UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

ESCUELA DE INGENIERIA INFORMATICA

Diseño y desarrollo de un videojuego en 3D basado en la defensa de una plataforma, haciendo énfasis en el estudio de controles para pantalla táctil de los dispositivos móviles.

Brayan Garzón Madroñero

Tutor: Agustín Trujillo Pino

Tutor: Antonio José Sánchez López

Las Palmas de Gran Canaria, Enero 2015

Agradecimientos

A la Universidad de Las Palmas de Gran Canaria, a la Escuela de Ingeniería Informática y a los docentes que han influido y participado durante mi formación a lo largo de esta carrera, que con su esfuerzo y dedicación han logrado transmitir su conocimiento y pasión por esta profesión.

A todos mis compañeros, aquellos que me han acompañado, con los que he compartido y disfrutado durante todos estos años.

A The Singular Factory y The Singular Social Games, por permitirme participar, introducirme y enseñarme el maravilloso mundo de los videojuegos.

A mis tutores Agustín Trujillo y Antonio José Sánchez por guiarme y facilitarme el conocimiento necesario para llevar a cabo este proyecto.

Y finalmente a mi familia, por estar siempre presente sin importar las circunstancias y darme su apoyo incondicional en cada momento, por creer en mí y permitirme seguir mis sueños.

Índice General

Tabla de contenido

Introducción

Actualmente nos encontramos en una sociedad rodeada de tecnología, donde la gran mayoría de las personas tiene un dispositivo móvil, dispositivo que en muchos casos ha pasado a ser una parte imprescindible de nuestras vidas y vayamos donde vayamos nuestro móvil está con nosotros. Esto nos lleva a que incluso nuestro tiempo de ocio o aquellos momentos donde simplemente tenemos que esperar por algo, lo pasemos mirando nuestro móvil revisando alguna red social, el correo o simplemente **jugando**.

Es en la última parte donde se pretende hacer énfasis con este proyecto, en el desarrollo de un videojuego donde se haga uso de las pantallas táctiles que nos ofrecen los dispositivos móviles, observando que es lo que espera el usuario, que le resulta más cómodo y lo más importante que le resulta divertido.

Motivaciones y Objetivos

Motivaciones

Una y quizá la más grande de las motivaciones es el hecho de que los videojuegos han ocupado una gran parte de mi vida, he pasado una gran cantidad de horas disfrutando y viviendo distintas aventuras, que hacen que te sumerjas en un mundo lleno de nuevas experiencias.

Otras de mis principales motivaciones son la curiosidad y las ganas de seguir aprendiendo, el querer despejar esas dudas de: ¿Cómo se hace? , ¿Qué debería hacer?, ¿De qué manera lo puedo hacer mejor?

Y finalmente, las ganas de usar todo lo aprendido durante los últimos años en un proyecto que me lleve largas horas de trabajo y en el que pueda disfrutar a la vez que aprender y aplicar los conocimientos adquiridos sobre ingeniería del software en el desarrollo de un videojuego.

Objetivos

En el desarrollo de este proyecto se pretenden cubrir o alcanzar los siguientes objetivos:

- Analizar y diseñar las distintas mecánicas del juego.
- Desarrollar y probar las mecánicas del juego.
- Analizar, diseñar e implementar los distintos tipos de controles.
- Estudiar los distintos componentes que influyen en el desarrollo de un videojuego.

Herramientas

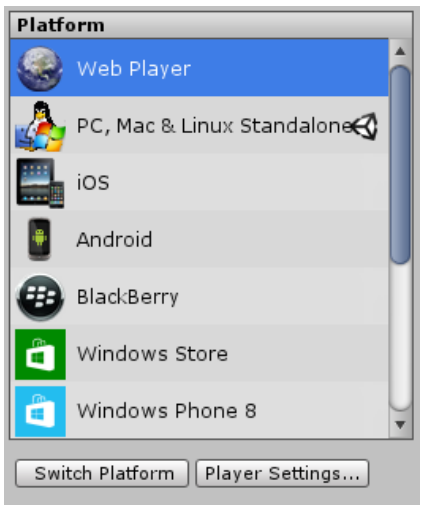
En este apartado mencionaré y explicaré un poco las distintas herramientas que fueron usadas para llevar a cabo el proyecto.

Unity 3D

El motor gráfico Unity 3D (de ahora en adelante lo llamaremos solo Unity) fue la principal herramienta usada durante el desarrollo de este proyecto, es un motor gráfico que permite fácil acceso a diverso contenido y una muy buena documentación, además posee una gran comunidad que facilita el aprendizaje y la resolución de problemas.

Unity tiene dos versiones, la versión free que fue la usada en este proyecto y la versión pro, que podemos adquirir pagando una cuota mensual o anual según conveniencia, la diferencia entre estas dos versiones es que la versión pro posee muchas más características que facilitan el desarrollo de un videojuego y permiten mejorar el rendimiento y calidad del mismo, tanto de manera gráfica como a nivel de programación, ofreciendo una gran cantidad de componentes.

Una de las principales razones para escoger este motor gráfico es su soporte multiplataforma, creamos el juego una vez y lo podemos exportar a un conjunto bastante amplio de plataformas y que cada vez va creciendo, aunque nos centraremos en las plataformas móviles.



Menú selección de plataforma.

Antes de empezar hay algunos términos que debemos tener claros ya que serán usados a lo largo del documento.

GameObjects

En Unity todos los elementos del juego son considerados GameObjects que tendrán uno o más componentes, como lo son, transform, colliders, rigidbodies, animators, mesh, materials, textures, shaders, terrains, scripts, etc, estos componentes se irán explicando más adelante a medida que se vea necesario.

Un GameObject con un conjunto de componentes con ciertos valores que nos ha costado equilibrar, puede llegar a ser bastante complejo y en ocasiones nos gustaría tener este mismo GameObject duplicado o incluso con pequeños cambios, es aquí donde aparecen los **Prefabs**.

Los **Prefabs**, como su nombre nos indica son GameObjects prefabricados, que nos permitirán acceder a un GameObject con ciertas características y componentes a modo de plantilla, del cual podremos crear tantas instancias como deseemos y modificarlas a gusto.

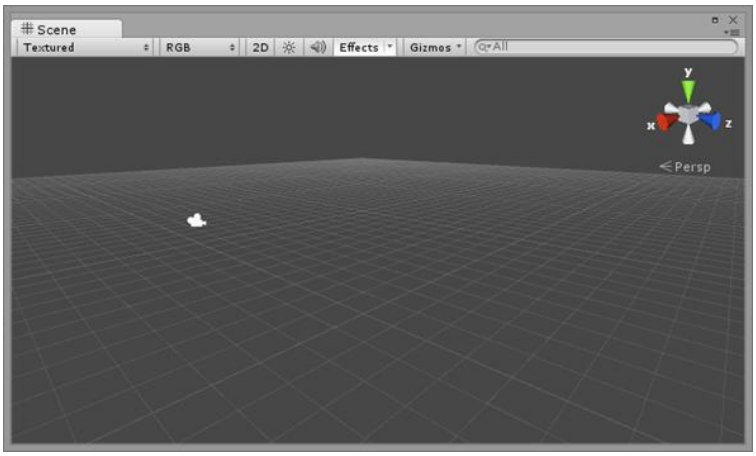
Para entender un poco más a fondo los prefabs imaginemos que en nuestro juego existirán un cierto tipo de enemigos, con la misma animación, mismo modelo 3D, mismo comportamiento, y todos los enemigos serán exactamente iguales, excepto que en algunos casos habrán unos más grandes que otros, bastará con crearnos un prefab de este enemigo e instanciarlo cuantas veces queramos y cuando queramos modificar algunos de ellos para que su tamaño varié lo haremos sobre las instancias, no sobre el prefab, ya que si modificamos el prefab modificaremos todas y cada una de las instancias.

Ventanas

Al iniciar Unity podremos observar distintas ventanas, a continuación se procederá a explicar algunas de ellas para que nos familiaricemos, ya que a lo largo del documento haremos referencia a algunas de ellas.

