Trabajo de Fin de Grado

GRADO DE INGENIERÍA INFORMÁTICA

Facultad de Matemáticas
Universidad de Barcelona

DISEÑO, DESARROLLO, MODELADO Y ANIMACIÓN DE UN JUEGO CON UNITY3D

Fernando Mateus Romero

Director: Àlex Pardo Fernández
Realizado en: Departament de Matemàtica Aplicada
i Anàlisi. UB

Barcelona, 27 de junio de 2015

Abstract

The video game industry is passing through a golden age which has been a long time coming. The sector is approaching slowly to a maturity that can only be reached when some barriers are broken after a long Trial and Error process, resulting in some approaching to the level of older industries with many more years at their backs, such as the cinematographic or musical industry.

There have been some facts that are directly responsible of that evolution. One is the constantly growing acceptation of video games in the homes all around the world, and other one is the fresh air that indie teams have bring to the development industry, being capable to take the risk of offer new experiences without be too much concerned of the economical side of the business nor the popular criticism. In that last one sense, the wide offer of free software development tools available to everyone has a lot to say.

Unity3D is the graphic engine more used by far in video game development at present, partly on being multi-platform (is able to build applications compatible with Android, iOS, PC, Mac, Linux and consoles) and its free licenses, that are not even limited in comparison with the purchasable licenses. Its use has grown widely in the last three years and statistics don't seem to say that the trend is going to change soon. Moreover, a lot of studios (and not only the independents ones) are putting their money in using Unity3D as their next game graphic engine.

With Unity3D Personal (the free license) as graphic engine and Blender as modeller and animator software, this project will start the development of a action-role PC game, creating some basic mechanics thinking in a future full development.

Resumen

La industria del videojuego está viviendo una edad dorada que ha tardado muchos años en alcanzar. El sector se está acercando lentamente a una madurez a la que sólo se puede llegar cuando se rompen ciertas barreras tras un proceso muy largo de prueba y error, acercándose al nivel de otras industrias con muchos más años a sus espaldas como la cinematográfica o la musical.

En dicha evolución han tenido mucho que ver por un lado la creciente aceptación del videojuego en los hogares de todo el mundo y por otro el soplo del aire fresco que brinda el desarrollo de videojuegos por parte de equipos independientes, que han sido capaces de arriesgar más en pos de ofrecer experiencias diferentes sin pensar demasiado ni en el dinero ni en la aceptación popular de sus creaciones. De esto último es responsable en parte las múltiples herramientas gratuitas de desarrollo que están al alcance de cualquiera.

Unity3D es el motor gráfico más utilizado en la actualidad en el desarrollo de videojuegos, en parte por ser multiplataforma (genera aplicaciones compatibles con Android, iOS, PC, Mac, Linux y consolas) y ofrecer licencias gratuitas que en absoluto se quedan limitadas respecto a las de pago. Su uso ha crecido a un ritmo vertiginoso durante los últimos tres años y los números no parecen indicar que la tendencia vaya a cambiar próximamente. De hecho, muchos estudios (no sólo independientes) están apostando por trabajar en sus nuevos proyectos con él.

Usando Unity3D Personal (versión gratuita) como motor gráfico y Blender para desarrollar los modelos en 3D y animarlos, el objetivo de este proyecto es iniciar el desarrollo de un juego de acción y rol para PC, creando algunas mecánicas básicas con vistas a un desarrollo completo posterior.

Índice de contenido

1. Introducción....2
    1.1 Motivación....2
    1.2 Objetivos....3
    1.3 State of the art....3
2. Tecnologías utilizadas....4
    2.1 Unity3D Personal....4
    2.2 C Sharp (C#)....5
    2.3 Blender....5
3. Planificación....6
4. Diseño inicial....8
    4.1 Jugabilidad....8
    4.2 Personaje....8
    4.3 Equipo....11
    4.4 Objetos....13
    4.5 Mundo....14
    4.6 Escenarios....15
5. Primeros pasos en Unity3D....17
6. Desarrollo 20
    6.1 Modelado....20
    6.2 Animación....21
    6.3 Importando a Unity3D....22
    6.4 Prefabs: pros y contras....23
    6.5 Montando y coordinando animaciones....23
    6.6 Construyendo el escenario....24
    6.7 Base de datos de objetos y enemigos....25
    6.8 El combate....26
    6.9 Gestionando las partidas guardadas....26
7. Resultados....28
8. Conclusiones y trabajo futuro....30
9. Bibliografía....31
10. Anexos....32
    ANEXO A: Manual de usuario....32
    ANEXO B: Galería de imágenes....36

1. Introducción

La ciencia y las tecnologías avanzan lentamente en pos de conseguir crear un mundo mejor en el que vivir... o al menos esa es la teoría en la que a todos nos gusta creer. Y las Tecnologías de la Información y todas sus formas de ser representadas no son una excepción, precisamente. Una de las ramas más en auge de la informática durante los últimos veinte años, la que hoy nos ocupa, es la del desarrollo de videojuegos.

La industria del videojuego ha evolucionado poco a poco, introduciéndose en los hogares de todo el mundo y pasando de ser la afición de unos pocos adinerados algo mal vista por el público en general a una forma de entretenimiento más conocida y aceptada, llegando hasta el punto de poder dedicarse profesionalmente a jugar a ciertos juegos de forma competitiva para ganarse la vida. Algo impensable hace cuarenta años, sin duda, y que todavía levanta algunas ampollas entre los no poco detractores que siguen sin aceptar la madurez que se ha demostrado alcanzar en el sistema.

Lo que antes era una pantalla en blanco y negro de resolución bajísima que recibía una entrada binaria ahora se ha transformado en más de dos millones de píxeles refrescándose sesenta veces por segundo. Lo que antes era una partida en solitario ahora son encuentros masivos con docenas de jugadores conectados simultáneamente a través de la red desde diferentes partes del mundo. En resumidas cuentas, lo que antes era un simple pasatiempo ahora es considerado una forma de arte a la altura del cine y el cómic. Y todo tiene sus consecuencias.

La explosión de la industria del videojuego se puede intentar explicar desde muchos puntos de vista y de muchas formas diferentes, pero lo que está claro es que una de ellas ha sido el brutal asentamiento de los ordenadores e internet en hogares de todo el mundo, abriendo muchas puertas tanto a consumir como a producir. Y lo que antes eran limitaciones de *hardware* y *software* que obligaban a los desarrolladores a exprimir su ingenio hasta la última gota para aprovechar el potencial de las herramientas disponibles ahora se ha convertido en una barra libre en forma de equipos sobrados de recursos y herramientas de todos los tipos y colores que hacen posible que cualquiera pueda intentar hacer sus pinitos creando sus propios proyectos.

Edición de imágenes, edición de sonido, modelado en tres dimensiones, cursos exprés de programación, motores gráficos... todos los frentes desde un punto de vista relativamente básico están cubiertos de forma gratuita. La eficiencia de dichas vías ya dependerá de otros factores, pero es un hecho que, si alguien si propone desarrollar un videojuego sin tener conocimientos en prácticamente ningún conocimiento en ningún campo relacionado, puede. Ahora bien, la calidad del resultado obtenido ya es harina de otro costal.

1.1 Motivación

Durante toda mi vida he sido aficionado al mundo de los videojuegos, e incluso me he estado ganando la vida durante varios años escribiendo profesionalmente en algunos medios del sector. Teniendo en cuenta esto, siempre me había atraído la idea de saltar la barrera entre la afición y el trabajo (una vez más) con el objetivo de intentar crear mi propio juego.

La asignatura Ingeniería del Software de cuarto curso del grado fue un primer acercamiento a lo que sería el desarrollo de un juego, pero entre que el resultado final dejó bastante que desear y que la asignatura estaba más enfocada a las metodologías de trabajo en grupo, me quedé con ganas de más. Y el Trabajo de Fin de Grado ha sido una oportunidad de ponerme manos a la obra.

1.2 Objetivos

El objetivo de este trabajo no es otro que demostrar que se puede desarrollar un juego para ordenador totalmente desde cero. Hilando un poco más fino, se tenía entre manos un juego de rol y acción que mezclara la profundidad de ambos géneros en un híbrido más sencillo. Diseñar las mecánicas, crear los modelos y animarlos usando Blender, desarrollar la lógica del juego y ponerlo todo en común con el motor gráfico Unity3D.

Ahora bien, sería equivalente a no tener ojos en la cara el ignorar que sobre el papel es un proyecto bastante ambicioso, y más considerando que mi experiencia en la mayoría de campos tiende a cero, por lo que el proceso de aprendizaje en todos ellos no abre muchas puertas a que la etapa de desarrollo final esté a la altura de lo esperado, algo que quedará patente en los apartados de planificación que se encontrarán posteriormente.

Dicho esto, se ha sido consciente en todo momento de que sería un proyecto de máximos, intentando en la medida de lo posible crear una base modular abierta a crecer en un futuro.

1.3 State of the art

Tal y como bien se encarga de demostrar en su página oficial la gente Unity Technologies, Unity3D es, de largo, la herramienta dominante a la hora de desarrollar videojuegos en todo el mundo. 600 millones de usuarios disfrutan al día de algún producto desarrollado con Unity3D. Los desarrolladores registrados se han multiplicado por cinco en los últimos tres años. Es el motor gráfico más utilizado en el mercado de juegos para plataformas móviles en todo el mundo. Acapara el 45% del mercado de motores gráficos, triplicando prácticamente a su competidor más cercano. Un dominio absoluto.

Su explosión en el mercado durante los últimos años ha coincidido y colaborado con la explosión de la vertiente independiente de una industria que necesitaba con urgencia un soplo de aire fresco, ideas nuevas y asumir ciertos riesgos que las empresas más importantes no se atrevían. El tener al alcance al alcance de la comunidad independiente un motor potente, a priori sencillo y relativamente gratuito ha empujado a ambos a evolucionar en la misma dirección de forma dependiente.

En la actualidad encontramos un número creciente de juegos desarrollados con Unity, que pese a todo, hasta hace un par de años era su principal talón de Aquiles, y ya no sólo por desarrolladores independientes de escasos fondos, sino por compañías de la talla de Obsidian Enternaiment o Eidos. Entre los más destacados encontramos Pillars of Eternity, Kerbal Space Program, Endless Space, Ori and the Blind Forest y Wasteland 2.

2. Tecnologías utilizadas

Las tecnologías empleadas en el desarrollo del juego son Unity3D como motor gráfico, C Sharp como lenguaje de programación y Blender como plataforma para crear modelos en tres dimensiones y realizar animaciones.

2.1 Unity3D Personal

Unity3D es un motor gráfico creado por Unity Technologies en 2005, en un principio siendo exclusivo para las plataformas de Apple pero llegando a muchas otras a lo largo de los años y las versiones. En estos momentos (Unity 5 es su última versión estable) sus proyectos son compatibles con más de una quincena de plataformas, incluyendo consolas, móviles, versiones web y diferentes sistemas operativos en lo que respecta al PC.

La clave de la importancia que ha cobrado Unity3D en el mercado de los motores gráficos se entiende por su modelo de negocio. Existen dos versiones del motor: la versión Personal y la versión Profesional. Mientras que la primera es gratuita y muy funcional teniéndolo en cuenta, la segunda es de pago y ofrece bastantes más funcionalidades y facilidades a la hora de trabajar en proyectos multiplataforma.

