Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior De Jaén

Trabajo Fin de Grado

DESARROLLO DE UN PROTOTIPO DE VIDEOJUEGO BASADO EN ROL DE ACCIÓN

Alumno: Gonzalo Martínez Romero

Tutor: Prof. D. Juan Roberto Jiménez Pérez

Dpto: Departamento De Informática

Junio, 2017

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Don Juan Roberto Jiménez Pérez, tutor del Proyecto Fin de Carrera titulado: Desarrollo de un prototipo de videojuego, que presenta Gonzalo Martínez Romero, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, Junio de 2017

El alumno: Los tutores:

Gonzalo Martínez Romero Juan Roberto Jiménez Pérez

Contenido

1. Introducción

Una de las principales razones por la cual elegí éste proyecto, viene motivada por los múltiples años que llevo en el sector del videojuego. En un primer momento, ésta dedicación, iba más orientada a la parte del jugador, que a la del desarrollador.

Conforme van pasando los años, el aprendizaje que he ido obteniendo sobre las distintas técnologías y disciplinas que gobiernan la informática, me han hecho ver el mundo de los videojuegos desde otro punto de vista.

Los intereses hacia éste sector, se decantan más hacia la parte tecnológica que hay detrás de cada escena, animación, personajes, inteligencia artificial o cualquier tipo de evento que pueda suceder durante una partida.

1.1 Estructura del documento

Para facilitar la lectura de éste documento, se va a comentar brevemente la estructura que se ha seguido:

Al ser un documento de tipo técnico, hay algunas palabras y términos, que requieren ser explicados para una mejor comprensión. Para ello, se han resaltado algunas palabras, utilizando la letra cursiva y se han añadido al final de cada apartado en una tabla, en la que aparece el término junto a su definición (Tabla 1).

Término	Definición

Tabla 1. Ejemplo tabla para la definición términos

En el punto 2, se han descrito todas las herramientas usadas durante el desarrollo. Previamente, se disponía de una idea general de los objetivos del proyecto, por lo que a partir de ahí se han considerado las herramientas a utilizar.

En el punto 3, se desarrollan de manera genérica y prototipada, cada uno de los elementos por los que va a estar formado el videojuego. Teniendo conocimiento de las tareas que se quieren realizar, se procede a planificarlas y organizarlas en el punto 4.

En el punto 5 y el punto 6, se explican de manera detallada la forma en la que han sido desarrollados los componentes del videojuego, profundizando en la parte de programación.

Cuando el prototipo esté finalizado, se procede a redactar una mini guía del juego en el punto 7 y su posterior evaluación en el punto 8.

En el punto 9, se ofrece como información extra al lector, el conjunto de técnicas que han sido usadas para mejorar el rendimiento del videojuego.

Para finalizar, en el punto 10, se describen los controles para poder jugar y se cierra con la conclusión en el punto 11.

1.2. Preludio a la motivación

Antes de comenzar la parte de motivación, hay que definir una serie de conceptos, para que toda la terminología se entienda de la mejor forma posible.

Los videojuegos se clasifican en un conjunto de categorías, como por ejemplo: acción, shooter, estrategia, deporte, carreras … etc. Cada una de estas modalidades, implican un modo de juego distinto, por lo que la forma de disfrutar de cada una ellas también será diferente.

Ilustración 1. Juego de acción

Dependiendo del tipo de juego que se vaya a desarrollar, los controles para poder jugarlo, también pueden llegar a ser distintos. En un juego de carreras, se puede utilizar un volante electrónico, sin embargo para jugar a un juego de tipo shooter, el volante no sería útil, por lo que se necesitaría un mando o un teclado, ya dependiendo de la platarfoma en la que se desarrolle.

Ilustración 2. Juego de estrategia

Otro tipo de categoría es la de **ROL** o **RPG** (Role Playing Game), la cual consiste en desempeñar un determinado papel dentro de una historia a través de un personaje que es controlado por el jugador. A lo largo de ésta, el personaje va evolucionando, al igual que la historia, lo que permite al jugador enfrascarse cada vez más en la historia conforme va avanzando el juego. Cabe destacar, que tambíen existen subgéneros, tales como rol-acción, rol-multiplayer, rol-tácticos… etc. (Wikijuegos, 2014)

Un **rol de acción**, combina los elementos típicos del ROL, junto con elementos tales como combates en tiempo real (Hack & Slash), que conducen al personaje a la obtención de armas, armaduras, items y demás accesorios que le permiten equiparlo, con el objetivo de poder combatir de la manera más satisfactoria posible.

Ilustración 3. Juego rol de acción

1.3. Motivación

Aprovechando que se disponen de ambos puntos de vista, el del jugador y el desarrollador, se ha visto una perfecta oportunidad para llevar a cabo el desarrollo de éste proyecto, el cual se ha enfocado al género **rol de acción**.

El motivo por el que se ha decidido realizar un videojuego de rol de acción, se debe a la libertad que éste otorga a la hora de implementar componentes y poder adaptarlos a una historia que ofrezca al jugador una buena experiencia durante el desarrollo del juego. Para aclarar ésta parte, lo mejor es poner un ejemplo: Un juego de coches, de deportes… etc, suelen ser juegos muy “estáticos”, en los que el objetivo siempre es el mismo, y el modo de juego suele ser repetitivo sin posibilidad de cambios muy notables. Por otro lado, los juegos de tipo ROL, ofrecen una variedad de objetivos más amplia y dinámica, junto con una mayor interactividad del jugador con los elementos del entorno,… etc.

En definitiva, para realizar de la manera más satisfactoria éste proyecto, se ha visto una muy buena opción desarrollar un videojuego de rol de acción, debido a que ofrece un abanico más amplio de posibilidades a la hora de implementar una mayor cantidad y variedad de elementos que, frente a otro tipo de géneros, como por ejemplo, los anteriormente comentados, cuyo desarrollo estaría más limitado debido al tipo de juego.

Para finalizar, éstos han sido uno de los motivos, aunque no el único, ya que, tambíen dispongo de más experiencia en éste tipo de videojuegos, lo cual, me ha servido de ayuda durante varias fases tanto en el proceso de creación como de desarrollo.

2. Herramientas usadas en el desarrollo

2.1. Elección del motor gráfico

Ilustración 4. Lista de motores gráficos

Uno de los principales componentes para el desarrollo de un videojuego, es el **motor gráfico**. En términos generales, un motor gráfico, provee al videojuego, un conjunto de recursos tales como renderizado de imágenes, físicas, sonidos, scripting, animaciones … etc. (Gregory, 2009)

Todo éstos elementos, permiten al desarrollador **abstraerse** durante el desarrollo, como por ejemplo, en la programación de físicas. En éste caso, no tendría que programarlas a bajo nivel, tan sólo tendría que utilizar el *API* que el motor le proporciona, para poder trabajar con la funcionalidades de éstas y enfocarse más a lo que viene siendo, el desarrollo del videojuego. (Unity, 2017a)

Término	Definición
API	Conjunto de funciones y procedimientos que ofrece alguna librería como capa de abstracción.

Tabla 2. Definición de API

A continuación se expone un **ejemplo** ilustrativo de las funciones de un motor gráfico:

Ilustración 5. Funciones de un motor gráfico

La capacidad, para gestionar todos éstos elementos de forma **eficiente**, junto con, una adecuada **flexibilidad** que se adecúe a las necesidades del videojuego, hacen decantarnos por el uso de unos u otros motores.

Para la **elección del motor**, se ha realizado una selección previa, formada por un conjunto de motores, que se adecúen a los requisitos del videojuego y la máquina en la que se va a desarrollar. Para ello, se han seleccionado aquellos cuya licencia estándar sea gratuita, tenga un renderizado 3D, que sea lo más eficiente posible y que el uso de librerias multimedia para *renderizado* de gráficos, sean *compatibles con el sistema operativo*. Lo ideal, es que también disponga de un amplio repertorio de documentación y una comunidad de usuarios lo más amplia y activa posible.

Término	Definición
Compatibles con el Sistema operativo	Tanto Direct3D como OpenGL son librerías que permiten la programación de gráficos, pero no todas se pueden usar en cualquier sistema operativo. Por ejemplo, Direct3D sólo está disponible para Windows, sin embargo, OpenGL lo está para Windows, Mac y Linux.
Renderizar	Generación de todos los elementos que forman una escena a partir de la ejecución de código.

Tabla 3. Definición compatibilidad entre S.O. y renderizar

Ilustración 6. Símbolo de Unity

Después de realizar la búsqueda con los criterios anteriormente dichos, **Unity** y **Unreal Engine 4,** han sido los más aptos:

En la parte de *scripting***, Unity**, nos ofrece la posibilidad de programar tanto en C# como en JavaScript. También nos permite un alto grado de abstracción, gracias a una amplia gama de APIs y recursos que evitan que el programador tenga que desarrollar algún módulo a un nivel más bajo, excepto para algún elemento que sea más específico y así lo requiera. (Unity, 2017a)

A la hora de **ejecutar** los scripts, Unity dispone de un recoletor de basura, por lo que no hay que preocuparse por liberar memoria. Tambíen dispone de un sistema, por el cual, cuando se genera un *GameObject*, se genera a la misma vez un puntero a dicho objeto. Esto permite tener una mayor eficiencia y menor consumo de memoria, ya que, cada vez que se quiera trabajar con dicho objeto, se hace a través de su puntero, evitando manejar íntegramente el objeto, que es más pesado. (Unity, 2017b)

Término	Definición
GameObject	En Unity, un GameObject es cualquier elemento del juego, desde una luz, un objeto… hasta una fuente de audio.
Scripting	Conjunto de órdenes asociadas a un componente

Tabla 4. Definición GameObject y Scripting

Unity contiene una gran cantidad de **documentación**, **tutoriales** y una comunidad de **usuarios** muy activa, lo que permite solucionar dudas o problemas un breve periodo de tiempo.

Lleva activo mucho más **tiempo** que Unreal, por lo que se presupone, que la comunidad de usuarios están más **experimentados** y el motor tendrá menos **bugs**.

Es **compatible** con las aplicaciones de modelado y animación 3D, tales como 3ds Max, Maya, Softimage, Cinema 4D, y Blender.

Ilustración 7. Símbolo de UnrealEngine

Por otro lado, **Unreal**, usa C++ y *BluePrints* para la parte de scripting. Tambíen cabe la posibilidad de usar únicamente BluePrints, sin necesidad de tener que programar en C++, aunque es un aspecto más orientado a desarrolladores principiantes. (UnrealEngine, 2017)

Término	Definición
BluePrints	Interfaz basada en nodos, que nos permite manejar los elementos del juego sin tener que tocar mucho código.

Tabla 5. Definición BluePrint

Unreal Engine 4, lanzó su primera versión totalmente **abierta** y **gratuita** en 2014, frente a Unity, la cual, si quieres una versión de código abierto, debes adquirir la versión de pago (Unity Pro).

Como **conclusión**, ambos motores, son muy parecidos, aunque finalmente se ha dicidido desarrollar el videojuego en Unity, ya que, a parte de lo anteriormente comentado, se tiene más experiencia en éste motor y es más amigable con software externo, como por ejemplo Blender o 3ds Max, aunque tambíen se prodría haber utilizado perfectamente Unreal, ya que ambos son muy buenos motores y los resultados los avalan.

2.2. IDE (Entorno de Desarrollo Integrado)

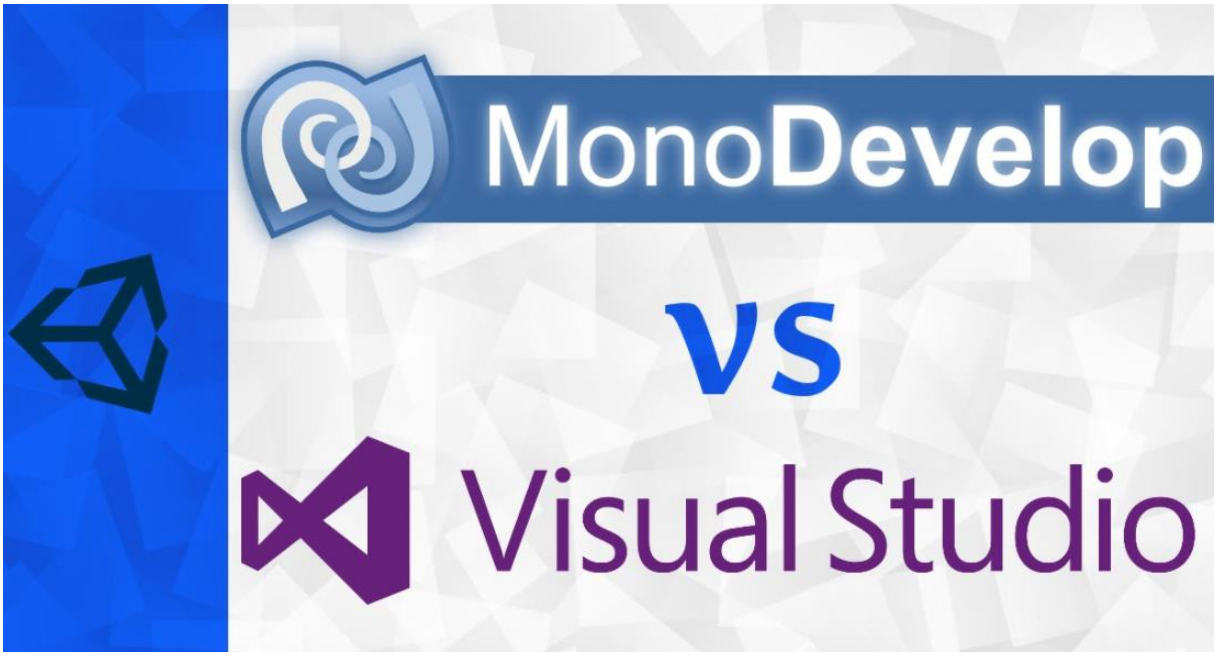


Ilustración 8. MonoDevelop vs VisualStudio

Para elegir el entorno de desarrollo más adecuado, la elección se situaba entre 2 modelos. Uno de ellos, MonoDevelop y el otro VisualStudio. Ambos IDEs permiten compilar y depurar código en C#, lo cual, es uno de los objetivos ya que el scripting va a utilizar ese lenguaje.

MonoDevelop, es un IDE que va principalmente enfocado a la programación en C# (también puede servir para .NET, Boo…etc), tiene un entorno sencillo, sin muchas funcionalidades y rápido de instalar. Posee las características básicas que se vienen buscando, sin sobrecargar mucho el entorno con plugins que luego no se usan y entorpecen el desarrollo. (MonoDevelop, 2017)

Por otro lado, **VisualStudio**, es un IDE mucho más completo, y por ende, mucho más pesado. La instalación de éste, requiere una gran cantidad de componentes y dependencias, por lo que a la hora de instalarlo se crean multitud de directorios, que dificultan su posterior desistalación. (Microsoft, 2017)

Ya que para el desarrallo, tan sólo se requiere un compilador básico de C#, con un IDE sencillo, que nos de una funcionalidad básica. Todos estos, han sido motivos suficientes para elegir como IDE a MonoDevelop.

2.3. Modelado y animaciones para modelos 3D

Ilustración 9.Logo de Blender

Blender, es un software muy potente que permite realizar todo tipo de acciones sobre modelos 3D, tales como: modelado, animaciones, simulaciones con fluidos…etc. Cabe destacar, que la propia aplicación, también dispone de manera integrada, de un motor propio, lo que amplía sus posibilidades de uso. (Blender, 2017)

De todas las características que nos ofrece, tan sólo han sido utilizadas las de modelado 3D, texturizado y *rigging and skinning* para hacer las animaciones y los modelos.

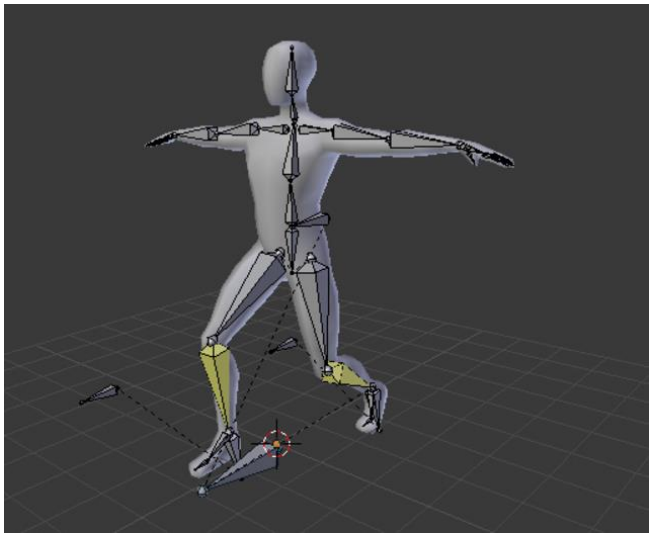
Término	Definición
Rigging and skinning	Es una técnica, por la cual, a un modelo 3D se le dota de una jerarquía de huesos, articulaciones y un conjunto de restricciones, con el objetivo de poder simular ciertos tipos de movimientos. Una vez esté añadido, moviendo dicho esqueleto, se mueve el modelo.  Ilustración 10. Ejemplo de rigging and skinning

Tabla 6. Definición rigging and skinning

Tanto Blender como Unity son totalmente compatibles, lo cual, facilita exportación de modelos entre ambos, motivo suficiente para agregarlo a la lista de software utlizado.

2.4. Diseño

Ilustración 11. Logo de VisualParadigm

Para el diseño de la estructura software que va a seguir el videojuego, se han utilizado multitud de diagramas: diagramas de clases, conceptuales, de interacción, máquinas de estado.. etc. Para poder abordar tal cantidad y variedad de diagramas, se va ha utilizar **Visual Paradigm**.

Visual Paradigm, es un software que satisface todas las necesidades anteriormente dichas, además de disponer de una interfaz completa y fácil de usar. Todas éstas características lo dotan para que haya sido elegido para el desarrollo de éste proyecto.

Tambíen se ha hecho uso de DrawIO (una herramienta con características similares a los de Visual Paradigm) pero en menor medida, ya que no ofrece la misma flexibilidad y comodidad que Visual Paradigm.

(Visual Paradigm, 2017)

2.5. Edición de audio

Ilustración 12. Logo de audacity

Una parte fundamental en los videojuegos es el sonido. Debido a ello, es muy importante que la duración y tonalidades de los sonidos se adecúen y sincronicen con los movimientos del personaje y la cámara o con los eventos que ocurren alrededor.

Para realizar la edición de sonido, se ha utilizado **Audacity**, un editor de audio gratuito, fácil de usar, que satisface todas las necesidades que se van a requerir durante el desarrollo de éste proyecto.

(Audacity, 2017)

3. Diseño inicial del videojuego

A continuación, se van a exponer un conjunto de acontecimientos previos, realizados antes de comenzar el desarrollo del videojuego.

3.1. Historia

El juego está basado en una historia, en la cual, el personaje principal es partícipe de ésta. Durante el juego, se va obteniendo información, de forma que la historia se va comprendiendo a medida que se va avanzando.

3.2. Desarrollo de la historia

Desde tiempos inmemoriables, las fuerzas del bien consiguieron expulsar al mal que reinaba en nuestro mundo, un mundo, donde sólo reinaba el caos y la destrucción. Los grandes guerreros, tras una gran batalla, consiguieron expulsar al mal. Para evitar que regresaran, hicieron uso del poder de los pilares maestros, que permitian sellar los portales, con el fin, de que el mal no pudiera regresar.

Ilustración 13. Portal del mal

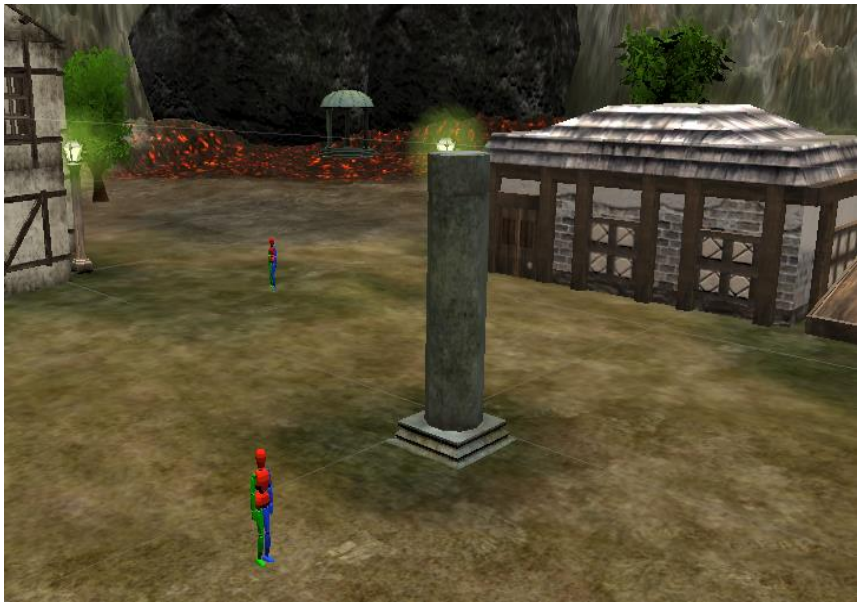


Ilustración 14. Pilar maestro

Ha sido asi durante miles de años, aunque al parecer, algo está pasando ya que, son, los numerosos pueblos, los que han sido arrasados de manera extraña y se piensa, que el mal, está intentado regresar, ya que a su paso, arrasaban los pilares maestros y a la población.

Cada vez que lo consiguen, la fuerza mágica que otorgan los pilares maestros para detener al enemigo, es cada vez menor. Si consiguen destruir la cantidad necesaria para volver a traer de nuevo a todo su ejército, volverá a reinar las guerras, destrucción y el caos.

Ilustración 15. Pueblo destruido

Los grandes guerreros no saben que hacer, debido a que la aparición de dichos portales se desconoce y el enemigo desaparece rápidamente, impidiendo tener la mínima capacidad de reacción, por lo que sólo consiguen ir detrás de sombras que se desvanecen.

La única solución antes de que el enemigo consiga estabilizar los portales, es conseguir acceder a través de uno de ellos al reino del mal y descubrir que es lo que está sucediendo.

Nuestro personaje principal, un guerrero, decide ir de forma solitaria en busca del misterio de la aparición de los portales que causa destrucción a su paso. Su plan, no es sólo combatir a los enemigos si no ir mucho más allá. Su objetivo será conseguir atravesar uno de esos portales para llegar al mundo del mal y poder descubrir cómo consiguen reabrir los portales y eliminar a los causantes de éstos acontecimientos.

Ilustración 16. Reino del mal

3.3. Prototipos a papel

A continuación, se van a exponer los **elementos**, **ideas** y **prototipos** que fueron **diseñados antes** de empezar con el desarrollo del videojuego. El objetivo de ésta fase, fue que, cuando se empezara la fase de desarrollo, la implementación de los elementos fuera lo menos improvisada posible y surgiera de una idea lo suficientemente sólida para que tan sólo se tuvieran que refinar algunos de los requisitos.

3.3.1. Prototipo a papel de los portales

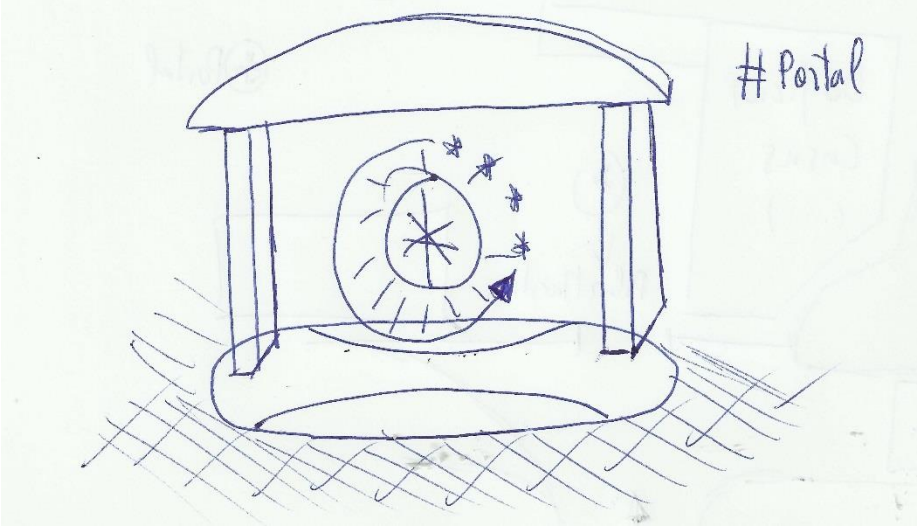


Ilustración 17. Prototipo portal del mal

3.3.2. Prototipo a papel de los pilares maestros

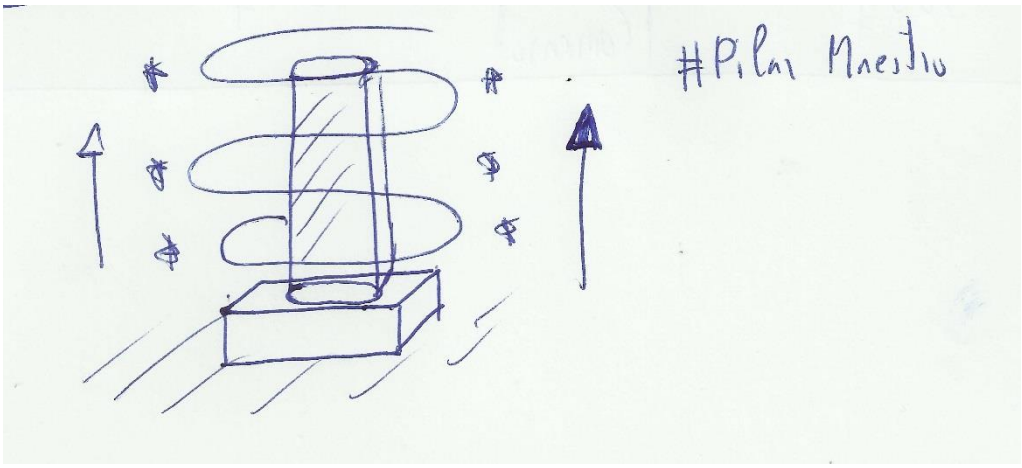


Ilustración 18. Prototipo pilar maestro

3.3.3. Prototipo a papel de la misión 1: “Defender el pilar”

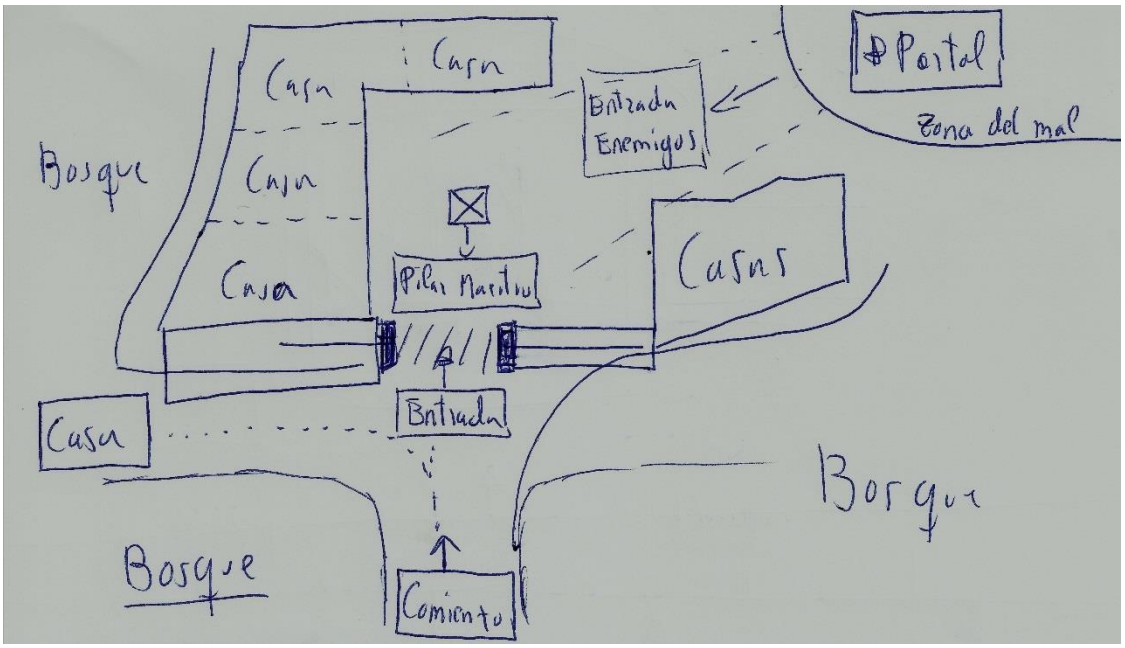


Ilustración 19. Prototipo mapa misión 1

3.3.4. Prototipo a papel de la misión 2: “La zona del mal”

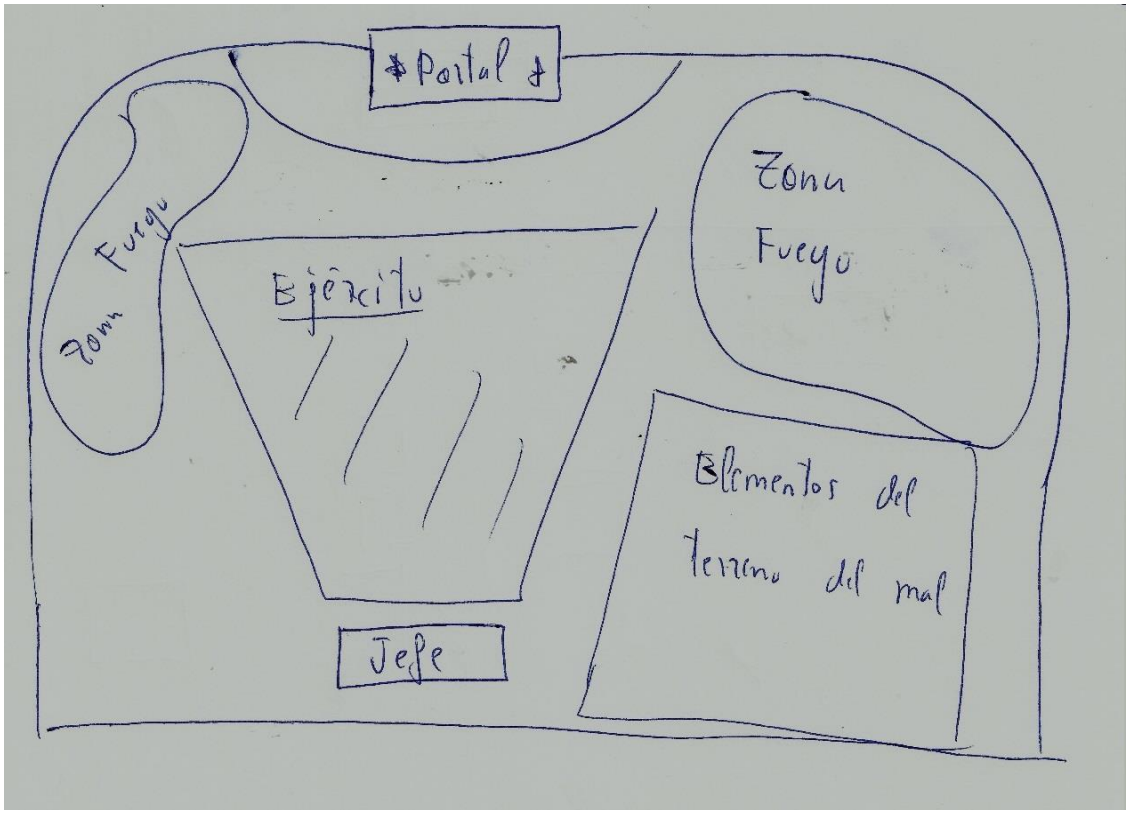


Ilustración 20. Prototipo mapa misión 2

3.4. Idea inicial del personaje principal

3.4.1. Modelo del personaje principal

En el siguiente diagrama se representa la **estructura** que va a tener el **modelo** del personaje y su relación con los distintos elementos que lo componen. No se han contemplado elementos externos tales como la interacción con los enemigos o la interacción con los *NPC (Non Person Character*), ya que éste diagrama se centra únicamente en el personaje y los elementos por los que va a estar formado.