Scene

En Unity cada vista o cada escenario serán construidos en lo que llamaremos una **escena** y todos los elementos que se encuentran en ella serán **GameObjects.**



Ventana Scene

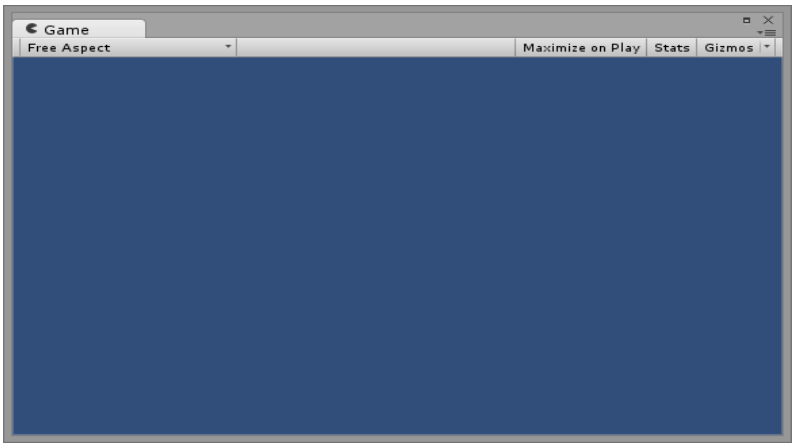
Como podemos observar en la imagen anterior, la escena no es más que un espacio con coordenadas donde iremos colocando todos los elementos de nuestro juego, con elementos nos referimos a los distintos GameObjects que podrán contener uno o más componentes, entre

estos elementos podemos identificar algunos como lo son, cámaras, personajes, enemigos, decoración, etc.

Este espacio o escenario puede estar en 3D o 2D, en nuestro caso emplearemos el modo de 3D.

Game

En la ventana Game podremos apreciar cómo se verá nuestro juego, se mostrara siempre lo que nuestra cámara principal está dibujando, es decir lo que se encuentra dentro de su campo de visión.



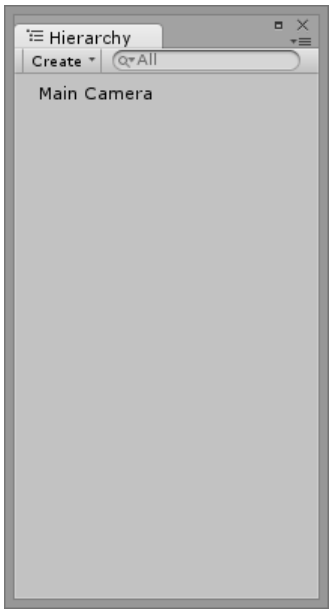
Ventana Game

Cada vez que creemos una escena nueva nos encontraremos con la ventana de Game y la ventana de Scene en el mismo estado, vacías, excepto por un único componente la Main Camera.

En la ventana de Game también podemos modificar la resolución de la vista, de esta manera podremos saber cómo se verá nuestro juego en distintas resoluciones y aspect ratios. Esta característica nos puede resultar muy útil a la hora de crear la interfaz gráfica de nuestro juego que es la que más se ve afectada en este aspecto.

Hirerarchy

Esta ventana nos permitirá saber que elementos se encuentran en nuestra escena actualmente, como ya mencioné previamente, en una escena vacía lo primero que veremos será la Main Camera.



Ventana Hirerarchy

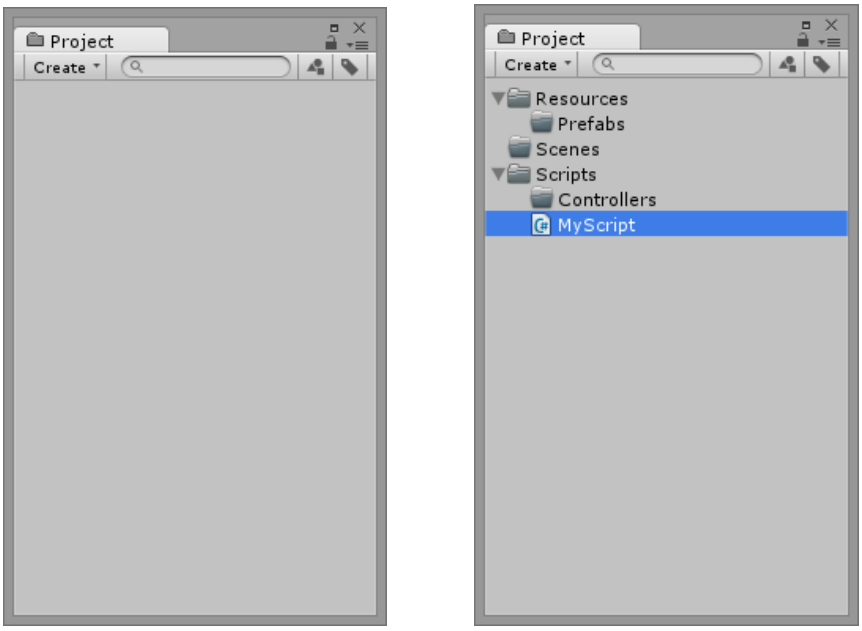
Cada una de nuestras escenas podrá ser tan grande como queramos, pero siempre debemos fijarnos a que dispositivos va orientado nuestro juego y que capacidad de procesamiento tienen estos dispositivos, ya que una escena con demasiados componentes, animaciones, partículas le exigirá mucho más y puede ocurrir que nuestro juego vaya muy lento.

Una forma de mantener nuestra Jerarquía de GameObjects ordenada es haciendo uso de los **Empty** GameObjects, que nos servirán como carpetas para agrupar y mantener una relación entre los distintos objetos.

Project

La ventana de Project nos mostrará todo el contenido de nuestra carpeta assets que es donde se encuentran todos los recursos que usaremos en el desarrollo del proyecto, al crear un nuevo proyecto esta ventana se encontrará vacía al igual que las demás, pero es recomendable generar una estructura de carpetas que iremos mejorando a medida que sigamos creando nuevos juegos.

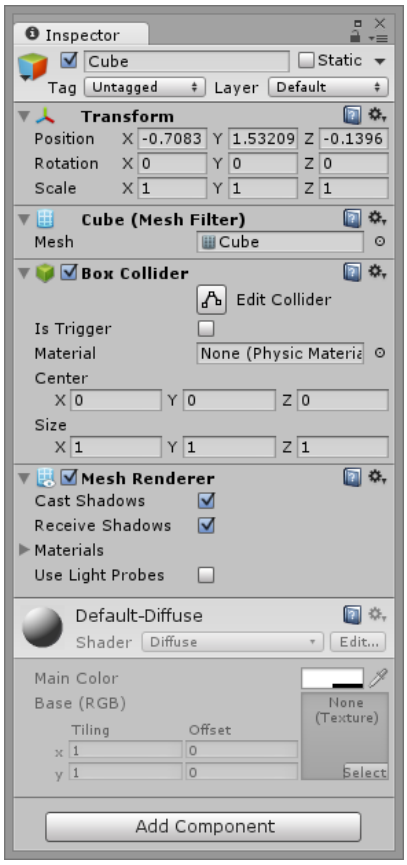
Para añadir algo nuevo al proyecto, por ejemplo una textura, basta con arrastrarla dentro de la ventana a la carpeta deseada.



Ventana Project

Unity soporta 3 Lenguajes de programación C#, Javascript y Boo, para crear un script de dicho lenguaje basta con hacer clic derecho > crear y nos aparecerá un menú de opciones con dichos lenguajes y un conjunto opciones para crear otro tipo de objetos dentro de Unity.