Con Unity Personal cualquiera puede lanzarse a la aventura de desarrollar una aplicación (no todo son juegos, al fin y al cabo) e incluso intentar comercializarla sin tener que pasar por caja de buenas a primeras. Si se generan beneficios por encima de los 100.000 dólares anuales con el producto de turno, entonces sí que se está obligado a adquirir una licencia de Unity Profesional, cuyo coste es una cuota anual de 1.500 dólares bastante asumibles. El modelo de negocio adoptado es, sin duda, muy tentador para cualquier pequeño equipo de desarrolladores, dando la oportunidad a proyectos escasos de fondo de intentar generar beneficios y minimizar las pérdidas en el caso de que las cosas no salgan bien.

Esto ha provocado que durante los últimos años haya aumentado considerablemente la cantidad de empresas de todo tipo que apuestan por Unity3D como plataforma de desarrollo y, por ende, ha colaborado en el hecho de que no sobre información sobre el motor gráfico en diferentes medios, siendo una de las razones principales por las que se ha escogido para llevar a cabo este proyecto. El otro motivo consiste en que ya se había trabajado anteriormente en alguna ocasión muy por encima con él, por lo que se ha preferido aprovechar el tiempo en perfeccionar un dominio sobre una plataforma antes que empezar de cero en otra igualmente válida.

Durante los primeros pasos del Trabajo de Final de Grado estuvo en el aire la posibilidad de conseguir una licencia para trabajar con Unity3D Profesional, pero dados los objetivos iniciales se descartó la idea por ser realmente innecesaria y porque muy posiblemente no se llegaría a sacarle partido en ningún sentido.

2.2 C Sharp (C#)

Los desarrollos con Unity3D son muy visuales y permiten trabajar con profundidad tanto a diseñadores como a programadores en un entorno común, lo que tiene bastante mérito. Aun así, sin programar scripts que gestionen la actividad de los elementos no se va a ningún lado, y en el motor gráfico de Unity Technologies se puede escoger entre tres opciones: Unityscript, C Sharp y Boo.

- Tras el nombre de Unityscript se esconde realmente Javascript, un lenguaje todoterreno normalmente interpretado por navegadores y usado en el mundo del desarrollo web. El pequeño cambio de nombre se debe a que su uso en el contexto que es Unity3D difiere ligeramente del habitual, por lo que tanto la comunidad como la gente de Unity Technologies prefieren hacer la distinción entre ambas versiones.

- C Sharp (C#) es un lenguaje de programación orientado a objetos bajo la firma de Microsoft, siendo parte de la plataforma .NET.

- Boo es otro lenguaje de programación orientado a objetos inspirado en la sintaxis de Python cuyo uso y reconocimiento está muy por debajo de las otras alternativas.

Unityscript cuenta con la facilidad que otorga una programación de tipificación dinámica y amigable, pese a que ello es un arma de doble filo que puede empujar al desarrollador a cometer fallos en lo que respecta a la optimización de código. C Sharp cuenta con el apoyo de un gigante de la talla de Microsoft y fue el lenguaje con el cual vio la luz la primera versión de Unity3D, por lo que soporte y documentación brillan por su abundante presencia. Boo es la oveja negra de la familia y prácticamente nadie lo usa con estos fines, sin tener siquiera representación en la API de Unity.

Se ha decidido desarrollar el proyecto usando C Sharp al descartarse por sí mismo Boo y al contar con poca experiencia en lo que respecta a Javascript.

2.3 Blender

Existen muchísimos programas de modelado en tres dimensiones, tanto gratuitos como de pago. Y, entre todos ellos, no son pocos precisamente los que ofrecen algún tipo de compatibilidad con Unity, por lo que la tarea de bucear en el mercado en busca de la mejor opción se antojaba un tanto titánica.

Se buscaba una herramienta que permitiera modelar y otra que permitiera animarlos (pese a que con Unity3D puede hacerse algo parecido, también había cierto interés por no cerrarse en un sólo entorno y sus posibilidades), sin necesidad de que una misma se encargara de ambas cosas. Pero Blender, desarrollado por Blender Foundation, es una plataforma completísima que incluso cuenta con un motor gráfico interno.

Ante la situación, se ha decidido apostar por Blender para modelar y animar modelos en tres dimensiones al ser gratuito y ya contar con cierta experiencia a la hora de trabajar con él, además de por contar con una cantidad de documentación abismal tras una veintena de años en el mercado.

3. Planificación

La situación del proyecto invitaba a usar alguna metodología de trabajo que ayudara a coordinar, planificar y sacar adelante todos los frentes establecidos en un principio (recordemos: programación, modelado, animación y un diseño que lo englobara todo). El problema es que existe una importante dependencia entre todos y cada uno de esos apartados, lo que hace prácticamente imposible realizar un guión de trabajo eficiente a corto plazo que se regulara por la prioridad de cada uno de ellos.

Otro aspecto a tener en cuenta es que hacía falta un proceso de documentación y estudio bastante importante para al menos tres de los cuatro apartados mencionados. Sumado al hecho de que sólo una persona se iba a encargar del desarrollo y que su duración no iba a superar los cuatro meses (sin que fuera a tiempo completo), resultaba complicado encontrar una metodología de trabajo que diera buenos resultados.

En los primeros pasos del desarrollo se consideró la idea de utilizar versión muy básica y minimalista de Scrum, con iteraciones de una semana de duración, pero el tiempo que se le podía invertir al proyecto era excesivamente irregular y no valía la pena ni el tiempo ni el esfuerzo que habría que dedicarle para obtener a priori unos resultados tan poco prometedores.

Dicho esto, no se ha utilizado ninguna metodología de trabajo como tal, sino el sentido común y la improvisación para adaptarse a los problemas que han ido surgiendo sobre la marcha, que no han sido pocos. Se ha procurado establecer una jerarquía y una cola de prioridades a la hora de determinar el camino a seguir. La mayor prioridad de todas fue cerrar el diseño del juego (ver apartado *Diseño inicial*), ya que es la piedra angular del proyecto y sobre lo que se sustenta todo lo demás. A partir de ahí, se ha ido intercalando el modelado y la animación con la programación de la lógica del juego, sobre la marcha, dependiendo de los resultados obtenidos.

El desarrollo ha durado unas 15 semanas, cuyo avance a grandes rasgos se puede resumir en el diagrama de Gantt de la Figura 2 y cuyo estudio económico es el siguiente:

Tarea	Horas dedicadas	Precio por hora	Coste
Documentación	60	5 euros	300 euros
Modelado	110	15 euros	1.650 euros
Animación	140	15 euros	2.100 euros
Programación	200	15 euros	3.000 euros
	510 horas		**7.050 euros**

Figura 1. Estudio económico del proyecto.

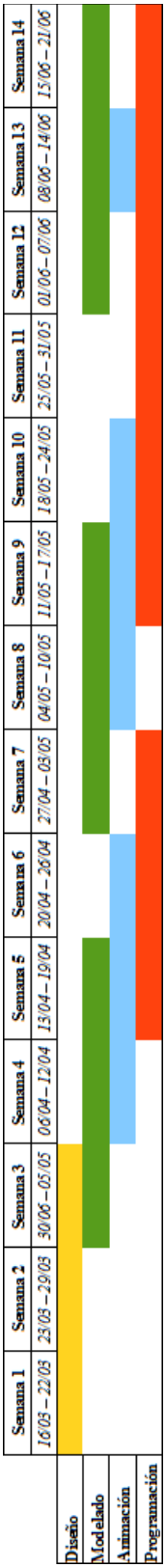

Figura 2. Diagrama de Gantt.

4. Diseño inicial

El diseño inicial recoge todo lo relacionado con el mundo en el que se desarrolla la acción, las mecánicas jugables, los objetos y el desarrollo del personaje.

4.1 Jugabilidad

El juego pretende combinar la profundidad del rol con la agilidad de la acción, añadiendo fases en las que hacer pensar al jugador con enigmas y puzzles en diferentes mazmorras. El objetivo es conseguir un género híbrido que recoja la esencia antes citada, ofreciendo una jugabilidad completa y equilibrada.

A grandes rasgos, controlamos a un personaje en tercera persona con una cámara isómetrica de altura regulable que no nos permite ver más allá de lo que el personaje tiene a su alrededor. Su aventura se lleva a cabo en un mundo de proporciones considerables en el que se encuentran entradas a diferentes mazmorras, que pueden estar conectadas entre sí. En ellas deberá enfrentarse a enemigos, conseguir nuevos objetos, subir de nivel y completar acertijos usando herramientas especiales con tal de poder enfrentarse al jefe final de cada nivel y llegar así al final de la aventura.

4.2 Personaje

El personaje cuenta con nivel, parámetros, atributos y experiencia. La experiencia lo rige todo: se puede invertir cierta cantidad para subir de nivel y ganar puntos de atributos con lo que mejorar sus parámetros.

4.2.1 Atributos

Cuando el personaje sube de nivel (hasta un máximo de nivel cien), ganará puntos de habilidad que gastar en una serie de atributos que afectarán a ciertos parámetros. La cantidad de puntos de habilidad ganados dependerá del rango de niveles en el que se encuentre.

Atributo	Descripción	Rango
Fuerza	Determina el daño físico que se realiza.	1-99
Inteligencia	Determina el daño mágico que se realiza.	1-99
Espíritu	Determina la cantidad de puntos de magia del personaje	1-99
Vitalidad	Determina la cantidad de puntos de vida del personaje	1-99
Defensa	Determina la resistencia a los ataques físicos enemigos y la cantidad de puntos de vida que se pierden por golpe.	1-99
Resistencia	Determina la resistencia a los ataques mágicos enemigos y la cantidad de puntos de vida que se pierden por golpe.	1-99

Aguante	Determina la cantidad de peso que el jugador puede llevar en el inventario.	1-99
Pericia	Determina el porcentaje de daños críticos realizados, la calidad y frecuencia de las recompensas que deja caer el enemigo y otros aspectos relacionados con la suerte.	1-99

Figura 3. Tabla de atributos del personaje.

4.2.2 Maestrías

Cada cinco subidas de nivel, el personaje ganará un punto de maestría con el que mejorar su manejo y control sobre armas, armaduras, accesorios, hechizos y otros aspectos que le rodean. Los puntos de maestría también se pueden conseguir cumpliendo algunas misiones especiales en forma de recompensa, pero el reto será mayúsculo.

Maestría	Descripción	Rango
Armas a una mano	Desbloquea acceso a diferentes armas de una mano y a sus habilidades. Afecta también a sus parámetros.	0-5
Armas a dos manos	Desbloquea acceso a diferentes armas de dos mano y a sus habilidades. Afecta también a sus parámetros.	0-5
Armas a distancia	Desbloquea acceso a diferentes armas a distancia y a sus habilidades. Afecta también a sus parámetros.	0-5
Armaduras ligeras	Desbloquea acceso a diferentes armaduras ligeras y a sus habilidades. Afecta también a sus parámetros.	0-5
Armaduras pesadas	Desbloquea acceso a diferentes armaduras pesadas y a sus habilidades. Afecta también a sus parámetros.	0-5
Hechicería	Desbloquea nuevas magias. Disminuye los requisitos de uso de hechizos de menor rango. Aumenta los puntos de magia.	0-5
Constitución	Disminuye los requisitos de uso de armas de menor rango.Aumenta los puntos de vida. Aumenta el daño físico.	0-3
Fortuna	Potencia los efectos del atributo Pericia.	0-3
Ranuras de hechizo	Añade más ranuras de hechizos, valor que determina la cantidad que pueden equiparse a la vez.	0-5
Bolsas de armas y accesorios	Añade más ranuras de equipamiento para armas y accesorios, lo que permite equiparse con más de un juego y cambiar a placer. Siguen pesando en el inventario y, salvo excepciones, sus efectos no se activan si no están en uso.	0-3

Figura 4. Tabla de maestrías del personaje.