Término	Definición
NPC	Siglas de Non Person Character (Personaje no jugador), es un personaje controlado por el juego.

Tabla 7. Definición NPC

Esta parte sólo sirve para tener una idea previa antes de comenzar a implementar al personaje principal. El **diagrama** mostrado a continuación, tan sólo **ofrecerá** una **idea** relativa, ya que en fases posteriores, se procederá a realizar un desarrollo más detallado, acorde el sistema sobre el que va a estar soportado.

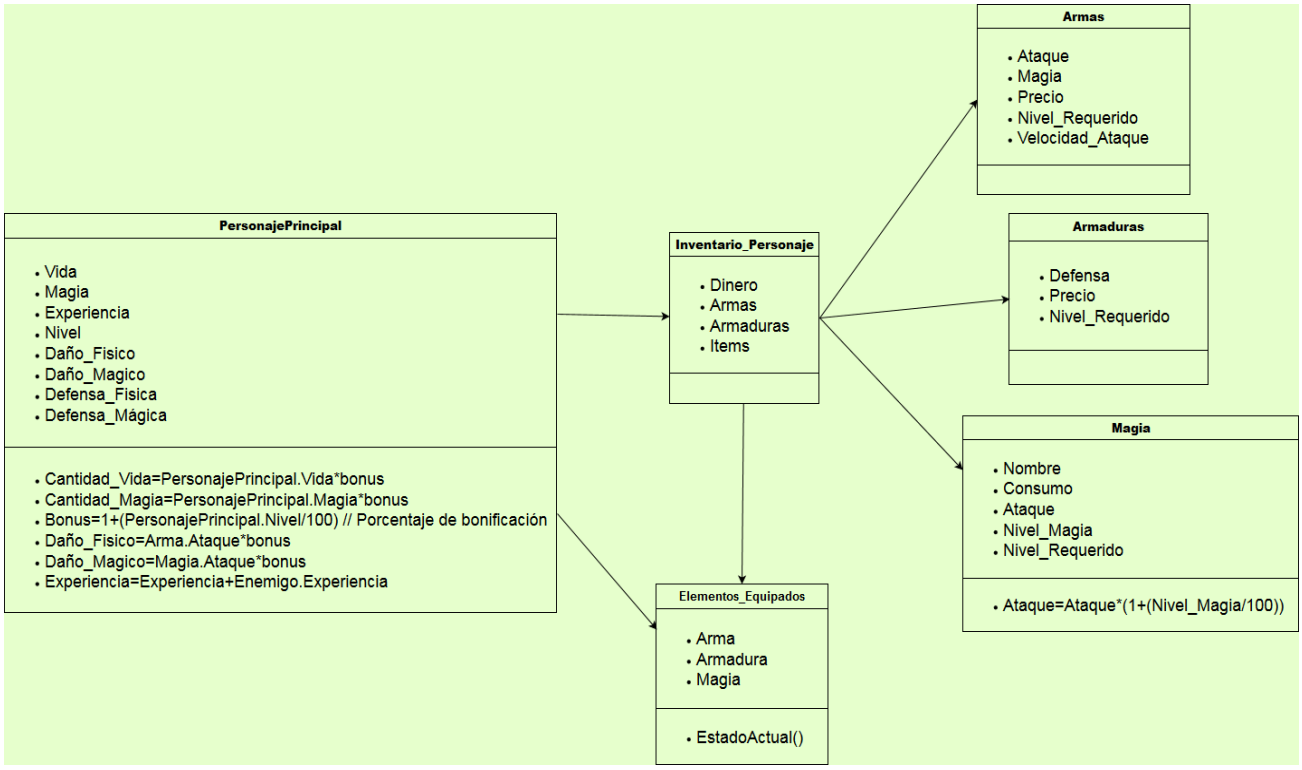


Ilustración 21. Prototipo diagrama del personaje principal

3.4.2. Animaciones del personaje principal

El personaje dispondrá de un conjunto de **animaciones**, que, para una mejor organización se han divido en **3 grupos**:

1. **Movimiento básicos**
    a. **Idle**: Modo reposo.
    b. **Walk**: Animación para andar. Sirve tanto para andar hacia delante como hacia atrás. Esto se debe a que haciendo uso de la cinemática inversa se puede generar el movimiento hacia atrás a partir del movimiento hacia delante.
    c. **Dead**: Animación de morir.

2. **Realizar ataques**
    a. **Ataque 1**: Ataque de tipo rápido que necesita poco recorrido del arma.
    b. **Ataque 2**: Ataque más poderoso con un mayor recorrido
    c. **Ataque 3**: Ataque frontal por el cual realiza una puntillada con el arma equipada.

3. **Realizar magias**
    a. Magia 1: Realiza un salto dando al arma equipada, un poder de ataque extra.
    b. Magia 2: Lanzar bola de energía.
    c. Magia 3: Realiza un salto, clavando el arma en el suelo, provocando una gran explosión a su alrededor.
    d. Magia 4: Movimiento giratorio con el arma extendida.

3.4.3. Modelo 3D

Para crear el **modelo 3D** del personaje, a modo de **prototipo**, se va a utilizar una estructura básica humana, con el objetivo de poder añadir en un futuro, a su jerarquía de componentes, elementos tales como armaduras, espadas, escudos… y demás elementos típicos en éste tipo de juegos.

Ya que el modelado requiere saber técnicas de dibujo y una visión más artística, se ha realizado un modelo simple, que ofrezca una estructura humana. Debido a ello, se ha escogido un boceto ya hecho, para generar el modelo 3D a partir de éste.

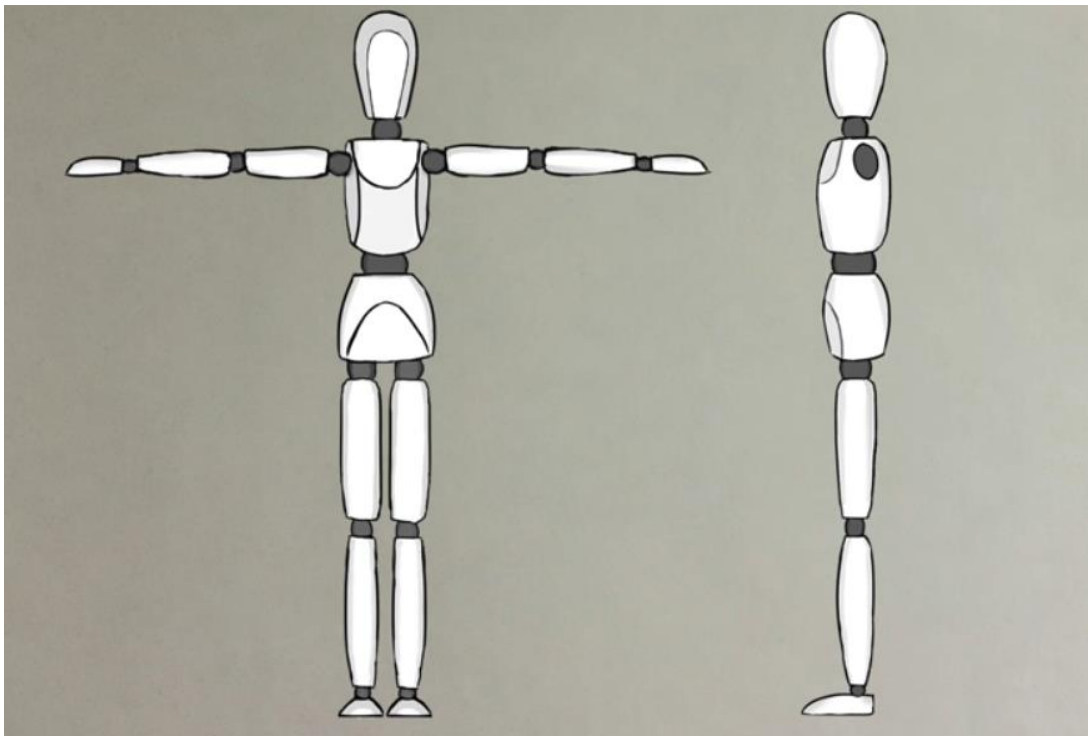


Ilustración 22. Prototipo modelo del personaje principal

3.4.4. Sistema de niveles y stats

Cada vez que el personaje sube de **nivel**, se van a desarrollar una serie de funciones que determinan la manera, en la que, los *stats* del personaje se van a ver afectados. Dependiendo del nivel del personaje , los stats se van a ver afectados de la siguiente manera:

Exp = Exp * (lvl / 1.5)

HP = HP* (lvl / 1.25)

SP = SP* (lvl / 1.25)

DmgFis = DmgFis * (lvl / 1.25)

DmgMag = DmgMag * (lvl / 1.25)

Def = Def*(lvl / 1.25)

Donde:

Exp = Experiencia necesaria para subir de nivel

Lvl = Nivel actual del personaje

HP = Cantidad de vida

SP = Cantidad de magia

DmgFis = Daño que causa el personaje al realizar un ataque físico

DmgMag = Daño que causa el personaje al realizar un ataque mágico

Def = Defensa frente ataques de enemigos

Término	Definición
Stats	Características del personaje, como por ejemplo: cantidad vida (HP), magia (SP), valores de ataque y defensa, experiencia …etc.

Tabla 8. Definición de stats

Éste tipo de sistema es muy rígido, pero fácil de implementar, ya que está basado en incrementar un tanto por ciento los valores anteriores, dependiendo del nivel en el que se esté.

Durante ésta fase, también se contempló la posibilidad de crear unas **tablas de correspondencia** en una **base de datos**. El objetivo de implementarlo, es el de eliminar las funciones anteriormente dichas y asociar a cada nivel y sus stats, unos valores ya prefijados que se obtendrían de una base de datos. Ésta forma sería más flexible, ya que, la evolución del personaje podría ser fácilmente modificada, con el simple hecho de cambiar unas tuplas, en lugar de modificar o tener que introducir más código.

Ejemplo de la tabla de correspondencia:

	Nivel 1	Nivel 2	Nivel lvl
Experiencia Necesaria	Exp = 1200	Exp = 3000	Exp = Valor
HP Máxima	HP = 1100	HP = 1500	HP = Valor
SP Máxima	SP = 1080	SP = 1200	SP = Valor
(…)	(…)	(…)	(…)

Tabla 9. Tabla de correspondencia

Cabe destacar, que el personaje subirá de nivel, cuando se haya alcanzado la experiencia necesaria, requerida por el nivel en el que se encuentre.

3.4.5. Inventario

El personaje dispondrá de un **inventario**, en el que se almacenarán todos los items adquiridos a lo largo de la partida. Se podrá interactuar a través de un canvas o interfaz que ofreza la posibilidad de poder visualizar dichos elementos y poder interactuar con éstos.

El *layout* del inventario también dispondrá de una parte en la que se puedan visualizar los stats del personaje, para así poder dar información al jugador.

Cada vez que se equipe un ítem, los valores y bonus que tenga, serán sumados a los stats del personaje. Por ejemplo, al equiparse un arma, aumentan los valores de ataque físicos y mágicos y con una armadura, vida y defensa.

Término	Definición
Layout	Forma en la que se organizan los elementos de una interfaz

Tabla 10. Definición de layout

3.5. Idea general de los enemigos

Los **enemigos** también dispondrán de una serie de stats, aunque ligeramente distintos a los del personaje principal. Se mantienen stats tales como HP, SP, valores de ataque ,Exp, siendo Exp la experiencia que se le va a sumar al personaje, una vez éste sea eliminado.

Todos los valores de los stats de un mismo tipo de enemigo, serán constantes e invariables y sólo serán distintos si la raza del enemigo tambíen lo es.

Cabe destacar, que el daño recibido, tanto por el enemigo como por el personaje, se calculará de la siguiente manera:

DañoRecibido = (AtaqueRecibido–Defensa)>0 ? (AtaqueRecibido– Defensa) : 0

Se establece que si el daño recibido no supera a la defensa, entonces es igual a 0, ya que, por cada golpe, le sumaría vida, lo cual, no es lo que se desea.

3.6. Idea general de las interfaces

3.6.1. Interfaz general

La **interfaz general,** va a ser la interfaz que va a estar visible siempre que se esté jugando, desde la cual se va a poder acceder al inventario y al menú a la vez que se va a poder visualizar stats tales como HP, SP , nivel y Exp en tiempo real, es decir, cada vez que sean modificados serán actualizados de manera inmediata.

Para la visualización de los stats, se utilizará la proporción obtenida de realizar el siguiente cálculo: (ValorActual / ValorMáximo).

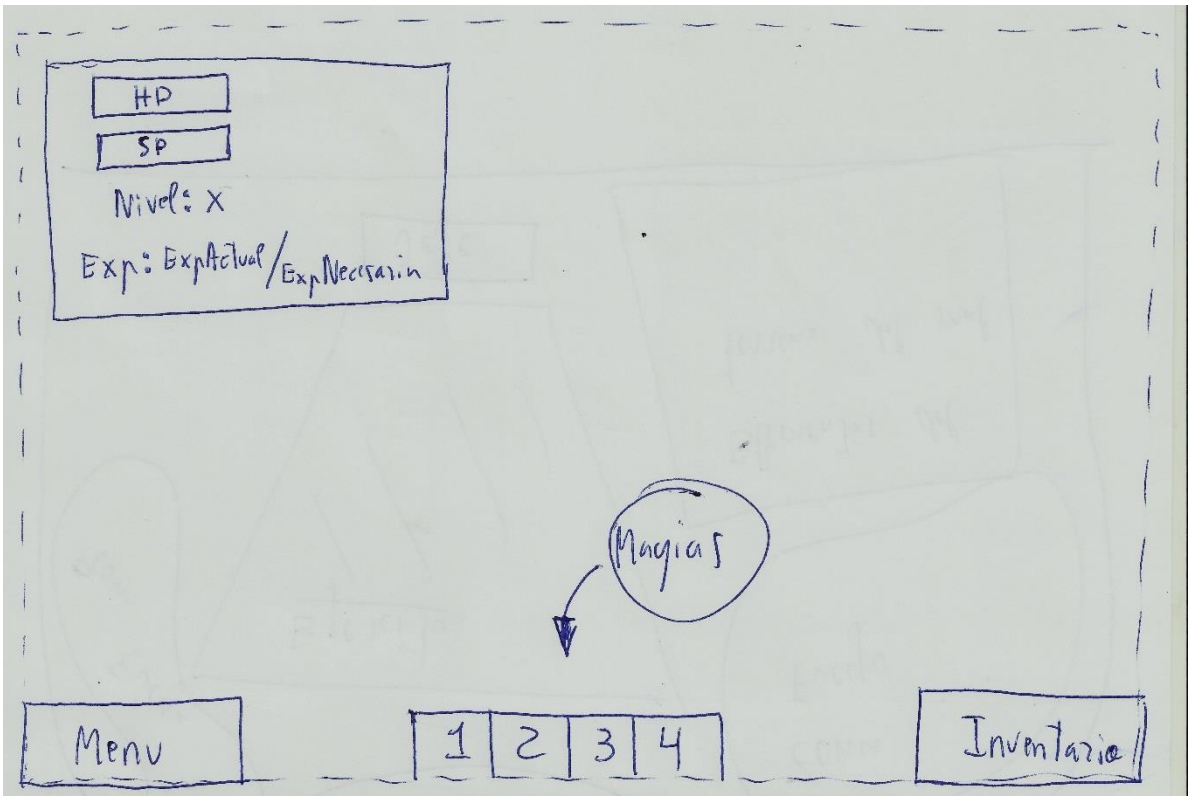


Ilustración 23. Prototipo interfaz general

3.6.2. Interfaz del inventario

Interfaz desde la que se ven los ítems almacenados y se podrá elegir cual llevar equipados. También se podrán visualizar los stats del personaje, con el objetivo, de que, el jugador pueda ver los cambios que surgen cuando se equipa un item.

A diferencia de los stats de HP y SP que se ven en la interfaz general, éstos, van a ser los **valores máximos** de los que dispone el personaje en el momento actual y no los valores actuales.

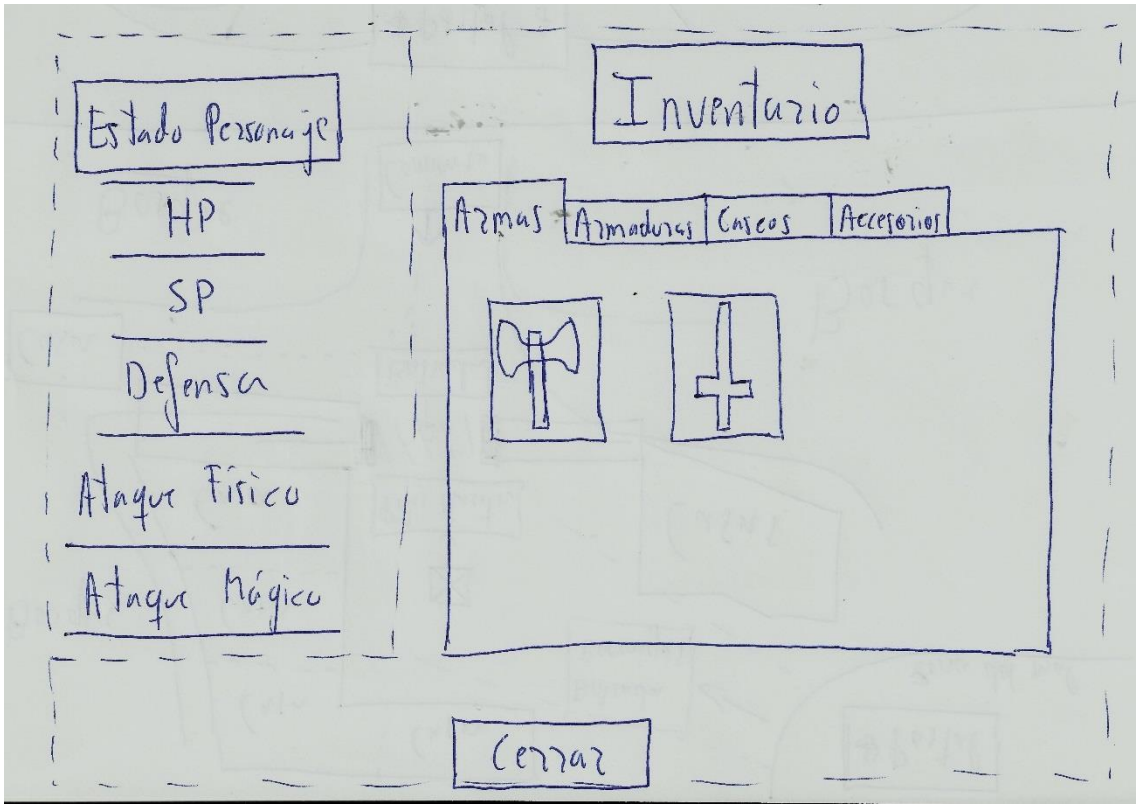


Ilustración 24. Prototipo del inventario

4. Planificación y organización

4.1. Distribución de las tareas

A continuación, se muestra un diagrama de tareas, en el que se exponen todos los procesos en los que se ha desglosado el proyecto y el orden de ejecución que se ha seguido.

La prioridad con la que cada tarea debe ser resuelta, se ha representado mediante lineas de flujo, que indican la tareas antecesoras y predecesoras.

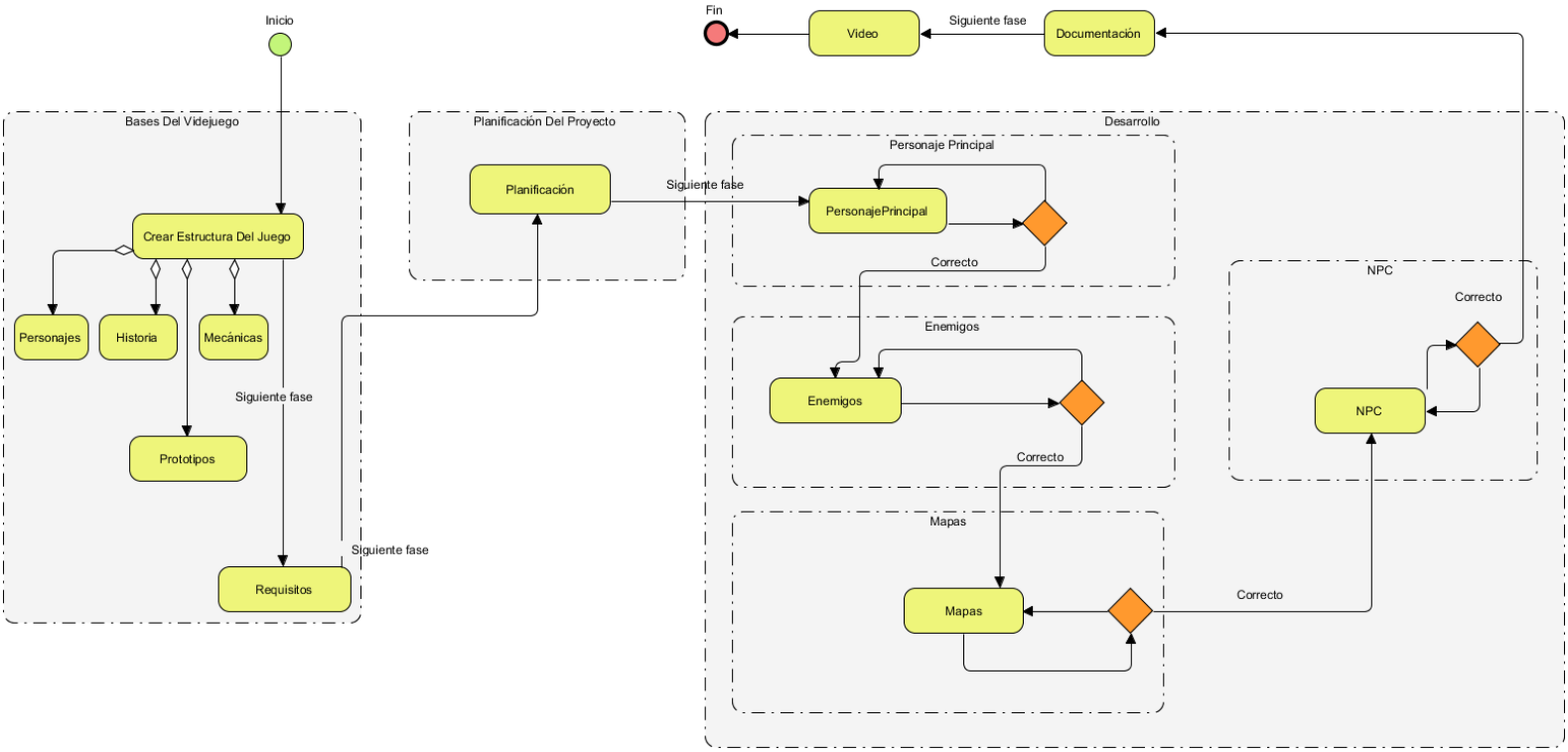


Ilustración 25. Diagrama de tareas

Las tareas contenidas en el paquete “Desarrollo”, se analizarán con más detalle en el punto 4.2.1 Incrementos.

En la primera sección se establecen los requisitos y la estructura que va a tener el videojuego. Una vez se sabe el objetivo que se quiere conseguir, se procede a realizar la planificación apoyándose en la metodología usada. Cabe destacar, que no todas las tareas se van a desarrollar usando dicha metodología. Tan sólo se va a aplicar para la partes funcionales del videojuego, no involucrando partes como la documentación, video o la creación de prototipos. Todas las tareas se van a ver reflejadas en el diagrama de Gantt.

4.2. Metodología usada para el desarrollo

Para el desarrollo de éste proyecto se ha utilizado el **método incremental** (OkHosting, 2016) (eumed, s.f.). La metodología de desarrollo que ofrece éste método, ha permitido finalizar con éxito el proyecto debido a las siguientes razones:

Cada vez que se finaliza un incremento, se tiene la posibilidad de ver una **parte funcional**. Esto ha permitido, que se puedan hacer modificaciones y correcciones de errores, antes de continuar con otras partes. Cada vez que se pase al siguiente incremento, se garantiza en cierto modo, que la parte implementada es totalmente funcional y cumple con los requisitos propuestos. Una vez que el proyecto esté finalizado, se realizarán test más en profundidad por parte de otras personas.

Realizar entregas de partes funcionales cada vez que se finaliza un incremento, permite **modularizar** el desarrollo del **proyecto**. Como medida de prevención frente a posibles adversidades que puedan retrasar la entrega del proyecto, es un punto muy a favor usar la metodología incremental. En caso de producirse cualquier problema durante el desarrollo, que pueda retrasar el proyecto, se pueden reconsiderar ciertas iteraciones para ser realizadas o no, ya que los resultados obtenidos en cada iteración son independientes. En caso de no tener tiempo para finalizar todos los incrementos, se tendría un proyecto totalmente funcional, aunque incompleto.

Para **organizar** cada uno de los **incrementos**, en primer lugar, se sitúan las partes más nucleares del proyecto, dejando para el final, las partes menos importantes.

Antes de comenzar con el desarrollo, la aplicación de ésta metodología requiere que se conozcan al principio, de manera genérica, los requisitos del proyecto. El análisis previo, realizado en el punto 3, ha permitido obtener la lista de requisitos necesarios, que a lo largo del proyecto, serán perfeccionados en cada uno de los incrementos.

Cada **incremento** está formado por **4 fases**: análisis, diseño, implementación y pruebas.

Durante la fase de **análisis**, se refinan los requisitos y se realizan algunos prototipos, necesarios para comprender, de la forma más completa posible, el resultado que se desea obtener al finalizar el incremento.

En la fase de **diseño**, se establece la estructura y el conjunto de diagramas necesarios, que se utilizarán como guía en la fase de implementación.

En la fase de **implementación**, se procede a realizar el código y todos los elementos necesarios basándose en los diagramas obtenidos en la fase de diseño.

Finalmente, en la parte de **pruebas**, se realizan los test necesarios para verificar que todo lo implementado funciona correctamente.

4.2.1. Incrementos

Los incrementos están formados por un conjunto de tareas que engloban una funcionalidad específica del videojuego.

En el **incremento 1**, se desarrolla el personaje principal y todos los elementos que lo forman.

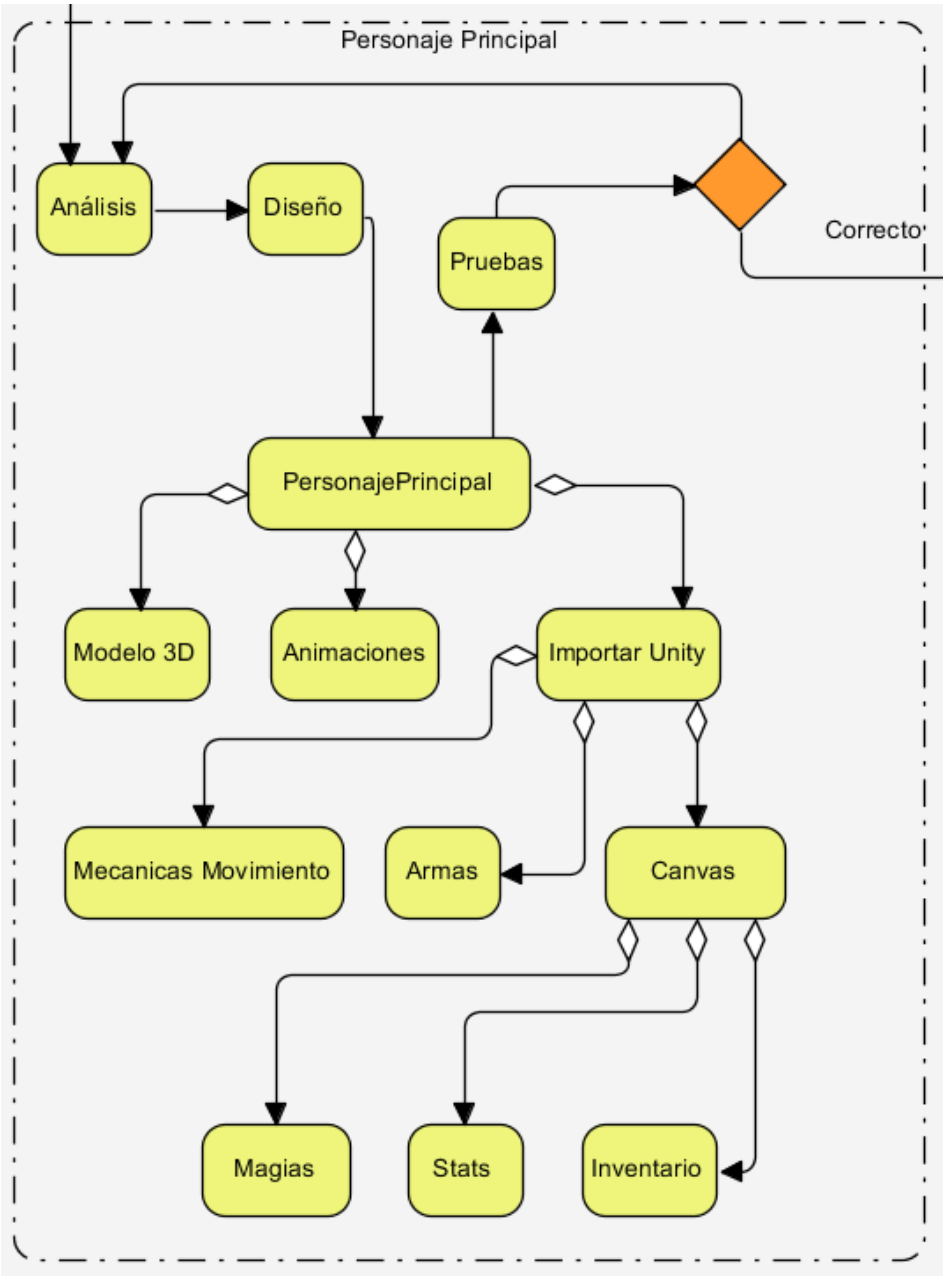


Ilustración 26. Diagrama de tareas personaje principal

En el **incremento 2**, se desarrollan a los enemigos y se comienza a hacer cuando el personaje principal es totalmente funcional. Todas las fases del incremento 1 deben estár finalizadas de manera satisfactoria, ya que cada incremento se construye sobre el anterior.

Algunas partes de la funcionalidades de los enemigos, utilizan recursos del personaje principal, por lo que se necesita que esté completo y funcional.