Inspector

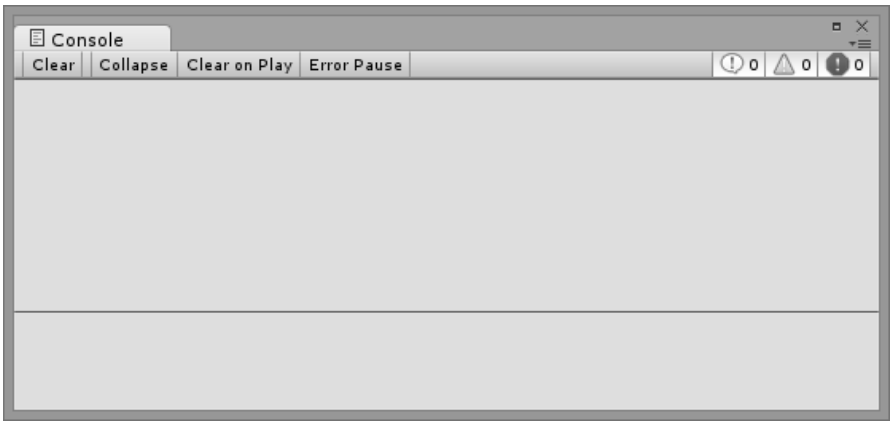


Ventana Inspector

En esta ventana veremos los distintos componentes del GameObject o del recurso de nuestro proyecto que tengamos seleccionado, también nos permitirá añadir más componentes y modificar sus distintas opciones para configurarlas a nuestro gusto.

Console

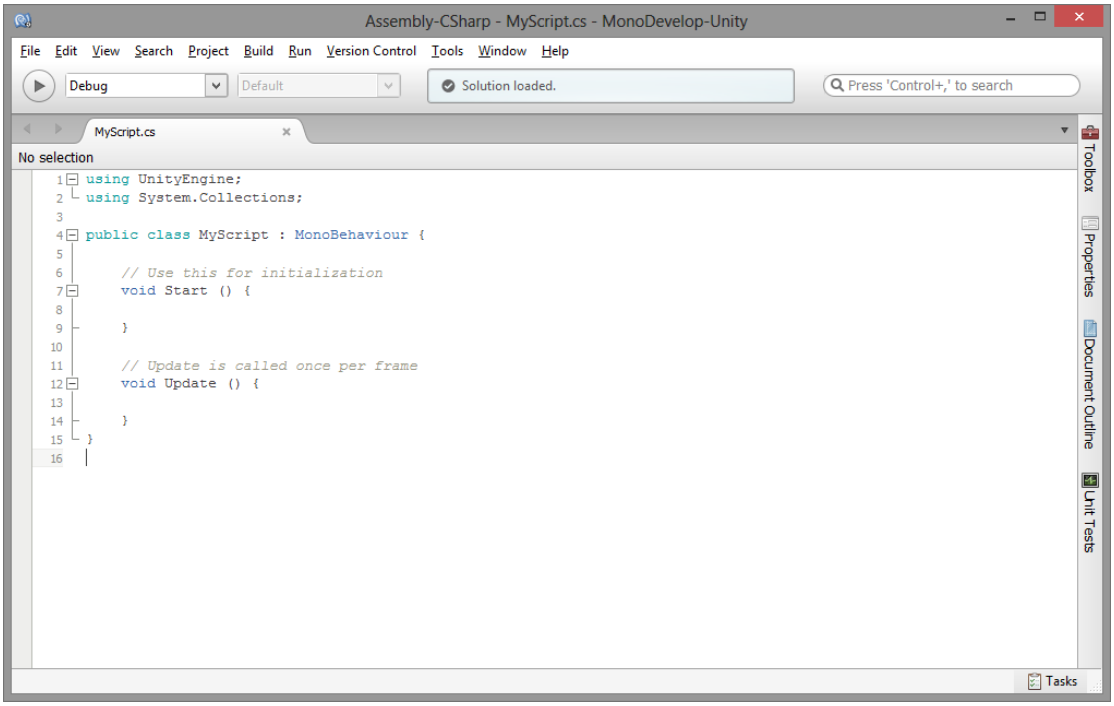
Por último, pero no menos importante, la consola, en esta ventana podremos ver los errores de compilación o de ejecución, alertas o mensajes que hayamos puesto que nos servirán en ocasiones como trazas para ver si el código se está ejecutando correctamente.



Ventana Console

Monodevelop

Monodevelop es el IDE por defecto que nos ofrece Unity para escribir nuestros scripts, aunque también podríamos usar el Visual Studio si disponemos de él.



Script en Monodevelop

Como vemos en la imagen anterior todos los scripts o clases que creemos en Unity, heredaran por defecto de la clase MonoBehaviour además tendrán dos métodos, Start y Update.

Con la clase MonoBehaviour podremos acceder a distintos métodos que nos ayudarán a implementar distintas funcionalidades, los métodos claves y a tener en cuenta son los siguientes:

- OnLevelWasLoaded
- OnEnable
- Awake
- Start
- Update
- LateUpdate
- FixedUpdate

Estos métodos se ejecutaran siempre en el mismo orden OnLevelWasLoaded> OnEnable> Awake> Start> Update> LateUpdate> FixedUpdate.

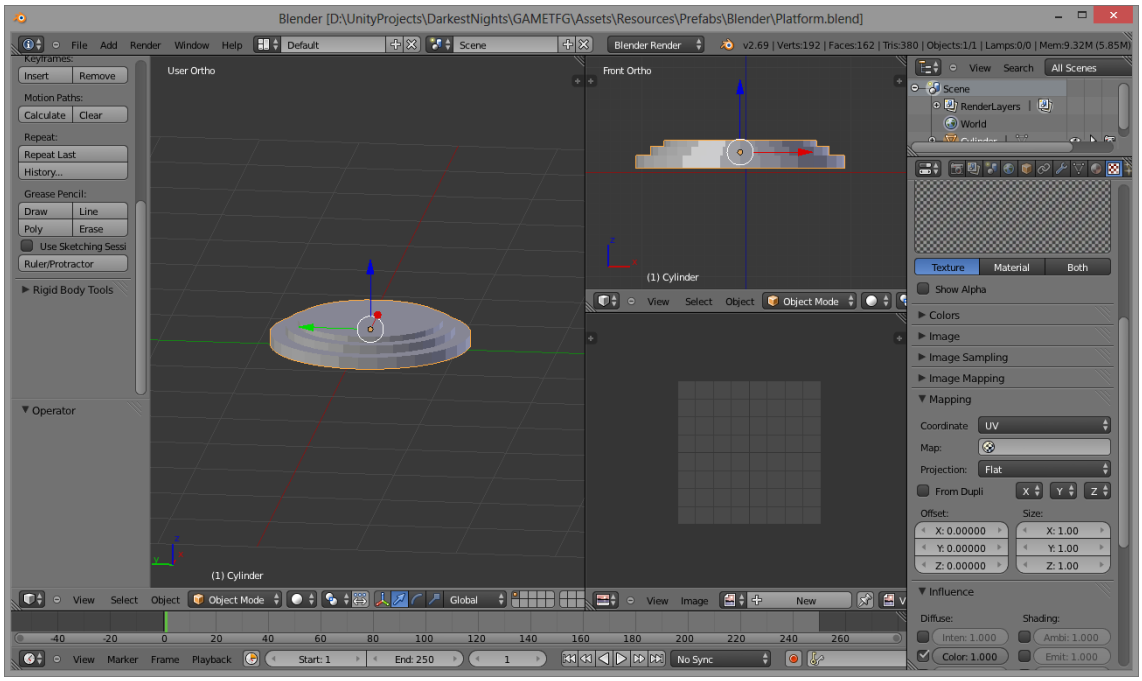
Así un gameObject que se encuentre dentro de nuestra escena y tenga un script asociado que contenga dichos métodos los ejecutara en ese orden. Debemos tener en cuenta que todos se ejecutaran una sola vez por cada gameObject, excepto por el Update y LateUpdate que se ejecutarán en cada fotograma y el FixedUpdate que lo hará cada cierto número de fotogramas.

Blender

Es una herramienta gratuita y muy potente para la creación y modelado en 3D, con un conjunto de características que la hacen muy versátil ofreciéndonos desde un renderizado foto realista y la creación de materiales hasta una gran cantidad de herramientas para poder esculpir esa figura o ese personaje que cobrara vida dentro de nuestro juego.