La idea es que un personaje no podrá llegar nunca a dominar todas las maestrías ni a tener al máximo todos los atributos, por lo que tendrá que intentar especializarse en los aspectos que más útiles considere para el arquetipo que se tenga en mente. Se ofrecerá alguna forma de reubicar puntos, pero su acceso será complejo e invitará a no volver a cometer errores en las subidas de nivel.

4.2.3 Parámetros

Los parámetros se ven determinados por el equipamiento, los atributos y las maestrías del personaje.

Parámetro	Descripción	Parámetros relacionados	Maestría relacionada
Daño físico	Valor de daño físico total, resultado del arma equipada y de los atributos del personaje.	Fuerza	Armas a una mano, Armas a dos manos, Armas a distancia
Daño mágico	Valor de daño mágico total, resultado del arma equipada y de los atributos del personaje. También dependerá de la magia a realizar.	Inteligencia	Armas a una mano, Armas a dos manos, Armas a distancia, Hechicería
Puntos de vida	Puntos que corresponden a la salud del personaje. Al llegar a cero, muere.	Vitalidad, Defensa, Aguante	Constitución
Puntos de magia	Puntos que se consumen al realizar ciertos ataques. Mágicos, normalmente.	Espíritu, Resistencia	Hechicería
Porcentaje de daño crítico	Posibilidad de realizar un ataque crítico (más dañino de lo normal).	Pericia	Fortuna, Armas a una mano, Armas a dos manos, Armas a distancia
Bonus de experiencia	Bonus a la experiencia ganada en combate.	Pericia	Fortuna
Peso	Cantidad de peso que puede llevarse en el inventario.	Aguante	Armaduras ligeras, Armaduras pesadas

Figura 5. Tabla de parámetros del personaje.

4.2.4 Experiencia

La experiencia la conseguimos venciendo enemigos, usando ciertos objetos consumibles y

como recompensa de algunas misiones secundarias. Las subidas de nivel no son automáticas, sino que debemos gastar la experiencia ganada en el proceso de forma manual. Al morir el personaje, la experiencia acumulada sin gastar se perderá o se reducirá a la mitad. Para complicar un poco más las cosas, por cada nivel por debajo que esté un enemigo del protagonista, éste recibirá un 20% menos de experiencia de él, de tal manera que a partir de los 5 niveles de diferencia las experiencia ganada será de un mísero punto.

Además de para subir de nivel, la experiencia será también la moneda del juego con la que interactuaremos con vendedores para adquirir nuevos objetos y equipo, cobrando así aún más importancia su papel en la jugabilidad.

La fórmula para establecer la subida de nivel es la siguiente: se parte de una diferencia mínima de 100 puntos entre nivel y nivel, sumándole una pequeña cantidad inicial de 10 puntos entre cada nivel. Cada 5 niveles, esa pequeña cantidad inicial multiplica su valor por 1.5. Considerando el hecho de que enemigos de menor nivel no serán útiles para entrenar, parece una escala coherente, aunque está abierta a modificaciones.

Nivel	Experiencia para subir	Nivel	Experiencia para subir	Nivel	Experiencia para subir
1	0	35	1.708	70	29.192
5	150	40	2.560	75	43.789
10	225	45	3.844	80	65.684
15	337	50	5.766	85	98.526
20	506	55	8.649	90	147.789
25	759	60	12.974	95	221.683
30	1.139	65	19.461	100	332.525

Figura 6. Tabla que determina la experiencia necesaria para subir de nivel.

4.3 Equipo

El equipo (armas, armaduras, accesorios) lo podremos comprar en tiendas distribuidas por las diferentes localizaciones del juego, descubrir en tesoros escondidos en los escenarios o conseguir como recompensa que dejen caer los enemigos.

El equipo puede tener requisitos de uso, que consisten en un valor mínimo en ciertos atributos, un valor mínimo en ciertas maestrías o un nivel mínimo de personaje. Su clasificación se rige por rangos, que determinan, por así decirlo, su calidad.

Además, el equipo evolucionará con el uso al cumplir ciertos requisitos, como consumir cierta experiencia para conseguir una subida de nivel o derrotar a un número de enemigos con el objeto equipado. Todo dependerá del objeto en cuestión. La evolución mejora los parámetros del equipo y puede desbloquear/potenciar habilidades únicas del mismo, que sólo podrán usarse si se

tiene equipado el elemento de turno.

El equipo puede agruparse en conjuntos, de tal manera que usar a la vez diferentes piezas de la misma familia (ya sean armas, armaduras o accesorios) proporcione ventajas adicionales. Son casos menos frecuentes, pero el esfuerzo de buscar el conjunto completo valdrá la pena.

4.3.1 Armas

Las armas serán las protagonistas de los combates y las encargadas de hacer daño al enemigo, normalmente físico. Encontramos tres categorías: a una mano, a dos manos y a distancia, aunque dentro de cada categoría podemos encontrar más medios para clasificarlas.

Tipo	Nombre	Descripción
A una mano	Espada	Rápida. Daño medio. Alcance medio.
	Hacha	Rápida. Daño medio-alto. Alcance bajo.
	Maza	Velocidad moderada. Daño alto. Alcance bajo.
	Lanza	Lenta. Daño moderado. Alcance alto.
	Vara	Arma necesaria para llevar a cabo los hechizos. Daño físico escaso.
A dos manos	Espadón	Velocidad moderada. Daño alto. Alcance medio.
	Maza	Velocidad lenta. Daño muy alto. Alcance medio.
	Hacha de batalla	Velocidad moderada. Daño alto. Alcance medio.
	Alabarda	Velocidad moderada. Daño alto. Alcance alto.
	Bastón	Rápido. Daño moderado. Alcance alto.
A distancia	Arco	Cadencia de tiro alta, daño moderado, bajo coste de munición.
	Ballesta	Cadencia de tiro baja, daño alto, bajo coste de munición.
	Arma de fuego	Cadencia de tiro alta, daño alto, alto coste de munición.

Figura 7. Tabla con la clasificación básica de los tipos de armas.

Aunque habrá armas clave en el argumento, dentro de una categoría podemos encontrar al misma arma con diferentes atributos. En otras palabras, será normal cambiar relativamente pronto de equipamiento al encontrar otros que se adapten mejor a nuestras necesidades.

Usando armas a una mano podemos equipar o bien un escudo o bien otro arma a una mano en la mano libre. Algunas armas pueden tener características especiales dependiendo de con qué mano se utilicen.

4.3.2 Escudos

Los escudos nos cubrirán de los ataques de los enemigos. Son “armas” de una mano, que

pueden equiparse en cualquiera de ellas. Los parámetros de los escudos son el porcentaje de defensa física y el porcentaje de defensa mágica que ofrecen. En otras palabras, la resistencia a los ataques enemigos.

4.3.3 Armaduras

El personaje podrá equiparse con hasta cuatro piezas de armadura a la vez (botas, coraza, guantes y casco) que reduzcan el daño recibido y potencien sus parámetros. Encontramos dos grupos de armaduras: las ligeras y pesadas. Como su propio nombre indica, las primeras pesarán menos y ofrecerán una resistencia menor que las segundas, pese a que habrá excepciones. Son diferentes, no mejores, y la elección a tomar sobre qué tipo de armadura escoger puede depender del arquetipo de personaje que queramos crear.

4.3.4 Accesorios

El personaje puede equiparse un accesorio cuyas propiedades no están ligadas a ningún género. Puede proporcionar habilidades especiales, mejorar parámetros, ofrecer bonificaciones…

4.4 Objetos

El personaje tendrá a su alcance un amplio abanico de objetos de diferentes tipos para poder superar los obstáculos que se le impongan durante la aventura más fácilmente.

4.4.1 Objetos consumibles

El personaje podrá usar objetos consumibles tales como pociones, hierbas o pergaminos que le ofrezcan ciertas mejoras de forma temporal. Tiene a su alcance tres ranuras de objeto que le permiten usarlos a través de acceso rápido, pero en caso contrario debería acudir al inventario para su uso.

Los objetos consumibles no son mejorables, aunque puede haber versiones mejores disponibles para conseguir a través de tiendas, tesoros y recompensas por abatir enemigos. Por ejemplo, una poción menor que cure una pequeña cantidad de vida y una poción superior que cure una cantidad superior.

4.4.2 Objetos clave y herramientas

Habrá ciertos objetos clave que una vez en nuestro inventario no podremos deshacernos de ellos y, como su propio nombre indica, son clave en el desarrollo de la aventura.

Por un lado tenemos libros, llaves y una serie de objetos cuyo uso está ligado a situaciones concretas, y por otro una especie de objetos necesarios para poder completar algunas partes de las mazmorras. Estos últimos, a los que denominaremos herramientas, podrán asignarse como si un objeto consumible se tratase, ya que su uso es activo y variado.

Un ejemplo de herramientas serían unos guantes que nos permitieran mover objetos pesados por el escenario, unas botas que nos permitieran correr durante una cierta cantidad de tiempo más rápido, una antorcha para iluminar una zona a oscuras, un látigo para interactuar con objetos a

distancia, etc.

4.5 Mundo

El mundo del juego se divide en tres secciones: exteriores, interiores y mazmorras. Las mazmorras y los interiores podrían considerarse lo mismo, pero por su uso es más conveniente separarlos.

El exterior del mundo es lo que su propio nombre indica. Zonas grandes, no necesariamente al aire libre, desde las que podemos acceder a ciudades y mazmorras. Hay enemigos, podemos encontrar puzzles, NPC… y está sujeto a cierta aleatoriedad en lo que respecta a los eventos que encontremos.

Los interiores son la parte interna de un edificio o cueva. Están aislados del exterior.

Las mazmorras son interiores con acertijos, puzzles y desafíos en las que al final aguarda un jefe final. Son parte vital de la aventura. Pueden tener más de una entrada y requerir que el usuario realice alguna acción fuera de la misma para superar algún obstáculo. También tienden a requerir el uso de ciertas herramientas para poder ser completadas.

El mapa de la demo será una porción de exterior, que contará con varias entradas a una única mazmorra, cuya resolución será el grueso de la demostración, mostrando algunos enemigos, el sistema de combate y evolución del personaje, el uso de alguna herramienta y un combate contra un jefe.

4.6 Escenarios

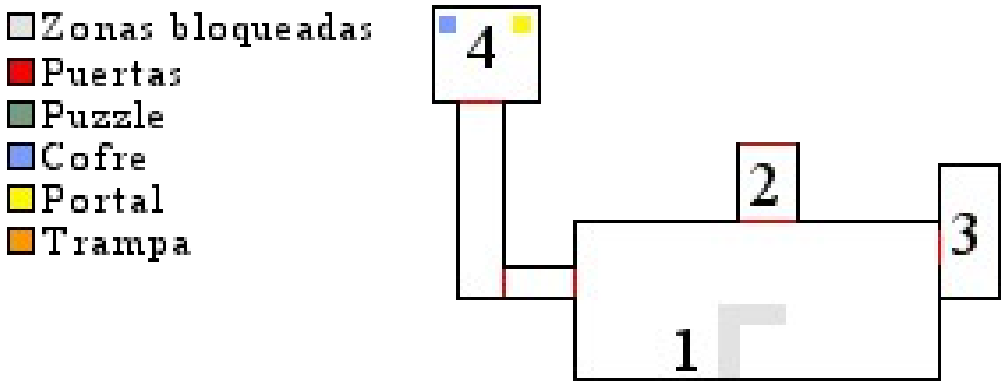

Figura 8. Diseño del mapa de la escena correspondiente al exterior.