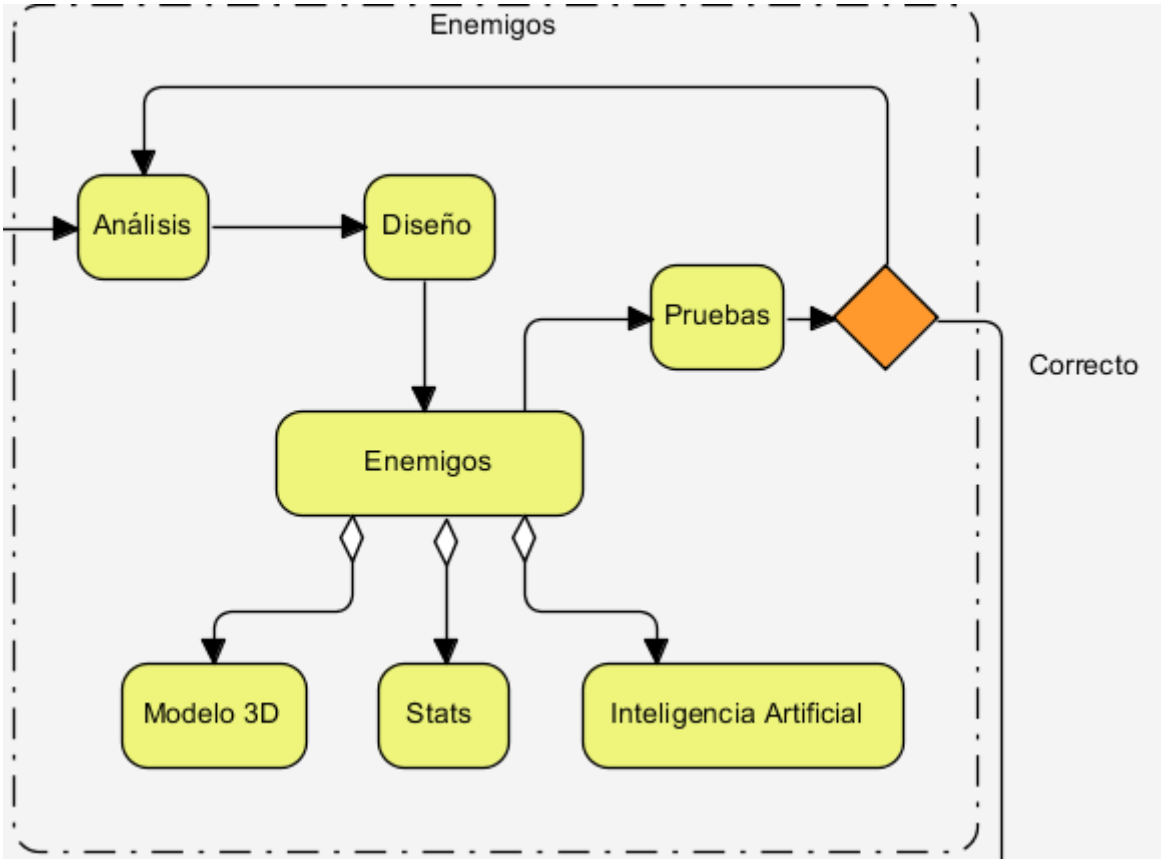


Ilustración 27. Diagrama de tareas de los enemigos

En el **incremento 3**, se desarrolla la creación de los mapas. Al realizar el mapa de la misión 1, se crean todos los elementos necesarios para poder realizar la misión, tales como el GameController y el MasterPillar, ambos incluidos en la tarea "Misión1".

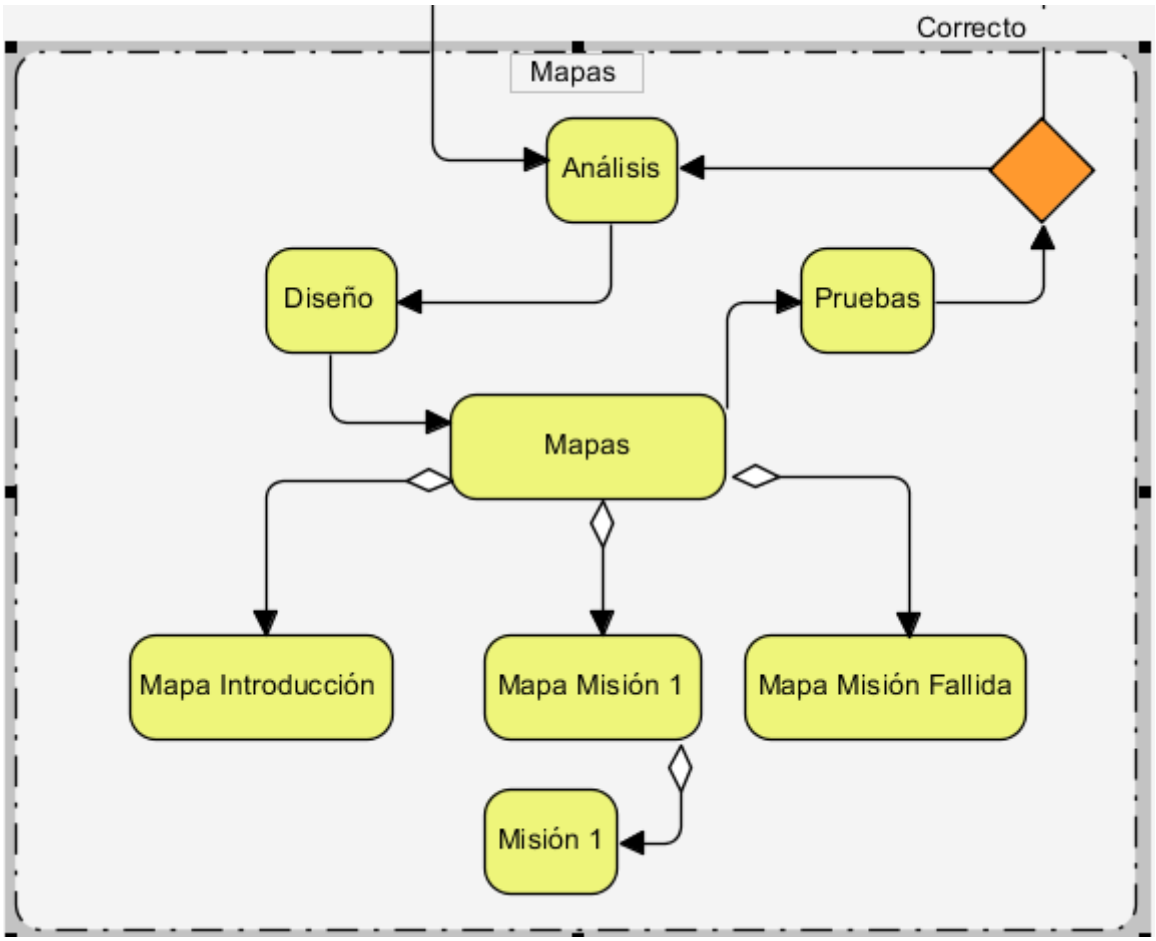


Ilustración 28. Diagrama de tareas de los mapas

El **incremento 4** está formado por creación de los NPC, un último aliciente para mejorar la experiencia de juego.

Los NPC defenderán y ayudarán al jugador a defender el pilar frente al ataque enemigo.

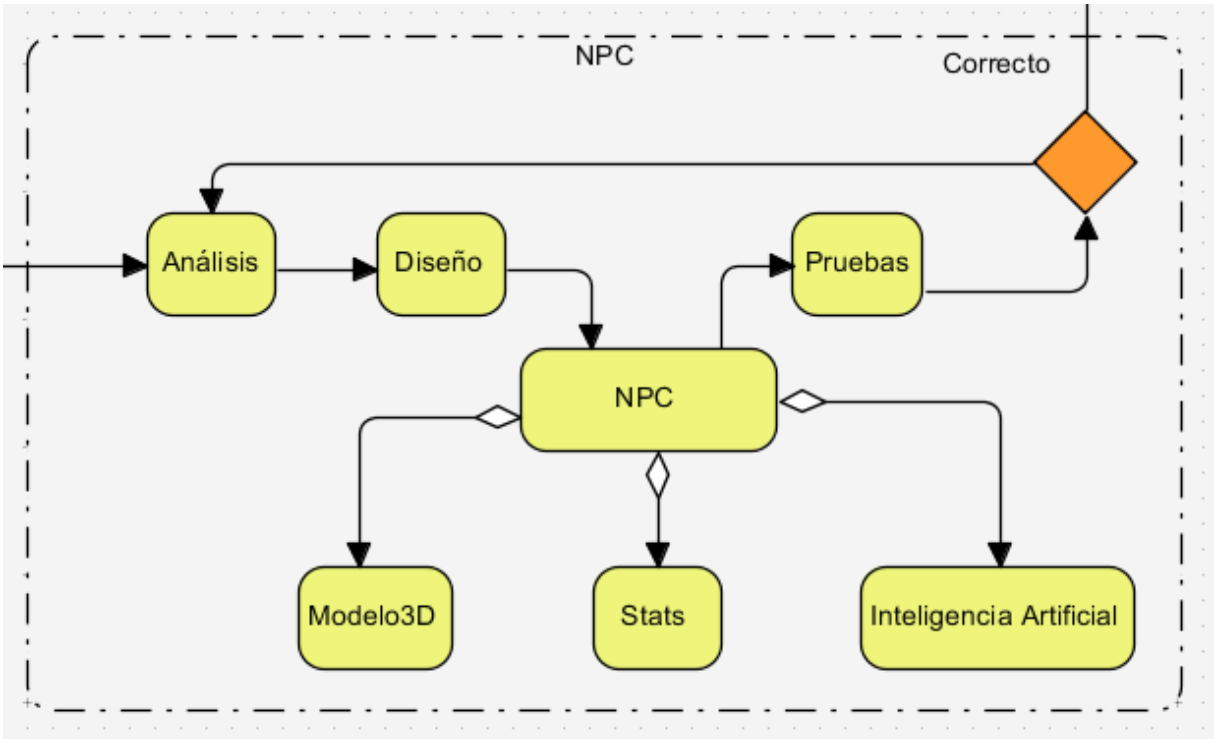


Ilustración 29. Diagrama de tareas de los NPC

4.3. Diagrama de Gantt

El **diagrama de Gantt** es una herramienta gráfica que permite ver el tiempo dedicado a cada una de las tareas realizadas. (Instituto de gestión de proyectos, 2000)

Se han considerado los tiempos dedicados a la planificación previa al desarrollo del proyecto, documentación y creación del video final, a parte del desarrollo del propio videojuego, ya que todo forma parte del mismo proyecto.

El tiempo dedicado a cada tarea, considera también el tiempo dedicado al aprendizaje necesitado para poder desarrollarla, ya que hay partes en las que no se disponían de conocimientos previos.

Para la crear la **estructura del videojuego**, se han necesitado 14 días, comprendidos entre el 1 y 14 de febrero.

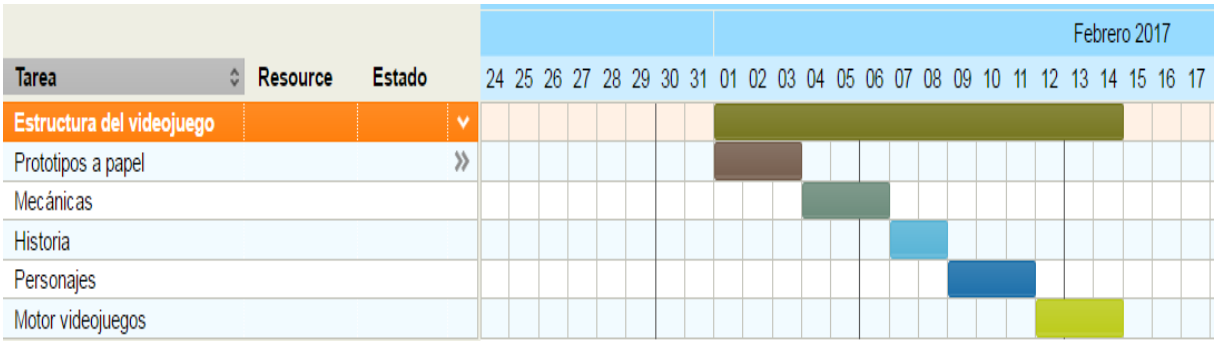


Ilustración 30. Diagrama de Gantt. Estructura del videojuego

Para la parte con mayor carga de trabajo, el **personaje principal**, se han invertido un total de 56 días.

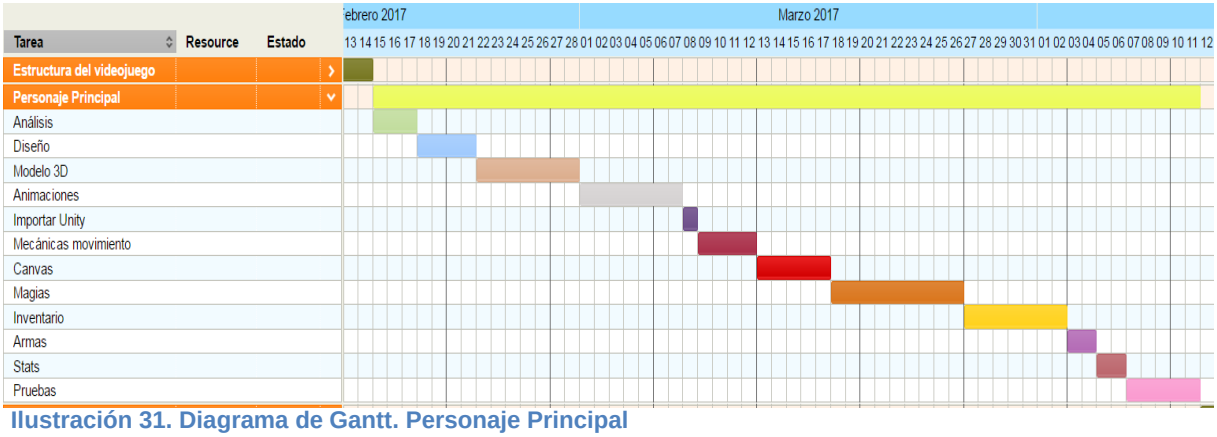


Ilustración 31. Diagrama de Gantt. Personaje Principal

Posteriormente, para los **enemigos**, se le ha dedicado un tiempo de 21 días.

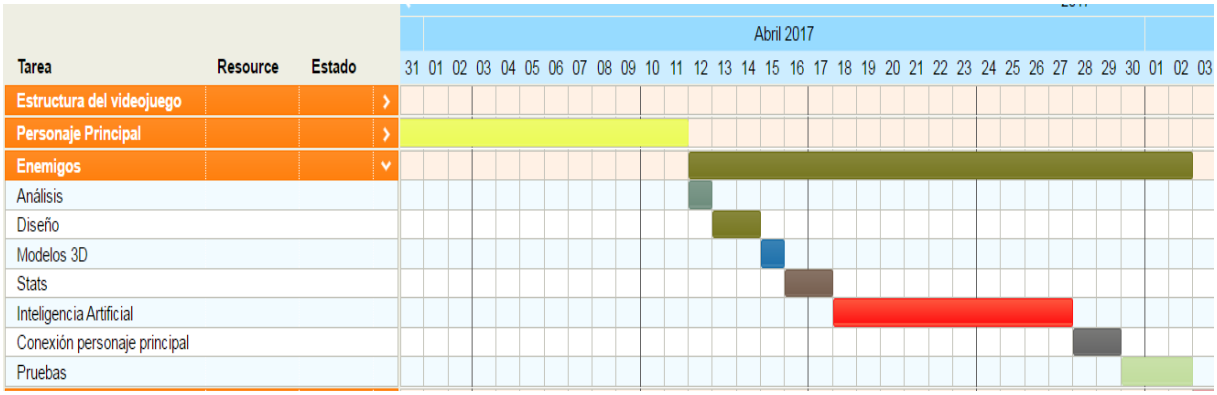


Ilustración 32. Diagrama de Gantt. Enemigos

Para la creación de los **mapas**, se han tardado unos 10 días.

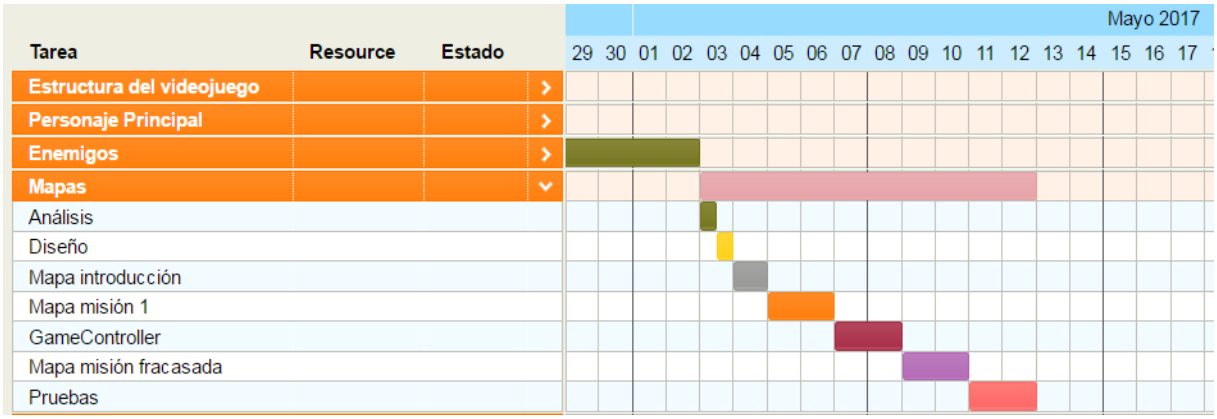


Ilustración 33. Diagrama de Gantt. Mapas

Para los **NPC**, se les ha dedicado 9 días.

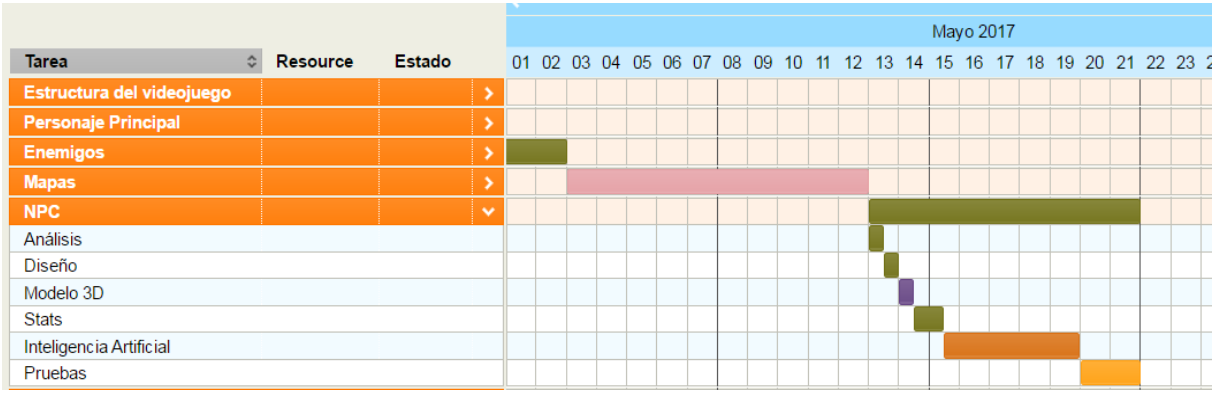


Ilustración 34. Diagrama de Gantt. NPC

La **documentación** se ha finalizado en 15 días, divididos en 3 bloques de 5, 6 y 4 días.

Para los **test** se han dejado unos 20 días.

Para la correción de **bugs** unos 3 días.

Para finalizar, se han empleado unos 6 días para la realización del **video** final.

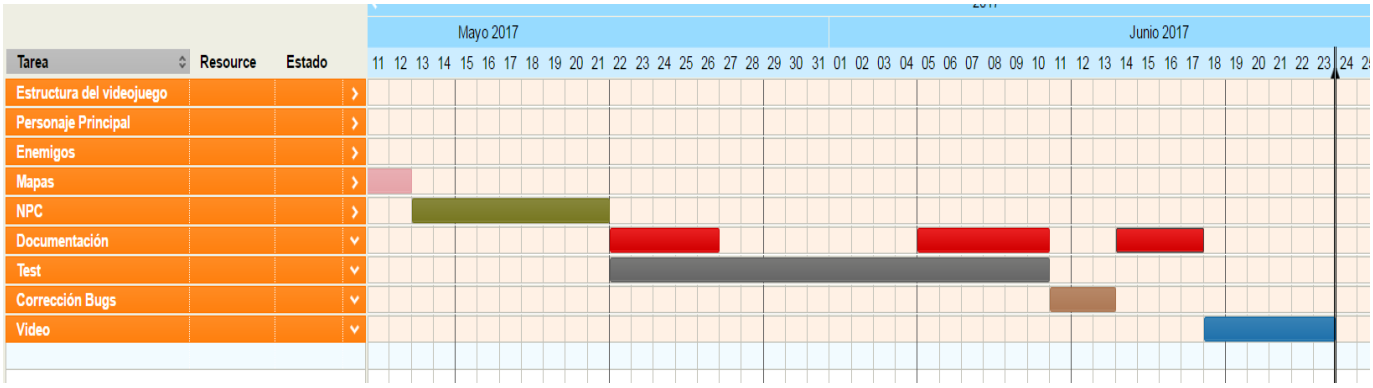


Ilustración 35. Diagrama de Gantt. Documentación, test y video

La tarea denominada como Documentación, está particionada debido a que el periodo del 27 de mayo al 4 de junio, ha sido la época de exámenes. Aún así, se ha aprovechado para que otras personas puedan ir testeando el juego, por ello, la duración de la tarea Test, se realiza de forma paralela y al mismo tiempo que la documentación.

Al finalizar, hay un breve periodo para corregir bugs e incidencias ocurridas en los test, durante los dias del 11 al 13 de Junio.

El resto de tiempo, ha sido para completar la documentación y hacer el video final.

4.4. Estimación de costes

Para desarrollar un videojuego, hay que tener en cuenta que es un proyecto extremadamente multidisciplinar. Un videojuego está formado por artistas, programadores, analistas, marketing…etc y demás roles, cuya aportación en el proyecto es indispensable. (Devueco, 2017)

A continuación se van a exponer los recursos y perfiles necesitados durante el desarrollo de éste proyecto.

4.4.1. Software

Recurso Utilizado	Definición	Coste	Coste Final
Blender	Modelado y animación 3D	Gratuito	
Unity	Motor gráfico	Versión profesional 111€/Puesto	111€ * 10 puestos = 1110€
MonoDevelop	IDE programación	Gratuito	
Visual Paradigm	Diseño software	Licencia estándar 400€	400€
Audacity	Modelado de sonido	Gratuito	
Windows 7 Ultimate	S.O.	60€/Puesto	60€ * 10 puestos = 600€

Tabla 11. Recursos software

4.4.2. Desarrolladores

Perfil	Objetivo	Coste	Coste Final
Programador Generalista	Programar lógica interna del videojuego	15€/hora	15€*150h=2250€
Diseñador De Software	Diseñar modelos software y relaciones entre componentes	10€/hora	10€*15h=150€
Programador De Interfaces	Programar los canvas	15€/hora	15€*30h=450€
Analista	Comprobación de la eficiencia y rendimiento	7€/hora	7€*5h=35€
Programador Cinemáticas	Cinemáticas internas del videojuego	15€/hora	15€*20h=300€
Programador de IA	Programación de la inteligencia artificial	20€/hora	20€*100h=2000€

Tabla 12. Desarrolladores

4.4.3. Artistas

Perfil	Objetivo	Coste	Número de horas
Diseño del videojuego	Diseñar la arquitectura del juego, relación entre sus componentes y historia	20€/hora	20€*20h=400€
Creación/Adaptación modelos 3D	Crear/Adaptar los modelos 3D de los personajes	10€/hora	10€*20h=200€
Modelado de mapas	Creación de los mapas	10€/Hora	10€*20h=200€
Animador modelos 3D	Animación de los modelos	10€/hora	10€*30h=300€

Tabla 13. Artistas

4.4.4. Marketing

Perfil	Objetivo	Coste
Publicidad	Realización de trailers, videos y publicación en redes sociales	500€/campaña

Tabla 14. Marketing

4.4.5. Testing

Perfil	Objetivo	Coste	Coste Total
Tester	Realizar pruebas fase final del proyecto antes de salir a producción	100€/tester	100€*3tester=300€

Tabla 15. Testing

4.4.6. Material

Tipo	Objetivo	Coste	Coste Total
Ordenador	Soporte para desarrollar el proyecto	700€/ordenador	700€*10ordenadores=7000€
Electricidad	Encendido de dispositivos	0.12381 €/kWh	De forma aproximada, un PC consume unos 0.43kWh. Los 10 PCs consumirán 4.3kWh. 9 horas/dia * 5 meses= 1350 horas encendidos. 4.3kWh*0.0123€kwh=0.0528 9€/hora 0.05289€/h* 1350h= 71€

Tabla 16. Materiales

4.4.7. Coste final

Sumando cada uno de los elementos previamente dichos, el coste estimado del proyecto es de **15.866€**

5. Componentes del videojuego

En ésta sección, se van a explicar de manera detallada, todas las partes que han sido desarrolladas en cada uno de los incrementos.

Las tareas dedicadas a la realización de pruebas, se van a tratar en el punto 8, dejándo para éste, los refinamientos de requisitos y su posterior implementación.

5.1. Personaje principal

5.1.1. Desarrollo

Para el **desarrollo** del modelo 3D del personaje principal, se ha comenzado a partir de un modelo primitivo de un modelo humano. Cuando se hace referencia a un modelo primitivo, quiere decir que contiene los componentes básicos que forman un cuerpo humano, sin resaltar ninguno de ellos, tan sólo tiene el objetivo de darle forma humana. (Sebastian League, 2016)

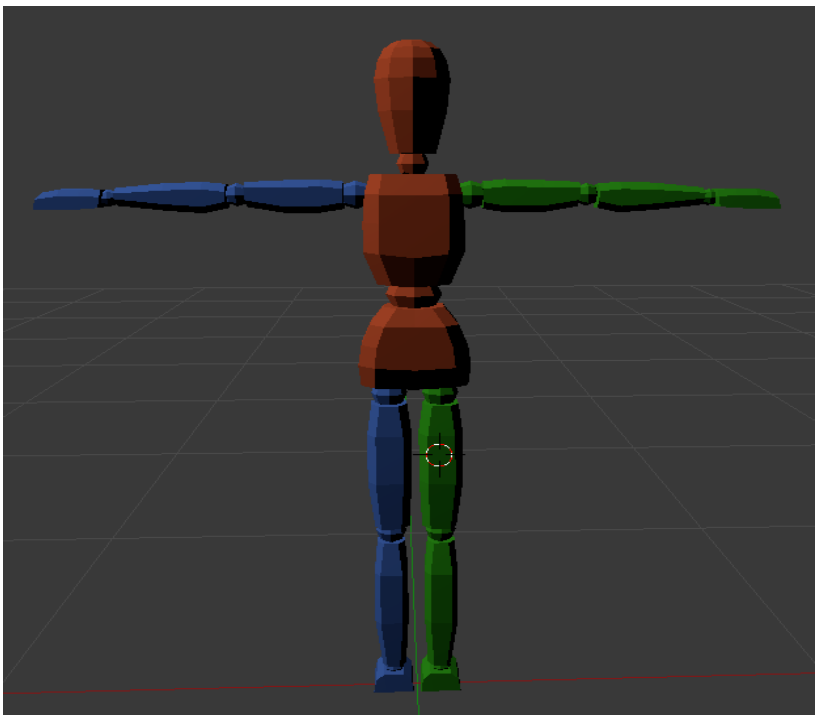


Ilustración 36. Modelo 3D del personaje principal

Para una primera versión, el modelo cumple con las funciones necesarias, pero, para un futuro, la función del modelo será la de dotar de estructura, a un conjunto de componentes que éste llevará asociados al cuerpo.

Esto quiere decir que, en lugar de cargar, por ejemplo, la parte del torso, se cargará la armadura que lleve equipada y se omitirá la visualización del torso. Esto nos va a ofrecer, poder visualizar la armadura y que ésta se comporte de la misma manera que el torso, ya que todos los movimientos y posiciones van asociados a éste. Ocurriría igual con el resto de elementos del cuerpo.

El desarrollo de cada una de esas partes, se realizarán en versiones más avanzadas del proyecto, por parte de un equipo artístico.

Otra opción, es adquirir un modelo ya hecho, pero se ha preferido desarrollar desde 0. Principalmente, se debe a que, normalmente, cuando importas un modelo realizado por terceros, no se sabe de la forma en la que se ha desarrollado o si se adaptará bien a los requisitos del proyecto. Para evitar posibles incompatibilidades o errores, se ha preferido hacer desde 0, para que todo el proceso esté totalmente controlado y adaptado a los requisitos que el videojuego solicita.

5.1.2. Rigging

A la hora de crear el modelo, hay que dotarlo de movimiento. Para ello, se ha utilizado la técnica de **Rigging**. Ésta técnica, permite generar un esquelo, formado por un conjunto de huesos y articulaciones, también sumado a un conjunto de restricciones (Blender, 2017). Las restricciones evitan que se produzcan movimientos “extraños”, como por ejemplo, mover la cabeza más de 180º.

Una vez creado el esqueleto, se tiene que fusionar con el modelo, para que a la hora de mover el esquelo, el modelo tambíen lo haga.

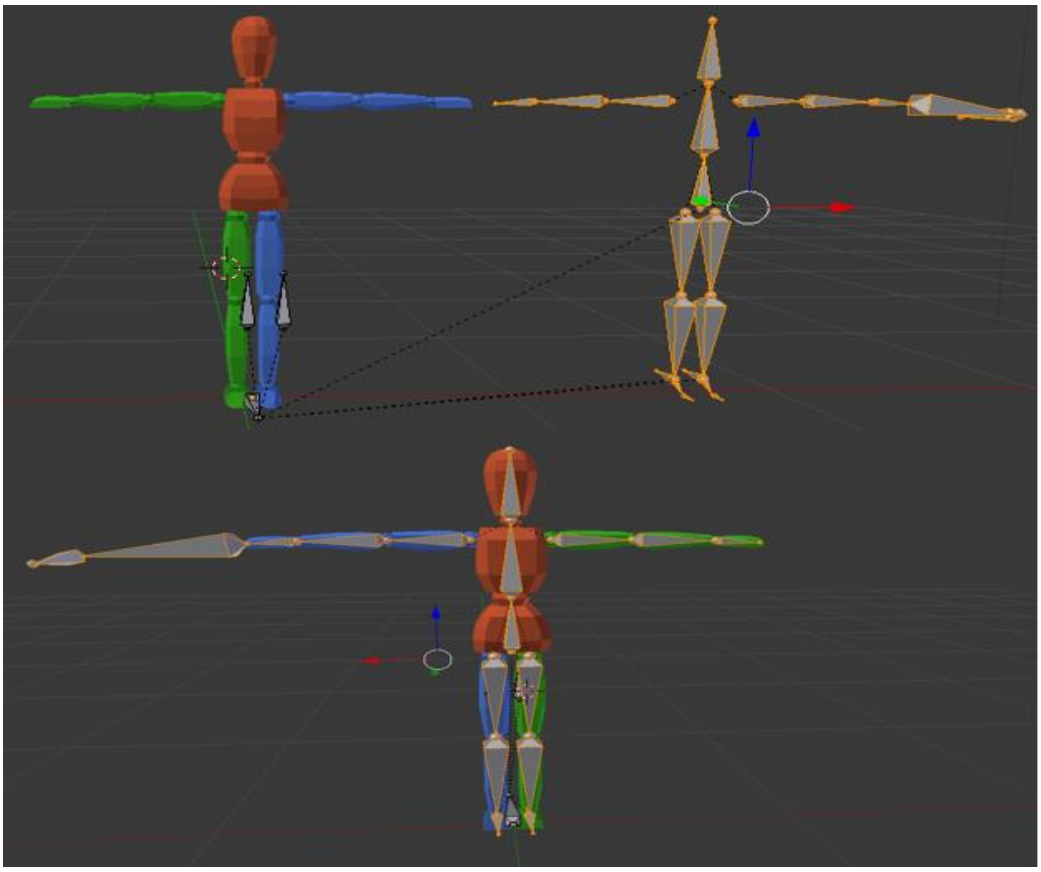


Ilustración 37. Fusión de rigging con el modelo

A la estructura generada, hay que añadirle en el **brazo dominante,** 2 huesos más. Esto se debe a que cuando se equipe un arma, la trayectoria que va a seguir ésta, viene dominada por la dirección de la muñeca y la dirección de la zona cortante. Se habla de zona cortante, debido a que las armas que va a llevar

equipadas son de tipo “espadas” y suelen tener una parte roma y otra cortante. Lo más realista, es que la dirección del ataque, se realice con la parte cortante.

Aunque parezca evidente que la dirección de la muñeca y la dirección del arma van a ser las mismas, no es del todo cierto. Esto se debe a que a la hora de empuñar un arma, no siempre se realizan los ataques manteniendo las mismas direcciones, por lo que tratarlas como dos vectores distintos, otorga un plus de realismo.