Además, blender se integra bastante bien con Unity, permitiendo guardar sus ficheros dentro del proyecto de Unity y generar distintos tipos de assets sin la necesidad de exportar e importar de un programa a otro como ocurre con otras herramientas de este estilo.

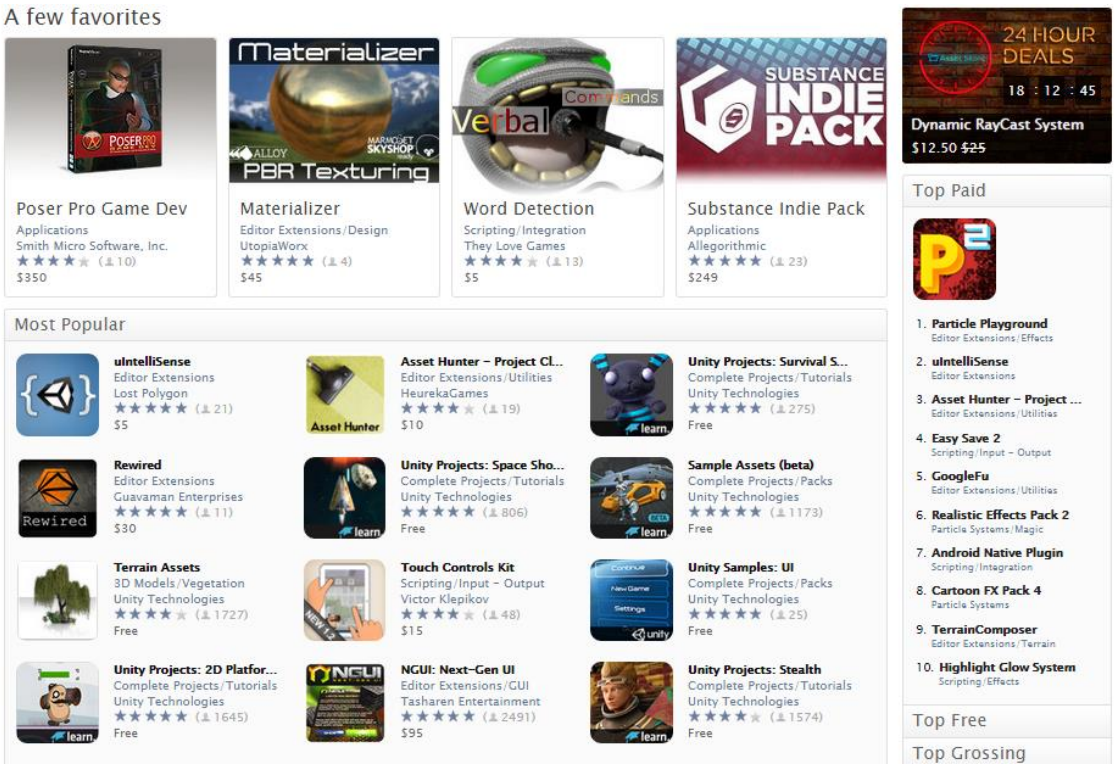
Además de permitir la creación de modelos, también permite crear su rigging (o esqueleto) que ayudara a que este modelo pueda ser animado fácilmente, blender también nos facilita la creación de animaciones desde su propio entorno.



Interfaz de blender

Posee varias ventanas para la edición de las mallas, donde podremos modificar cada arista, vértice o cara del modelo 3D en el que estemos trabajando e incluso podremos tener distintos puntos de vista del mismo modelo al mismo tiempo.

Asset Store



Aunque no es una herramienta considero que debo mencionar la Asset Store que posee Unity, la Asset Store no es más que un sitio donde podremos encontrar distintos recursos que nos facilitarán el desarrollo de nuestro videojuego, ofreciéndonos desde un simple pack de modelos 3D de árboles, hasta proyectos enteros y plugins para toda clase de idea que se nos pueda ocurrir, algunos de ellos gratuitos y con una calidad más que aceptable otros aunque no son gratuitos tendrán un precio asequible.

Para este proyecto ha sido la fuente principal de recursos ya que de aquí se han seleccionado un número de modelos 3D con sus animaciones, que nos facilitarán la creación de enemigos y darán más riqueza al juego, el modelo 3D del personaje, sistemas de partículas que servirán para la creación de las magias, etc.

Competencias

A continuación se detallan algunas competencias cubiertas durante el desarrollo del proyecto.

- **G1.Poseer y comprender conocimientos en un área de estudio (Ingeniería Informática) que parte de la base de la educación secundaria general, y se suele encontrar a un nivel que, si bien se apoya en libros de texto avanzados, incluye también algunos**

aspectos que implican conocimientos procedentes de la vanguardia de su campo de estudio.

En este proyecto se cubre esta competencia ya que en él se hace uso de distintas técnicas de ingeniería del software aplicadas a un campo específico como lo es el desarrollo de un videojuego.

- **G3.Reunir e interpretar datos relevantes (normalmente dentro de su área de estudio) para emitir juicios que incluyan una reflexión sobre temas relevantes de índole social, científica o ética.**

Con el desarrollo de este trabajo se intenta conseguir a partir de la experimentación con usuarios y observando su comportamiento, un tipo de controles que resulten cómodos a la hora de jugar un juego de este estilo, analizando lo que funciona y lo que no de cada uno de ellos.

- **G4.Transmitir información, ideas, problemas y soluciones a un público tanto especializado como no especializado.**

La idea del proyecto es desarrollar un videojuego a la vez que intentar conseguir un tipo de control que resulte fácil e intuitivo para todo tipo de usuarios, experimentados y novatos.

- **G5.Desarrollar aquellas habilidades de aprendizaje necesarias para emprender estudios posteriores con un alto grado de autonomía.**

Al inicio de este proyecto mis conocimientos sobre el desarrollo de videojuegos eran escasos, así que este trabajo ha supuesto un proceso de aprendizaje, donde cada día que pasaba era un conjunto nuevo de conocimientos adquiridos.

- **N3. Contribuir a la mejora continua de su profesión así como de las organizaciones en las que desarrolla sus prácticas a través de la participación activa en procesos de investigación, desarrollo e innovación.**

Como se ha comentado anteriormente la finalidad de este proyecto es implementar, evaluar y analizar distintos tipos de controles, probando nuevas formas que varían la jugabilidad para al final conseguir uno que se adecue correctamente.

- **T8. Conocimiento de las materias básicas y tecnologías, que capaciten para el aprendizaje y desarrollo de nuevos métodos y tecnologías, así como las que les doten de una gran versatilidad para adaptarse a nuevas situaciones. (G3, N3).**

En este proyecto se han utilizado distintas técnicas para el desarrollo de un software sostenible intentando mantener un código fácil y sencillo de entender a la vez que la tecnología de las pantallas táctiles de los móviles. Con el estudio realizado de los distintos tipos de controles se ha conseguido un conocimiento que facilitará la implementación de nuevos juegos de este estilo y que implementen un tipo de control parecido.

- **TFG01. Ejercicio original a realizar individualmente y presentar y defender ante un tribunal universitario, consistente en un proyecto en el ámbito de las tecnologías específicas de la Ingeniería en Informática de naturaleza profesional en el que se sinteticen e integren las competencias adquiridas en las enseñanzas**

Esta competencia será completada el día que se defienda ante el tribunal asignado, pero sí, es una de las principales competencias a cumplir con este proyecto.

Análisis y Diseño del juego

El análisis y diseño del juego se han realizado para lo que sería una primera versión del juego Darkest Nights dentro de la empresa The Singular Social Games, pero el proyecto no abarca esta primera versión del juego en su totalidad, el proyecto pretende centrarse en el desarrollo de un videojuego donde procuraremos encontrar un tipo de control que resulte fácil e intuitivo haciendo uso las pantallas táctiles de los móviles.