Punto de interés	Descripción
1	Punto de inicio del juego.
2	Entrada a la mazmorra.
3	Contenido por determinar.
4	Cofre y portal a la segunda entrada de la mazmorra.

Figura 9. Tabla de los puntos de interés de la Figura 8.

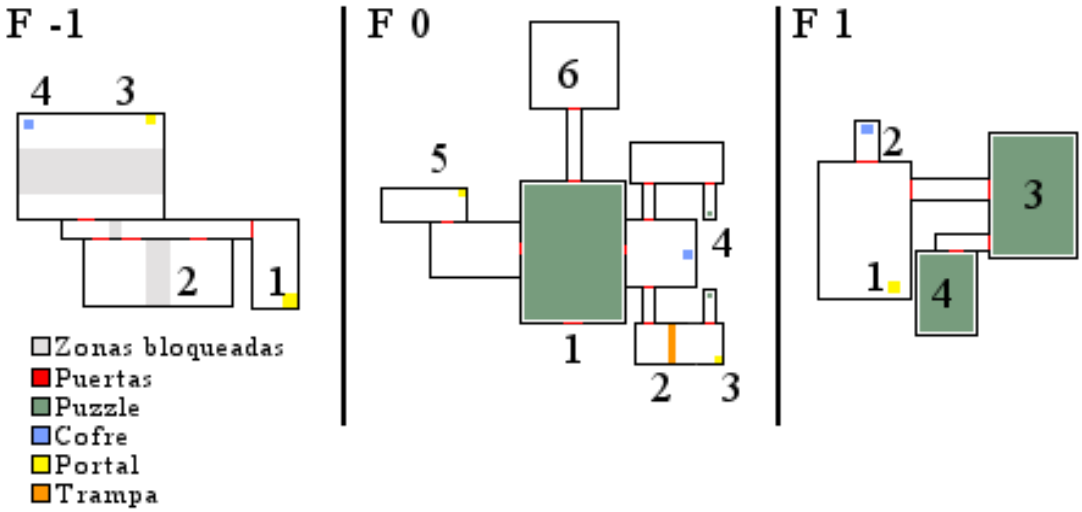

Figura 10. Diseño del mapa de la escena correspondiente al exterior.

Planta	Punto de interés	Descripción
F -1	1	Portal que conecta con F0.3.
F -1	2	Lugar de la caída de la trampa de F0.2.
F -1	3	Entrada secreta que se encuentra en el exterior de la mazmorra.
F -1	4	Cofre.
F 0	1	Entrada principal de la mazmorra. Puzzle relacionado con el de F1.4.
F 0	2	Trampa. Se cae a F-1.2.
F 0	3	Portal que conecta con F-1.1.
F 0	4	Pequeño puzzle de palancas. Hay que activarlas para que aparezca el cofre en la sala centaral.
F 0	5	Portal que conecta con F1.1.
F 0	6	Sala del jefe final.
F 1	1	Portal que conecta con F0.5.
F 1	2	Cofre.
F 1	3	Puzzle por determinar con el uso de alguna herramienta.
F 1	4	Puzzle por determinar con el uso de alguna herramienta enlazado con el puzzle de F0.1.

Figura 11: Tabla de los puntos de interés de la Figura 10.

5. Primeros pasos en Unity3D

Antes de comenzar a explicar los detalles del desarrollo del proyecto, vale la pena dejar algunos conceptos claros sobre cómo funciona internamente Unity3D y familiarizarse con sus elementos más comunes.

Todo el motor gráfico gira en torno a la existencia de los GameObjects, un término que puede definirse como cualquier objeto de la escena en la que transcurre la acción, aunque técnicamente se trata de un contenedor, algo vacío por dentro sin ningún tipo de significado ni función por sí solo. Por otra parte encontramos los Components, que son las piezas funcionales de los GameObjects, los elementos que dotan de sentido su existencia. Sin ir más lejos, no puede existir un GameObject totalmente vacío: siempre trae consigo un Transform Component, que encargado de determinar el tamaño, rotación y posición del GameObject en la escena. Ahora bien, la relación de dependencia es mutua: un Component necesita tener un GameObject sobre el que funcionar.

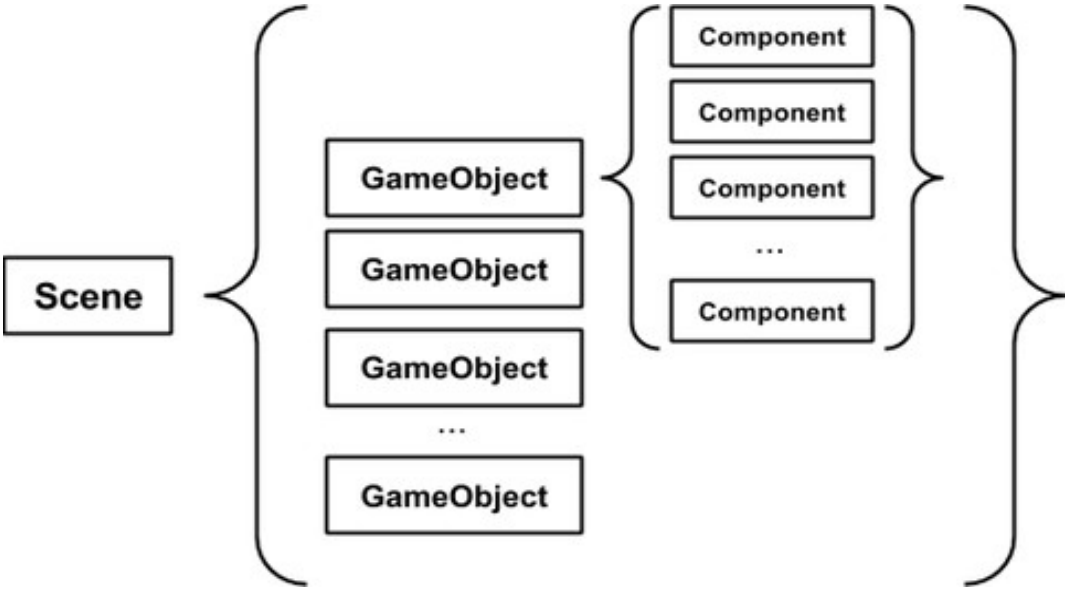

Figura 12. Relación entre escena, GameObject y Components en Unity3D.

Dicho esto, ¿dónde entran los scripts en este esquema? ¿De dónde sale el código que se ejecuta? Pues también son Components, aunque unos algo especiales. Las clases que implementemos han de heredar de la clase MonoBehaviour, que nos da todas las herramientas necesarias para, ahora sí, poder interactuar con el resto de elementos de la escena y entrar de lleno en su flujo de ejecución.

Los objetos MonoBehaviour cuentan con algunas peculiaridades si los comparamos con los objetos clásicos de C Sharp y JavaScript. No tardamos en darnos cuenta, por ejemplo, que no tienen algo tan básico en el Paradigma de la Programación Orientada a Objetos como un constructor. Ahora bien, sí que nos ofrece una serie de métodos alternativos para conseguir realizar las operaciones relativamente equivalentes.

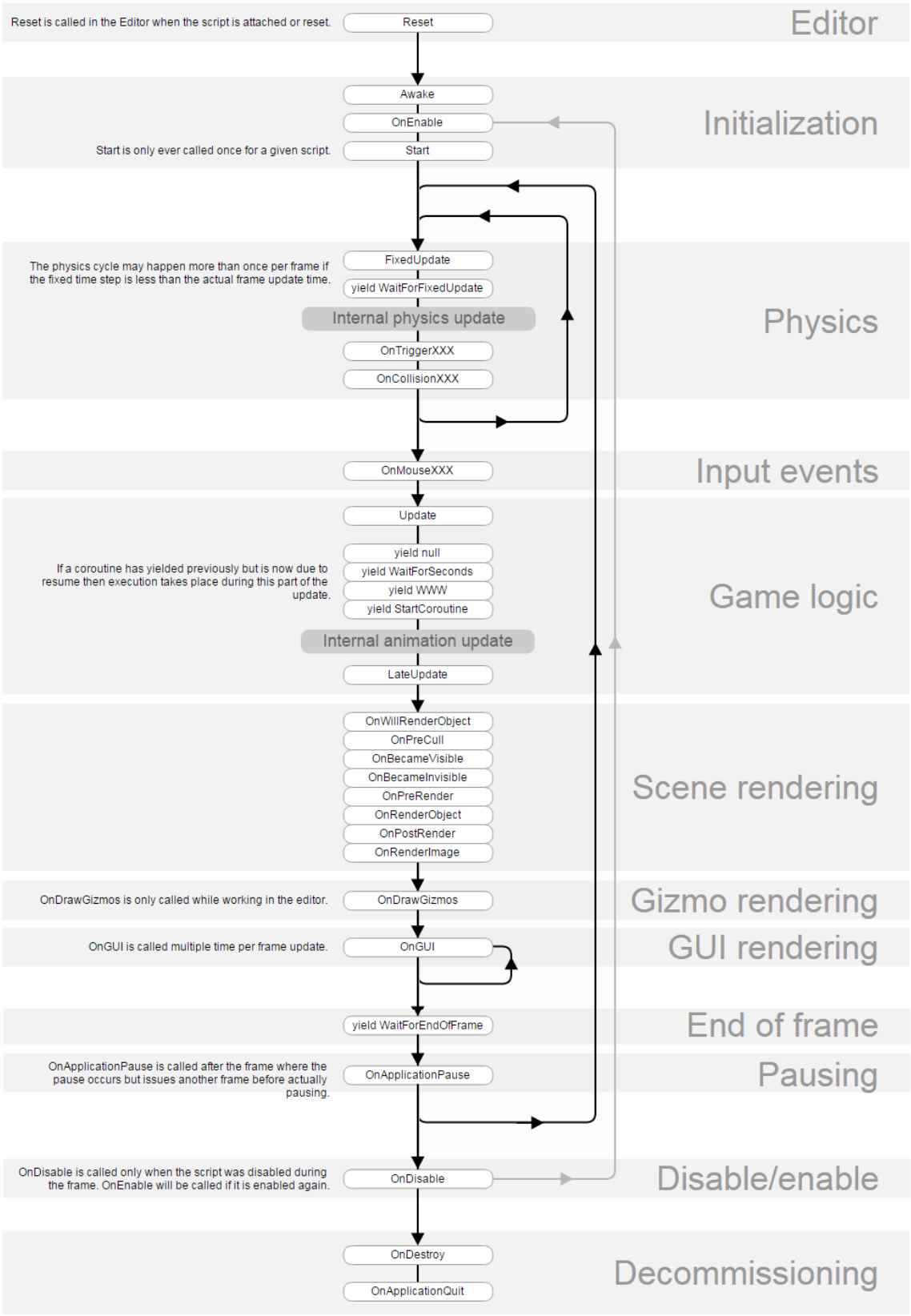

Figura 13.Ciclo de vida de la ejecución de una escena en Unity3D. [6]

El *pipeline* de la Figura 12 concreta todas las fases por las que pasa una escena de Unity3D. Cuando una escena se carga, todos los objetos presentes en ella ejecutan su método Awake, posteriormente el método OnEnabled y finalmente el método Start, correspondiendo este tridente a un constructor tradicional en el que, si es necesario, se recoge la referencia de otros Components a tratar como variables. Si los Components están definidos antes de la ejecución, se les puede pasar parámetros (públicos) a través del Inspector y evitarnos algunas complicaciones, pero en el caso de ser creados al vuelo el procedimiento es algo más pesado.