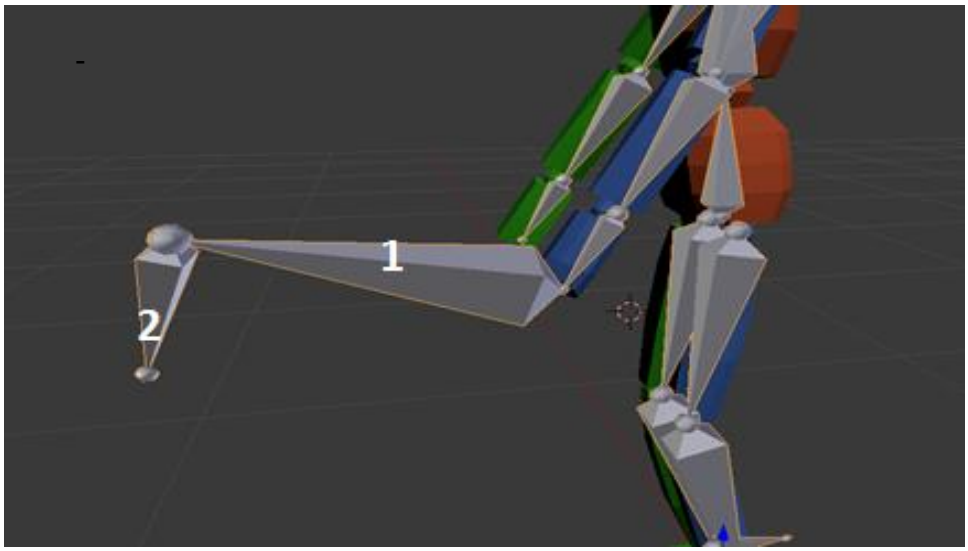


Ilustración 38. Vectores dirección arma del personaje principal

La **parte 1**, simula la posición del arma, mientras que la **parte 2**, la dirección de ésta. Por defecto, el vector que indica la dirección de la parte cortante, apunta hacia abajo. Todo esto, sirve para que a la hora de realizar las animaciones, se tenga una referencia de la dirección hacia dónde va a ir el arma y hacia dónde va a estar dirigida, ya que durante el juego, serán sustituidos por un arma.

Durante la experiencia de juego, que un personaje lleve un arma y la trate como tal, parece obvio, pero en la implementación, son elementos totalmente desacoplados (el modelo del personaje y las armas son distintos). Para integrarlos de la mejor forma posible, éste es un buen comienzo, aunque todavía quedan más cosas por hacer, que se comentarán en los siguientes apartados.

5.1.3. Animaciones

Una vez tenemos todo lo anterior montado, se procede a realizar las **animaciones**. En un análisis previo al desarrollo (punto 3.4.2), se han establecido las animaciones que se quieren realizar, por lo que tan sólo habría que realizar los movimientos que se adapten a las especificaciones y almacenarlos.

Para ello, **Blender** dispone de un repertorio de herramientas para poder realizar dicha labor. La forma de hacer las animaciones, consiste en almacenar unas posiciones clave, en las que se almacena la posición y rotación de todos los elementos del personaje. Cuando se obtienen un conjunto de éstos, utilizando algún tipo de interpolación, se generan todos los fotogramas intermedios. En éste caso, se ha usado una interpolación lineal. (Blender, 2017) (Sebastian League, 2016)

A continuación, se muestra un ejemplo del resultado de la animación de andar:

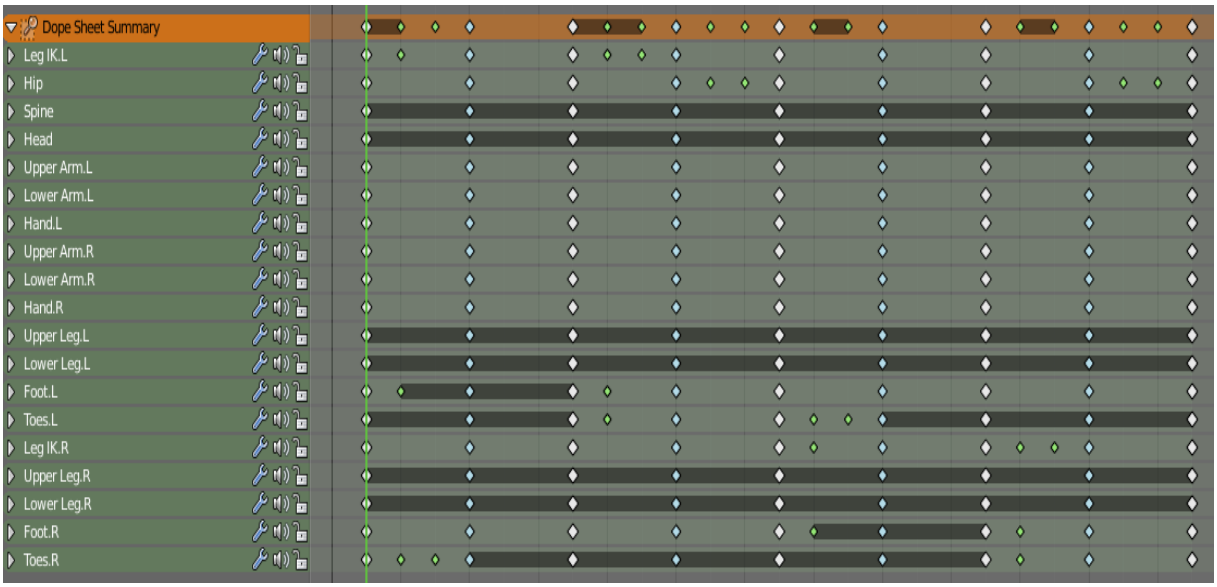


Ilustración 39. Diagrama fotogramas clave para animación de andar

En la parte izquierda se aprecian los **componentes**, y a la derecha, en forma de rombos, los **fotogramas clave**. En los espacios sin rellenar, se calcula la posición mediante el uso de la interpolación lineal, por consecuente, mientras más fotogramas clave se tenga, mayor control sobre el movimiento del modelo se tendrá.

No es aplicable a todas, pero si la animación es de tipo repetitiva, como por ejemplo, la de andar, correr …etc, da un plus de realismo poner como primer y último fotograma clave, el mismo.

5.1.4. Exportar a Unity

Llegados a éste punto, tenemos un modelo 3D, con un conjunto de animaciones, que hay que insertar en Unity. Éste proceso, se va a dividir en varias fases, debido a que consta de múltiples etapas que hay que detallar. Para éste apartado, sólo nos vamos a centrar en exportar el modelo, para que se pueda visualizar y que se obtenga toda la toda la jerarquía de elementos por la que está formado.

Para ello, se ha cargado el fichero generado por Blender en Unity, que gracias a su compatibilidad, permite una sencilla adaptación. Una vez estamos en Unity, hay que configuar el modelo. El primer paso, es cargarlo como un modelo 3D que pueda ofrecer una jerarquía de componentes, para poder en un futuro, añadirle elementos.

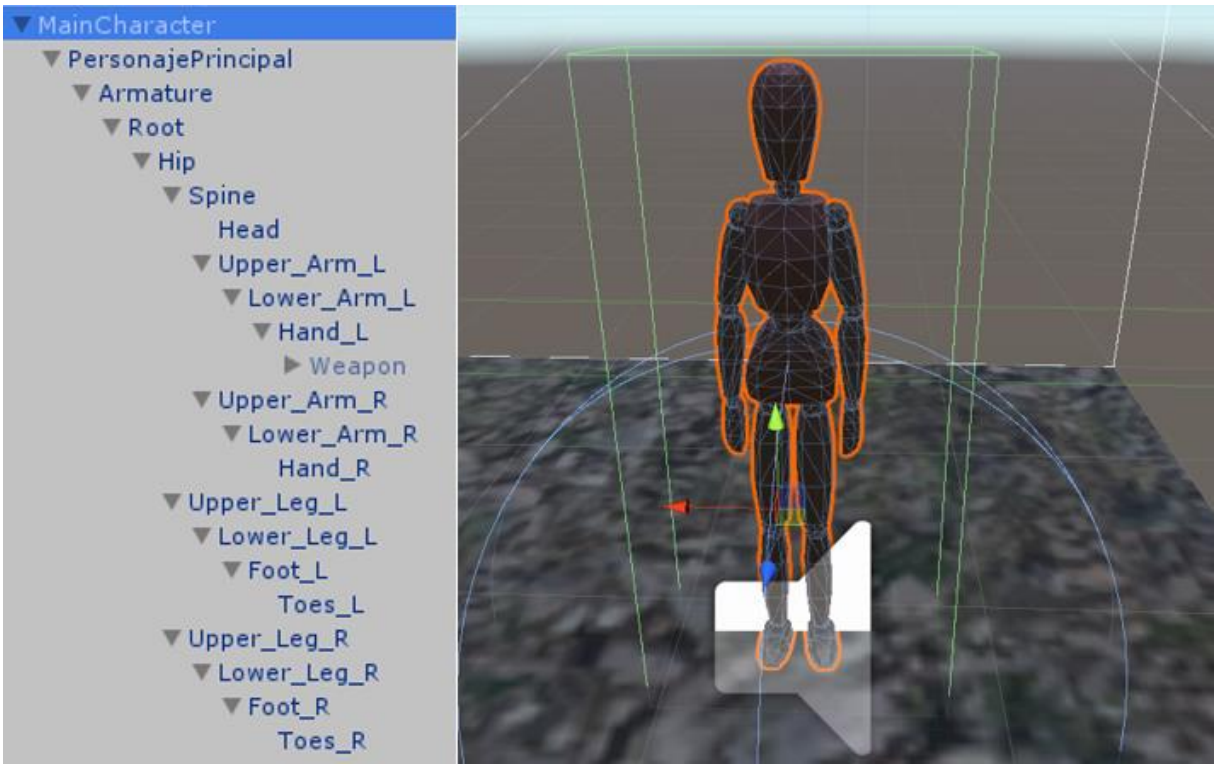


Ilustración 40. Jerarquía de componentes del personaje

Una vez configurado, de manera interna **Unity** realiza 2 procesos para que el modelo se pueda renderizar y las animaciones puedan ser aplicadas de manera eficiente al modelo. Para ello, hace uso de **MeshRenderer** y **Mecanim**, cuyos parámetros habrá que adaptar a los requisitos del proyecto.

En **primer lugar,** se genera la *mesh* del modelo y la asocia a un GameObject. Ese GameObject, contiene un **MeshRenderer** cuya función es la de renderizar la deformación que se produce en la maya cuando a ésta se le aplican las animaciones. MeshRender también aplica a la malla, efectos de iluminación y sombreado. (Unity, 2017c)

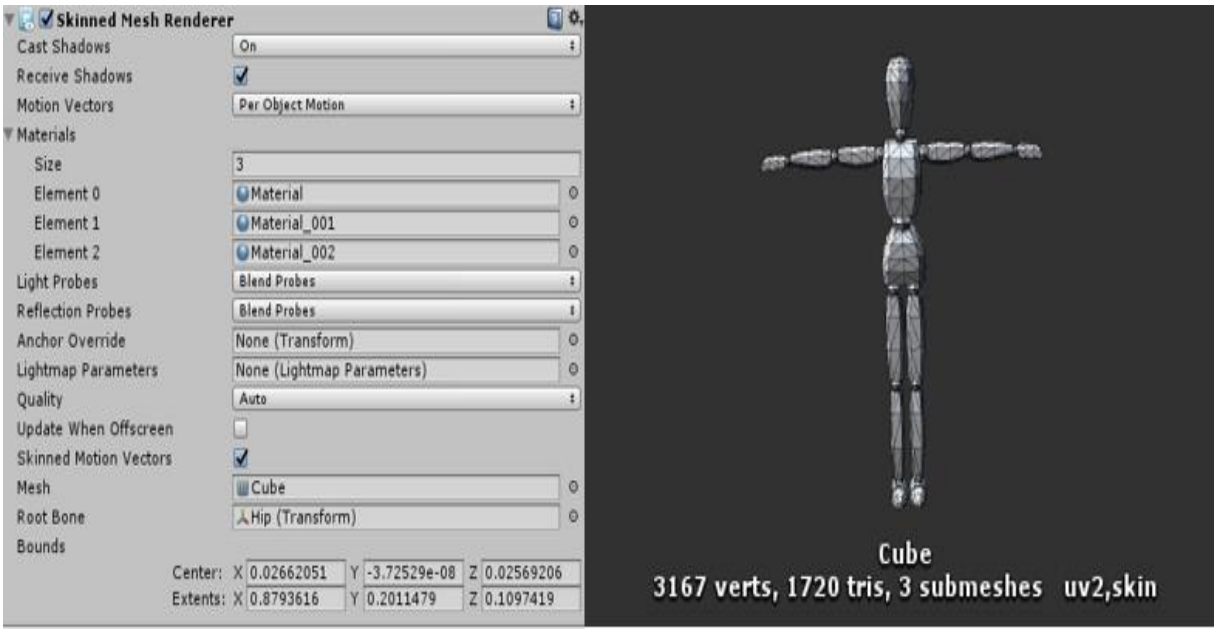


Ilustración 41. Mesh render en Unity

Al mismo tiempo, haciendo uso de **Mecanim**, se generan las animaciones en forma de animations, un componente de Unity que permite manejarlas como si fueran un conjunto de transiciones entre nodos, los cuales representan la animación, lo que facilita su manejo. (Unity, 2017d)

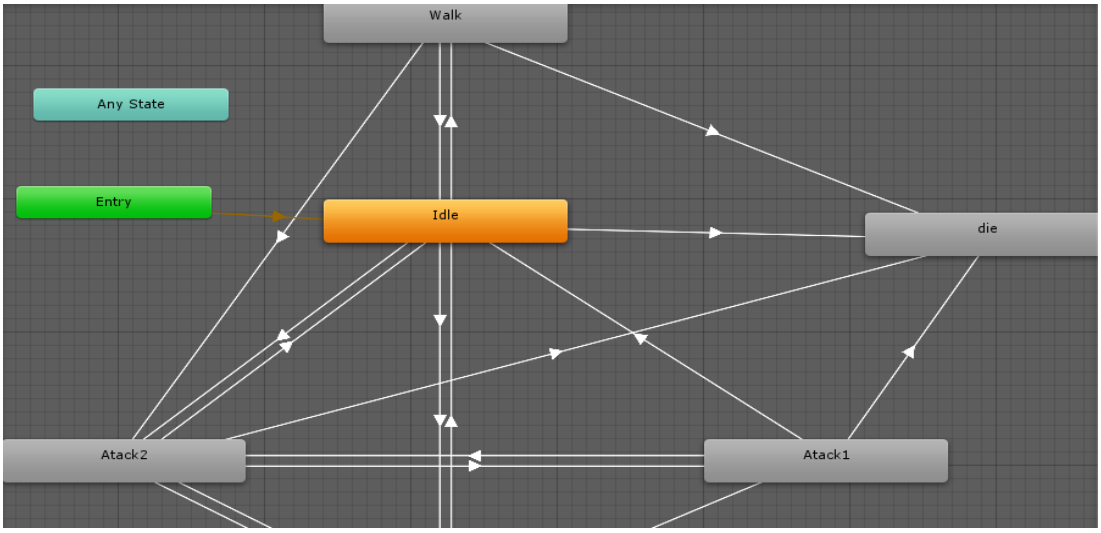


Ilustración 42. Transiciones entre animaciones con mecanim

Término	Definición
Mesh	Malla de triángulos utilizada para crear los modelos 3D

Tabla 17. Definición de mesh

Término	Definición
Mecanim	Sistema para representar y manejar las animaciones

Tabla 18. Definición de mecanim

5.1.5. Mecánicas de movimiento

Para implementar el movimiento del personaje, hay que programar un serie de eventos y restricciones que ocurren cuando se activa un input determinado. Se entiende como un input, cualquier acción proveniente de un periférico del ordenador, como por ejemplo, el teclado, un ratón o un joystick.

El sistema que compone las mecánicas de movimiento, está formado por **varios bloques**, con el objetivo de tener una modularidad entre componentes.

Si establecemos una jerarquía, por la cual, los componentes se comporten como un único sistema, pero siendo independientes, en la estructura, estarían los periféricos de entrada, los inputs asociados a dichos periféricos, las funciones de movimiento y ejecuciones de los movimientos.

Los **periféricos de entrada**, son asignados por el jugador, de forma que, puede elegir y configurarlos de la manera que desee. Por ejemplo, si utiliza el teclado, puede configurar las teclas que mejor se adapten a su modo de juego, para realizar cierto tipo de acciones. Si la acción de andar, está por defecto en la tecla “W”, puede modificarla para que sea cualquier otra, siempre y cuando no entre en conflicto con otras.

Los **inputs**, se activan cuando se pulsa el comando (tecla, botón…) que ha sido asignado previamente por el jugador. En caso de no haber sido asignado, se usarían los valores por defecto. Por ejemplo, el input “Walk”, tiene asociada la tecla “W”, de forma que, cuando ésta se pulse, el input de tipo Walk, se activará. (Unity, 2017e)

Situándonos en un nivel inferior, habría que asociar a cada input que esté activado, una **función**. Para desarrollar ésta parte, el motor, pone una serie de obstáculos, que impiden el mejor desarrollo posible. Esto se debe a que, para realizar la comprobación de los inputs, hay que ir comprobando 1 a 1 si está activado o no. El problema, es que en cada frame, se tiene consultar el estado de todos los inputs para comprobar su estado, lo cual, consume una cantidad de recursos que podrían evitarse. Una forma de evitarlo, sería usar un sistema de

eventos, por el cual, cuando se active un input, no haya que consultarlo, si no que, tenga asociada una función que se ejecute cuando éste se active.

Éste sistema, evitaría estar consultando en cada frame el estado de todos los inputs, que normalmente, la mayoría estaría a false, o cómo máximo, 2 a true. Esto se debe a que el *buffer* de inputs, tan sólo es capaz de recoger 2 inputs a la vez, siendo el tercero ignorado. (Microsoft, 2013)

Término	Definición
Buffer	Espacio reducido de memoria en los que los que se almacenan elementos de forma temporal

Tabla 19. Definición de buffer

A continuación (ilustración 43), se muestra el diagrama en el que se resume el funcionamiento de éste mecanismo:

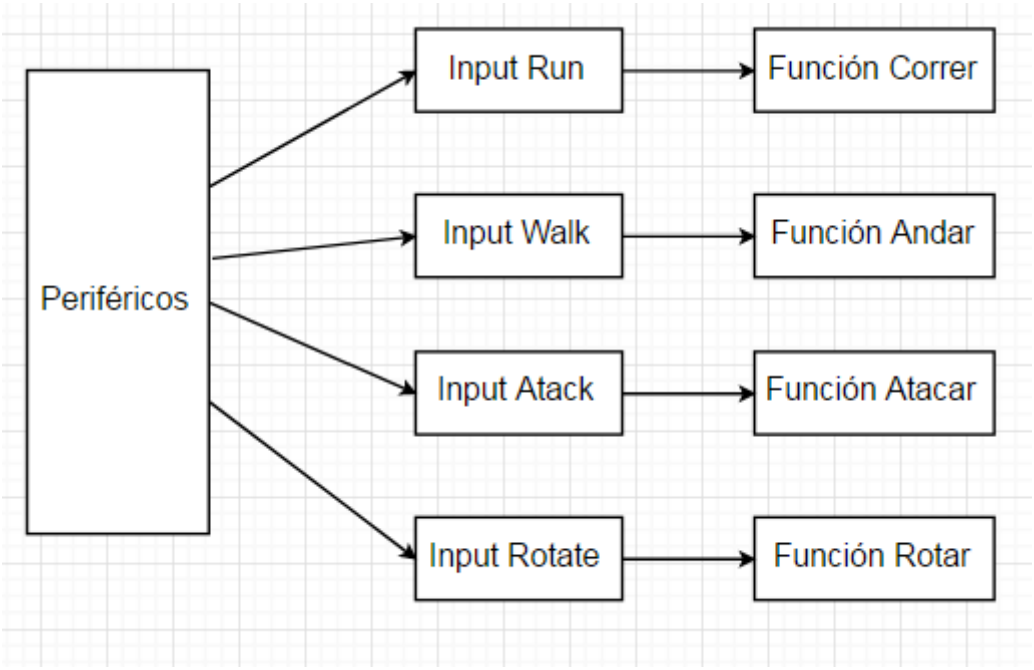


Ilustración 43. Esquema de mecánicas de movimiento

Parte del código asociado a ésta implementación a modo de ejemplo:

```
if (Input.GetButton ("Walk")) {

    actionMove ("SpeedCharacter","CameraMove", this.walk, this.velocityWalk);


/*
    Realiza la accion de correr
    animationCharacter nombre de la animacion
    valueRun valor de transicion de la animacion
    velocityRun velocidad de avance del personaje
*/
private void actionMove(string animationCharacter,string animationCamera,float valueRun,float velocityRun){

    /*
        Comprueba que el personaje no esta realizando otra animacion que no sea la de movimiento.
        Evita, por ejemplo, que mientras el personaje este atacando se desplace sin realizar la
        animacion del movimiento
    */

    if(animatorCharacter.GetCurrentAnimatorStateInfo (0).IsName("WalkRunAnim")){

        animatorCamera.SetFloat (animationCamera,valueRun);
        animatorCharacter.SetFloat (animationCharacter, valueRun);
        transform.Translate (Vector3.forward * Time.deltaTime * velocityRun);

    }
```

Ilustración 44. Ejemplo de código de mecánicas de movimiento

Cada vez que se ejecuta una **función** asociada al movimiento, se deben de activar varios elementos. Ese conjunto de elementos, se activan a través de la clase **MoveBehaviour**, mediante la cual, permite acciones tales como: la transformación de la posición del personaje, la animación asociada a dicho movimiento, sonido y la animación de la cámara. La sincronización y correcta ejecución de éstos elementos recae sobre la clase MoveBehaviour.

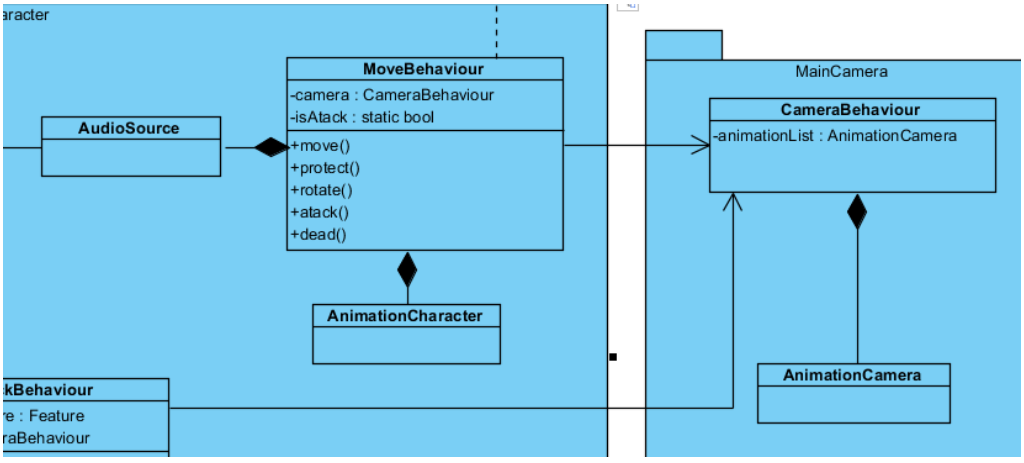


Ilustración 45. Diagrama de clases move behaviour

La animación de la camara, permite que ésta tenga distintos comportamientos, dependiendo de la acción que esté realizando el personaje. Esto se debe a que las sensaciones a la hora de jugar, son mejores que, si la cámara estuviera siempre fija.

Para resolver éste problema, se ha utilizado el **patrón** estructural **GoF Bridge**. Nos permite desacoplar una abstracción de su implementación (Larman, 2003). En éste caso, la aplicabilidad se basa en que, cambios en la implementación de un método, no debe repercutir en las clases que la utilizan.

Cuando se activa un input que ejecuta una acción, por ejemplo, la de andar, la forma de ejecutarse dicha función, puede ser modificada sin tener que modificar también el input que la utiliza. Ocurriría igual con la asignación de teclas asociadas a los inputs, la modificación de la entrada no afecta a la forma de funcionar del input.

5.1.6. Implementación de los stats

La implementación de los **stats** consisten en, asignarle al personaje una serie de atributos que determinan el estado en el que se encuentra. En éste módulo, va a haber atributos tales como HP (cantidad de vida), SP (cantidad de magia), lvl (nivel actual), dmg_fisic (daño físico), dmg_magick (daño mágico)…etc

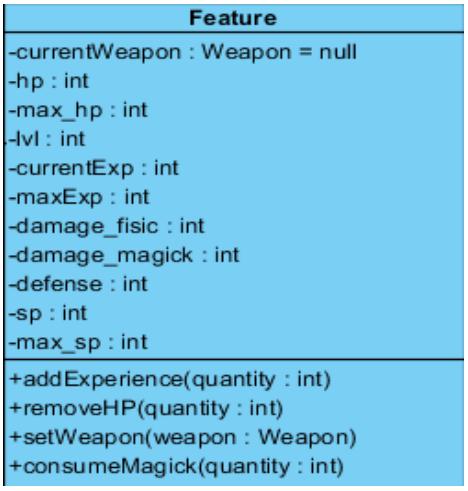


Ilustración 46. Clase feature

Alguno de éstos valores, van a ser modificados, cuando ocurran una serie de acontecimientos que afecten al personaje. Un ejemplo, podría ser el evento de llevar un arma, por el cual, los valores de ataque mágico y físico, se ven modificados dependiendo del arma que se lleve equipada.

Si un personaje lleva un arma que dé de ataque físico +200, éste, se le sumará al que ya lleve por defecto. Ocurriría igual con resto de atributos, que dependiendo del item equipado, se verán afectados unos atributos u otros.

El valor de algunos de éstos atributos van a desencadenar una serie de acontecimientos que afectan a la jugabilidad. En caso de que el HP, sea menor que 0, se van a ejecutar las correspondientes acciones para indicar que el personaje está muerto y por consucuente, la partida finaliza.

```
public void removeHP(int quantity){

    int totalQuantity=(quantity-defense)>0 ? (quantity-defense) : 0;

    hp -= totalQuantity;

    if (hp < 0) {

        //Muerto
        hp = 0;
        controller.characterIsDead ();

    } else {

        mainCanvas.setTextDamage (totalQuantity);
    }

    mainCanvas.updateHPCanvas();

}
```

```
public void addExperience(int quantity){

    int totalExperience = currentExp + quantity;

    if (totalExperience > maxExp) {

        lvl += 1;
        currentExp = totalExperience % maxExp;
        maxExp *= 2;
        hp += 100;
        sp += 100;
        damage_magick += 50;
        damage_fisic += 150;
        mainCanvas.updateLvlCanvas ();

    } else {

        currentExp = totalExperience;

    }
    mainCanvas.updateExp ();

}
```

Ilustración 47. Ejemplos de código stats del personaje

Para tener una correcta asignación de responsabilidades, todos los eventos que ocurren, cuando se modifican algunos de los atributos, se ejecutan dentro de la propia clase. Éste tipo de arquitectura, nos recuerda al **patrón GRASP: Experto en información**, el cual indica que: "*La responsabilidad de la creación de un objeto o la implementación de un método, debe recaer sobre la clase que conoce toda la información necesaria para crearlo".* (Larman, 2003)

De este modo, obtendremos un diseño con mayor cohesión y así la información se mantiene encapsulada (disminución del acoplamiento).

5.1.7. Implementación del inventario

Esta parte, está formada por un conjunto de elementos que trabajan entre sí, para proporcionar al jugador un sistema de inventario por el cual, se puedan agregar armas y equiparlas. Es totalmente flexible para que se puedan agregar más tipos de items, pero, debido a la planificación, sólo se ha desarrollado la parte asociada a las armas.

Lo primero que se ha hecho, ha sido **representar** las **armas del juego** como una jerarquía de clases que permitan una total interacción con éstas:

StatsWeapon: Representa valores tales como la cantidad de daño mágico y físico, junto con algún posible bonus, como por ejemplo, un incremento de HP y/o de SP.

3D_Model: Clase encargada de cargar la mesh asociada al arma, para que se pueda visualizar el modelo 3D cuando sea equipada.

Weapon: Clase base, que contiene la referencia a los stats y carga el modelo 3D asociado. También es la responsable de crear el sprite, una imagen en 2D para poder visualizar el aspecto del arma en el inventario.

WeaponDrop: Ésta clase, surge como consecuencia de querer representar un arma que aún no tiene el personaje, pero, se visualiza en el juego, con la intención de que pueda ser recogida. Para ello, tan sólo se necesita cargar el modelo 3D, ya que el resto de elementos, no nos sirven para éste caso. Eso no quita, que tenga una referencia a Weapon, ya que, cuando el personaje realiza la acción de recoger dicha arma, se necesitan el resto de atributos para que pueda ser almacenada de forma correcta en el equipo. También se ha añadido un tiempo, el cual indica cuánto va a estar el arma visible, de forma que al finalizar dicho tiempo, el arma desaparece.

Las **diferencias** que hay entre las clases WeaponDrop y Weapon, son que **WeaponDrop** representa el modelo del arma, para que el personaje pueda visualizarla y adquirirla en el juego. Por otro lado, **Weapon**, representa la estructura completamente funcional del arma, es decir; está vinculada a la jerarquía de componentes del personaje, tiene unos stats que afectan a los del personaje, tiene un detector de colisiones (collider) para los ataques… etc.

Ilustración 48. Ejemplo del tipo Weapon

Ilustración 49. Ejemplo del tipo DropWeapon

A continuación (ilustración 50), se muestra el diagrama de clases de ésta parte, que representa la interacción y la forma que tiene de comunicarse con los elementos anteriormente dichos:

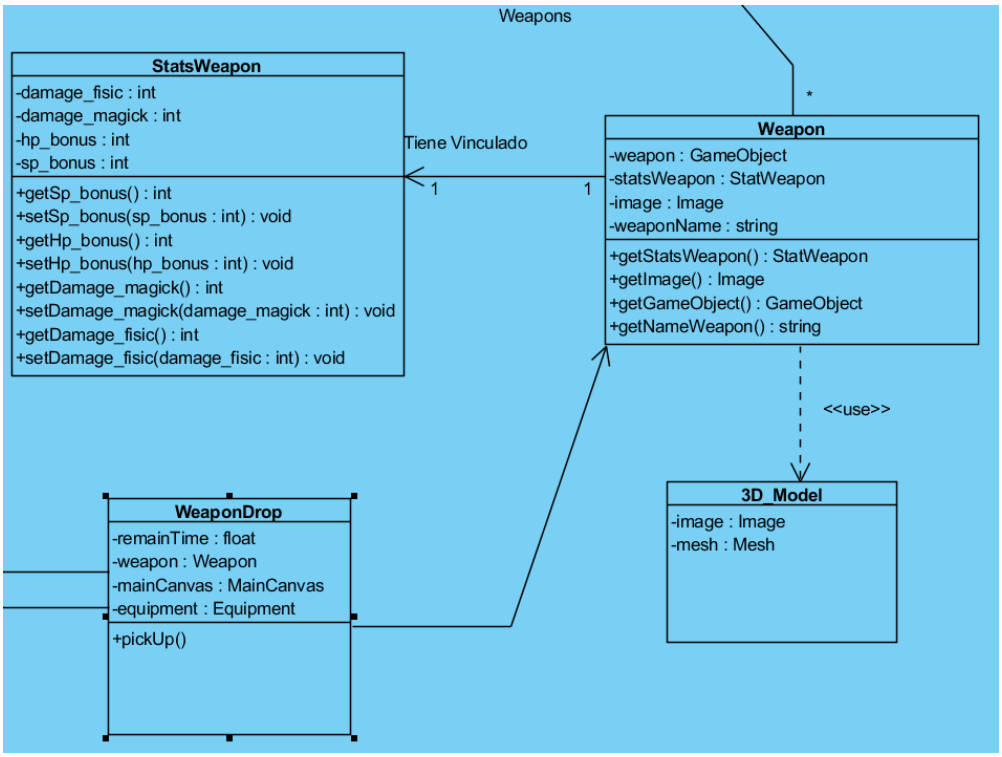


Ilustración 50. Paquete de clases Weapons

El siguiente paso, es tener algún tipo de **contenedor en el personaje**, para que pueda almacenar las armas obtenidas a lo largo del juego. Para ello, se ha implementado la clase **Equipment**, la cual, permite almacenar, buscar, borrar y devolver una lista de todas las armas obtenidas.