Ficha del juego

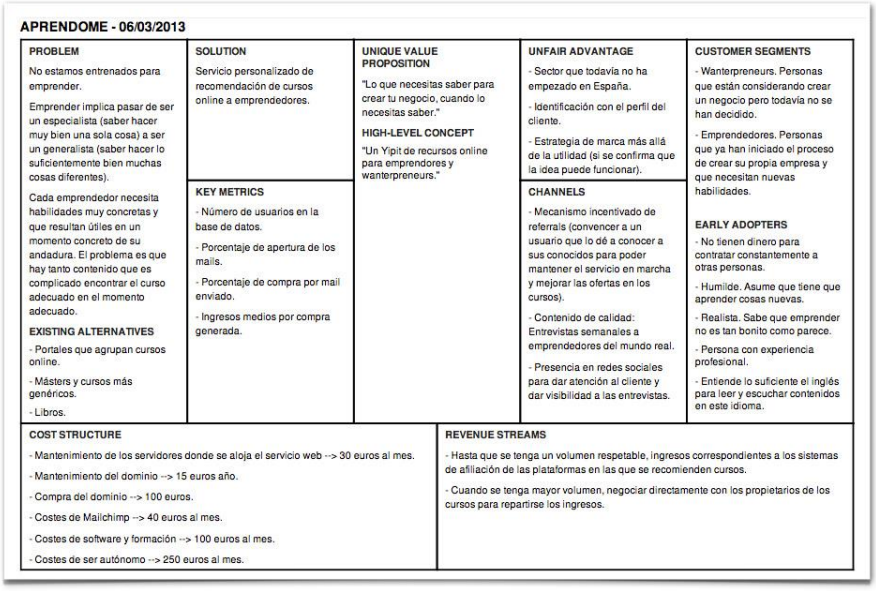
Darkest Nights pretende ser un juego social donde predomine la dominación territorial, los jugadores pertenecerán a una facción, luz u oscuridad, e irán conquistando distintos puntos geo posicionados que les permitirá ampliar el control de su facción. Para conquistar dichos puntos deberán enfrentarse a un reto, sobrevivir a un grupo de oleadas de la facción contraria mientras defienden una plataforma, o mientras intentan conquistarla, haciendo uso de objetos, magias y cualquier recurso que esté disponible.

Análisis

Para empezar el desarrollo del videojuego, se decidió hacer un análisis previo donde se identificó un conjunto de factores que afectaron el diseño y la implementación del mismo.

Canvas Gamificado

Para realizar este análisis se hizo uso de un canvas similar al del método Lean pero en su versión Gamificada.



APRENDOME - 06/03/2013

PROBLEM

No estamos entrenados para emprender.

Emprender implica pasar de ser un especialista (saber hacer muy bien una sola cosa) a ser un generalista (saber hacer lo suficientemente bien muchas cosas diferentes).

Cada emprendedor necesita habilidades muy concretas y que resultan útiles en un momento concreto de su andadura. El problema es que hay tanto contenido que es complicado encontrar el curso adecuado en el momento adecuado.

EXISTING ALTERNATIVES

- Portales que agrupan cursos online.
- Másters y cursos más genéricos.
- Libros.

SOLUTION

Servicio personalizado de recomendación de cursos online a emprendedores.

KEY METRICS

- Número de usuarios en la base de datos.
- Porcentaje de apertura de los mails.
- Porcentaje de compra por mail enviado.
- Ingresos medios por compra generada.

UNIQUE VALUE PROPOSITION

"Lo que necesitas saber para crear tu negocio, cuando lo necesitas saber."

HIGH-LEVEL CONCEPT

"Un Yipit de recursos online para emprendores y wanterpreneurs."

UNFAIR ADVANTAGE

- Sector que todavía no ha empezado en España.
- Identificación con el perfil del cliente.
- Estrategia de marca más allá de la utilidad (si se confirma que la idea puede funcionar).

CHANNELS

- Mecanismo incentivado de referrals (convencer a un usuario que lo dé a conocer a sus conocidos para poder mantener el servicio en marcha y mejorar las ofertas en los cursos).
- Contenido de calidad: Entrevistas semanales a emprendedores del mundo real.
- Presencia en redes sociales para dar atención al cliente y dar visibilidad a las entrevistas.

CUSTOMER SEGMENTS

- Wanterpreneurs. Personas que están considerando crear un negocio pero todavía no se han decidido.
- Emprendedores. Personas que ya han iniciado el proceso de crear su propia empresa y que necesitan nuevas habilidades.

EARLY ADOPTERS

- No tienen dinero para contratar constantemente a otras personas.
- Humilde. Asume que tiene que aprender cosas nuevas.
- Realista. Sabe que emprender no es tan bonito como parece.
- Persona con experiencia profesional.
- Entiende lo suficiente el inglés para leer y escuchar contenidos en este idioma.

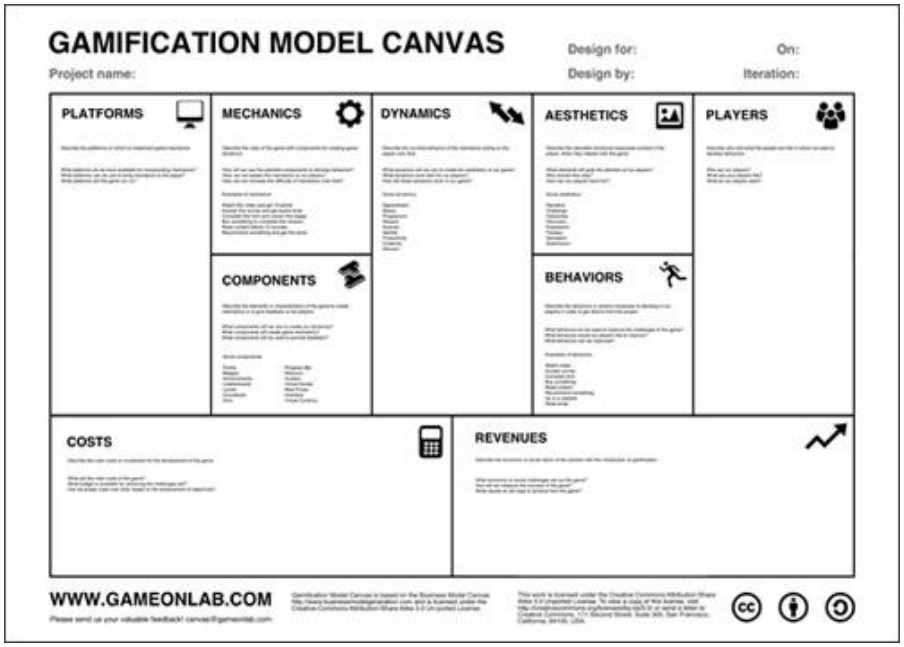
COST STRUCTURE

- Mantenimiento de los servidores donde se aloja el servicio web --> 30 euros al mes.
- Mantenimiento del dominio --> 15 euros año.
- Compra del dominio --> 100 euros.
- Costes de Mailchimp --> 40 euros al mes.
- Costes de software y formación --> 100 euros al mes.
- Costes de ser autónomo --> 250 euros al mes.

REVENUE STREAMS

- Hasta que se tenga un volumen respetable, ingresos correspondientes a los sistemas de afiliación de las plataformas en las que se recomienden cursos.
- Cuando se tenga mayor volumen, negociar directamente con los propietarios de los cursos para repartirse los ingresos.