El ciclo de vida de una escena de Unity3D es largo y complejo, pero nos podemos saltar bastantes eslabones de su *pipeline* a la hora de trabajar si realmente nuestro desarrollo no guarda relación con ellos. Por ejemplo, el método FixedUpdate se ejecuta más de una vez por frame y se encarga de realizar cálculos relacionados con la física del motor gráfico, mientras que Update se ejecuta una única vez por frame, gestionando la lógica de la aplicación. Tras este último se ejecuta LateUpdate, también una vez por frame, que tiende a ser útil para realizar operaciones dependientes de otras realizadas en el método anterior.

Hilando un poco más fino en el ciclo de vida de la escena, cabe destacar que con un *pipeline* tan denso y complejo se corre el riesgo de que el desarrollador inconscientemente sobrecargue los métodos derivados de Update. Siendo la ejecución secuencial y considerando que todo ha de ejecutarse entre frame y frame, si el volumen de operaciones a realizar es considerablemente elevado puede saltarse este marco temporal, conllevando bajones en la tasa de frames por segundo y, en resumen, empañando el rendimiento del juego. Para solventar este problema existen, además de las buenas prácticas a la hora de estructurar y programar el código, las coroutines.

Las coroutines son un tipo de método que permite pausar su ejecución a medias (tras varios segundos, hasta el final del método FixedUpdate o hasta el final del frame), quedarse en segundo plano y volver a retomar la ejecución desde el mismo punto una vez vuelva a ser llamado. Es una práctica interesante e inteligente de aliviar la carga de los métodos más pesados en lo que respecta a algunas operaciones de menor importancia.

6. Desarrollo

Tal y como se ha comentado en el apartado Planificación, el flujo de trabajo ha dado prioridad al modelado de escenarios, objetos y enemigs, para soltar posteriormente a la animación necesaria de cada modelo y, finalmente, a su integración a la lógica del juego. Aun así, en algunos momentos los tres apartados han avanzado en paralelo.

6.1 Modelado

Las dos máximas que han regulado mis decisiones de diseño en el modelado de todo el juego han sido la sencillez por un lado y la intención de escapar de un aspecto hiperrealista por otro. Bien podrían significar ambas cosas lo mismo, pero la segunda hace hincapié en aspectos como la iluminación y la renderización, por lo que vale la pena separar los conceptos.

Mi intención en todo momento ha sido evitar el uso de los shaders comunes y realistas en lo que respecta a cómo responden ante la iluminación para usar *toon shaders*. Esta técnica de renderización calcula para cada píxel del modelo en cuestión su valor dependiendo del color del material asignado y del ángulo de la normal de su cara respecto a la iluminación recibida, pero a posteriori cuantifica todos los valores (que pueden definirse usando una pequeña imagen que responde al nombre de *Toon Ramp*) antes calculados en un rango considerablemente más pequeño, consiguiendo un efecto la mar de curioso en el que las zonas que comparten una iluminación similar muestran el mismo color. El efecto de "dibujo animado" puede potenciarse remarcando los bordes de los modelos con líneas de grosor y color configurable.

Figura 14. Ejemplo de Toon Ramp.

Sobre la mesa en un principio estaba la idea de desarrollar un *shader* personalizado para conseguir potenciar el efecto antes explicado, pero Unity3D por defecto trae consigo cuatro shaders que cumplen a la perfección con la intención, por lo que no ha sido necesario.

Con este aspecto gráfico en mente, se ha evitado tener que trabajar con texturas, aunque todo tiene su lado bueno y su lado malo. En lo que respecta a lo primero se ahorran muchas horas de edición de imágenes intentando generar texturas (algo que no afecta a la jugabilidad, pero si al apartado estético), aligerando así el proceso de desarrollo de modelos. En lo que respecta a lo segundo, hay que tener un mínimo de buen gusto y arte para conseguir crear modelos sencillos que queden bien con dicha técnica y con una complejidad poligonal baja, complicando el conseguir un buen resultado. También hay que añadir en este segundo grupo que los escenarios se ven perjudicados al ser planos, por lo que en ellos se ha saltado la norma para conseguir un efecto algo más vistoso.

Para que este estilo pueda ser llevado a cabo con relativo éxito, el modelo ha de tener asignado a cada una de sus caras un material, pese a que su uso, que define el comportamiento de un cuerpo en muchos sentidos, no es independiente del uso de texturas. A la hora de realizar el modelado (recordemos, con Blender), se han de crear tantos materiales como colores queramos, y asignar a cada cara del modelo uno de ellos. Aquí se nos abren varias formas de trabajo: o definir los materiales al completo desde Blender o crear plantillas en blanco a definir una vez el proyecto se importe a Unity3D. Se ha optado por lo segundo, prefiriendo dar más responsabilidades al motor gráfico que a herramientas externas. Además, esta segunda opción daba un mayor margen de maniobra.

Consultar el anexo para ver todos los modelos realizados.

6.2 Animación

Como he explicado en el apartado *Tecnologías utilizadas*, he decidido trabajar con Blender para modelar y animar al ofrecerme herramientas suficientes para realizar ambos procesos desde una misma plataforma, por lo que se entiende a priori que debería dar bastantes facilidades. Y realmente lo hace. Más o menos.

Existen varias formas de crear animaciones, pero me he decantado por la que se basa en huesos. La teoría es sencilla: consiste definir una serie de huesos (elementos con tamaño, rotación y localización como si de cualquier otro modelo se tratase, pero invisible fuera de los programas de edición) de forma jerárquica que se distribuyan según nuestras necesidades en un modelo concreto, creando así un armazón.

Una vez distribuidos, hay que establecer los pesos que tendrá cada hueso en el modelo, lo que significa decidir qué conjuntos de vértices del modelo reaccionarán (y cómo) acorde al hueso cuando este reciba alguna modificación en su forma. Blender ofrece una serie de herramientas para distribuir automáticamente el peso de los huesos, pero el resultado no tiene a dejar buenos resultados, generando deformaciones muy extrañas en puntos con algo de complejidad. Por eso, existe la posibilidad de pintar a mano la relación entre cada hueso con todos los vértices del modelo. Una tarea mucho más tediosa y compleja, pero que me he visto obligado a usar en algún momento concreto para arreglar alguna animación.

Teniendo el armazón terminado, llega la hora de crear las diferentes animaciones, algo en esencia sencillo pero cuya dificultad crece muchísimo en cuanto intentamos hacer algo mínimamente complicado, en lo que tampoco ayuda que la interfaz de Blender sea poco usable y no colabore en ningún momento en el proceso. La idea es partir de una pose inicial, definir una duración para la animación y establecer una posición, tamaño y rotación para cada hueso en una serie de puntos clave (*key frame*), entre los cuales se llevará a cabo una transición. Ahora bien, la jerarquía definida por los huesos, las limitaciones establecidas en lo que respecta a sus transformaciones y el peso de cada hueso en el modelo determinan el resultado final.

Se pueden crear tantas animaciones como nos venga en gana, pero para que se puedan importar hay que forzar a Blender a que su *garbage collector* no decida eliminarlas al sólo estar en activo una de ellas. Es decir, internamente la herramienta se encarga de gestionar cuántos bloques de memoria está usando cada elemento creado (material, animación, mapa UV...). Si el resultado es igual a cero, internamente decide ignorarlo a la hora de guardar e importar el proyecto, ya que podría decirse que no está siendo usado.

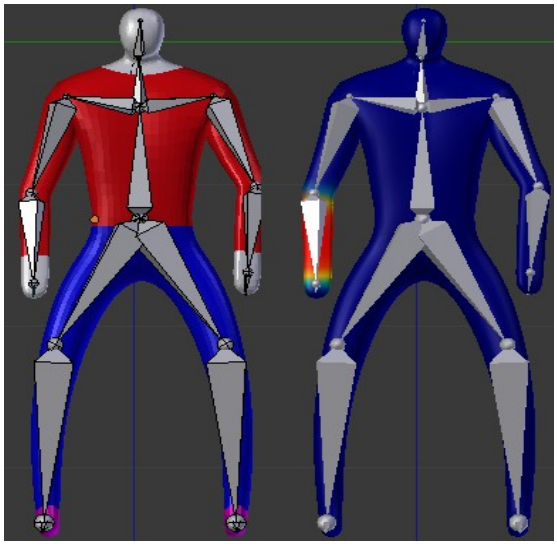

Figura 15. Modelo con su armazón a en el lado izquierdo de la imagen. El mismo modelo en el lado derecho, mostrando el peso de uno de los huesos de su brazo derecho respecto al resto del cuerpo.

Es una forma de proceder bastante lógica, pero a la que cuesta entender de buenas a primeras y la plataforma no es demasiado amigable en el proceso de aprendizaje, más bien todo lo contrario. Una vez comprendida la idea, el procedimiento consiste en crear usos falsos de estos elementos para que a la hora de importar el proyecto podamos trabajar con toda la información sin que se pierda nada por el camino.

La parte más complicada en lo que respecta a las animaciones ha sido el personaje principal. Las armas que el personaje utiliza son independientes del personaje, por lo que se anclan en un punto concreto de su armazón desde Unity3D y comparten los movimientos del modelo. Para conseguir anclarla se ha utilizado un hueso (pequeñísimo, sin peso) que únicamente sirve como punto orientativo para saber cómo y dónde situarlo.

6.3 Importando a Unity3D

Existen muchos formatos en los que importar el modelo y su animación a Unity3D, pero el que menos complicaciones me ha dado en un principio es el formato FBX (Filmbox). Es un formato propietario desarrollado por Autodesk y Filmbox en 2006 que se ha vuelto un estándar en el desarrollo de aplicaciones en tres dimensiones. Al ser un formato propietario trae consigo ciertas restricciones en forma de un SDK que nos ata a la hora de trabajar con estos archivos, pero tanto Blender como Unity3D son totalmente compatibles con él.

Al importar un modelo en FBX empaquetamos todo lo que hay en la escena (cámaras y luces incluidas, por ejemplo) en un sólo archivo que Unity interpreta, dejándonos modificar todo su interior. En mi caso, estos archivos incluyen el mesh del modelo, su Avatar (que define el armazón y sus huesos) y las diferentes animaciones que se hayan encontrado. Junto al archivo se generará una carpeta con los materiales que necesita, existiendo varias posibilidades de configuración al respecto jugando con el nombre de cara material y posibles rutas para almacenarlos.

6.4 Prefabs: pros y contras

Unity nos da la posibilidad de montar una estructura de datos especial que responde al nombre de Prefab, que no es otra cosa que un GameObject con unos components concretos y una configuración determinada por el desarrollador, todo bien empaquetado y organizado en un sólo archivo. Su uso es muy útil cuando un objeto tiene varias instancias, y replicar su existencia a mano es una tarea tediosa y un tanto inútil, si realmente todas sus instancias van a hacer lo mismo.

También gana protagonismo cuando llega la hora de instanciar un GameObject con código. Si estamos desarrollando algo utilizando el Inspector y otras herramientas bastante visuales que nos ofrece Unity3D, con un simple copiar y pegar nos basta para duplicar un elemento, pero si estamos trabajando desde código, la aproximación es muy diferente, siendo infinitamente más sencillo instanciar un prefab que cualquier otro procedimiento.

En el código podemos ver ambas aproximaciones. Por ejemplo, el personaje principal se monta a través de código totalmente. Se carga su mesh, se le añaden componentes y se configura al instanciarse en un procedimiento bastante pesado en situaciones relativamente complejas, cuando podría haberse realizado en dos simples líneas si se hubiera optado por usar un prefab con anterioridad. Por otra parte, los enemigos se cargan instanciando los prefabs tras haberlos configurado visualmente en la interfaz gráfica y su empaquetado posterior.