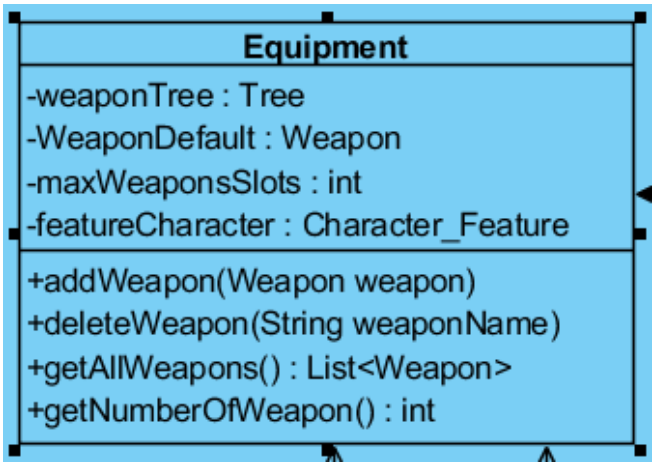


Ilustración 51. Clase equipment

Cabe destacar, que en ésta parte, se debería haber implementado algún tipo de API, que permita una interacción con alguna base de datos, de forma que al almacenarlas, se guarden ahí. El objetivo es que a la hora de guardar la partida, las armas se almacenen en la base de datos, y cuando se cargue la partida, realizar algún procedimiento de cargado para obtenerlas de ahí.

Ya que es un prototipo cuyo objetivo es mostrar funcionalidades y no un desarrollo final, se dejará la parte de persistencia para futuras versiones.

En lugar de usar persistencia en una base de datos, los guardados se hacen en memoria, a través de una estructura de datos de tipo arbórea, en la que los datos se almacenan con la tupla “clave, dato”. Para la clave, se utiliza el nombre del arma, y para el dato, el tipo Weapon. En C# se implementa como **Dictionary**<Key,Value>, que según la documentación, garantiza búsquedas e insercciones con una eficiencia en O(1). (Microsoft, s.f.)

Una vez finalizada ésta parte, tan sólo queda implementar el inventario, para que se puedan ver las armas disponibles y poder interactuar con ellas a través de una interfaz.

El **inventario**, se ha implementado como parte del canvas, ya que el objetivo es que el usuario pueda interactuar con él. Para ello, se ha desarrollado la clase **Inventory**, cuyo objetivo es cargar en un *grid layout*, las armas que el personaje ha obtenido. Para ello, se le solicita a Equipment, la lista de todas las armas. Una vez obtenidas, se genera por cada una de ellas un botón, al que se le asigna la imagen, el nombre de ésta, y una referencia al arma.

Término	Definición
Grid layout	Organización de los elementos en forma de matriz, es decir, por filas y columnas.

Tabla 20. Definición grid layout

La referencia al arma, sirve para que cuando se active el botón, se visualice el modelo 3D del arma en el personaje y se asigne el arma, lo cual, se hace mediante

la clase Feature, para que los stats del personaje se actualicen, con los valores que tenga el arma.

A continuación (ilustraciones 52,53,54), se muestran los diagramas de clases y de secuencia realizados:

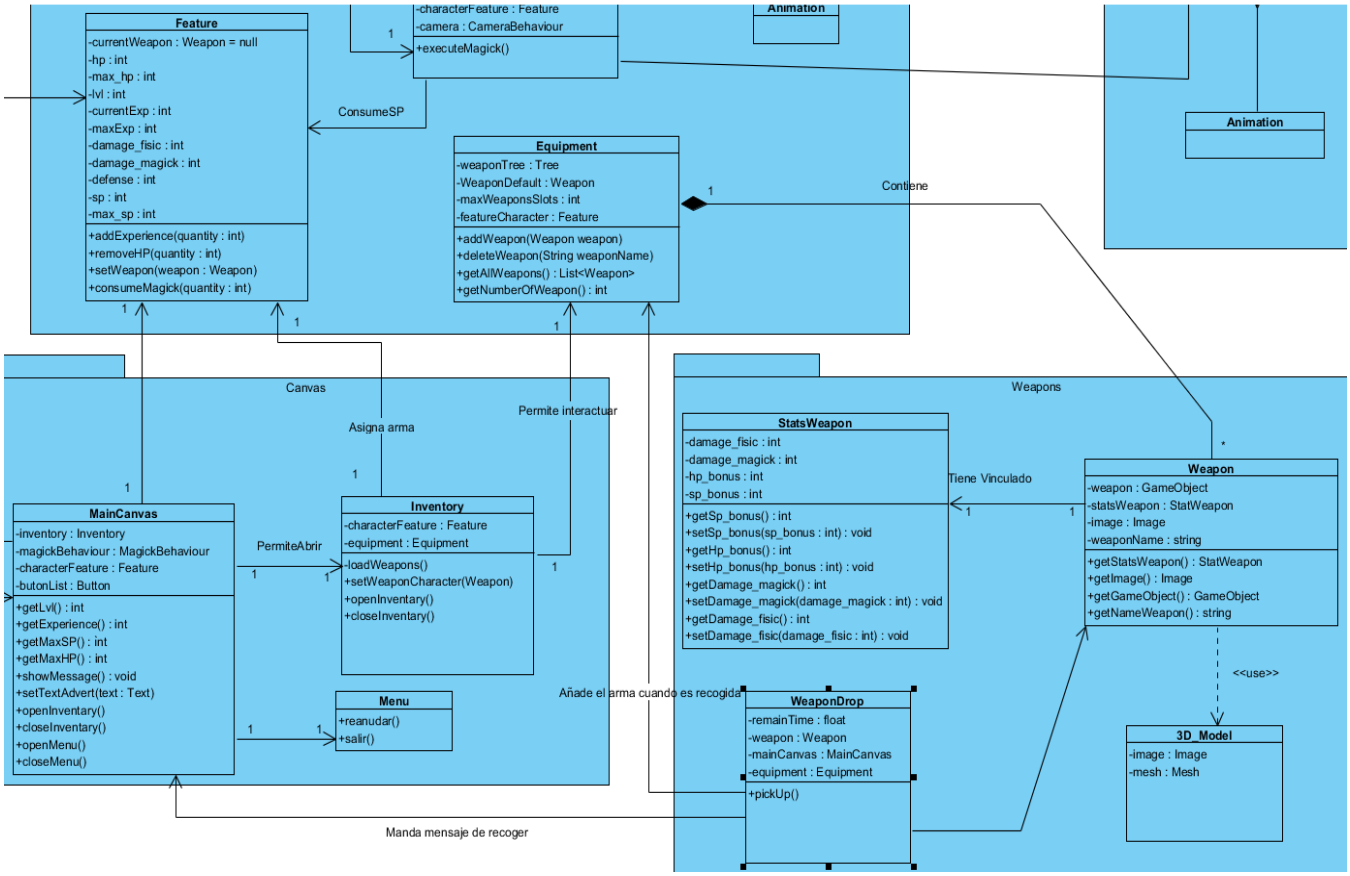


Ilustración 52. Diagrama de clases del sistema inventario

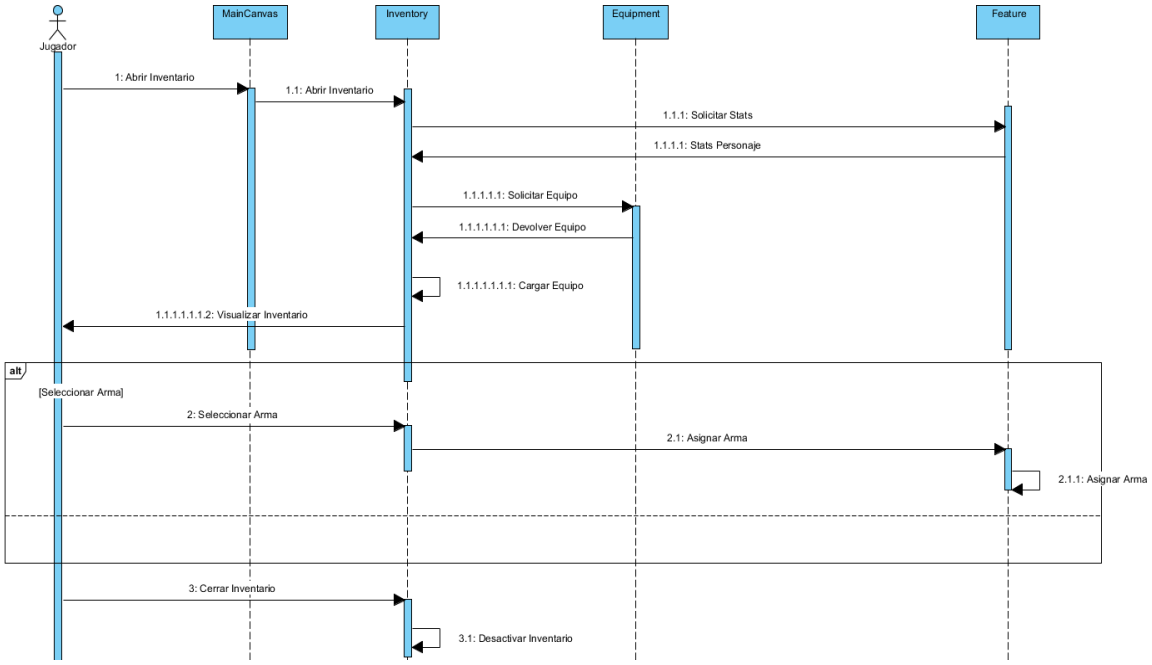


Ilustración 53. Diagrama de secuencia abrir inventario

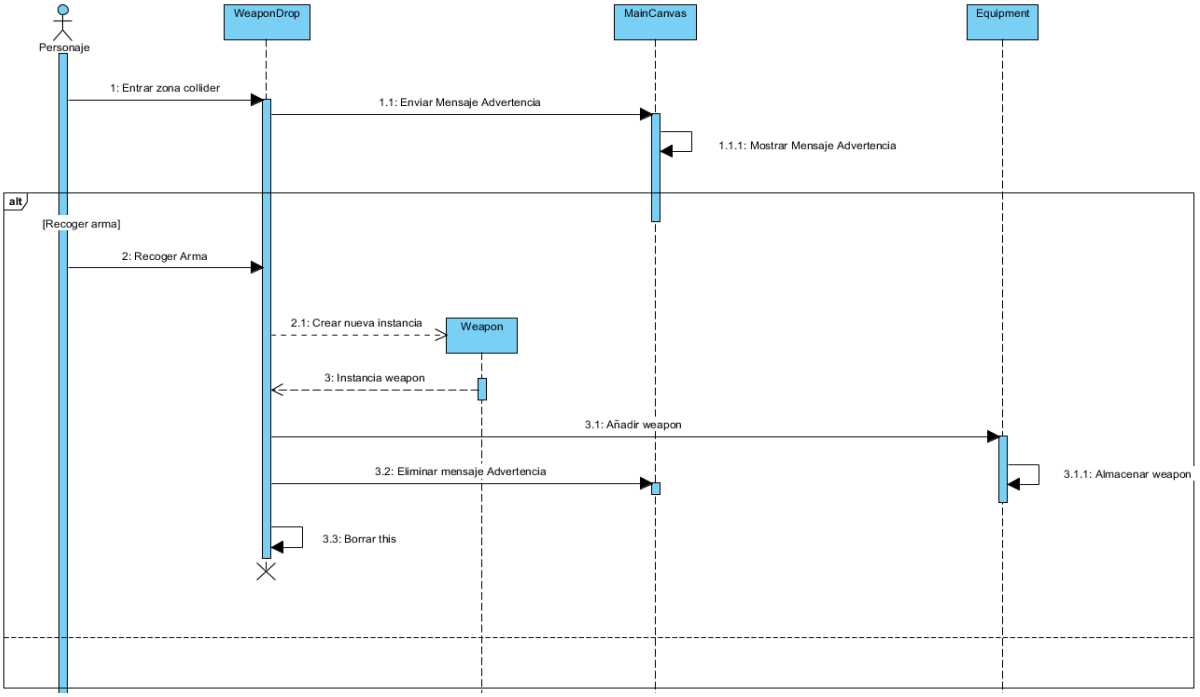


Ilustración 54. Diagrama de secuencia recoger arma

Para finalizar (ilustración 55), se muestra el resultado de abrir el inventario durante el juego:

Ilustración 55. Inventario del juego

5.1.8. Implementación de las magias

Al igual que para el inventario, también se ha creado un sistema para ejecutar las magias. A continuación, se va a explicar el proceso que ocurre detrás de la ejecución de cualquiera de ellas, desde que se presiona el botón para ejecutarla, hasta que ésta se ejecuta.

En primer lugar, partimos del **MainCanvas**, una clase que representa toda la jerarquía de botones y estados con los que el jugador puede interactuar y tener información sobre algunos stats del personaje. Dentro del patrón de diseño **MVC** (Modelo Vista Controlador), se puede interpretar como la **vista**.

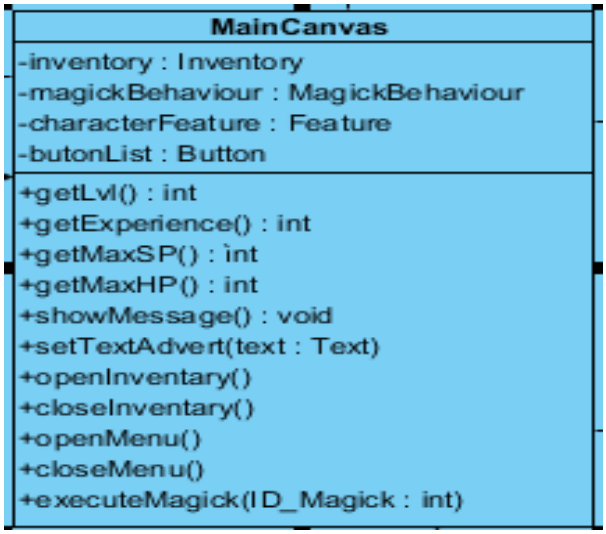


Ilustración 56. Clase MainCanvas

Ilustración 57. Representación del MainCanvas

Los **botones** situados en la parte **central** de **abajo** (ilustración 57), son los encargados de recoger la acción de ejecutar las magias. Cada botón, tiene asignado una función, que es llamada cuando éste se activa. Cada una de esas funciones, realizan las llamadas y procedimientos necesarios para ejecutar cada una de las magias, ya que tienen comportamientos distintos y no se pueden generalizar.

Para ofrecer una mejor experiencia, cada botón dispone de un mecanismo por el cual, se solapan 2 imágenes, una más oscura que la otra, de forma que, cuando

se activa la magia, ofrece al jugador la información visual, en la que se intuye el tiempo restante, para poder volver a realizar la magia.

Ilustración 58. Regeneración de magias

El siguiente paso, va a venir determinado, por las funciones a las que se llama, cuando se ejecuta cada botón, las cuales, son las responsables de controlar el estado de la ejecución de las magias. Para ello, se ha implementado la clase **MagickBehaviour**, la cual, contiene una referencia a todas las magias que se pueden hacer y es la única responsable de controlar su ejecución. Dentro de la arquitectura MVC, ésta clase se sitúa como **controlador**.

Éste diseño, evita que se realicen llamadas directamente a las magias, evitando situaciones no deseadas. A la hora de ejecutar las magias, se debe tener un total control sobre éstas, debido a que detrás de cada una, se ejecuta una animación, un sonido, un sistema de partículas…etc que deben estar totalmente sincronizados y bajo ningún concepto deben solaparse con otras magias. En definitiva, evita que dos magias se ejecuten al mismo tiempo, a la vez que consulta al stat del personaje (Feature) si contiene suficiente SP para poder realizarlas y si el tiempo de regeneración de la magia ha concluido.

La única forma de ejecutar las magias, es a través de la clase **MagickBehaviour**, proporcionando éste tipo de diseño, un modelo robusto, a prueba de situaciones anómalas.

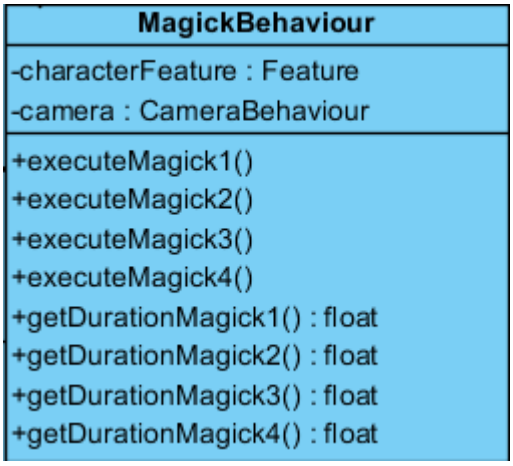


Ilustración 59. Clase MagickBehaviour

Para mejorar la experiencia de juego, se dispone de una referencia a la cámara, con el objetivo de realizar una animación a la hora de realizar la magia, en caso de que así se requiera. La cámara, dispone de una clase llamada **CameraBehaviour**, cuya función es muy similar a la de ésta clase, en cuanto a control de eventos se refiere.

Para el último nivel, se ha desarrollado la clase **MagickFeature**. Ésta clase, tiene asociados un conjunto componentes, tales como ParticleSystem y AudioSource, los cuales ofrecen los efectos visuales y sonoros que representan la ejecución de la magia en sí. También tiene **atributos** tales como:

sp_consume: Cantidad de SP que necesita para ejecutarse.

duration: Tiempo que permanece la magia estando ejecutada antes de desaparecer

damage: Daño infligido por la magia a los enemigos. Cabe destacar que éste no es el daño final, ya que el total, sería el resultante de sumar el daño de la magia y el de los stats del personaje.

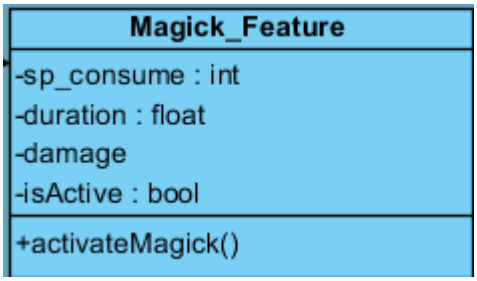


Ilustración 60. Clase MagickFeature

También contiene un conjunto de colliders, para detectar a los enemigos que han sido alcanzados por la magia efectuada, en caso de que sea necesario, ya que hay algunas que no lo necesitan. Para ese caso, el collider queda desactivado de manera permanente.

A la hora de ejecutarse, es la encargada de sincronizar todos los efectos que tenga asociados la magia.

La clase **MagickFeature** actuaría como **modelo**, dentro del patrón MVC.

A continuación (ilustraciónes 61,62) se muestran los diagramas de clases y de secuencia:

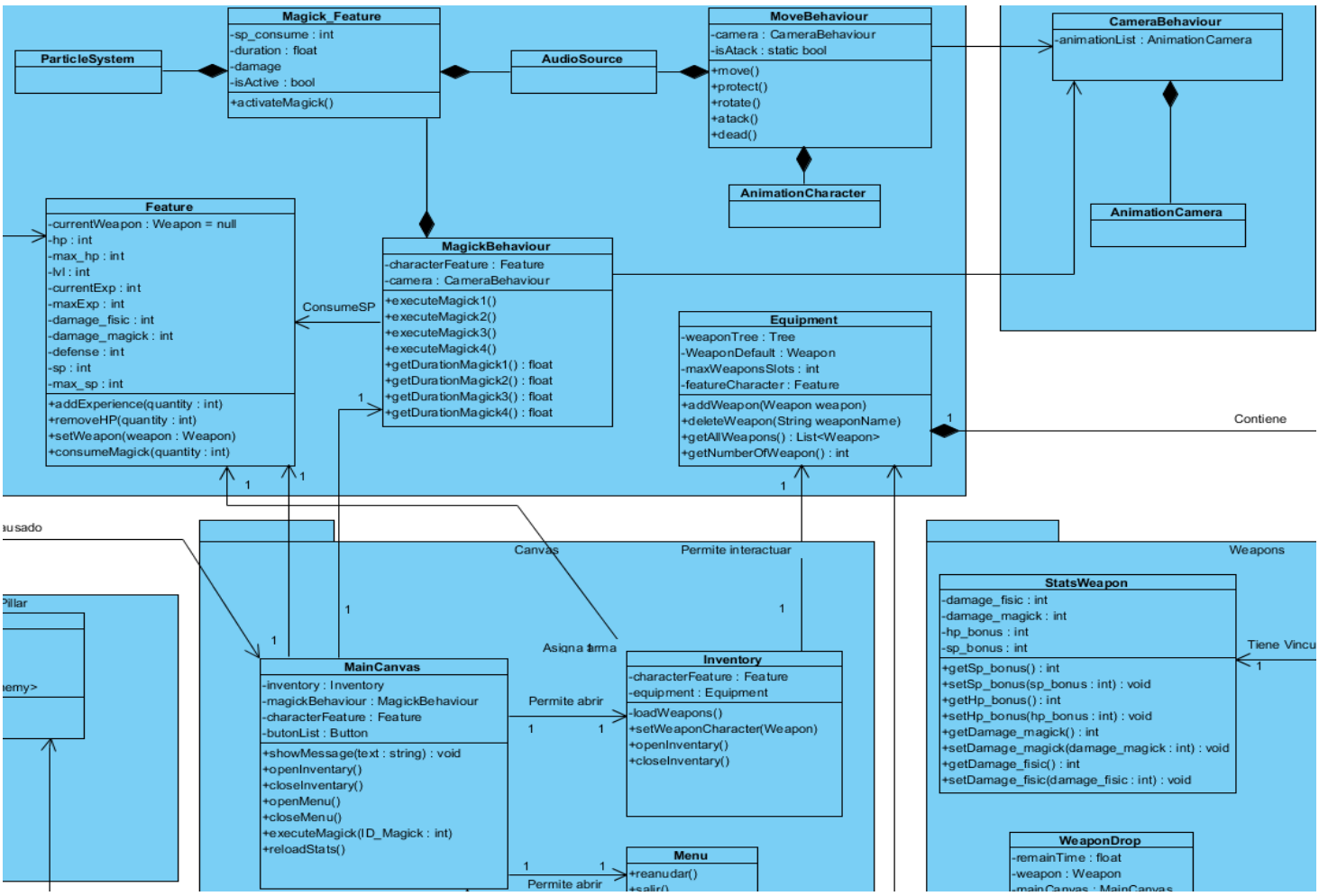


Ilustración 61. Diagrama de clases de las magias

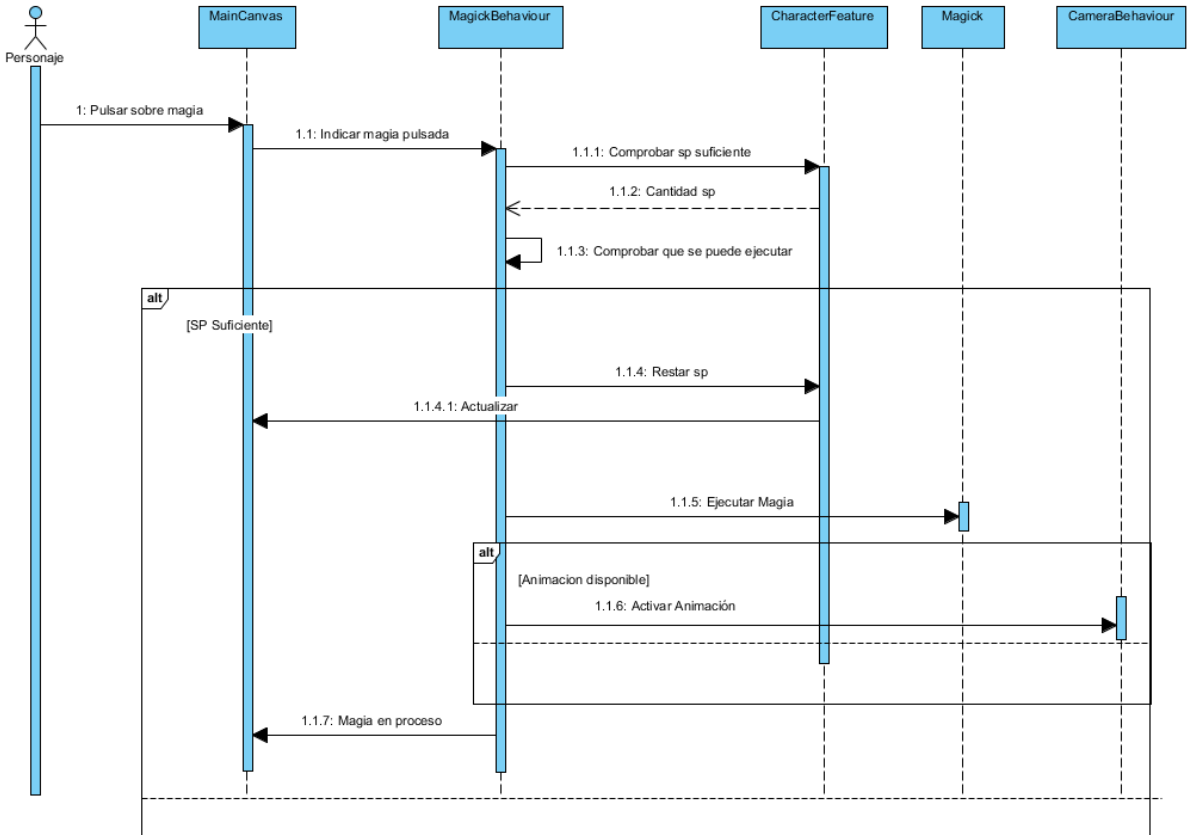


Ilustración 62. Diagrama de secuencia. Ejecutar Magia

5.2. Enemigos

A diferencia del personaje principal, los modelos 3D de los enemigos han sido importados del asset store, un recurso web, en el que la comunidad de Unity, se dedica a subir todo tipo de implementaciones para contribuir en el desarrollo de videojuegos. (Unity Asset Store, s.f.)

Ilustración 63. Enemigos del juego

No todos los modelos son aptos para ser importados a éste proyecto. En **aspectos visuales**, se han buscado modelos que ofrezcan una sensación de maldad, con el fin de poder adaptarse a la historia lo mejor posible y dar una buena impresión al jugador.

A lo referente en **aspectos técnicos**, se han buscado modelos que ofrezcan una mesh (malla de triángulos para visualizar el modelo), con un índice de poligonalización bajo. Esto quiere decir, que la mesh contenga un número lo más reducido de triángulos, ya que el uso que se le va a dar, está orientado a crear múltiples copias, por lo que se busca un modelo lo más simple posible, pero que también ofrezca una buena calidad de visualización.

También es importante destacar, que el modelo debe contener un "bone system", es decir, una jerarquía de huesos, similar al rigging de Blender, en caso de que se quiera crear alguna animación o modificar alguna que ya venga de serie.

5.2.1. Implementación de los Stats

Como los enemigos son un tipo de personaje que pueden interactuar con otros personajes del juego, deben de tener una serie de atributos que permitan saber las características que tienen.

Tanto para realizar ataques, como para recibirlos, deben de tener una serie de stats que permitan saber su estado para así, poder realizar, cierto tipo de acciones. Por ejemplo, cuando realiza un ataque, hay que saber el daño que hace, al igual, que, si el daño es recibido, saber cuánta defensa tiene para saber el daño total que le ha sido causado.

Para implementar éste sistema, se ha creado la clase **EnemyFeature**, cuyo funcionamiento es muy similar al de la clase Feature del personaje principal. Ofrece una serie de atributos básicos como HP, Damage … etc cuya manipulación se realiza dentro de la propia clase, para controlar los eventos que puedan surgir frente a distintos rangos de valores. Si al indicarle, que ha sufrido un daño, y éste produce que el HP sea menor que 0, esta clase es la encargada de indicar que el personaje está muerto, y no habría que comprobar de manera externa que HP<0.

Hay que recordar, que también se a aplicado el mismo **patrón** que en la clase Feature, **GRASP**: Experto en Información.

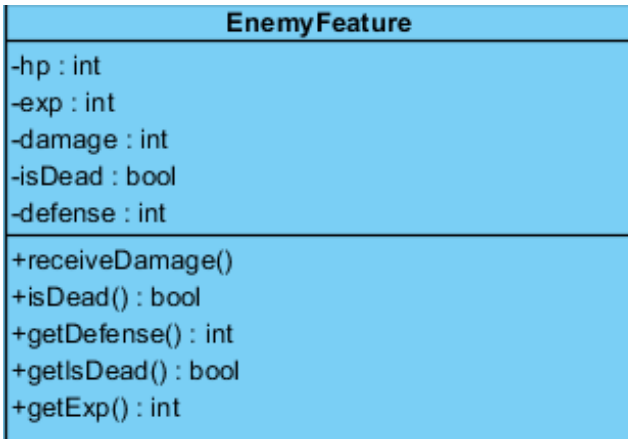


Ilustración 64. Clase EnemyFeature

5.2.2. Atacar y ser atacado

En éste apartado, se van a explicar las clases y métodos aplicados, para que a la hora, tanto de realizar como de recibir un ataque, puedan ser efectuados. La forma en la que se realizan los ataques se explicarán en el siguiente punto: "Implementación de la IA".

La acción de **realizar ataque**, es únicamente llamada desde el sistema que gestiona la IA, la cual indica al agente la acción que debe realizar. La idea es tener desacoplados elementos que pertenecen a la IA, de los comportamientos propios del agente, en este caso, el enemigo. Éste sistema, se ha diseñado de forma que, la acción final que tome el personaje viene determinada por la IA, pero la ejecución recae sobre la clase EnemyBehaviour**.**

La clase **EnemyBehaviour**, es la encargada de controlar acciones básicas del personaje tales como realizar ataques, recibir daño o ejecutar el estado de muerto. Este tipo de diseño, permite tener un sistema robusto a la hora de ejecutar cierto tipo de eventos, ya que esta clase, es la única encargada de gestionar las acciones que va a realizar el enemigo. Por ejemplo, cuando se activa la acción de atacar al personaje, es la responable de indicar al personaje dañado, la cantidad de daño que va a recibir, ejecutar la animación de ataque y reproducir el sonido.

Si éstos elementos se trataran por separado, el sistema podría ser inestable, a parte de ser más difícil de depurar, debido a que los eventos que ocurren deben estar sincronizados y perfectamente acoplados. Saber el estado de cada uno de ellos evita llegar a situaciones anómalas tales como la de realizar un ataque al personaje, estando el propio enemigo muerto.