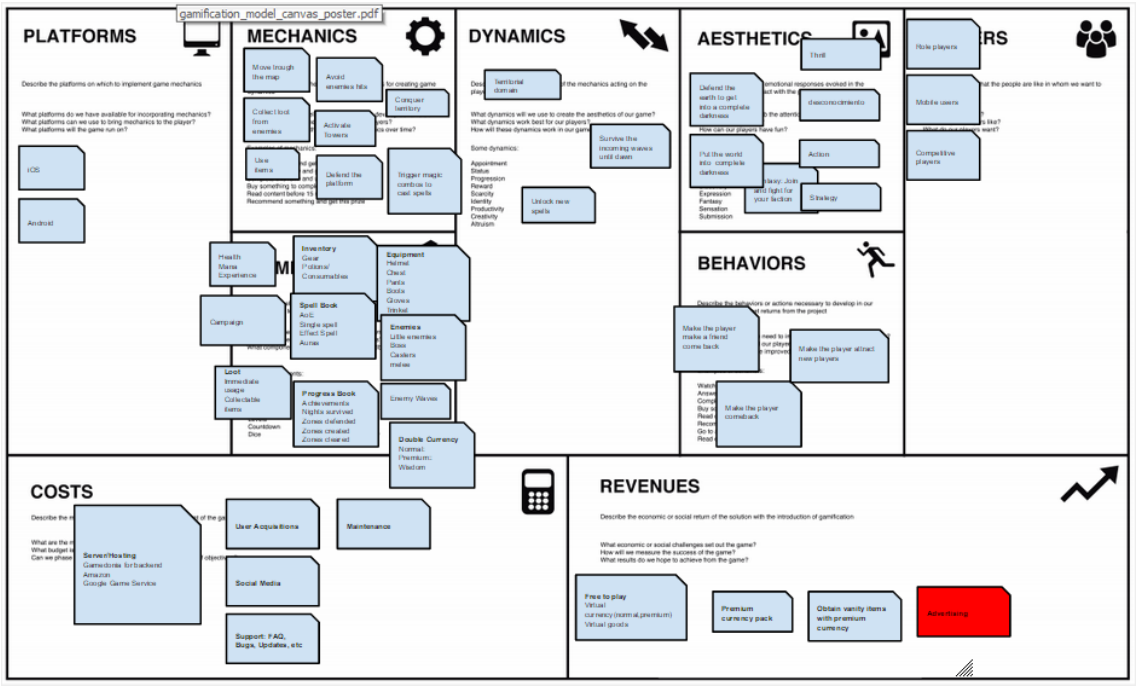
Canvas en método Lean



Canvas Gamificado

En esta Gamificación del canvas se cambian los títulos de distintas casillas sustituyéndolas por: Plataformas, Mecánicas, Componentes, Dinámicas, Estéticas, Comportamientos y Jugadores.

En cada una de estas secciones definimos un conjunto de características o ideas que nos ayudarán a enfocar nuestro trabajo y facilitar la toma de decisiones.



Canvas gamificado Darkest Nights

Dejándonos al final con una visión general de lo que queremos y de lo que esperamos conseguir con este juego.

Para completar cada una de las distintas secciones deberemos responder a varias preguntas, a continuación se explicarán algunas de estas secciones.

Mecánicas y Dinámicas

Aunque en el canvas se muestran en secciones distintas están muy relacionadas entre sí, ya que las mecánicas son aquellas acciones, comportamientos, técnicas y mecanismos de control que se utilizan para convertir en juego una actividad. Todas estas mecánicas juntas crean una experiencia para el jugador. Mientras que las dinámicas son ese efecto que se desea conseguir en el usuario, la motivación y el deseo de seguir experimentando a través de las distintas mecánicas.

Algunas preguntas que deberemos hacernos para completar la sección de mecánicas son:

- ¿Cómo usaremos los componentes seleccionados para crear un comportamiento?
- ¿Cómo podemos explicarles las mecánicas a nuestros jugadores?
- ¿Cómo podemos aumentar la dificultad de las mecánicas con el tiempo?

Ejemplos de mecánicas:

- Compra un objeto y obtén 10 puntos.
- Recomienda esto para obtener un premio.
- Contesta esto y obtén un nivel.

Y para la sección de dinámicas:

- ¿Qué dinámicas podemos usar para crear la estética de nuestro juego?
- ¿Qué dinámicas funcionan mejor con nuestros jugadores?
- ¿Cómo funcionan estas dinámicas en nuestro juego?

Ejemplos de dinámicas:

- Mejor estatus.
- Saciedad.
- Recompensa.
- Creatividad.
- Progresión.

Componentes

Son los elementos o características del juego que nos permitirán crear mecánicas o dar un feedback al jugador.

Las preguntas que deberemos hacernos para esta sección son:

- ¿Qué componentes nos permitirán crear dinámicas?
- ¿Qué componentes nos permitirán crear mecánicas de juego?

- ¿Qué componentes usaremos para dar feedback al jugador?

Ejemplos de componentes:

- Puntuación.
- Medallas.
- Logros.
- Niveles.

Comportamientos

Son las acciones que queremos nuestros jugadores realicen cuando interactúen con el juego.

Estética

En estética describiremos las sensaciones que queremos tengan los jugadores al interactuar con nuestro juego.

- ¿Por qué deberían jugar?
- ¿Cómo se pueden divertir nuestros jugadores?
- ¿Qué les debería llamar la atención?

Ejemplos de estéticas:

- Fantasía
- Reto
- Exploración
- Narrativo

Jugadores

Los jugadores son el público al que va dirigido nuestro juego, y tener claro quiénes es, es una parte fundamental en este proceso, ya que de esto dependerá en gran medida el desarrollo del juego.

Debemos siempre tener esta sección presente y responder a lo siguiente:

- ¿Quiénes son nuestros jugadores?
- ¿Qué es lo que esperan los jugadores?
- ¿Cómo son nuestros jugadores?

Poner una cara e imaginarnos su personalidad nos ayuda a ponernos en su posición y nos permite mantener siempre una mente abierta a la hora de tomar decisiones.

Plataformas

Por último pero no menos importante, las plataformas, para completar esta casilla deberemos responder a las siguientes preguntas

- ¿En qué plataformas se ejecutara el juego?
- ¿Qué plataformas tenemos disponibles para incorporar las mecánicas?

Esto nos ayudará a definir con que tecnología podemos contar y cuáles son nuestros recursos para llevar a cabo el desarrollo del videojuego.

De esta manera se completaron las distintas secciones del canvas quedando cada una de ellas con la siguiente información.

Jugadores

- Usuarios de móviles.
- Jugadores de rol.
- Jugadores competitivos.

Estéticas

- Desconocimiento
- Thrill.
- Acción.
- Estrategia.
- Fantasía.

Comportamientos

- Hacer que el jugador regrese al juego.
- Hacer que el jugador atraiga más jugadores.
- Hacer que el jugador haga regresar a otros jugadores.

Dinámicas

- Desbloquear nuevos hechizos
- Sobrevivir oleadas
- Dominio territorial.

Mecánicas

- Moverse a través del mapa.
- Evitar ataques de enemigos.
- Conquistar territorio.
- Usar objetos.
- Defender una plataforma.

- Activar torres.
- Recoger objetos.