Trabajar con prefabs tiene numerosas ventajas, pero también algún que otro inconveniente. Es un procedimiento cómodo y que además nos permite trabajar de forma muy visual, pero su principal talón de Aquiles es la conexión del prefab con todos sus hijos. Es decir, si cambiamos algún aspecto del prefab, se verá reflejado en todas sus instancias. Es una característica que de por sí no es negativa (de hecho, es un comportamiento que se agradece, ya que en parte responde coherentemente a la naturaleza de este tipo de archivos), pero sí que requiere tener cierta consciencia de ello, obligándonos a editar vía código algún elemento instanciado de esta forma si tenemos la necesidad de ello. Por otra parte, si partimos de la rama visual, podemos desvincular de su prefab a cualquier objeto para que sea un elemento libre, y posteriormente transformarlo en otro prefab con total libertad.

6.5 Montando y coordinando animaciones

Para que las animaciones importadas funcionen (mejor dicho, para que cualquier animación en Unity3D funcione) es necesario añadir el componente Animator al GameObject correspondiente, que se encargará de enlazar el *mesh* del objeto con un Animator Controller que tendremos que crear para poder gestionar los diferentes estados del objeto. En otras palabras, estamos creando una máquina de estados.

El componente Animator Controller genera tantos estados como animaciones tenga el modelo que le enlacemos, aunque tenemos cierta libertad para crear algunos por nuestra cuenta, pese a ser algo que a prior no suene demasiado lógico. Con los estados en la mano, nos encargamos de definir las transiciones entre ellos. Definir una animación de entrada, definir una animación de transición, definir una animación de salida del sistema... también podemos crear variables desde las que acceder externamente que sean las que se encarguen de gestionar los pasos de una transición a

otra, e incluso controlar otros aspectos de las transiciones como su duración o si pueden ser cortadas a medias o deben terminar antes de dar paso al siguiente estado.

Tal y como está definido y diseñado el juego, las animaciones cobran una importancia considerable. No son un simple aspecto visual, sino que también afectan a las mecánicas del juego. Por eso, es necesario por interactuar con el Animator Controller a través de algún script que gestione las variables que determinan los cambios de estado. El sistema que se ha montado respecto a los enemigos se compone de dos scripts que se comunican el uno con el otro: uno que gestiona las animaciones y otro que gestiona la inteligencia artificial de los enemigos. En lo que respecta al personaje principal, este último es innecesario, cambiándolo por el componente que gestiona el movimiento del mismo.

Para poder entablar comunicación con el Animator Controller se usa un sistema bastante curioso. Cada estado de la máquina de estados se identifica con una cadena de texto que es la unión del nombre de la capa en la que se encuentra el estado, un punto y el nombre del estado en sí. Dicha cadena se convierte en un valor *hash*, que será el valor que devolverá el Animator Controller cuando se le pregunte por su estado actual. Realizando externamente la misma operación (lo que nos obliga de antemano a conocer internamente las animaciones existentes) y comparando resultados podemos saber así en qué estado se encuentra y actuar en consecuencia.

En lo que respecta a las variables internas de la máquina de estados el funcionamiento es similar. Conociendo con anterioridad los parámetros existentes podemos leer su valor o sobreescribirlo, permitiéndonos así gestionar de forma manual las transiciones entre estados usando.

Cabe destacar que también existe la posibilidad de forzar estados manualmente, pero es una práctica poco recomendada al romper el funcionamiento del sistema, siendo algo más propio de versiones más antiguas de Unity3D en las que el aspecto de las animaciones no se había llegado a depurar tanto.

6.6 Construyendo el escenario

Unity3D es un motor gráfico que invita tanto a programadores como a artistas a trabajar con una interfaz común, dando bastantes libertades e información visual para llevar a cabo un sinfín de configuraciones. Por supuesto, hay un gran repertorio de acciones que no se pueden realizar si no se mete mano al código, pero no se puede obviar la tentadora facilidad que supone montar un escenario arrastrando el ratón con total comodidad y con un resultado inmediato.

Dicho esto, ¿qué es más inteligente? ¿Aferrarse al principio de montar todas las escenas y situaciones a través de código? ¿Procurar estructurarlo todo de tal manera que haya que configurar lo menos posible vía scripts? Realmente no existe una respuesta perfecta a esa pregunta, y la clave es adaptarse a la situación según convenga. Es algo parecido a lo que ocurre con los prefabs.

Al principio del desarrollo me abracé totalmente a la filosofía de evitar utilizar en la medida de lo posible las facilidades visuales que ofrece Unity3D. Todo iba a montarse tras leer una serie de archivos CSV que dictarían la posición de los diferentes planos, dónde estarían las puertas, los márgenes de la cámara... pero cuando un pequeño cambio conllevaba encontrar la línea en cuestión del archivo y volver a ejecutar todo el proyecto para comprobar si el resultado era el deseado, pronto me percaté de que no valía la pena.

La distribución de los escenarios está realizada por planos independientes entre ellos. El personaje puede moverse por todo el plano y la cámara le seguirá, pero contará con un rango de

movimiento limitado (del centro del plano hasta los extremos menos un margen en los tres ejes) que se recalculará cada vez que se entre en un nuevo plano. Los enemigos de cada plano también son independientes, desapareciendo/apareciendo los enemigos cada vez que salimos/entramos de una habitación.

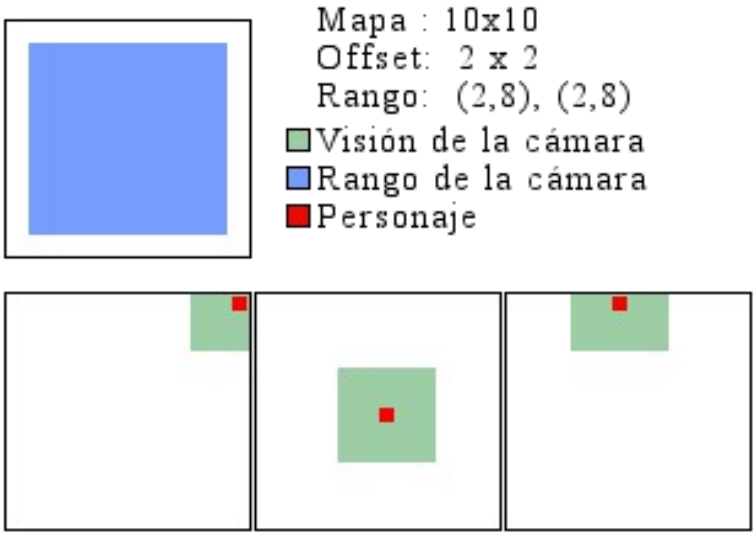

Figura 16. Funcionamiento de la cámara dependiendo del tamaño del plano y de los márgenes establecidos.

En lo que respecta a los eventos, existía la intención de que los interiores y exteriores estuvieran relacionados, añadiendo complejidad a las mazmorras, pero no ha dado tiempo a lograr algo similar más allá de un pequeño puzzle en el que tenemos que activar un par de palancas para que aparezca un cofre, que a su vez está cerrado y necesitamos una llave que se encuentra escondida en lo más profundo de la mazmorra, a donde se llega desde una entrada secundaria escondida en el exterior del nivel.

6.7 Base de datos de objetos y enemigos

Aunque montar el escenario a partir de la lectura de un archivo CSV se antojara demasiado complicado, no significa que la técnica sea inútil, ni mcho menos. Por ello hay dos especies de bases de datos en el juego: la de enemigos y la de objetos.

En lo que respecta a los objetos existe una clase estática que lee de un archivo de texto todos los objetos del juego, crea su instancia en una clase del Modelo y los agrupa en un diccionario al que se accederá para todo lo relacionado con ellos. Cada objeto posee un identificador en forma de número, y para tratar con objetos se usa dicho valor. Por ejemplo, cuando el personaje recoge un objeto de un cofre, recoge su identificador, que es consultado en el diccionario de objetos para recibir un clon del mismo.

Dependiendo del tipo de objetos (consumibles, clave, armas, equipo) los parámetros que conforman el CSV cambian y se adaptan a las necesidades. Por ejemplo, un arma necesita una ruta al prefab para poder ser instanciada, además de sus valores de ataque, mientras que una armadura sólo necesita sus valores de defensa.

En lo que respecta a los enemigos, al cargar el escenario se recorre otro archivo CSV que contiene la información de todos los enemigos que aparecen en cada punto del juego (es decir, un enemigo X de tipo Y de nivel Z en la posición XYZ), generando un diccionario de enemigos por plano para poder cargar así los enemigos que aparecen cuando cambiamos de animación.

6.8 El combate

El combate, aunque no ha terminado llegando a buen puerto por culpa de las animaciones, se basa en un sistema de colliders entre el arma y los modelos. Cada una de las armas cuenta con un prefab montado manualmente en el que se ha establecido un área en forma de ortoedro (Box collider) que define el área de contacto del arma. Por otra parte, los modelos de enemigo y personaje siguen el mismo patrón, pero indicando en su caso las zonas en las que puede recibir daño.

Los colliders se pueden usar desde dos aproximaciones diferentes. Por un lado existe la posibilidad de gestionar cuando otro objeto entra en su área a través de los métodos OnTriggerEnter, OnTriggerStay y OnTriggerExit (que gestionan entrada, permanencia por frame y salida de un GameObject en el espacio del collider, respectivamente), convirtiéndolo en una especie de chivato. Por otro puede enlazarse a un rigid body (componente que aplica la física del motor gráfico en un GameObject) y detectar colisiones entre dos colliders o rigid bodies con los métodos OnCollisionEnter, OnCollisionStay y OnCollisionExit de forma prácticamente idéntica al caso anterior.

Si el personaje del jugador se encuentra atacando y existe una colisión entre el arma y el enemigo, se procede a herir a este último. El daño impartido se debería calcular dependiendo de las estadísticas del jugador, del tipo de ataque, de las estadísticas del enemigo y de las características del arma. El sistema está montado para que sea funcional, pero no se ha establecido ninguna fórmula ni procedimiento definitivo para calcular dicho valor al haber dejado de lado el progreso del personaje en el desarrollo a última hora para poder centrarse en otras prioridades.

De la misma forma que el jugador hiere a los enemigos, los enemigos pueden herir al jugador si entran en contacto con él. Dependiendo de en qué estado de su máquina de estados se produzca el contacto, el daño será uno u otro, respetando las mismas normas que habría que haber establecido en el caso anterior.

6.9 Gestionando las partidas guardadas

Para gestionar las partidas guardadas se ha optado por la solución de guardar el estado del jugador (inventario, nivel, vida, posición, escenario...) en un archivo binario, para lo que se ha necesitado por un lado que en las clases que contienen dicha información no se encontrara ninguna clase que no fuera serializable. La serialización es un proceso mediante el cual transformamos el estado de un objeto en una secuencia de byte que, a posteriori, puede ser deserializada para reconstruir el objeto en cuestión. El problema es que todas las clases de Unity3D son no serializables, por lo que los estados del jugador han de reducirse a su mínima expresión para que esta técnica funcione.