Ejemplo del código donde se ejecuta la acción de **atacar al personaje**:

```
/*
    Realiza un damage al personaje
*/
public void atackCharacter(){

    //Comprobar que el propio agente no este muerto
    if (!isDead) {

        characterFeature.removeHP (enemyFeature.getDamage ());
        animator.SetBool ("Atack",true);
        impactSound.Play ();

    }

}
```

Ilustración 65. Fragmento de código atacar enemigo

A continuación (ilustración 66), se muestra un fragmento del **diagrama de clases**:

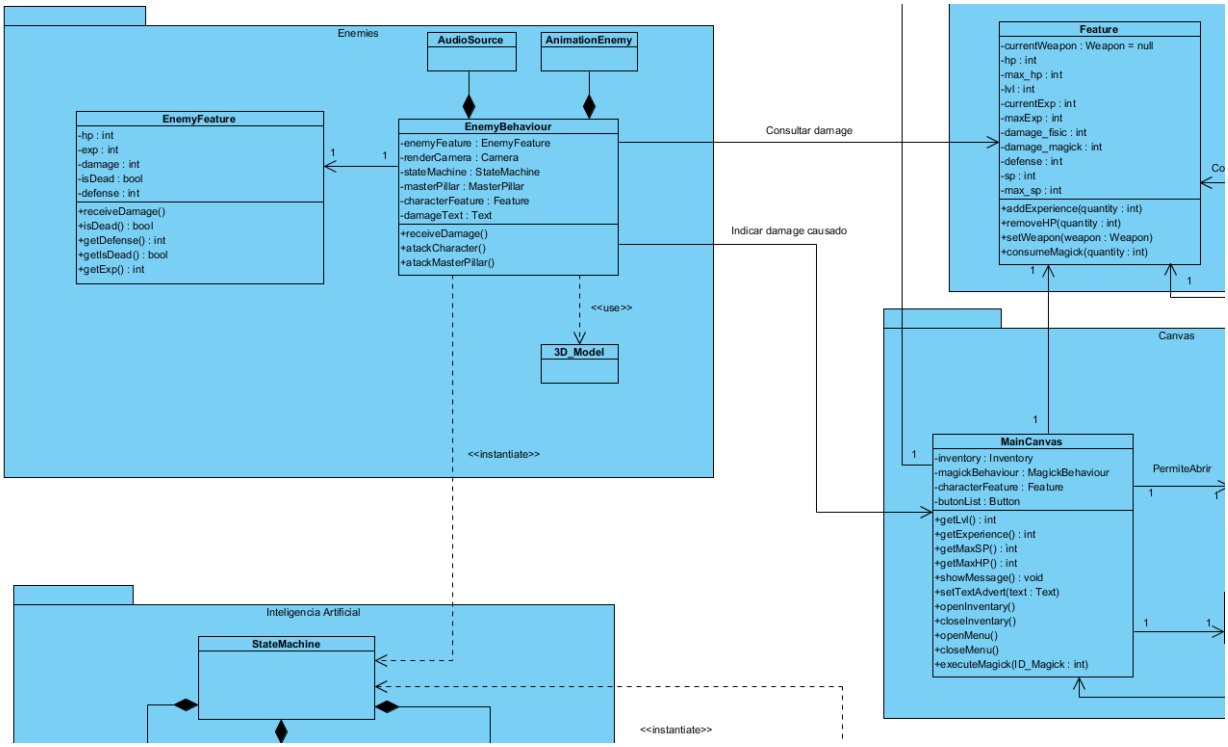


Ilustración 66. Diagrama UML de Enemy

La **referencia** que tiene al **MainCanvas**, se debe a que, cuando al enemigo se le aplica un daño, éste aparece en forma de texto para que pueda ser visualizado por el jugador. Ocurre igual cuando el daño es recibido por el jugador.

Ilustración 67. Texto en canvas para mostrar daño

5.2.3. Implementación de la IA

Antes de empezar a desarrollar la IA, se necesita saber cuales son los recursos de los que se dispone, para dar soporte a esta implementación. Por parte de **Unity**, nos ofrece recursos tales como **NavMesh** y **NavMeshAgent**.

El componente **NavMesh**, permite saber por qué parte del terreno se puede mover al agente. Eso dependerá de la configuación de éste, ya que, dependiendo del tamaño, podrá o no acceder a unas determinadas zonas (Unity, 2017f). A continuación, se muestra en color azul, la NavMesh que ha sido generada:

Ilustración 68. Representación del NavMesh

El componente **NavMeshAgent**, permite desplazar por la NavMesh, a cualquier elemento del juego que lo tenga asociado. Para ello, se genera un cilindro, cuya función es la de envolver al agente que se desea mover, por lo que, hay que adaptarlo a su tamaño. Tambíen se suele poner un ángulo de inclinación, para determinar las zonas por las que puede subir. (Unity, 2017g)

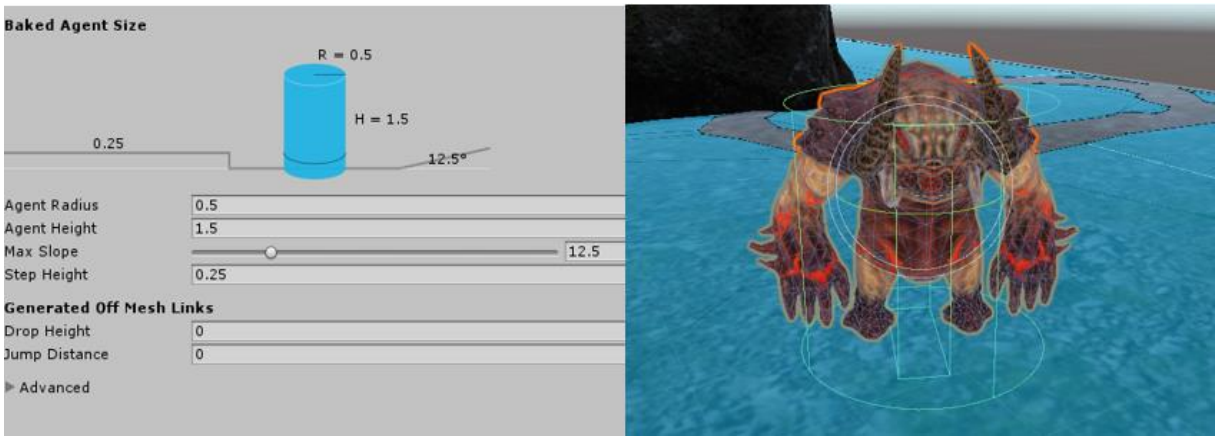


Ilustración 69. Cilindro envolvente del NavMesh

Con éstos recursos que nos ofrece el motor, ahora, el siguiente paso ha sido implementar un sistema que indique a esos agentes, dónde se tienen que mover, cúando deben pararse, establecer la velocidad con la que lo hacen… etc.

Para todo ello, se ha implementado un sistema de IA lo más flexible posible. Con esto, se quiere decir que, los elementos por los que está formado, puedan ser reutilizados por cada agente que lo use evitando hacer una IA específica para cada uno.

Para ello, se han establecido funciones básicas que establecen el estado en el que el agente se encuentra. El estado "**StateGoTo**", permite desplazar al agente, "**StateIdle**", estado de reposo y "**StateAtack**" permite realizar un ataque a un objetivo. Con éstos 3 estados, se puede programar una IA que indique cual de ellos se debe hacer en un momento determinado.

Para desarrollar la IA, se ha tomado la decisión de hacerla usando una **máquina de estados**, ya que permite satisfacer las necesidades para resolver el problema.

La máquina de estados, nombrada como **StateMachine,** permite una implementación específica para determinar el comportamiento que va a seguir cada tipo de agente, decidiendo para ello, qué estado ejecutar en un momento dado. (Asociación de estudiantes de videojuegos, s.f.)

En éste caso, el comportamiento que van a seguir los enemigos va a ser el representado por el siguiente diagrama de estados:

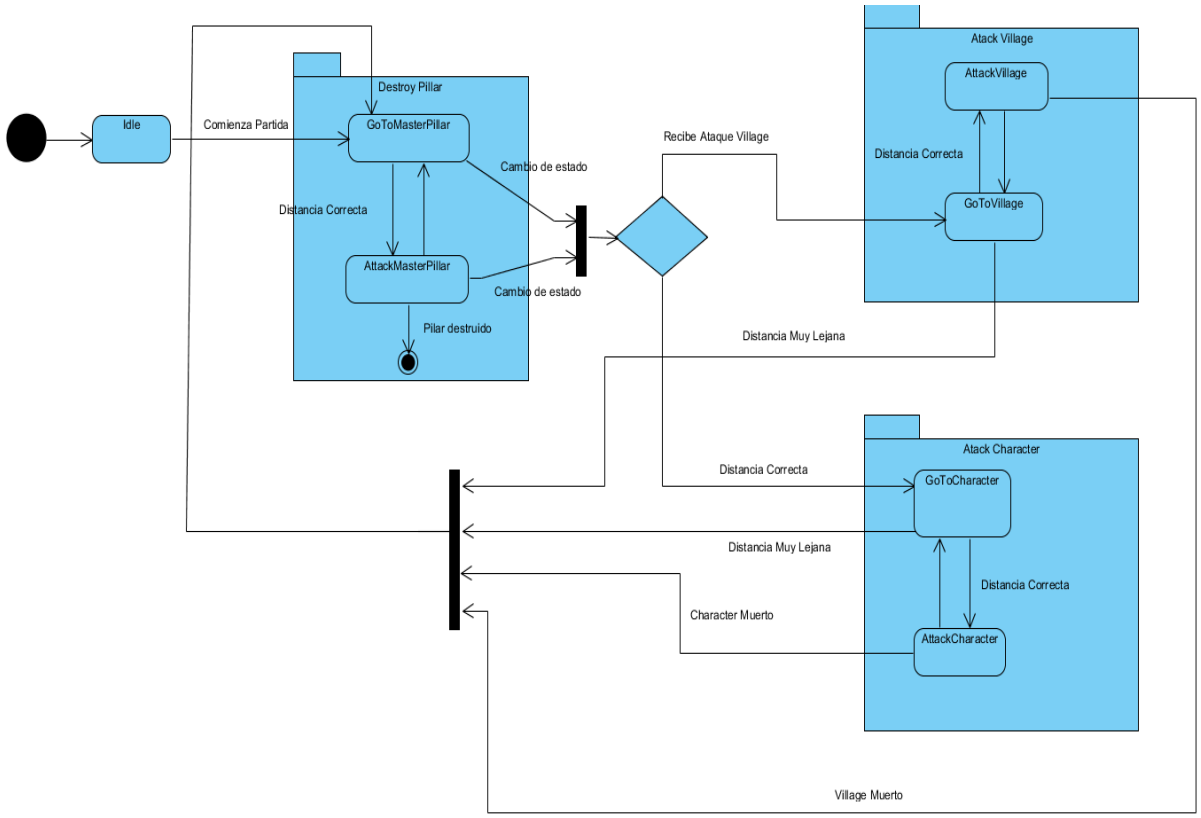


Ilustración 70. Máquina de estados de enemy

A continuación, se va a explicar el **desarrollo** que ha tenido la **máquina de estados** del enemigo y los parámetros de entrada de los que ésta dispone:

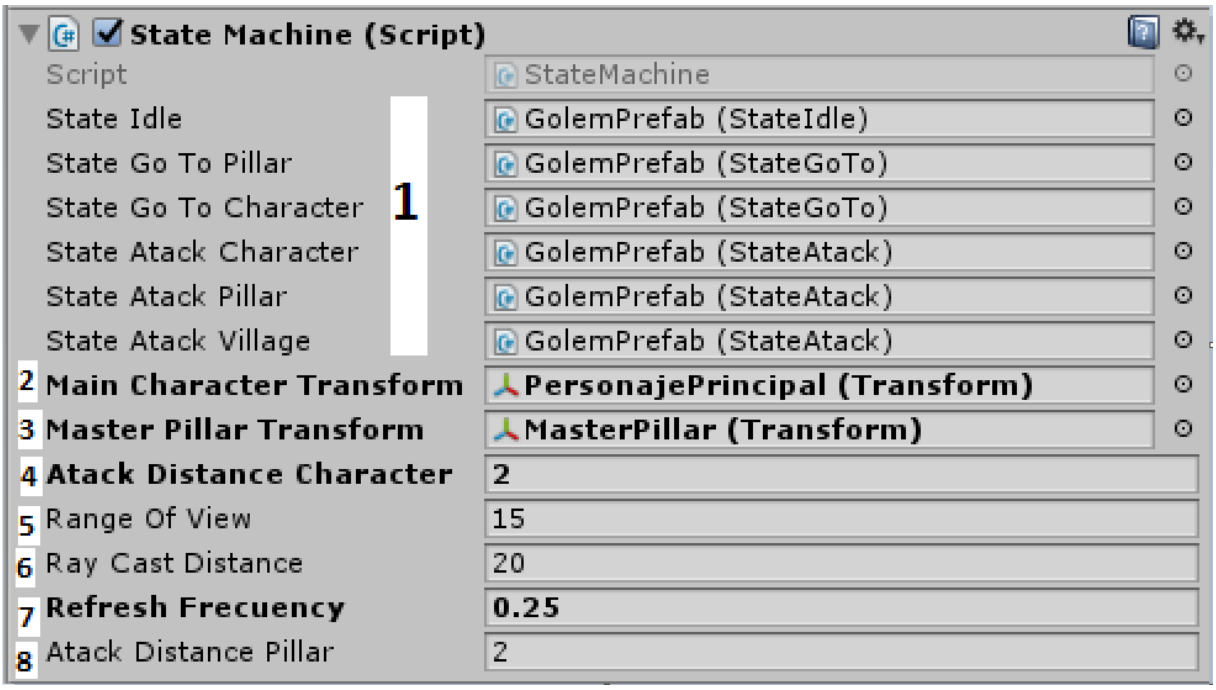


Ilustración 71. Máquina de estados de enemy

En la **parte 1,** se cargan todos los estados posibles en los que puede encontrarse el enemigo. Como se puede ver, haciendo uso del "**StateGoTo**", tiene 2 objetivos posibles, ir al "**Pillar**" o ir hacia el "**Character"**, personaje principal que maneja el jugador.

El resto de estados, son de tipo ataque, cuyos objetivos pueden ser el Character, Pillar o Village. Los Village, son otros personajes del juego que se explicarán más adelante. También se ha añadido el estado idle, contemplando el caso de que el agente no pueda realizar ninguna acción y tenga que estar en reposo.

La **parte 2**, sirve para tener una referencia de la posición del personaje principal. Sirve calcular la distancia para comprobar si debe de atacarlo, o está demasiado lejos para hacerlo. También sirve para que en caso de tener que realizar el ataque, saber hacia que posición del mundo debe desplazarse el agente.

La **parte 3**, tiene un objetivo similar a la parte 2, pero en lugar del personaje, la posición es la del pilar.

La **parte 4**, establece la distancia a partir de la cual se debe cambiar del estado GoTo al estado Atack, para atacar al personaje.

La **parte 5** y **parte 6**, asignan la distancia de visión del enemigo. Si la distancia del personaje es menor que ésta, comienza a perseguirlo. La cuestión por la que se establecen 2 distancias, está explicada en la parte de "9.2.2 Cálculo de distancias".

La **parte 7**, es un valor orientado al rendimiento, el cual establece, cada cuánto tiempo se debe refrescar la máquina de estados para comprobar si debe realizar algún cambio. El motivo tambíen está explicado en el apartado "9.2.1 Actualización de la máquina de estados".

La **parte 8**, es similar a la parte 4, la diferencia es que la distancia de ataque la establece sobre el pilar, en lugar del jugador.

Por último, se va explicar el **funcionamiento de los States** y la forma en la que se han **implementado** para el enemigo:

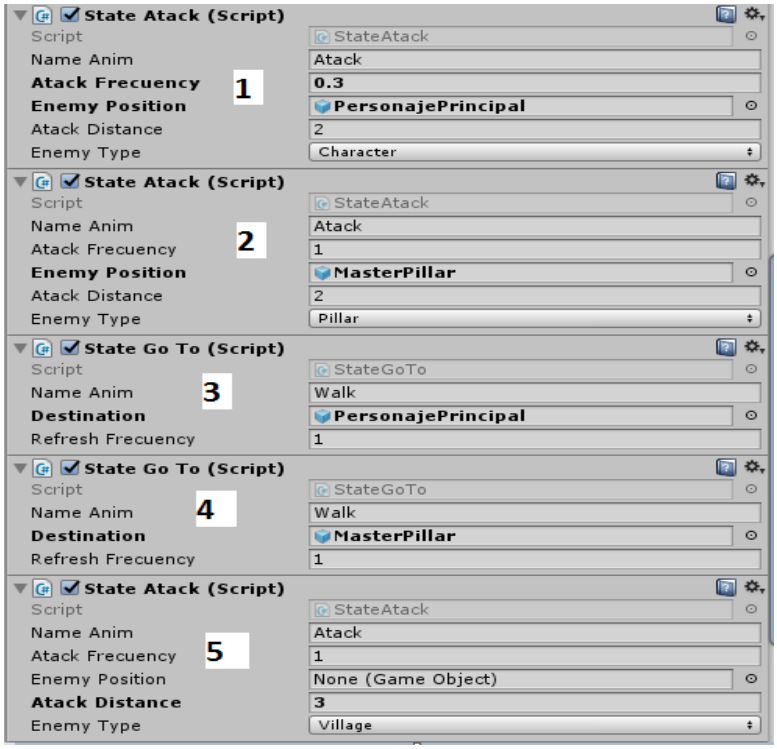


Ilustración 72. States del enemigo

En la **parte 1**, **parte 2** y **parte 5**, se establece el estado mediante el cual, el agente realizará un ataque.

El primer campo, "**Name anim**", hace referencia al nombre de la animación que se va a realizar cuando se ejecute el estado.

"**AtackFrecuency**", establece la frecuencia con la que va a realizar el ataque.

"**EnemyPosition**", la posición dada en coordendas donde está el agente al que se quiere atacar.

"**AtackDistance**", mediante la posición obtendia de EnemyPosition, se establece la distancia a partir de la cual, el agente, puede empezar a realizar el ataque.

"**EnemyType**", sirve para seleccionar al tipo de agente sobre el que recae el ataque. Este campo, sirve para indicar a qué clase se debe establecer una referencia, ya que como anteriormente se ha explicado, las clases responsables de restar vida, son aquellas que contienen el atributo de vida. Es por eso, por lo que se necesita saber a qué clase se debe llamar para decrementar la vida.

Por ejemplo, si se quiere realizar un ataque sobre el personaje principal, se deberá tener una referencia a Feature, y hacer uso del método removeHP();

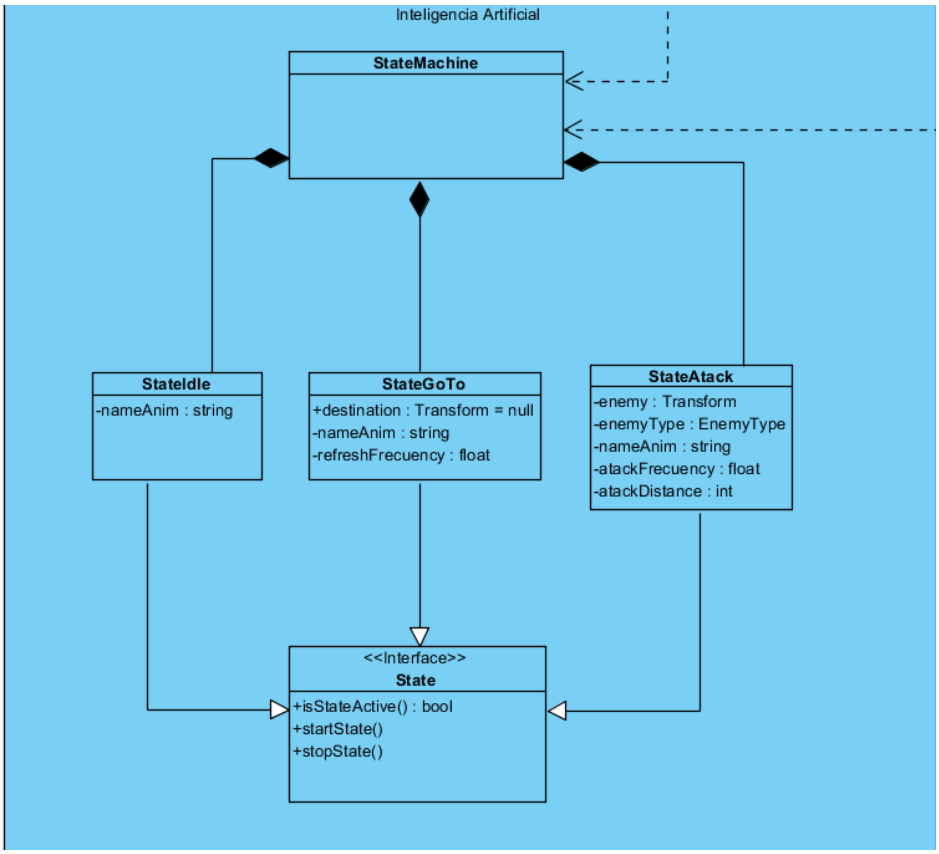


Ilustración 73. Diagrama de clases de la máquina de estados

Como se puede ver, tanto en StateMachine, como StateAtack, se utiliza la variable “atackDistance”. En un primer momento, puede parecer redundante, pero en StateMachine se utiliza para decidir que estado ejecutar, y en StateAtack comprueba que la distancia es la correcta para poder realizar el ataque (ejecutar la animación de ataque y restar vida al agente que convenga).

Con todo lo explicado anteriormente, junto con el funcionamiento desarrollado en el diagrama de máquinas de estado, se ejecuta la inteligencia artificial de los enemigos tipo “Golem”.

5.3. Interfaz de usuario (Canvas)

En puntos anteriores, ya se han explicado elementos que componen la interfaz de usuario, como por ejemplo, el MainCanvas o el Inventory. No está de más, profundizar en éstos elementos, junto con otros, que todavía no han sido comentados, de forma que en éste apartado se hará especial énfasis en aquellos no comentados aún.

5.3.1. MainCanvas

El componente principal de la interfaz de usuario, está formado por la clase **MainCanvas**. Esta clase, permite al jugador, tanto obtener información del personaje, como la de poder acceder a otros canvas, como por ejemplo el menú o el inventario.

La información que ofrece al jugador, son los valores de HP, SP, Nivel y Experiencia que en ese momento posee el personaje. Éstos valores se actualizan cada vez que sufren algún tipo de modificación, por lo que los cambios se visualizan en tiempo real. La forma en la que se ven modificados estos atributos, es la misma que la propuesta en el punto 3.6.1, en la que se detallan las ideas sobre la interfaz.

Para ello, la clase Feature , cada vez que sus atributos sufren algún tipo de modificación, se notifica a la clase MainCanvas para que sean actualizados.

Ilustración 74. Cuadro de stats

Aprovechando que es el canvas principal, se ha implementado un método, mediante el cual, se puedan **visualizar advertencias** en formato texto. Esto permite, que si otras clases tienen que ofrecer algún tipo de información relevante al jugador, lo hagan a través del MainCanvas, evitando tener que crear cada una, un método específico para poder visualizar texto. De ésta manera, la responsabilidad de visualizar advertencias, sólo recae sobre la clase MainCanvas.

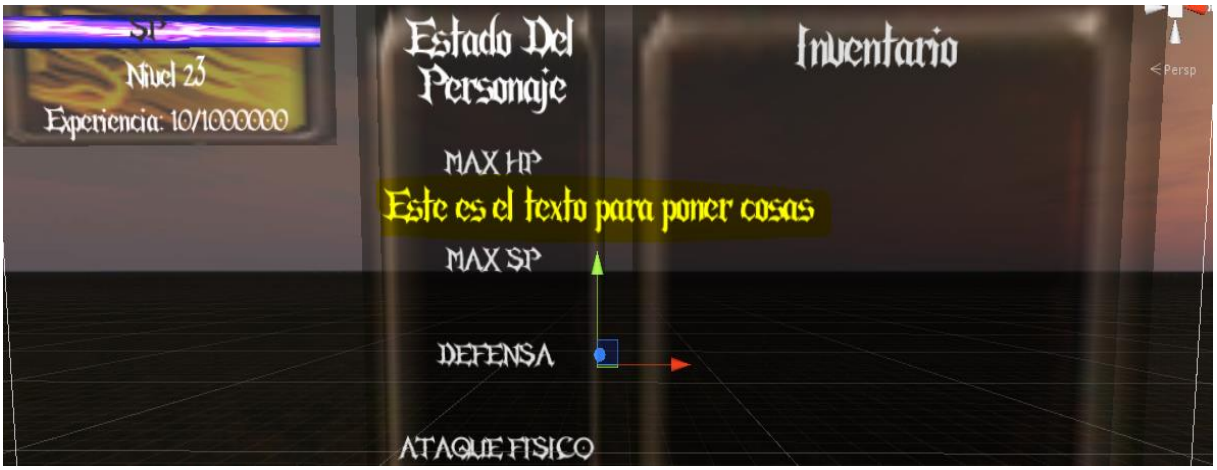


Ilustración 75. Texto del canvas

El texto resaltado en amarillo (ilustración 75), será sustituido por el que le indique la clase que haga uso de éste método (showMessage() de la clase MainCanvas).

5.3.2. Inventory

Mediante una referecia al canvas Inventory, se ha implementado un botón que permite abrirlo y poder acceder a él. La explicación del funcionamiento de éste canvas, ya se ha hecho en el punto “5.1.7 Implementación del inventario”.

5.3.3. Menú

A través del MainCanvas se puede acceder al menú, cuya funcionalidad principal es la de poder salir del juego.

Ilustración 76. Acceso al menú desde el MainCanvas

Ilustración 77. Representación del menú

A continuación (ilustración 78), se expone el **diagrama de clases** resultante de éste sistema:

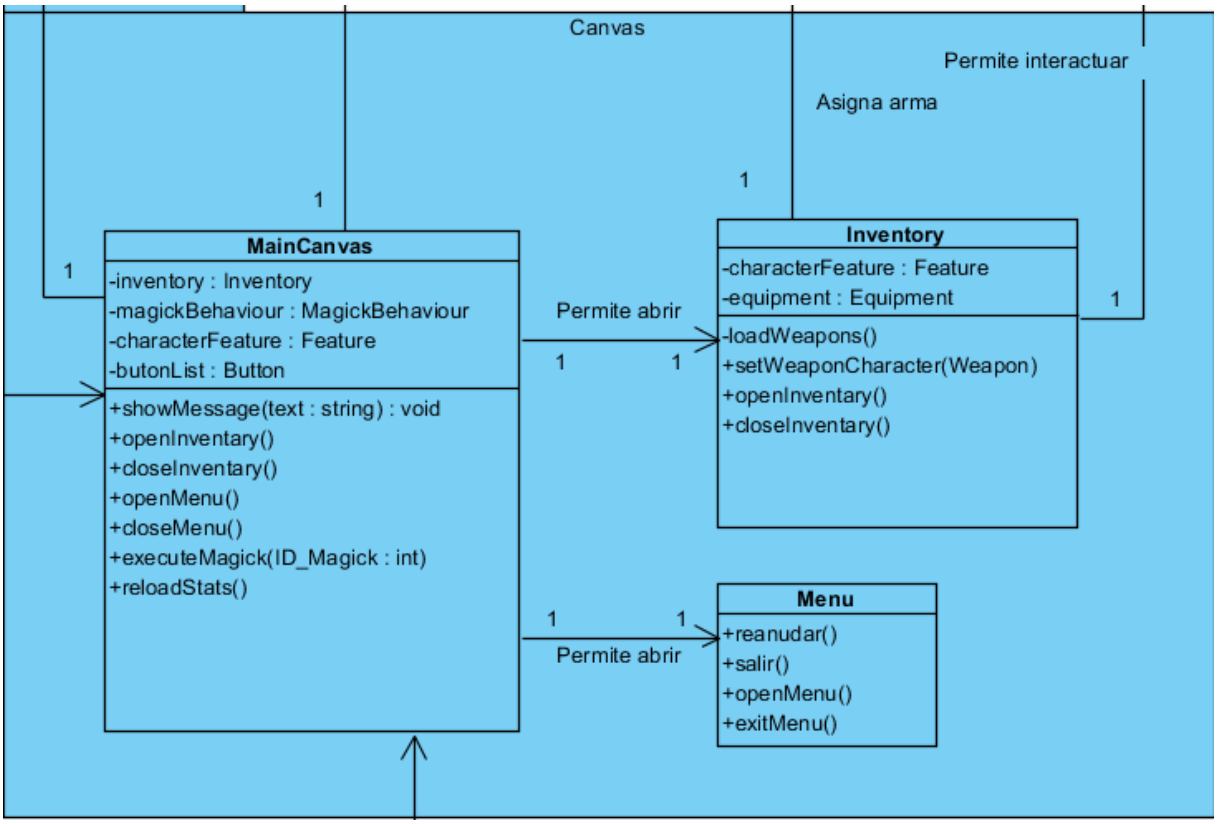


Ilustración 78. Diagrama UML del canvas

5.4. Village

Ilustración 79. Representación de los village

Ya que la misión 1, se desarrolla en un pueblo, se ha visto conveniente la creación de unos NPC (Non Person Character) que representen a los habitantes, llamados “village”. El objetivo que tienen, será el de proteger el pilar frente a cualquier ataque enemigo y el de estar merodeando por el pueblo y sus alrededores.

5.4.1. Implementación de la IA

Cualquier comportamiento que no sea controlado por un jugador, requiere el uso de una IA. Aprovechando los recursos que nos ofrece la IA ya implementada, se va a reutilizar para usarla con los village, al igual que se ha utilizado con los enemigos.

Ya que, el funcionamiento de la máquina de estados, para poder desarrollar una IA, ha sido explicada en el punto 5.2.3, tan sólo se va a proceder a explicar la adaptación de la misma para los village.

En primer lugar, se ha creado el **diagrama de la máquina de estados**, para representar el comportamiento que se desea que éstos tengan:

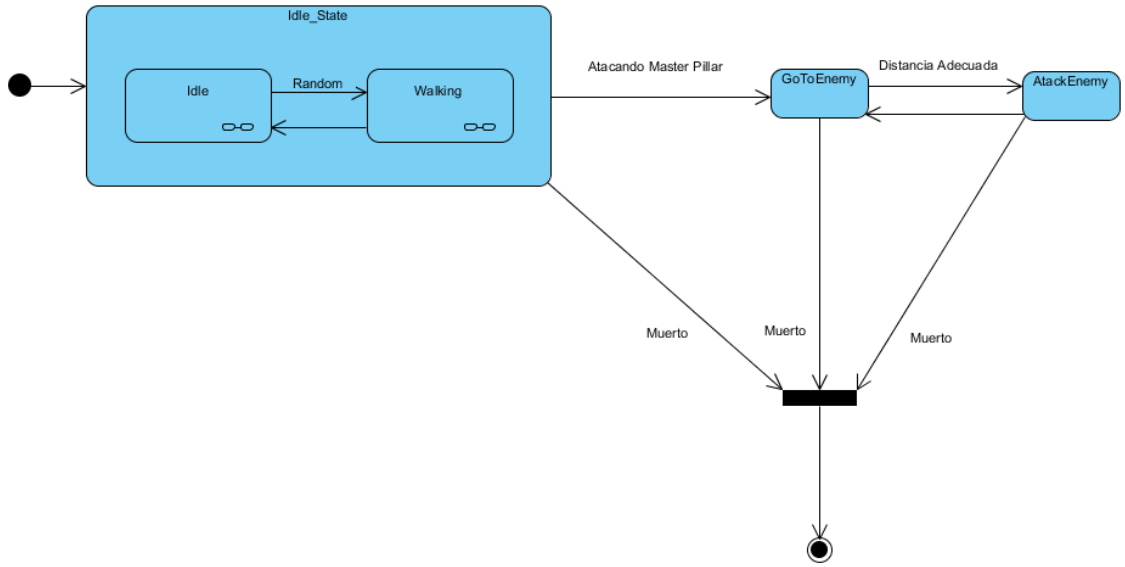


Ilustración 80. Diagrama de estados de los village

La implementación de la **máquina de estados** es la siguiente:

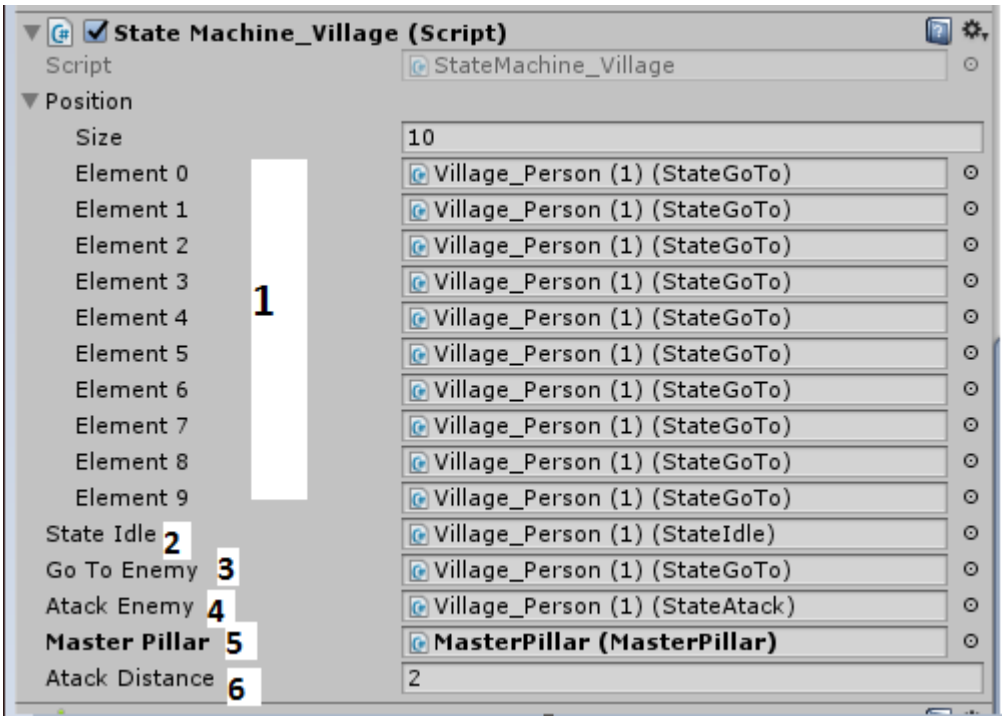


Ilustración 81. Máquina de estados de los village

La **parte 1**, es una lista de estados de tipo “StateGoTo”, cuya función es la de establecer una serie de puntos por los que el agente pueda pasar. A la hora de tener que ejecutar ésta parte, se elige un punto de manera aleatoria para que los agentes no realicen todos el mismo recorrido y dé una mejor sensación a la hora de jugar. (Asociación de estudiantes de videojuegos, s.f.)