Componentes

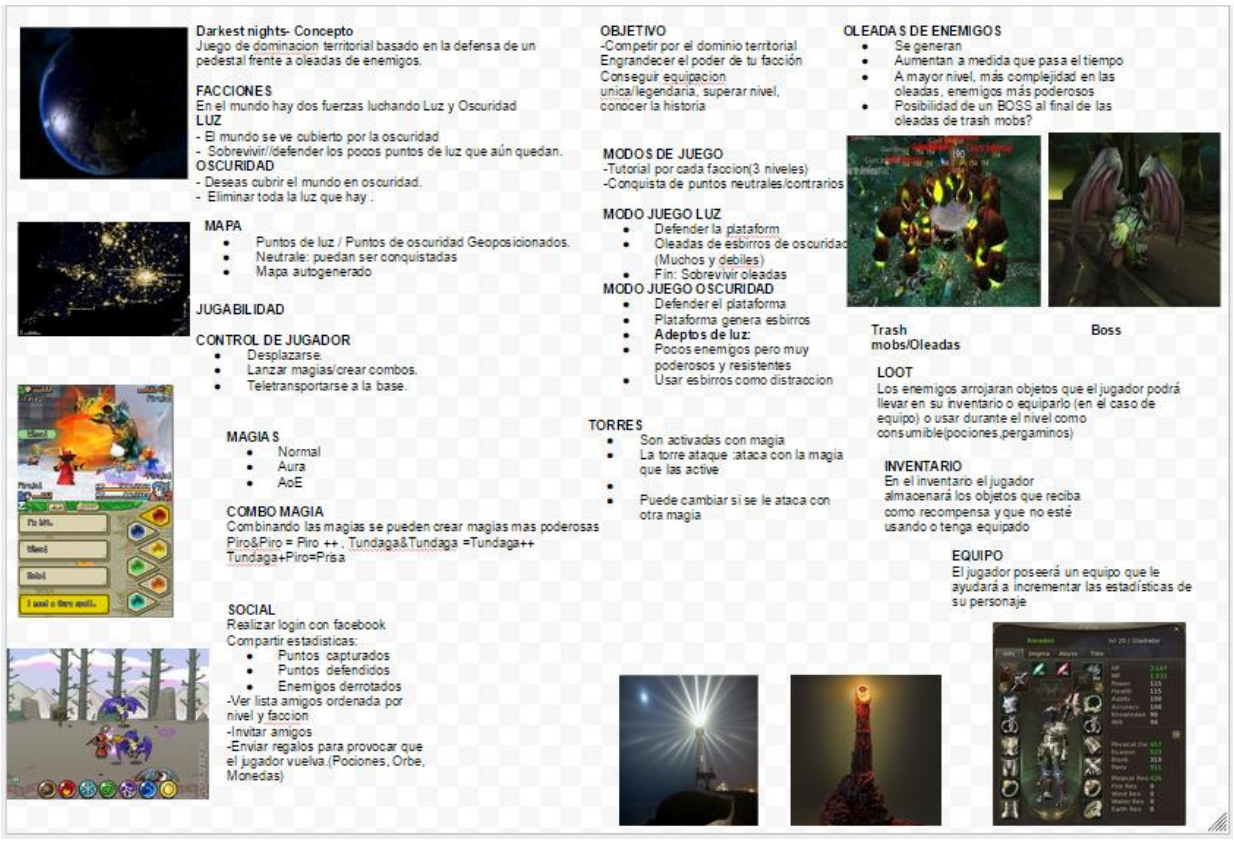
- Vida.
- Mana.
- Experiencia.
- Enemigos.
- Oleadas de enemigos.
- Monedas InGame.
- Libro de hechizos.
- Inventario.

Plataformas

- iOS.
- Android.

Ficha de concepto

Para tener una visión más cercana y clara de lo que se quiere, se realizó también una ficha de concepto donde se reunía toda la información relevante, tomando como ejemplos y fuentes de inspiración otros juegos con características similares.



Ficha concepto Darkest Nights.

Como podemos ver, se toman imágenes de referencia que aunque no conservan el mismo estilo nos sirven para explicar distintos apartados de la ficha.

Dentro de la ficha se definieron los siguientes apartados que se consideraron eran los más importantes para el análisis y posterior diseño del juego.

Concepto

Una pequeña definición sobre en qué consiste el juego.

Objetivo

Las acciones principales que debe llevar a cabo el jugador.

Facciones

Una breve descripción sobre las distintas facciones que podemos encontrar, en este caso solo dos, Luz y Oscuridad, especificando las intenciones de cada una.

Modos de juego

Son las distintas formas que el usuario tiene para interactuar con el juego y variarán en función de la facción que escoja

Mapa

Ya que se trata de un juego con puntos geo posicionados nos hacía falta mencionar que elementos debería contener el mapa, se definieron los puntos de luz, oscuridad y neutrales.

Control de jugador

En principio se definió un comportamiento por defecto donde lo único que se esperaba era que el jugador se moviera por el mapa, y lanzara hechizos para derrotar a sus enemigos y defender el pedestal.

Magias

Como se comentó anteriormente el jugador hará uso de hechizos para derrotar a sus enemigos, se definieron 3 categorías de hechizos:

Normales, que afectarán a un solo objetivo.

Con efecto de área, que afectarán a varios objetivos.

Y Auras, que proporcionaran beneficios al jugador.

Torres

Dentro de cada nivel habrá torres que ayudarán al jugador a defender la plataforma. Que se podrán activar con los hechizos del jugador.

Oleadas de enemigos

Durante cierto periodo de tiempo, vendrán varias oleadas de enemigos para romper nuestra plataforma.

Estas oleadas estarán compuestas de distintos tipos de enemigos que irán incrementando en número y dificultad a medida que avance el tiempo.

Loot

Serán objetos que el jugador podrá despojar de los enemigos que ha derrotado y que podrá usar para su beneficio, pociones, pergaminos, etc.

Inventario

Será un sistema que nos permitirá almacenar los distintos objetos que hayamos obtenido a lo largo del juego como pociones y pergaminos, que podremos usar para nuestro beneficio.

Social

Se definieron también un conjunto de operaciones que permitirán conectar más fácilmente a los usuarios. Como realizar login con Facebook, ver su lista de amigos, ver los puntos conquistados de su facción, compartir estadísticas, etc.

Diseño

Esta parte la enfocaremos más al diseño de software, también se mencionaran las curvas de interés que resultan convenientes tener presente a la hora de diseñar un videojuego.

Primero me gustaría mencionar que Unity utiliza una **arquitectura basada en componentes**, donde los elementos de nuestro juego se crean con GameObjects a los que uniendo distintos componentes como son los colliders, riggidbodies, mesh renderer etc., nos permite crear un objeto más complejo.

Debido a lo anterior y al ser el primer proyecto que realizaba en Unity, en algunas ocasiones me encontraba con dudas acerca de cómo debería implementar una funcionalidad, que estructuras de clases seguir ¿era en realidad necesario utilizar una estructura de clases?

La segunda opción era crear scripts que simplemente fueran pequeños comportamientos reutilizables y que no dependieran entre sí, por ejemplo se tendrían distintos scripts para un sistema de salud, caminar, disparar, saltar o morir.

Seguro que tener este tipo de componentes reutilizables puede resultar bastante cómodo, ya que si imaginamos que tenemos que crear distintos tipos de enemigos y entre estos tenemos enemigos que vuelan y otros que no, bastaría con cambiar su componente de movimiento. Pero surgía un nuevo inconveniente, ¿cómo hacer que estos scripts se comunicaran entre sí en determinado momento?

Para solucionar este problema Unity nos ofrece un método llamado **SendMessage (string name)** que realiza un broadcast a los scripts del gameobject sobre el que se ha lanzado e invoca al método que coincida con la cadena pasada como parámetro. Hacer uso de esto, para mí como programador y con los conocimientos que tengo sobre ingeniería del software, era una aberración, pero es una herramienta más que nos ofrece Unity. Por esta razón opte en un principio por hacer una estructura basada en una jerarquía.

El hecho de decidir si usar una u otra, no las hace exclusivas entre sí, de hecho en algunos momentos y para ciertos casos concretos era conveniente realizar un pequeño script en lugar de crearnos una estructura completa de clases, ya mencionaremos estos casos más adelante.

Diagrama de clases inicial

Con el análisis que habíamos realizado y una vez habíamos tomado la decisión de crear una arquitectura basada en una jerarquía, nos quedaba identificar las distintas clases necesarias para el desarrollo del videojuego.

Siguiendo el análisis anterior podíamos identificar las siguientes clases:

- Jugador.
- Enemigo.
- Torre.
- Plataforma.
- Magia.

Y como era obvio necesitaríamos un par de controladores para gestionar la lógica del juego:

- Controlador de juego
- Controlador de nivel
- Controlador de GUI

Tendiendo estas clases como base podríamos empezar el desarrollo del juego para ir poco a poco enriqueciendo y complementando la estructura.