A la hora de gestionar las partidas guardadas existe una clase estática que guarda hasta tres partidas diferentes en tres ranuras diferentes. A su vez, esta clase también está pluriempleada y

colabora en ciertos sentidos con el desplazamiento entre escenas del personaje. La clase guarda una copia temporal del jugador entre los cambios de escena, y los scripts que montan al personaje en cada una de ellas actúan en consecuencia dependiendo de su valor. Si es nulo, significará que el personaje no viene de otra escena, sino que comienza una partida nueva o bien está cargando otra. Para esto último también cuenta con una variable booleana que hace de auxiliar en el sistema.

Lamentablemente, el uso de este sistema de archivos para guardar y cargar partidas es compatible con todas las plataformas con las que trabaja Unity3D excepto una: el navegador web. La solución sería implementar o bien un sistema que guardara los datos en cookies (con todo lo que ello conlleva) o bien gestionar las partidas guardadas desde el lado del servidor. No se ha llevado a cabo ninguna de las dos soluciones.

7. Resultados

Los resultados obtenidos con el proyecto no han sido satisfactorios. Los problemas que ya se adelantaban en la planificación inicial han demostrado que los temores tenían un origen razonable. El resultado es una demostración inestable y que en ningún momento representa las horas de trabajo que hay tras su desarrollo. Por ello, a continuación se resumirá en la medida de lo posible los mayores obstáculos que se han planteado en el desarrollo.

El reto de desarrollarlo todo desde cero

El desafío de este proyecto era construir absolutamente todo lo que lo conforma desde sus cimientos. Un reto cuyas consecuencias son agridulces: se tienen controlados todos los detalles del juego... pero reinventar la rueda no tiende a ser algo inteligente, por definición.

Usando *assets* de terceros se habría obtenido un resultado mucho más vistoso y funcional en muchísimo menos tiempo. Si así hubiera sido, no se habrían perdido meses peleando con caras invisibles de modelos porque sus normales miraban hacia su interior, con animaciones que deformaban modelos y desaparecían al ser importadas, con materiales que desaparecían de la misma manera que las animaciones o con archivos FBX mal importados, por citar algunas cosas.

Sistema complejo y dependiente

Habría sido sencillo establecer un collider predeterminado que realizara unos movimientos similares a los de las animaciones de ataque independientemente del arma que llevara el jugador. Así se habrían separado responsabilidades y el sistema de combate no se vería perjudicado por el unas animaciones rebeldes. Pero las intenciones iniciales apuntaban a un sistema de combate complejo, en el que cada arma fuera diferente entre sí y en el que cada tipo de arma tuviera un conjunto de movimientos diferentes.

Se han perdido muchas horas intentando entender el por qué los modelos importados no compartían la transformación geométrica del padre cuando en un principio la configuración era idéntica. Se han perdido aún más horas aprendiendo a distribuir el peso de los huesos de una animación para que el modelo no se deformara de forma anormal. Y como resultado se ha visto perjudicado, por ejemplo, el sistema de combate.

Combinación de animación y *meshes*

Una de las primeras ideas del proyecto era que el equipamiento que llevara el personaje equipado (arma, casco, coraza, botas, guantes...) se reflejara de forma visual.

Sobre el papel era tan sencillo como, en vez de modelar un personaje completo, modelar todas las partes por separado y después montar el personaje vía código, cambiando los meshes al vuelo dependiendo del equipo del personaje. Cada una de las partes comunes compartirían los mismos movimientos, por lo que debería ser fácil cambiar un cuerpo por otro manteniendo la estructura de la animación.

La aproximación inicial fue montar el cuerpo a partir de cinco GameObjects diferentes, cada uno correspondiente a una parte del cuerpo diferente. Trabajando con coordenadas locales y con cálculos precisos sería posible colocarlos en condiciones para formar un cuerpo a partir de pedazos.

Sin problemas en ese sentido.

El primer aviso llegó al descubrir la íntima relación entre un armazón y el mesh al que está ligado. Obviamente, pueden cambiarse vía código, pero los vértices del segundo no guardan ningún tipo de relación con la animación, por lo que el resultado implica o bien deformaciones totalmente aleatorias o bien una ristra de errores.

La segunda aproximación fue crear las animaciones de forma independiente a los modelos. Animar un brazo derecho, animar una cabeza... y una vez estando finalizado ese proceso, llevar a cabo a mano la fusión entre armazón y mesh para cada uno de los modelos. Tanto en teoría como en práctica es un procedimiento costoso, pesado y largo, además de muy propenso a errores. Exportar estructuras de Blender de un proyecto a otro no es una práctica muy precisa y, en el contexto deseado, unas pocas unidades de diferencia en posición, escala o rotación puede destrozar el resultado final.

Otros puntos débiles de esta segunda idea son la complejidad que implica coordinar todas las animaciones del cuerpo como si sólo se tratara de una por un lado y lo complicado de desarrollar las animaciones en sí por otro. En un sistema normal existe una jerarquía entre los huesos que toma parte en las transiciones de la animación de forma natural, sin necesidad de configurar nada por defecto. Partiendo el cuerpo en pedazos esa relación desaparece y, aunque algunas efectos podrían emularse de alguna manera, se pierde la riqueza que ofrece la herramienta para elaborar animaciones realistas.

Tras muchos intentos fallidos, pese a que la segunda aproximación sea bastante acertadas, se decidió no implementar dicha funcionalidad por todos los problemas que conllevaba, siendo el más crítico el tiempo que ocupaba conseguir un resultado mínimamente correcto. Y, por supuesto, ni siquiera se llegó a considerar válido el desarrollar todas las posibles combinaciones como modelo de cuerpo entero, lo que sería una auténtica locura.

8. Conclusiones y trabajo futuro

El proyecto pecaba de ambicioso a nivel de diseño y complejidad, mientras que los medios eran escasos. Había poco músculo para afrontar tantos frentes con tan poco tiempo, y más necesitando todos ellos de un periodo de formación y documentación inicial que se ha quedado corto. La planificación no ha sido la mejor y el resultado final lo ha reflejado con claridad.

Pese a que no se haya logrado cumplir el objetivo del proyecto, sí que se ha demostrado que a partir de herramientas gratuitas al alcance de cualquier usuario se puede llevar a cabo el desarrollo de un juego desde cero, aunque la curva de aprendizaje obliga a muchas horas de documentación y preparación para afrontar según qué partes del desarrollo con éxito.

También cabe destacar que hay algunos apartados que están estrictamente ligados a unas competencias artísticas que no tienen por qué mejorar a base de muchas horas de práctica y esfuerzo. Aun así, los conocimientos adquiridos en el campo del modelado en tres dimensiones y su posterior animación no se pueden considerar en absoluto baladí.

Las bases y algunas mecánicas de lo que sería el juego propuesto inicialmente están definidas, y no sería difícil proseguir con su desarrollo a posteriori, por lo que se puede considerar que no ha sido un fracaso absoluto. Liberar responsabilidades a los modelos y animaciones para centrarse en la programación pura y dura sería una buena línea de futuro, aunque por ello se perdieran algunas de las intenciones originales.

9. Bibliografía

- [1] Tutoriales oficiales de Unity3D [en línea]. Disponible en: https://unity3d.com/es/learn/tutorials/modules
- [2] Tutoriales oficiales de Blender [en línea]. Disponible en: https://www.blender.org/support/tutorials/
- [3] *Blender addon: Low Poly Rocks* [en línea]. Disponible en: http://blenderaddonlist.blogspot.com.es/2013/11/addon-low-poly-rock.html
- [4] Tutorial de C Sharp [en línea]. Disponible en: http://www.tutorialspoint.com/csharp/
- [5] Documentación de Unity3D [en línea]. Disponible en: http://docs.unity3d.com/Manual/index.html
- [6] *Execution Order of Event Functions* [en línea]. Disponible en: http://docs.unity3d.com/es/current/Manual/ExecutionOrder.html

ANEXOS

ANEXO A: Manual de usuario

Tras ejecutar el archivo ejecutable y configurar la resolución (los controles no están gestionados, así que no es recomendable tocarlos), la pantalla del menú principal será la primera en aparecer.

Figura 17. Menú principal.

Se interactúa con los menús del juego a través del ratón. Las posibilidades no tienen misterio: crear nueva partida, cargar alguna existente, acceder a las opciones o salir de la aplicación.

Figura 18. Carga de partidas guardadas desde el menú principal.

Figura 19. Pantalla de opciones del menú principal.

Una vez el juego está en marcha, el sistema de controles es sencillo:

- Movimiento horizontal y vertical con las flechas direccionales.
- Ataque horizontal con la tecla Z.
- Ataque vertical con la tecla X.
- Acceder al menú de opciones con la tecla Escape.
- Aumentar/disminuir el zoom de la cámara con la rueda del ratón.
- Accionar palancas, puertas, portales y cofres con la tecla Espacio.

Figura 20. Situación de juego con el personaje en el escenario.

Con el botón de la esquina inferior izquierda se accede al inventario, desde el que se puede cambiar el equipamiento del personaje y comprobar su estado. Para salir del inventario hay que volver a pulsar el botón.

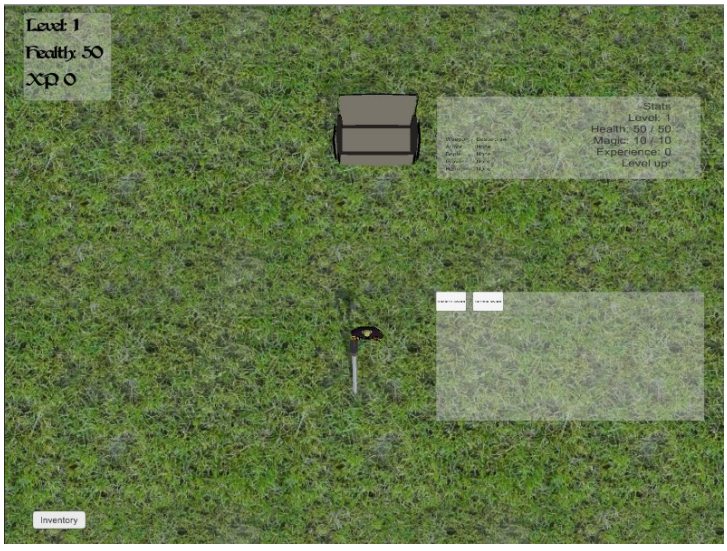

Figura 21. Situación de juego con el inventario abierto.

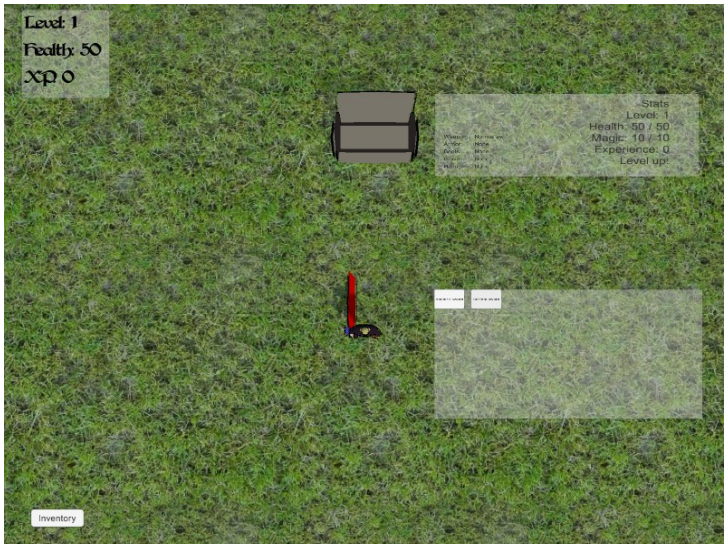

Figura 22. Arma cambiada desde el inventario tras haberse encontrad en un cofre.

Figura 23. Menú del juego.