La **parte 2**, es el estado “StateIdle”, el estado de reposo, que se ejecuta cuando se llega a alguno de los puntos establecidos del apartado anterior.

A lo largo del juego, los agentes van modificando su trayectoria con una regularidad establecida mediante un intervalo de tiempo, la cual, sólo se ve perturbada, cuando estén atacando al pilar. Cuando ocurre, se selecciona a uno de los enemigos que ha atacado al pilar, y se ejecuta la **parte 3** y cuando esté a la distancia adecuada, lo cual se establece en la **parte 6**, se ejecuta la **parte 4**.

La **parte 5** hace referencia a la clase MasterPillar, que es la encargada de indicar si está siendo atacada, para que el agente modifique el estado en el que se encuentra y también tiene la labor de devolver a los enemigos que han realizado el ataque para que los village sepan a quién atacar.

Puede darse el caso, de que el primer enemigo que ataque el pilar, se le asignen a todos los village para que sea atacado. Estas **sobrecargas**, provocan que el propio jugador no pueda atacar a los enemigos, debido a que, todos los village están atacándolo y el espacio que queda para poder hacerlo, es insuficiente.

Para evitarlo, MasterPillar, dispone de una lista con los enemigos que lo están atacando. Cada vez que un village, cambie de estado para defender el pilar, le solicita a MasterPillar un enemigo, que extrae de esa lista. Esto permite que la cantidad de village que atacan a un enemigo se distribuya, para que el jugador pueda participar en el ataque de una forma más cómoda.

5.4.2. Stats village

Como toda entidad que tenga una relación con otros elementos del juego, debe tener unos stats que identifiquen su estado. Para los village, también se ha implementado una clase para representar sus stats. Debido a que los stats de los village sólo contiene el atributo HP, se ha hecho una clase que empaqueta todos los comportamientos de éste, permitiendo que cualquier interacción que se haga con éstos, se realice a través de esta clase, denominada **VillageBehaviour**.

También contiene una referencia a la máquina de estados para que al inicializar al personaje, pueda arracar la máquina y comenzar a ejecutarse su IA.

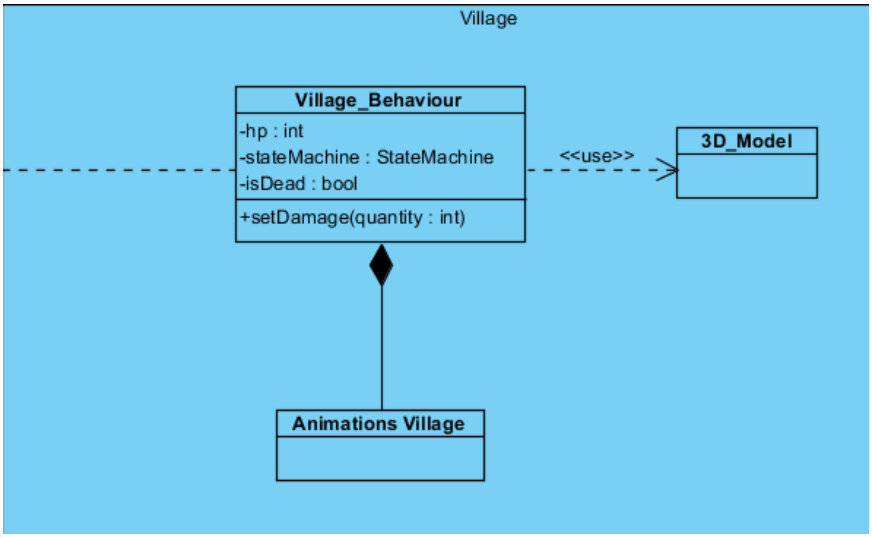


Ilustración 82. Diagrama UML de los village

5.5. Master Pillar

Ilustración 83. Master pillar

El objeto **MasterPillar**, es una de las partes más importantes de la misión 1, ya que el objetivo del jugador es evitar que los enemigos lo destruyan.

Para desarrollarlo se ha utilizado un modelo 3D del asset store, el cual, se ha adaptado perfectamente a los requisitos del proyecto, como se puede ver en la imagen.

Como ya se ha comentado en otras ocasiones, todos los elementos con los que se pueden interactuar, deben tener implementada una clase que permita saber su estado y ofrecer mecanismos para poder interactuar. Para ello, se ha desarrollado la clase **MasterPillar**, que representa el estado interno del pilar junto con una serie de métodos que nos permiten su correcta manipulación.

Ésta clase es usada por la IA del enemigo, con el fin de poder localizar la posición del objeto y poder atacarla.

También es usada por la IA de los village, para saber si el pilar está siendo atacado, permitiendo obtener la referencia del enemigo que le ha realizado el ataque. Para ello, se ha desarrollado una Queue<Enemy>, en la que se van almacenando los enemigos que atacan al pilar. Cuando un Village, decide atacar al

enemigo, lo solicita a MasterPillar y ésta le devuelve a un enemigo extraido de la cola. Si el village o el jugador no consiguen eliminar al enemigo, y sigue atacando el pilar, vuelve a ser introducido en la cola para que pueda volver a ser objetivo de otro village.

En caso de que MasterPillar sea destruido, es el responsable de indicar su estado al **GameController** para que pueda realizar los eventos necesarios para dar la partida como fracasada.

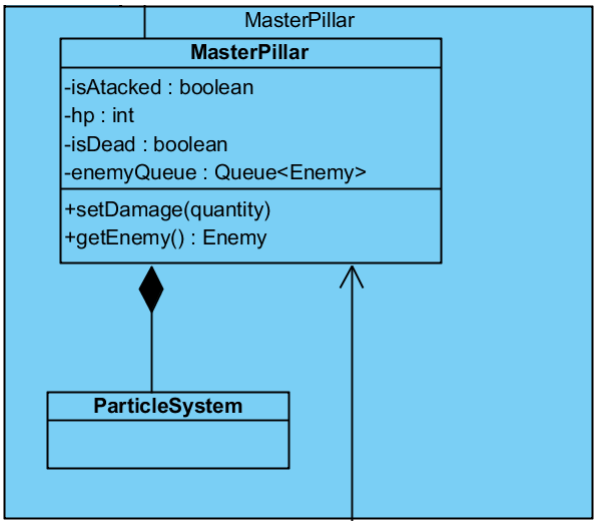


Ilustración 84. Diagrama UML MasterPillar

5.6. Diagrama de clases final

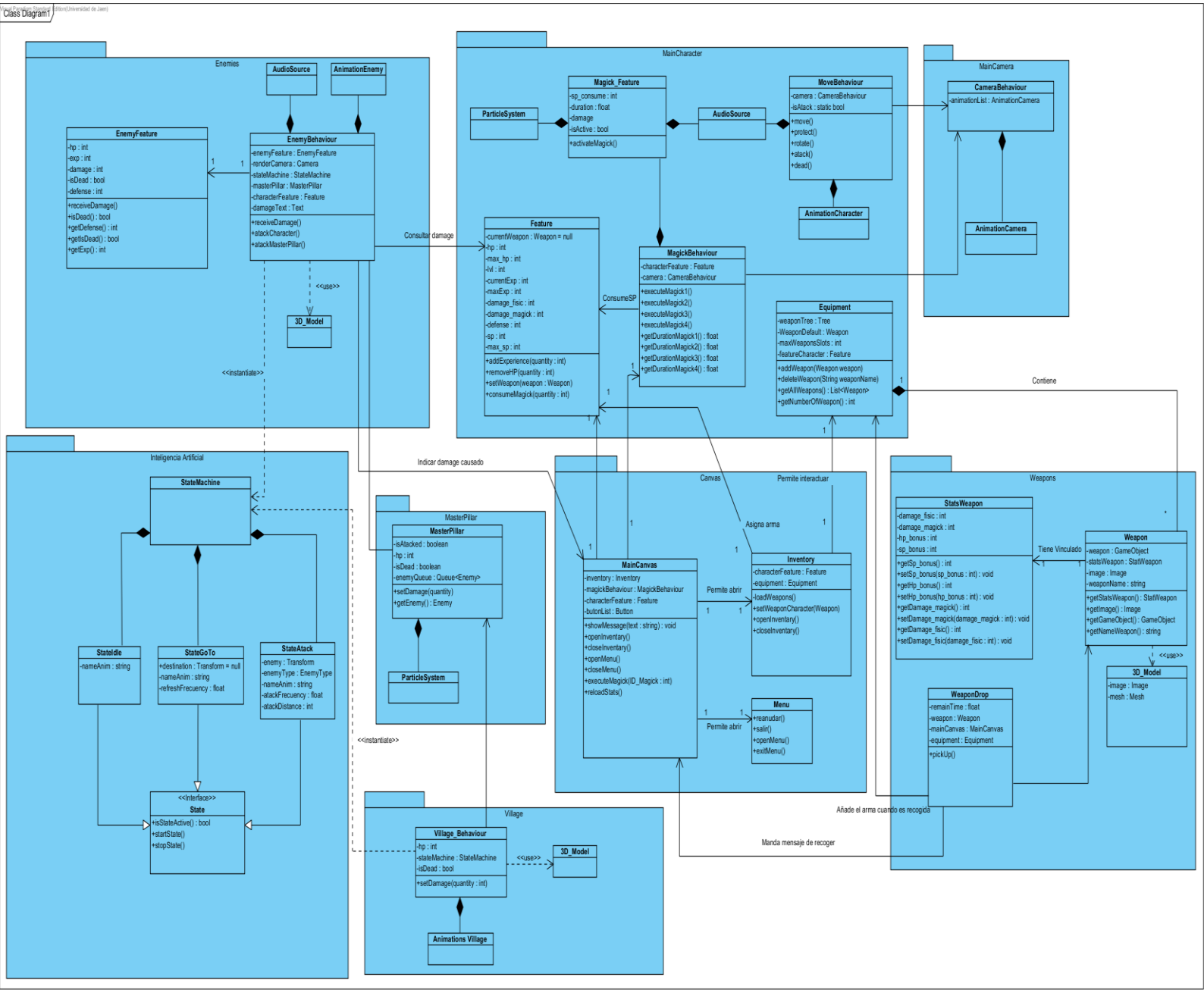


Ilustración 85. Diagrama de clases principal

5.7. Game Controller

El **GameController**, es un componente del videojuego que permite controlar todos los eventos que van surgiendo a lo largo de éste (Unity, 2017h). Cada misión está compuesto por uno, por lo que su implementación dependerá del tipo de misión.

Para éste proyecto, sólo se ha implementado el de la Misión 1: Defender el pilar, el cual, permite controlar eventos tales como cinemáticas, generación de enemigos e identificar criterios de éxito o fracaso. A continuación se detallará cómo se han desarrollado estos elementos:

Durante la partida, cuando se atraviesa una determinada zona, el modo de juego cambia, y comienzan a venir enemigos, dando lugar al comienzo de la misión.

Para ello, se han establecido colliders para detectar que el personaje está en la zona adecuada. Una vez detectado, se activa una cinemática y se da comienzo a la generación de los enemigos. Lo que ocurre después, es controlado por las IAs de los respectivos agentes. Todos estos eventos, son controlados por el GameController.

El GameController, contiene una referencia a los elementos clave del juego, que sirven para determinar si la partida ha **fracasado** o por el contrario, ha tenido **éxito**. Dichos elementos son el pilar maestro y el jugador principal. Para evitar estar continuamente consultando el estado en el que se encuentran, más que nada, por aspectos de eficiencia, los responsables de indicar un estado crítico, en éste caso, estar muerto o ser destruido, son los objetos referenciados y no la propia clase.

Dependiendo de quién indique primero el estado en el que se encuentra, el GameController, hará que en el juego ocurra un evento u otro, ya que, no es lo mismo que destruyan el pilar, que eliminen al jugador.

A continuación, se muestran los inputs que se requieren para que el GameController pueda funcionar correctamente:

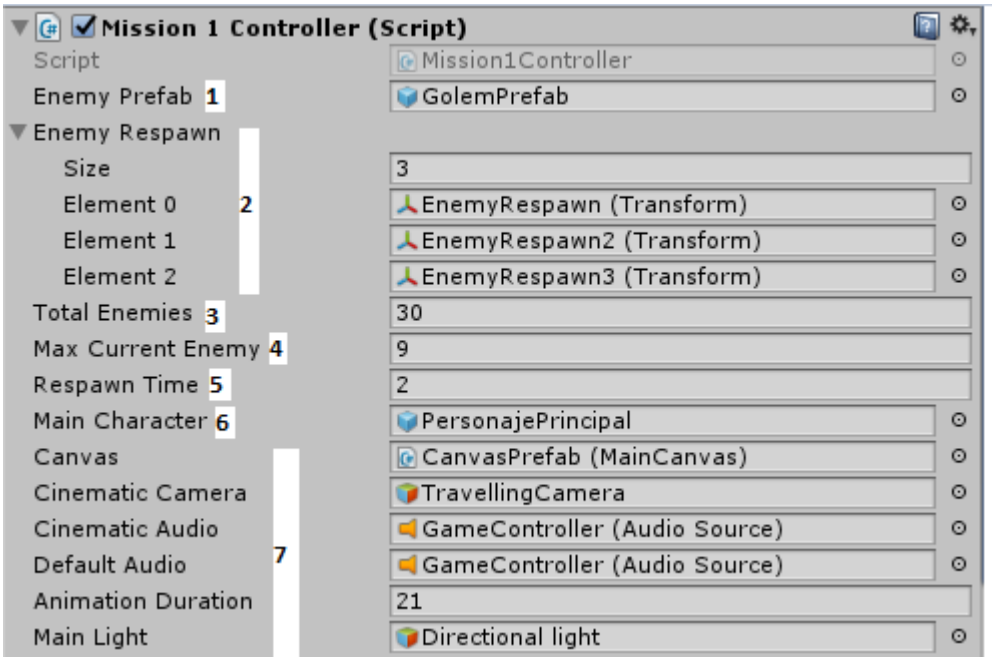


Ilustración 86. GameController script

La **parte 1**, contiene la referencia al GameObject del enemigo para poder instanciar múltiples copias.

La **parte 2**, son las zonas de respawn de los enemigos, es decir, las posiciones dentro del mapa desde donde son instanciados.

La **parte 3**, establece el número de enemigos que van a ser generados, de forma que, cuando todos estén eliminados, se habrá completado la misión 1.

La **parte 4**, establece el límite de enemigos que puede haber en el juego al mismo tiempo.

La **parte 5**, es una variable de tiempo, que asigna cada cuánto tiempo se genera un nuevo enemigo.

Las **partes 3, 4 y 5**, permiten tanto regular aspectos de **rendimiento**, como de jugabilidad, ya que un número elevado de enemigos puede provocar caidas de rendimiento, junto con una sobrecarga de enemigos que impidan una correcta

jugabilidad. Estos valores, pueden ir siendo modificados en la fase de pruebas y testing, con el objetivo de llegar a ofrecer una dificultad agradable al jugador.

Las **partes 6 y 7** son utilizadas para controlar los eventos que se producen a la hora de activar la cinemática. Se tiene que tener una referencia al personaje principal, con el objetivo de desactivarlo, ya que no conviene que el jugador siga teniendo el control de éste durante ese tiempo. El resto de elementos, tales como la cámara, sonido, iluminación son componentes involucrados para que la cinemática se ejecute de forma correcta.

5.8. Mapas

Los mapas han sido elaborados acorde los prototipos expuestos en los apartados 3.3.3. “Prototipo a papel de la misión 1” y 3.3.4. “Prototipo a papel de la misión 2”.

Unity proporciona un conjunto de herramientas para modelar terrenos, las cuales, han permitido elaborar de forma satisfactoria los mapas propuestos.

Para la creación del **mapa de la misión 1**, se han utilizado un conjunto de objetos y texturas, previamente importadas del AssetStore de Unity, para ofrecer la máxima fidelidad acorde lo estipulado en los prototipos. (Unity Asset Store, s.f.)

Ilustración 87. Mapa misión 1

Cabe destacar que durante su elaboración, se han tenido que modificar algunos aspectos de configuración asociados a algunas texturas, debido a que ofrecían un resultado poco realista.

Ilustración 88. Textura suelo mal configurada

La excesiva refracción de la luz, por parte de algunas texturas, ha sido el problema más habitual. Para solucionarlo, se ha modifado la forma en la que la

textura es renderizada, realizando una modificación en los *shaders* para que la cantidad de luz arrojada sea menor. (Unity, 2017i)

Ilustración 89. Textura del suelo bien configurada

Para la creación del **mapa de la misión 2**, se han utilizado las mismas técnicas que para el mapa de la misión 1, ofreciendo finalmente el resultado de la ilustración 90:

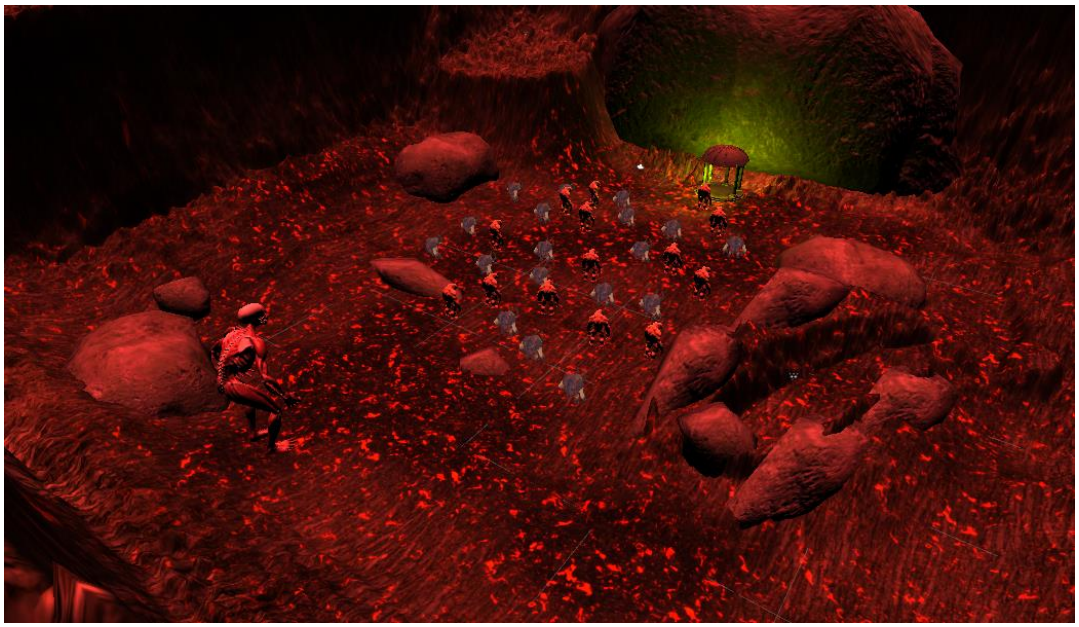


Ilustración 90. Mapa del mal

Término	Definición
Shader	Tecnología utilizada para calcular el color de cada pixel renderizado, basándose en la iluminación y la configuración del material.

Tabla 21.Definición de shader

6. Patrones de diseño aplicados

A continuación, se van a explicar los patrones de diseño que han sido utilizados durante el desarrollo de éste proyecto. Para ello, se va mencionar el tipo de patrón y el lugar donde han sido aplicados.

6.1. Patrones GRASP

Los patrones **GRASP** (General Responsability Assignament Software Patterns), mas que patrones que indiquen la forma de solventar un determinado problema, ofrecen una serie de pautas y buenas prácticas para un desarrollo software orientado a objetos. (Larman, 2003)

No siempre se pueden aplicar todos los patrones, ya que, dependiendo del comportamiento que tengan las clases, se podrán usar unos u otros.

A continuación, se van a explicar los patrones GRASP más comunes y algún ejemplo de las clases en los que han sido usados, ya que no tiene sentido mencionar todas las clases debido a que los aspectos funcionales son similares:

Experto en información: "*Nos indica, que la responsabilidad de la creación de un objeto o la implementación de un método, debe recaer sobre la clase que conoce toda la información necesaria para crearlo*" (Larman, 2003)

Utilizado en todas las clases, ya que cada una es responsable de manejar los atributos por los que está compuesta, no dejando esa labor a otras entidades.

Controlador: " *Sirve como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es la que recibe los datos del usuario y la que los envía a las distintas clases según el método llamado*" (Larman, 2003)

Este patrón ha sido utilizado a la hora de ejecutar las magias, mediante la cual, se ha establecido como interfaz, el MainCanvas, y el controlador MagickBehaviour. Mediante una señal del MainCanvas, se solicita a MagickBehaviour la ejecución de una determinada magia (MagickFeature), el cual, determina si se puede realizar o no. La parte relacionada con el sistema de magias ha sido explicado en el punto 5.1.8.

Alta cohesión y bajo acoplamiento: Se ha realizado un desarrollo lo más cohesivo posible en todas las clases, ya que, todos los métodos implementados en una clase están totalmente relacionados con dicha clase y con los elementos que almacena. Como anteriormente se había comentado, todas las clases siguen la filosofía del patrón Experto en información, lo cual conduce a obtener modelos altamente cohesivos.

Por otro lado, el acoplamiento se ha reducido lo máximo posible, haciendo que las dependencias entre clases sean mínimas para permitir su correcto funcionamiento.

6.2. Patrones GoF

Los patrones GoF (Gang of Four), ofrecen soluciones frente a un determinado problema software. A diferencia de los GRASP, estos ofrecen una solución implementable. (Larman, 2003)

A continuación se van a explicar algunos de los patrones GoF usados en el desarrollo de las clases:

Singleton: El patrón singleton “garantiza una instancia única y un acceso globlal”. (Larman, 2003)

El patrón singletone ofrece las características necesarias para implementar la clase encargada de controlar cada una de las misiones del juego.

El GameController de la misión 1 ha utilizado éste patrón, ya que no puede haber más de 1 instancia de ésta clase. Esto se debe a que podría conducir al sistema a una situación inestable. Por definición, el patrón expone que tiene que ser de acceso global, aunque para éste caso, se ha adaptado y se ha realizado un acceso más específico, concretamente a las entidades que pueden producir variaciones en el curso del juego, como por ejemplo, el MasterPillar o el MainCharacter.

```
public static Mission1Controller instance = null;

//Singletone pattern
void Awake(){

    if (instance == null) {

        instance = this;

    } else {

        if(instance!=this){

            Destroy (gameObject);
        }
    }
}
```

Ilustración 91. Ejemplo código patrón singleton

Prototipo (Prototype): “Crea nuevos objetos clonándolos de una instancia ya existente”. (Larman, 2003)

El patrón prototype ha sido utilizado, al igual que el Singletone, en la clase GameController de la misión 1. Se ha establecido como la responsable de crear múltiples copias de los enemigos a partir de una instancia obtenida a partir de un *prefab*.

Término	Definición
Prefab	En Unity, un prefab es un mecanismo a través del cual, te permite realizar múltiples copias de un mismo objeto con la particularidad de que al modifcar el prefab, el resto de copias también se modifican, evitando tener que hacerlo de una en una.

Tabla 22. Definición de prefab

Modelo Vista Controlador: “Separa los datos y la lógica de negocio de la interfaz de usuario” (E. Gamma, 2003)

La utilización de éste modelo ya ha sido previamente explicado en el apartado 5.1.8, en la implementación de las magias. Cabe destacar, que dentro de las formas de desarrollar éste tipo de patrón, se ha utilizado el estilo moderno:

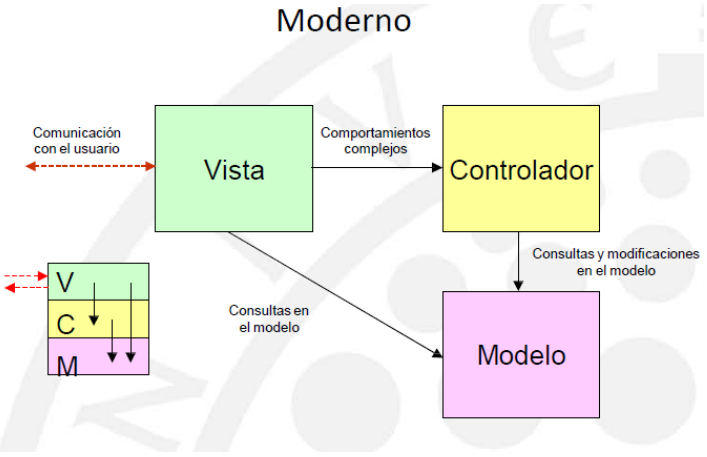


Ilustración 92. Ilustración MVC estilo moderno

Bridge: “Desacopla una abstracción de su implementación”. (Larman, 2003)

Este patrón ha sido utilizado en el punto 5.1.5, Mecánicas de movimiento. Se ha mantenido la aplicabilidad del patrón, aunque la estructura ha tenido que ser adaptada. Esto se debe a que no se puede tener acceso al código de todas las partes que se ven involucradas, ya que son elementos que pertenecen al motor gráfico, como por ejemplo, los eventos tipo Inputs.

Fachada: “Provee de una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema.” (Larman, 2003)

Utilizado para la implementación del MainCanvas, el cual funciona como interfaz principal desde la que se puede acceder al resto de interfaces, tales como el menu o el inventario.

Peso Ligero (Flyweight) : “Reduce la redundancia cuando gran cantidad de objetos poseen idéntica información.” (E. Gamma, 2003)

Este patrón ha sido utilizado a la hora de implementar los States de la máquina de estados. En lugar de implementar una IA específica para cada tipo de agente, con sus correspondientes estados, se han creado un tipo de estados de carácter genérico. Esto permite, que los estados puedan ser reutilizados evitando la redundancia de tener que implementar uno específico para cada agente.

Lo que se ha hecho, es crear unos estados básicos (GoTo, Idle, Atack), a partir de los cuales se pueda desarrollar una máquina de éstados que ofrezca todos los comportamientos deseados posibles.

Observador: “ Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualicen automáticamente todos los objetos que dependen de él.” (E. Gamma, 2003)

La aplicabilidad de este patrón ha sido utilizada en la clase Feature, la cual, notifica al MainCanvas si ha habido cambios en los stats para poder refrescarlos en tiempo de ejecución.

Estrategia: “Permite disponer de varios métodos para resolver un problema y elegir cuál utilizar en tiempo de ejecución.” (Larman, 2003)

La aplicabilidad de éste patrón ha sido implementada en la máquina de estados, la cual, dependiendo de la situación en la que se encuentre el agente, ejecutará unos comportamientos u otros.

7. Walkthrought

A continuación, se va a exponer una pequeña guía en la que se van a ir contando de manera descriptiva las distintas partes por la que está formado el videojuego. También va a servir de ayuda a los jugadores menos experimentados, ya que se van a explicar los pasos que hay que seguir para completar la misión, así como lugares de interés y la utilización de ciertas técnicas que facilitan la jugabilidad.

7.1. Introducción

La introducción del videojuego, se presenta como una cinemática, cuyo objetivo es la de poner al jugador en contexto de la situación. A lo largo de ésta, se explica brevemente la situación actual en la que se desenvuelve la historia y se muestra el reino enemigo junto con sus criaturas.

Ilustración 93. Reino del mal

7.2. Misión 1: Defender el pilar

Esta es la primera misión del juego, la cual da comienzo justo después de la introducción.

El personaje comienza en medio de un camino, teniendo de fondo un pueblo. Mientras se dirige hacia el pueblo, se puede ver una birfurcación antes de llegar.

Ilustración 94. Inicio del juego

Si se decide tomar la bifurcación, se llegará a una casa en la hay un conjunto de armas que serán de ayuda para lo que viene a continuación. Ya que es un prototipo, se han puesto todas las armas que hay disponibles en el juego, con el objetivo de que puedan ser probadas. Cuando el proyecto se encuentre en una fase más avanzada, se desarrollarán otros modos para obtener las armas, como por ejemplo, al derrotar a un jefe o a un conjunto de enemigos. Aunque sólo se hable de armas, el sistema es totalmente flexible para implementar armaduras, cascos u otros items, pero debido a que no han sido desarrollados, sólo se hablará de armas.

Ilustración 95. Bifurcación entrada del pueblo

Ilustración 96. Set de armas del juego

Cuando se llega a la entrada del pueblo, da comienzo una cinemática en la que se puede observar como los enemigos comienzan a llegar a través del portal para destruir el pilar maestro. El objetivo será evitar que lo destruyan, luchando contra ellos, aunque no será el único que participe en el ataque. En ésta ofensiva, también participarán los habitantes del pueblo, aunque son extremadamente débiles y no

poseen armas. Con su ayuda, tan sólo podrán frenar el ataque enemigo para hacer que la batalla sea más fácil. Aún así, si el enemigo consigue eliminar a todos los habitantes, el único que podrá evitar la destrucción del pueblo será el personaje principal.

Ilustración 97. Pueblo

En caso de que destruyan el pilar, el pueblo será arrasado por los enemigos y la misión habrá fracasado.

Ilustración 98. Pueblo destruido

7.3. Misión 2: El contraataque

Esta parte, da comienzo cuando el jugador logre completar la misión 1, justo cuando cruce por el portal.

En ésta misión, el personaje principal se adentrará en el reino enemigo para derrotar a los causantes de la apertura de los portales, teniendo que luchar antes contra el ejército que los protege.

El desarrollo de ésta fase, queda fuera del ámbito de éste proyecto.

8. Evaluación del prototipo

Para evaluar el prototipo antes de lanzarlo como versión 1, se ha realizado un test con el objetivo de corregir bugs y/u otras anomalías que se hayan podido dar durante la experiencia de juego. Para ello se ha proporcionado la siguiente plantilla con el objetivo de facilitar a los testers su labor.