Pulsando la tecla Escape se accede al menú, desde donde se puede guardar la partida en una de las tres ranuras existentes.

Cuando se pueda interactuar con algún elemento del escenario aparecerá algún texto descriptivo que así lo indique.

Figura 24. Situación de combate.

Si el personaje se queda sin vida, morirá y saltará la pantalla de fin del juego, desde la que se puede volver a comenzar la partida o cargar otra.

Figura 25. Pantalla de fin del juego.

El juego no tiene final al no haberse cumplido todos los objetivos marcados por el proyecto.

ANEXO B: Galería de imágenes

Figura 26, Modelo del personaje con un arma.

Figura 27. Modelo del enemigo Watcher.

Figura 28. Modelo del enemigo Watcher disparando.

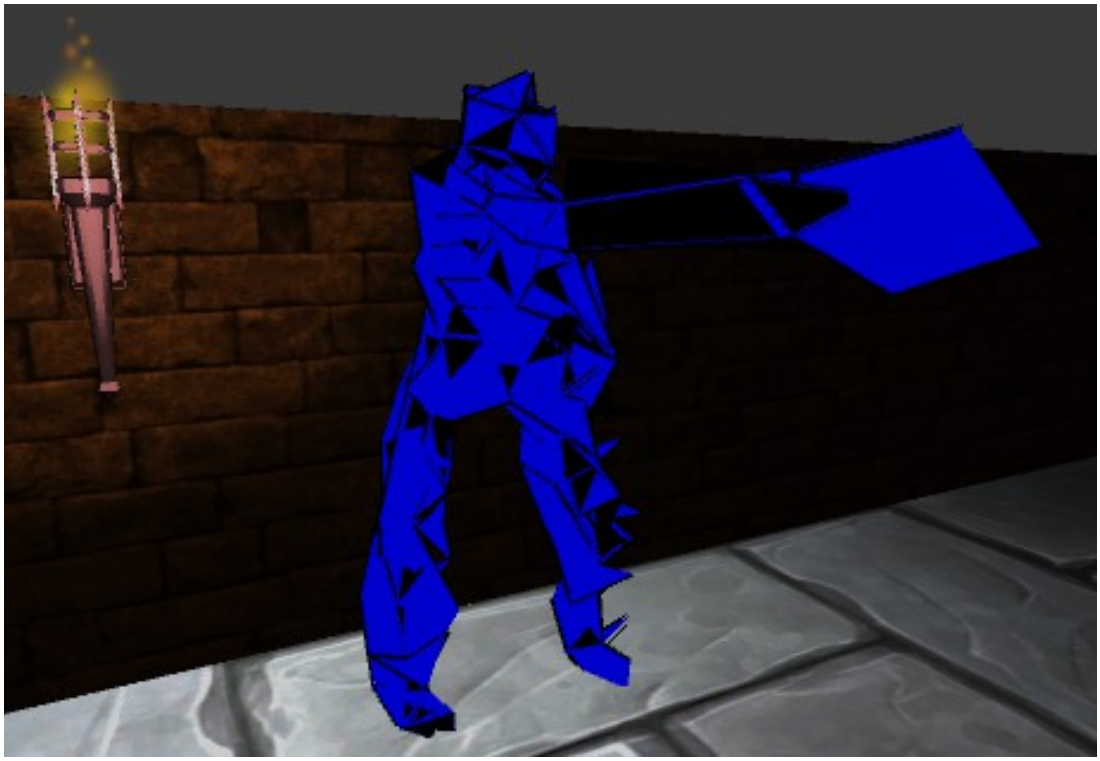

Figura 29. Modelado del enemigo Demon.

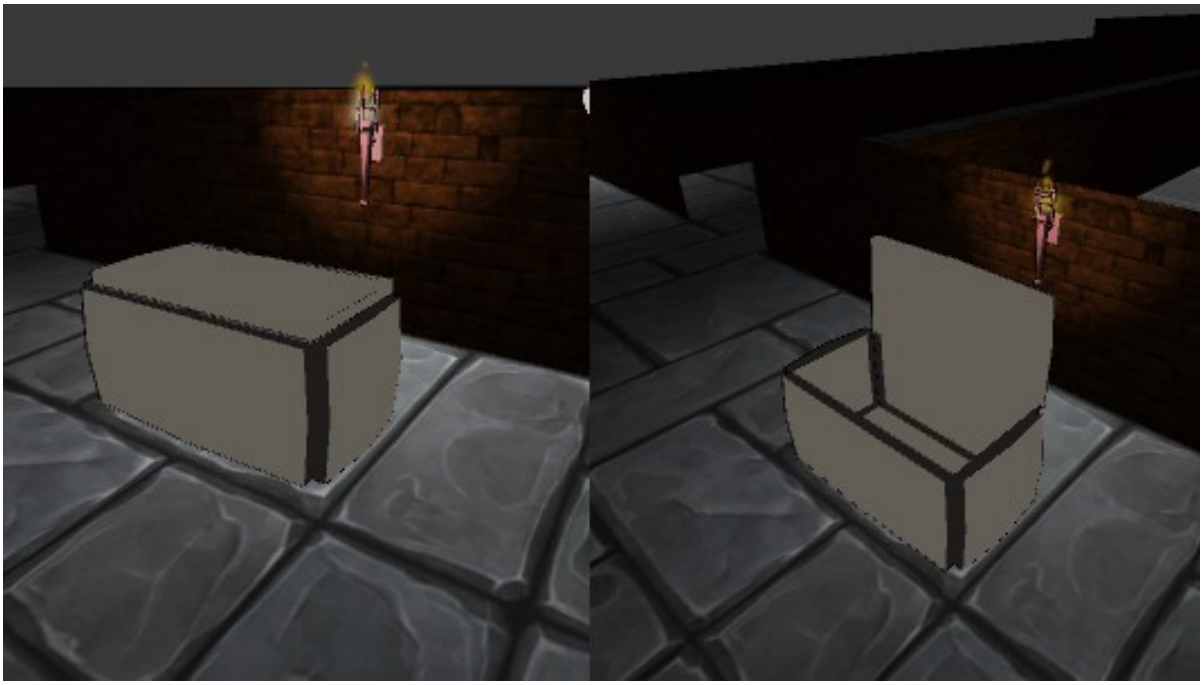

Figura 30. Cofre cerrado. Cofre abierto.

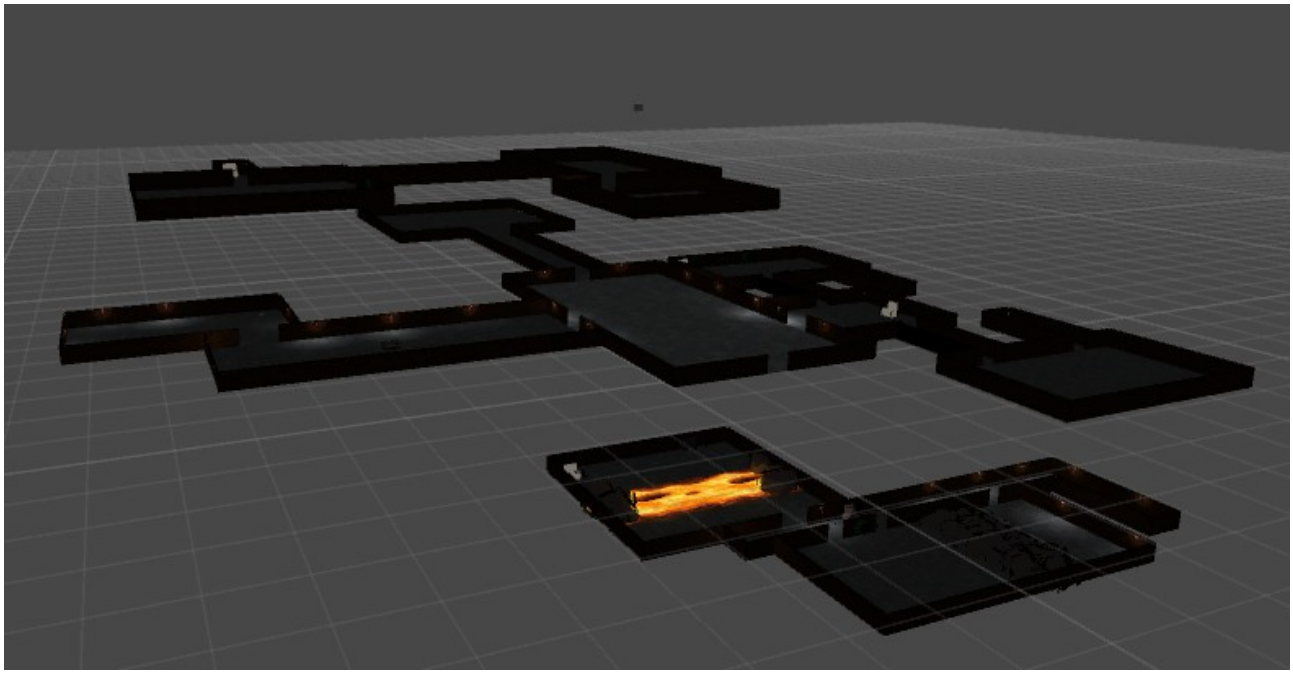

Figura 31. Vista de águila de la mazmorra.-

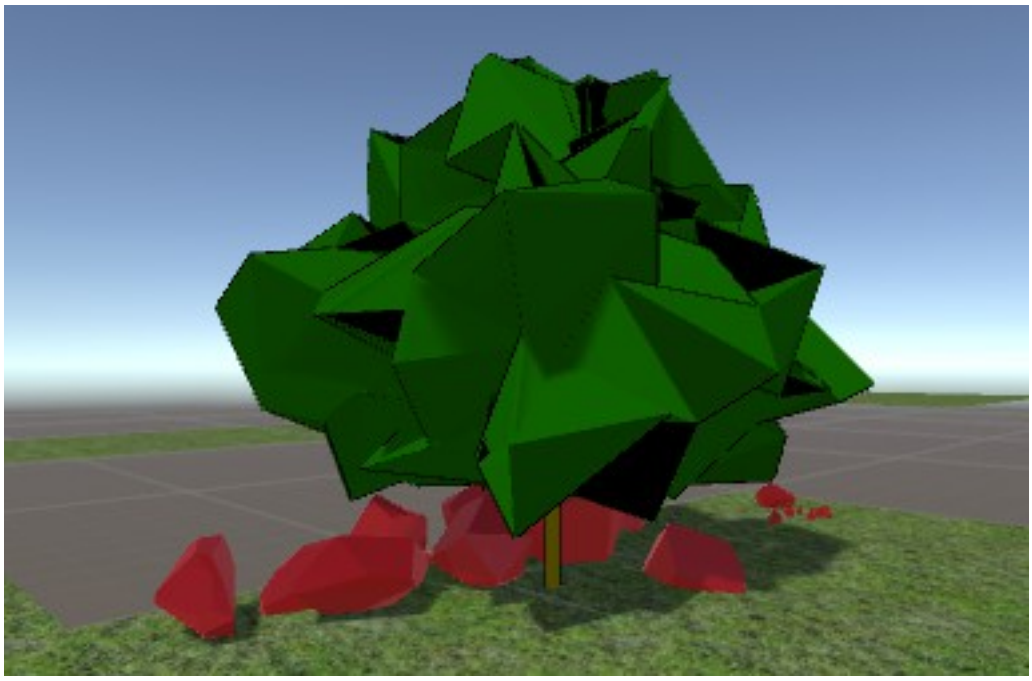

Figura 32.Rocas descansando bajo la sombra de un árbol.