Cosas que hay en el juego y me gustaría cambiar	Cosas que no están y me gustaría que estuvieran	Bugs e incidencias

Tabla 23. Plantilla tester

8.1. Tester 1

Cosas que hay en el juego y me gustaría cambiar	Cosas que no están y me gustaría que estuvieran	Bugs e incidencias
• Poder moverse mientras atacas • Poder mover la cámara hacia más direcciones • Al finalizar la misión, poner algo que indique que la has completado con éxito • Los ataques puedan ser más rápidos	• Personaje pudiera saltar • Al coger una espada hiciera algún sonido • Poner un pause • Efecto sonoro al atacar a algún monstruo o al recibir algún ataque • Master pillar se pudiera ver la vida que le queda • Asignar botones a las magias	• Personaje tarda en caer • Si no pasas justo en medio del pueblo no se activa la misión • NPC los puedes atravesar

Tabla 24. Resultados tester 1

8.2. Tester 2

Cosas que hay en el juego y me gustaría cambiar	Cosas que no están y me gustaría que estuvieran	Bugs e incidencias
• Modelos de los personajes más realistas • Interactuar con los NPC	• Poder recuperar HP y SP • Minimapa	• NPC andan pero no se desplazan

Tabla 25. Resultados tester 2

8.3. Tester 3

Cosas que hay en el juego y me gustaría cambiar	Cosas que no están y me gustaría que estuvieran	Bugs e incidencias
• Los enemigos al morir puedan dar armas, vida o algún item de ayuda	• Crear un sistema monetario para poder comprar armas • Tener en el inventario armaduras y más items	• Algunas paredes de las casas se pueden atravesar

Tabla 26. Resultados tester 3

8.4. Corrección de bugs

Cuando los **NPC** se quedan con la animación de andar **sin llegar a moverse**, se debe a que, al NavMeshAgent, al indicarle que detenga el movimiento del agente, elimina la ruta que tiene asociada. Esto produce que al reiniciar la marcha, no tiene una ruta establecida y el agente activa la animación de andar, pero sin llegar a moverse. Como **solución**, se ha usado un **ResetPath** para asignar otra ruta cada vez que el movimiento de cualquier agente se detenga. (Unity, 2017j)

Para evitar que se puedan evitar **atravesar objetos**, se han establecido triggers que permiten detectar colisiones entre objetos. Tan sólo se han puesto en los objetos más relevantes, dejando tales detalles para versiones más avanzadas. (Unity, 2017k)

Para evitar que el **personaje** parezca que esté "**levitando**", se ha aumentado la gravedad.

Para activar la primera misión, se ha puesto un collider a la entrada del pueblo, de forma que, si tomas otra ruta para entrar, no detecta la presencia del personaje y la misión 1 no se activa. Para solucionarlo, se ha puesto el collider en la zona central del pueblo.

8.5. Configuraciones

Todas las pruebas realizadas han utilizado la configuración que se expone a continuación:

- **Antivirus desactivado** para evitar borrados de archivos del juego.
- **Ejecución** usando el **procesador gráfico,** posteriores al año 2010 y con una capacidad superior a 256MB.
- **Resulución** de pantalla a **1920x1080**.
- **Calidad** gráfica comprendida entre "**High**" y "**Extreme**".
- **Sistema operativo Windows 7** o **Windows 10** con arquitectura x86_64.
- Memoria **RAM** superior a **4GB**

9. Aspectos sobre el rendimiento

9.1. Recoger armas

Cuando se genera un arma para que el personaje pueda recogerla, hay que planificar una serie de acontecimientos previos para que, en caso de recogerla, el videojuego no sufra bajadas en el rendimiento.

Para ello, cada vez que el personaje recoge un arma, tan sólo se ejecuta la operación de añadir al equipo. Dicha operación, añade la referencia del arma, que previamente ya ha sido cargada. Cuando se genera un objeto de tipo arma, hay que cargar el modelo 3D, obtener la localización donde va a ser renderizada (por ejemplo el brazo del personaje), cargar los stats del arma…etc y finalmente generar el GameObject para que pueda ser utilizado por Unity. Todos estos pasos se realizan **antes** de que el arma sea recogida.

Si todos éstos acontecimientos no se precargaran, habría que ejecutarlos dentro de la misma función en la que se recogen los eventos producidos por los Inputs. Se recuerda, que la función en la que se recogen esos eventos, es una función de tipo Update. Éste tipo de funciones, se les llama en cada frame, por lo que interesa tener la mínima cantidad de código en ellas.

Realizando la carga completa del arma antes de que sea recogida, favorece el rendimiento, ya que tan sólo hay que ejecutar una sola linea de código. Cabe destacar, que la estructura de datos utilizada, garantiza un orden de eficiencia de O(1) para insercciones.

```
// Update is called once per frame
void Update () {

    if (Input.GetButton ("PickUp")) {

        if (pickUp) {

            //Se anade al equipo
            equipment.addWeapon (weaponReference);

        }
    }
```

Ilustración 99. Ejemplo código función recoger arma

En caso de que no sea recogida, Unity dispone de un recolector de basura que identifica los objetos creados que no van a ser utilizados y aprovecha los momentos de menor carga computacional, para realizar los borrados realizando una correcta liberación de memoria. (Unity, 2017l)

9.2. Inteligencia artificial

En éste apartado, se van a explicar porqué se han implementado cierto tipo de elementos, de una manera y no de otra. Se van a detallar los recursos utilizados, con el objetivo de demostrar, que el método utilizado, permite obtener un mayor rendimiento.

9.2.1. Actualización de la máquina de estados

La máquina de estados, comienza a funcionar, justo en el momento en el que se inicializa el agente. La acción final que debe realizar el agente, se genera a partir de unos parámetros de entrada, como por ejemplo, posiciones de otros agentes u objetos, estados en los que se encuentra el juego… etc. Todo ello conduce a que la acción que vaya a hacer el agente, se producirá cuando la máquina de estados disponga de esos parámetros. Llegados a éste punto, hay que decidir, de qué forma se podrían actualizar.

La **primera forma**, es utilizar la función **Update**(), ya explicada anteriormente. Cada vez que se llame a la función Update, se podrían obtener los parámetros de entrada y hacer que la máquina de estados calcule el siguiente estado en el que se podría encontrar el agente. Hacer uso de éste método, sería **extremadamente ineficiente**, ya que se tendría que calcular el estado de cada agente en cada frame. Hay que destacar, que un videojuego cuya velocidad sea de 60 FPS (Frames Per Second), se dispondría de un margen de tiempo de unos 0.016 segundos por frame. En ése tiempo, habría que calcular rendering, animaciones, físicas… etc (Unity, 2017). Esto lleva a la conclusión, de que, si hacemos uso del Update para actualizar la máquina de estados, se necesitará más tiempo para cada frame, lo que producirá una bajada de FPS. (Unity, 2015)

La **otra forma** es evitar usar la función Update. Se ha pensado que, no se llegan a producir cambios significativos en el comportamiento de los agentes, en intervalos de tiempo inferiores a 1 segundo. Aprovechando ésta idea, se tendría que usar algún mecanismo, que permita realizar llamadas cada cierto tiempo. Unity dispone de un sistema de **corroutinas** (Unity, s.f.), que permite realizar llamadas a funciones en intervalos de tiempo. Para éste caso, se ha utilizado un tipo de corroutina, llamada InvokeRepeating, que permite actualizar la máquina de éstados en el intervalo de tiempo que se le haya indicado. Un tiempo muy reducido, simularía una función Update, mientras que un tiempo muy amplio, produciría comportamientos tardíos en el agente, por lo que habría que establecer un tiempo óptimo, como por ejemplo 1 segundo.

Aplicando la última forma, se reduce la carga de cómputo por cada frame, lo que permite usar múltiples IAs, sin tener que verse afectados de manera considerable los FPS.

9.2.2. Cálculo de distancias

Normalmente, para saber si un agente puede ver a otro, se utiliza el recurso raycast. El funcionamiento consiste en lanzar un rayo desde una posición, y obtener una lista de los elementos con los que intersecta el rayo. Si en esa lista está el objeto que se busca, entoces se podrá afirmar que el agente puede ver al objeto deseado y la máquina de estados actuará de una determinada forma en función de la distancia a la que se encuentren.

Lo ideal, para tener una máxima eficiencia, sería utilizar Raycast a partir de una distancia lo más corta posible. Esto se debe a que, cada vez que se lanza un rayo, se necesitan realizar operaciones, tales como intersecciones y disponer de estructuras de datos que permitan almacenar los objetos y las distancias con las que se ha producido la colisión.

2 agentes que estén muy lejanos, se lanzarían una cantidad de rayos innecesaria, ya que no empiezan a actuar hasta que se supera el rango de visión.

Para evitarlo, se ha implementado un sistema que distingue 2 fases, uso de resta de vectores y raycasting.

Si la distancia es superior al rango de visión, se utiliza una **resta de vectores**, de forma que el vector origen y destino serían las posiciónes de los agentes. El módulo del vector resultante sería la distancia.

Cuando dicha distancia es menor que el rango de visión, se usa **raycast** para ver con más detalle si realmente un agente puede ver a otro. Puede darse el caso de que se encuentren en el rango de visión, pero puede haber objetos por medio, que impiden que puedan verse, por eso, en éste caso, conviene usar el raycast.

Aplicando los métodos anteriormente comentados, se reduce el uso del raycast, lo que produce una mejora en el rendimiento para el cálculo de distancias y colisiones.

9.3. Planificador de escenas

A lo largo del juego, se suceden diversas escenas, cuya totalidad de componentes deben estar cargados cuando vayan a ser ejecutadas. Esto produce, que el paso de una escena a otra, genera una bajada de FPS muy considerable. Durante una partida, dicha bajada es tan acusada, que el juego se detiene, dando la sensación de haberse colapsado.

Unity dispone de un mecanismo, que permite realizar cargas de escenas de manera asíncrona (LoadSceneAsync), lo que permite tener cargadas varias escenas. Cuando la carga esté completa, se llama al método que contiene la referencia de la escena y se le indicaría que se visualizara. (Alan Zucconi, 2016)

Aprovechando lo anterior, se ha implementado un método de carga de escenas encadenada. El método consiste en que cuando se cargue una escena, al mismo tiempo se va a cargar la siguiente y así sucesivamente. Esto va a producir que la primera escena tenga un índice de carga muy elevado. Para enmascararlo, se ha creado una escena auxiliar, en la que se visualiza una imagen que indica que se están cargando la escenas.

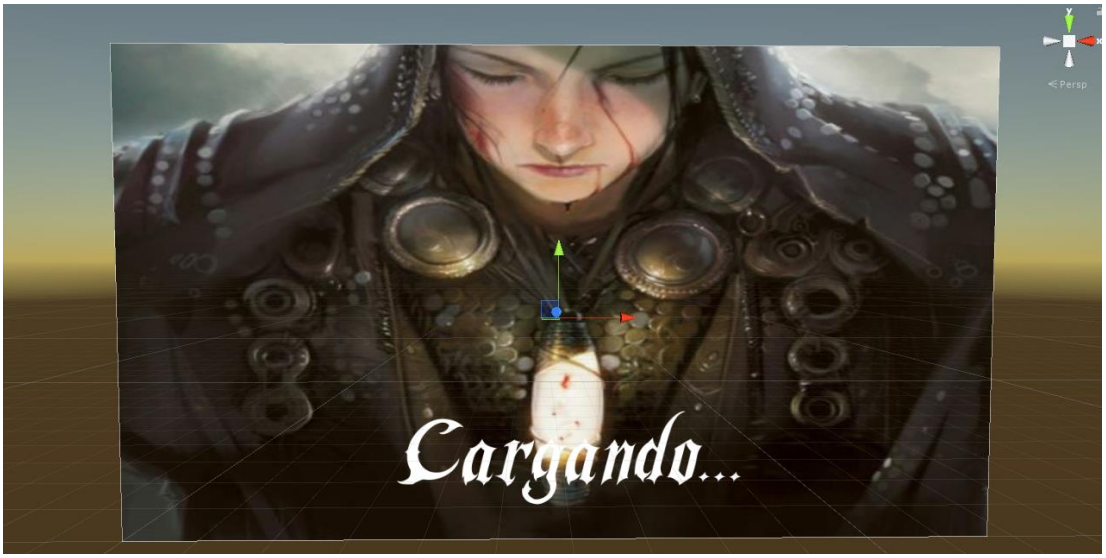


Ilustración 100. Escena de cargar

Una vez estén cargadas, pasar de una a otra, va a ser mucho más rápido. Éste método, hace que se consuma más memoria, pero por otro lado, se obtienen mejores tiempos de respuesta, que finalmente, es lo que interesa.

9.4. Configuraciones de Unity

A continuación, se van a exponer algunas de las configuraciones realizadas en Unity para obtener un mejor rendimiento.

9.4.1. Matriz de colisiones

Para el cálculo de colisiones, Unity hace uso de una matriz que relaciona los objetos que pueden colisionar y los que no (Unity, 2017m). Establecer una configuración que reduzca el número de colisiones, permite que a la hora de ejecutarse la física, se realicen muchas menos comprobaciones lo que se traduce en tener una mayor eficiencia.

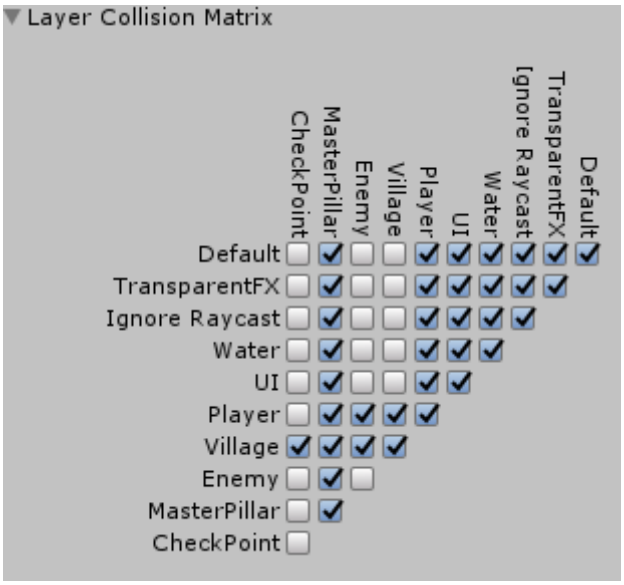


Ilustración 101. Matriz de colisiones de Unity

9.4.2. Static

En cada escena, se deben calcular las colisiones que ha habido. Dicho proceso consta de dos fases, la fase ancha y la fase estrecha. En la **fase ancha**, se examina toda la geometría de la escena con el objetivo de descartar los elementos no involucrados en la colisión. En la **fase estrecha**, se calculan de manera más detallada la colisiones que ha habido entre los objetos.

El algoritmo de fuerza bruta de la fase ancha, tiene un tiempo de eficiencia O(n^2+nm), siendo n los objetos móviles y m los estáticos. Se deduce que es costoso, por lo que lo ideal es que el “m” sea mayor que el “n”. (Berg, 2008)

La configuración Static de Unity, permite establecer que objetos son estáticos y cuales no, lo cual, es muy importante indicarlo si se quiere tener un buen rendimiento.

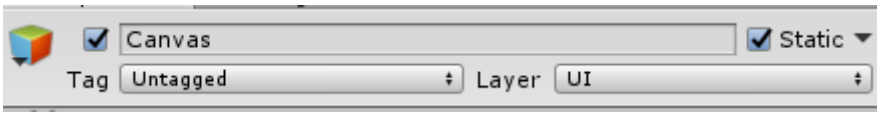


Ilustración 102. Configurador GameObjects

El método Static, no sólo se centra en el aspecto de las colisiones, también es usado en la parte de **render**. Los objetos marcados como estáticos, se precalculan antes de ejecutar la escena, lo que resta el número de objetos que se tienen que renderizar. (Unity, 2017n)

Hay muchas más consecuecias de marcar objetos como Static (iluminación, sombreado…), pero generalmente van asociadas a mejoras en el rendimiento.

10. Manual de juego

El juego ha sido principalmente diseñado para ser usado con el ordenador, a través de periféricos tales como el ratón y el teclado. A continuación se van a exponer los controles por defecto.

Ilustración 103. Ratón

Botón 2: Permite hacer zoom sobre el personaje

Botón 3: Mientras se pulsa, permite mover la cámara

Ilustración 104. Teclado

Teclas A,S,D,W: Permiten mover al personaje

Barra Espaciadora: Permite atacar

Enter: Se utiliza para recoger las armas

Teclas Q, E: Permiten ver desde otra perspectiva al personaje

Tecla T: Cubrirse

11. Conclusiones

El desarrollo de éste proyecto, ha permitido la mejora y la obtención de un conjunto de habilidades que anteriormente no se poseían, o al menos, no tan desarrolladas, como a la finalización del mismo.

Los proyectos que se habían realizado con anterioridad, estaban basados en un guiones y estructurados por entregas limitadas en tiempo. El objetivo era cumplir con los plazos propuestos y entregar un software funcional, que cumpliera con los requisitos propuestos.

A la hora de enfrentarse a un proyecto, en el que tan sólo se exponen unos objetivos muy generales, dejando cierta libertad al desarrollador para encauzar el proyecto en una dirección u otra, obliga, en cierto modo, a obtener un grado de madurez mayor.

El enfrentarse a un proyecto de ésta índole, la mentalidad, tiene que cambiar del “qué hay que hacer” al “cómo se va a hacer”. Ese cambio, incluye tener que pensar una organización y una planificación previa, antes de empezar a implementar cualquier elemento. Evidentemente, antes de planificar, hay que saber de antemano, los objetivos a los que se quiere llegar, contemplando limitaciones como recursos y tiempo.

Dentro de esa planificación, también hay que tener en cuenta aspectos tales como: la metodología que se va a seguir, los tipos de diagramas que se van a utilizar, elección de las herramientas más adecuadas, patrones de diseño…etc. La utilización de éstos elementos, proporcionados mayoritariamente por la rama de ingeniería del software, han demostrado un alto índice de proyectos finalizados con éxito.

Una vez está toda la parte de organización y planificación completadas, es cuando se procede a desarrollar el proyecto en sí. Ese cambio de mentalidad, el de ver, que dedicar cierto tiempo a organizar y planificar, va ofrecer muchas más ventajas, frente a otros proyectos que no lo hayan hecho, ha sido unos de los aspectos más importantes que se han adquirido durante el desarrollo el proyecto.

Ilustración 105. Planifique anticipada-mente

La creación de videojuegos, no es tarea fácil, ya que para obtener un buen producto que guste a los jugadores, hace falta un equipo multidisciplinar. Eso no quiere decir que una sola persona, no sea capaz de hacerlo, como bien se ha demostrado en éste proyecto, aunque tiene como consecuencia, la dedicación de muchas horas.

También hay que tener en cuenta, que este proyecto, es un prototipo y no una versión final lista para ser comercializada. Los resultados obtenidos, ofrecen una arquitectura, que en un futuro, podrá servir de base para seguir ampliándolo y mejorándolo, hasta llegar a ser un producto final.

Como conclusión, no es aconsejable que una sola persona se aventure a la realización de un videojuego, sobre todo por la dificultad que conlleva abarcar todas

las áreas por la que está formado, como por ejemplo, la parte de programación, artística, musical, marketing…etc. En definitiva, desarrollar un videojuego, es un trabajo de equipo.

Ilustración 106. Perfil de hombre orquesta

Sabiendo que éste tipo de proyectos suele extenderse a años de trabajo, se han obtenido unos muy buenos resultados frente a la adversidad que ha supuesto el tiempo. Aún así, gracias a la gran dedicación que se le ha otorgado, se han conseguido finalizar todos los objetivos propuestos.

Inevitablemente, se dejan muchas cosas por el camino, aunque, no se descarta que algún día, se llegue a conseguir una versión comerciable que pueda competir en el mercado de los videojuegos.

Índice de tablas

Índice de ilustraciones

Bibliografía

Alan Zucconi. (2015, 6 17). *Shaders in Unity3D*. Retrieved from http://www.alanzucconi.com/2015/06/17/surface-shaders-in-unity3d/

Alan Zucconi. (2016). *Scene management in Unity*. Retrieved from http://www.alanzucconi.com/2016/03/23/scene-management-unity-5/#part5

Asociación de estudiantes de videojuegos. (n.d.). *Creación de puntos de control*. Retrieved from http://aev.org.es/unity-creacion-de-puntos-de-control-checkpoints/#comment-23798

Asociación de estudiantes de videojuegos. (n.d.). *Desarrollo de una máquina de estados*. Retrieved from http://aev.org.es/unity-desarrollo-de-una-sencilla-maquina-de-estados/

Audacity. (2017). *Audacity*. Retrieved from http://www.audacityteam.org/

Audacity. (n.d.). Tutorial de edición de sonido. www.jesusda.com/docs/ebooks/ebook_tutorial-edicion-de-sonido-con-audacity.pdf.

Berg, M. (2008). *Computational geometry: Algorithms and applications.* Berlin Heidelberg.

Blender. (2017). *Manual usuario*. Retrieved from https://wiki.blender.org/index.php/Doc:ES/2.6/Manual

Blender. (2017). *Rigging and animation with blender*. Retrieved from https://sites.google.com/site/machinimist/

C#. (n.d.). *Tutorial C#*. Retrieved from https://msdn.microsoft.com/es-es/library/aa288436(v=vs.71).aspx

Devueco. (2017). *Roles en la creación de videojuegos*. Retrieved from http://www.devuego.es/blog/2014/12/18/roles-en-la-creacion-de-videojuegos-i-el-diseno/

E. Gamma, R. H. (2003). *Patrones de diseño.* Addison Wesley.

eumed. (n.d.). *Modelo incremental*. Retrieved from http://www.eumed.net/tesis-doctorales/2014/jlcv/software.htm

Film Storm. (n.d.). *Youtube*. Retrieved from https://www.youtube.com/channel/UCH1svLGqmyVuCnHCHDr0EXw

freelancer. (n.d.). *¿Qué metodología utilizar?* Retrieved from https://www.freelancer.es/community/articles/proceso-del-desarrollo-software

Gregory, J. (2009). *Game Engine Arquitecture.* A.K.Peters, Ltd.

Hagamos Videojuegos. (n.d.). *Youtube*. Retrieved from https://www.youtube.com/channel/UCBhkLrsmV9PVQMpT3qe-toA

Instituto de gestión de proyectos. (2000). *Guía para la gestión de proyectos.* Newton Square, Pennsylvania USA: Aserpro.

Kripto. (n.d.). *Youtube*. Retrieved from https://www.youtube.com/channel/UCmL_uEJWUN0eyCsh4kEgSUA

Larman, C. (2003). *UML y patrones.* Prentice-Hall.

Matt Smith, C. Q. (n.d.). *Unity 5.x Cookboock.* Packt.

Microsoft. (2013). *Keyboard ghosting demonstration*. Retrieved from https://www.microsoft.com/appliedsciences/content/projects/KeyboardGhostingDemo.aspx

Microsoft. (2017). *Visual Studio* . Retrieved from https://www.visualstudio.com/es/?rr=https%3A%2F%2Fwww.google.es%2F

Microsoft. (n.d.). *C# Data structures*. Retrieved from https://msdn.microsoft.com/en-us/library/ms379570(v=vs.80).aspx

Microsoft. (n.d.). *Dictionary documentation*. Retrieved from https://msdn.microsoft.com/en-us/library/xfhwa508(v=vs.110).aspx

MonoDevelop. (2017). *MonoDevelop*. Retrieved from http://www.monodevelop.com/

Mundo Informático. Patrones de diseño. (n.d.). Retrieved from https://infow.wordpress.com/category/patrones-de-disenogof/

Nodician. (n.d.). *Youtube*. Retrieved from https://www.youtube.com/channel/UC393VScZGe721UVQf3u1NZw

OkHosting. (2016). *Metodologias desarrollo software*. Retrieved from https://okhosting.com/blog/metodologias-del-desarrollo-de-software/

Pixelsandcoffee. (n.d.). *Unity shader introduction*. Retrieved from http://www.pixelsandcoffee.es/unity-shaders-1-introduccion/

Sebastian League. (2016). *Youtube*. Retrieved from https://www.youtube.com/user/Cercopithecan/videos

Sergio Osma. (n.d.). *Youtube*. Retrieved from https://www.youtube.com/channel/UCRIA6rBETl1GCD8xiDPh5hw

StackOverFlow. (n.d.). *A star vs Dijkstra*. Retrieved from https://stackoverflow.com/questions/13031462/difference-and-advantages-between-dijkstra-a-star

Stuart Spence. (n.d.). *Youtube*. Retrieved from https://www.youtube.com/channel/UCb0DBpjN2lZhJCflvMUCJpw

Two Steeps From Hell. (2015). *Youtube*. Retrieved from https://www.youtube.com/channel/UC3swwxiALG5c0Tvom83tPGg

Unity. (2015, Diciembre 23). *Blogs 1k update calls*. Retrieved from https://blogs.unity3d.com/es/2015/12/23/1k-update-calls/

Unity. (2015, Diciembre 23). *Unity Blogs*. Retrieved from https://blogs.unity3d.com/es/2015/12/23/1k-update-calls/

Unity. (2017). *Execution order of event functions.* Retrieved from https://docs.unity3d.com/Manual/ExecutionOrder.html

Unity. (2017). *Pipeline execution order*. Retrieved from https://docs.unity3d.com/Manual/ExecutionOrder.html

Unity. (2017a). *Scripting APIs*. Retrieved from https://docs.unity3d.com/ScriptReference/

Unity. (2017b). *Scripting*. Retrieved from https://unity3d.com/es/learn/tutorials/s/scripting

Unity. (2017c). *Mesh Renderer Manual*. Retrieved from https://docs.unity3d.com/Manual/class-MeshRenderer.html

Unity. (2017d). *Animaciones Generic en Mecanim*. Retrieved from https://docs.unity3d.com/es/current/Manual/GenericAnimations.html

Unity. (2017e). *Input Manager Manual*. Retrieved from https://docs.unity3d.com/Manual/class-InputManager.html

Unity. (2017f). *Building a NavMesh*. Retrieved from https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html

Unity. (2017g). *Creating a NavMesh Agent*. Retrieved from https://docs.unity3d.com/Manual/nav-CreateNavMeshAgent.html

Unity. (2017h). *Writting the game manager*. Retrieved from https://unity3d.com/es/learn/tutorials/projects/2d-roguelike-tutorial/writing-game-manager

Unity. (2017i). *Shader overview.* Retrieved from https://docs.unity3d.com/es/current/Manual/ShadersOverview.html

Unity. (2017j). *Nav Mixing components*. Retrieved from https://docs.unity3d.com/Manual/nav-MixingComponents.html

Unity. (2017k). *Colliders as triggers*. Retrieved from https://unity3d.com/es/learn/tutorials/topics/physics/colliders-triggers

Unity. (2017l). *Gestión automática de memoria*. Retrieved from https://docs.unity3d.com/es/current/Manual/UnderstandingAutomaticMemoryManagement.html

Unity. (2017m). *Collider overview*. Retrieved from https://docs.unity3d.com/Manual/CollidersOverview.html

Unity. (2017n). *StaticObjects*. Retrieved from https://docs.unity3d.com/Manual/StaticObjects.html

Unity Asset Store. (n.d.). *Double axe*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/4039

Unity Asset Store. (n.d.). *Fantasy ambient pack*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/49863

Unity Asset Store. (n.d.). *Fantasy RPG Icon set*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/43810

Unity Asset Store. (n.d.). *Fantasy weapon pack*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/11325

Unity Asset Store. (n.d.). *Free shader fx mask*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/71311

Unity Asset Store. (n.d.). *Fx explosion pack*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/30102

Unity Asset Store. (n.d.). *Fx magick*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/23978

Unity Asset Store. (n.d.). *Golem*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/33260

Unity Asset Store. (n.d.). *Lighting shader*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/3003

Unity Asset Store. (n.d.). *Magick weapon icons*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/15419

Unity Asset Store. (n.d.). *Monster*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/48933

Unity Asset Store. (n.d.). *Nightmare creature*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/44477

Unity Asset Store. (n.d.). *Orc Sword*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/61472

Unity Asset Store. (n.d.). *Ruins creation kit*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/2235

Unity Asset Store. (n.d.). *Town creator kit*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/25574

Unity Asset Store. (n.d.). *VFX*. Retrieved from https://www.assetstore.unity3d.com/en/#!/list/7122-vfx

Unity. (n.d.). *Awake and start*. Retrieved from https://unity3d.com/es/learn/tutorials/topics/scripting/awake-and-start?playlist=17117

Unity. (n.d.). *Best practices*. Retrieved from https://unity3d.com/es/learn/tutorials/topics/best-practices

Unity. (n.d.). *Best way to animate* . Retrieved from http://answers.unity3d.com/questions/234395/rpg-armors-and-clothes-best-way-to-animate-each-pa.html

Unity. (n.d.). *Coroutines*. Retrieved from https://docs.unity3d.com/Manual/Coroutines.html

Unity. (n.d.). *Destroy*. Retrieved from https://unity3d.com/es/learn/tutorials/topics/scripting/destroy?playlist=17117

Unity. (n.d.). *GetComponent*. Retrieved from https://unity3d.com/es/learn/tutorials/topics/scripting/getcomponent?playlist=17117

Unity. (n.d.). *GPU instancing*. Retrieved from https://docs.unity3d.com/Manual/GPUInstancing.html

Unity. (n.d.). *Graphics performance*. Retrieved from https://docs.unity3d.com/es/current/Manual/OptimizingGraphicsPerformance.html

Unity. (n.d.). *Instantiate*. Retrieved from https://unity3d.com/es/learn/tutorials/topics/scripting/instantiate?playlist=17117

Unity Institute. (n.d.). *Youtube*. Retrieved from https://www.youtube.com/channel/UCxDZ3d_9hF55liPZkzpBgMg

Unity. (n.d.). *Invoke*. Retrieved from https://unity3d.com/es/learn/tutorials/topics/scripting/invoke?playlist=17117

Unity. (n.d.). *Issues scenes manager*. Retrieved from https://issuetracker.unity3d.com/issues/loadsceneasync-allowsceneactivation-flag-is-ignored-in-awake

Unity. (n.d.). *Lighting*. Retrieved from https://docs.unity3d.com/Manual/LightingOverview.html

Unity. (n.d.). *Multi scene editing*. Retrieved from https://docs.unity3d.com/es/current/Manual/MultiSceneEditing.html

Unity. (n.d.). *NavMesh with other components*. Retrieved from https://docs.unity3d.com/Manual/nav-MixingComponents.html

Unity. (n.d.). *Optimization graphics*. Retrieved from https://unity3d.com/es/learn/tutorials/temas/performance-optimization/optimizing-graphics-rendering-unity-games?playlist=44069

Unity. (n.d.). *Rigibody*. Retrieved from https://docs.unity3d.com/es/current/Manual/class-Rigidbody.html

Unity. (n.d.). *Scripts and behaviour components*. Retrieved from https://unity3d.com/es/learn/tutorials/modules/beginner/scripting/scripts-as-behaviour-components?playlist=17117

Unity. (n.d.). *Surface shader example.* Retrieved from https://docs.unity3d.com/Manual/SL-SurfaceShaderExamples.html

Unity. (n.d.). *Swappable armor RPG game*. Retrieved from https://forum.unity3d.com/threads/making-swappable-armor-clothing-for-rpg-game.125995/

Unity. (n.d.). *Time Frames*. Retrieved from https://docs.unity3d.com/es/current/Manual/TimeFrameManagement.html

Unity. (n.d.). *Understanding memory usage*. Retrieved from https://docs.unity3d.com/es/current/Manual/UnderstandingAutomaticMemoryManagement.html

Unity. (n.d.). *Vector maths*. Retrieved from https://unity3d.com/es/learn/tutorials/topics/scripting/vector-maths?playlist=17117

UnrealEngine. (2017). *Unreal Engine Documentation*. Retrieved from https://docs.unrealengine.com/latest/INT/Programming/Introduction/index.html

Visual Paradigm. (2017). *Visual Paradigm*. Retrieved from https://www.visual-paradigm.com/

Wikijuegos. (2014). *Wikijuegos*. Retrieved from http://es.videojuegos.wikia.com/wiki/G%C3%A9neros_de_videojuegos