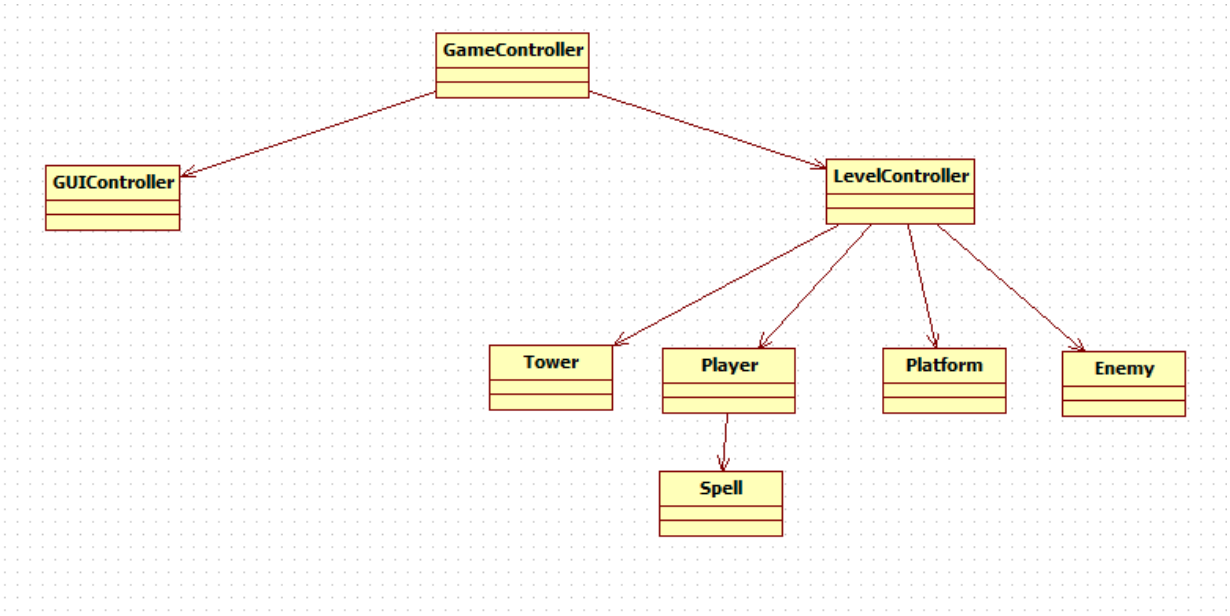
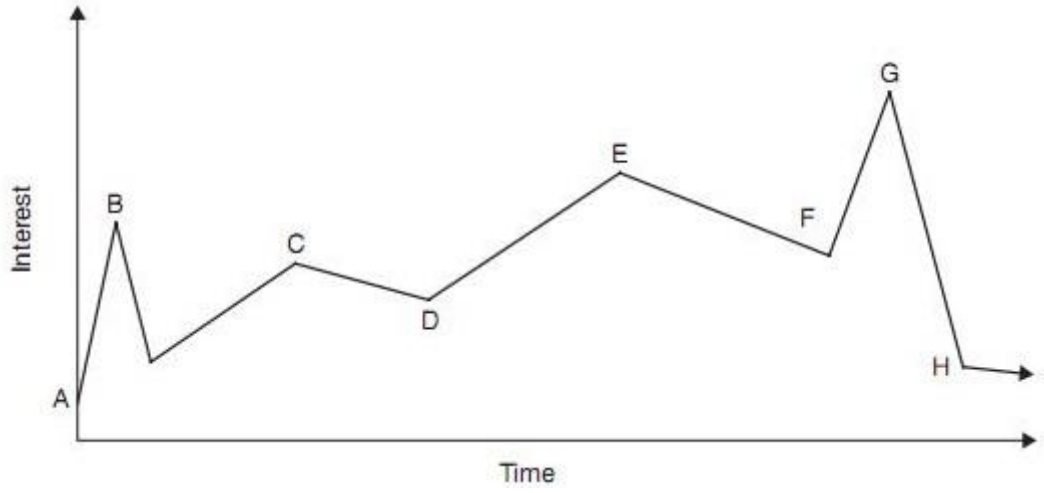


Diagrama de clases inicial

Curvas de interés



Curva de interés

Con las curvas de interés, lo que pretendemos transmitir es un conjunto de sensaciones que mantengan al jugador atrapado ofreciéndole en distintos instantes de tiempo emociones variadas. Suponemos también que si el jugador ha entrado por primera vez es porque ya tiene un punto inicial de interés. Partiendo de este punto inicial queremos que el jugador se enganche, y para lograr esto deberemos mostrarle algo que lo motive a quedarse, una cinemática con muchos efectos, una historia interesante o algo que genere un pico, que si nos fijamos en la gráfica se ve representado por el punto B.

A partir de ahí deberemos jugar con el interés del jugador y crear nuestra curva de interés. No es necesario y de hecho no es conveniente mantener al jugador en un estado que genere demasiada excitación o visto desde el punto de vista de una curva que tenga muchos picos

seguidos. Es recomendable que existan valles e incluso que su interés baje para darle un momento de descanso y relajación.

Para finalizar una curva de interés y quizá una de las partes más difíciles, es tener ese momento de clímax que le da un cierre a la historia o al juego, teniendo como objetivo final que el jugador se sienta realizado, en la gráfica los puntos G y H.

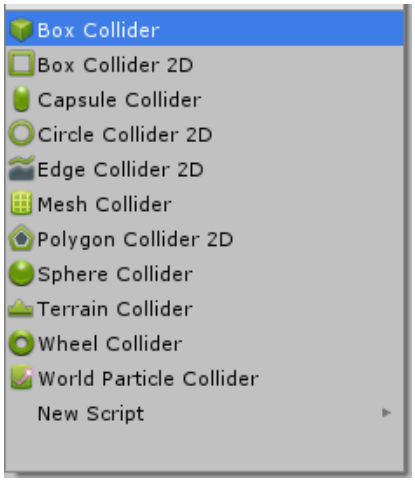
En este proyecto dicha curva de interés se puede conseguir con las oleadas de enemigos, variando su dificultad a lo largo del tiempo, cantidad de enemigos por oleada, ofreciendo distintos tipos de enemigos, donde las primeras estén compuestas por enemigos fáciles de derrotar y las ultimas presenten una mayor dificultad que ofrezcan un reto a las habilidades del jugador.

Implementación en Unity

Para la implementación en Unity nos hará falta conocer algunos de los componentes básicos que nos ofrece este motor.

Colliders

Los colliders son componentes que permitirán que nuestros objetos no puedan ser atravesados, es decir y como su nombre lo indica hará que otros objetos colisionen contra ellos. Unity nos ofrece colliders con distintas formas.



Lista de colliders

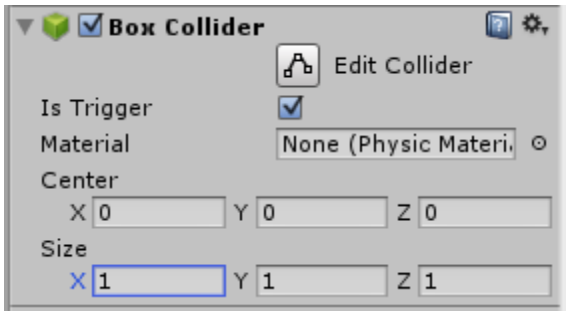
Para gestionar las colisiones Unity nos ofrece varios métodos. Que se disparan cuando ocurren determinados eventos.

- OnCollisionEnter(Collision collision)
- OnCollisionStay(Collision collision)
- OnCollisionExit(Collision collision)

Debemos tener en cuenta que los métodos OnCollisionEnter y OnCollisionExit se ejecutaran una sola vez por evento, mientras que OnCollisionStay se ejecutara constantemente mientras sigan colisionando los objetos.

Triggers

Los triggers son una variación de los colliders, con la diferencia de que si se puede pasar a través de ellos. Se suelen utilizar como zonas de detección y disparadores de eventos.



Collider como trigger

Para crear un trigger en Unity se usan los mismos componentes que los colliders y basta con marcar el checkbox **Is Trigger**.

Al igual que los colliders los triggers también tienen sus métodos para la gestión de eventos.

- OnTriggerEnter (Collider collider).
- OnTriggerStay (Collider collider).
- OnTriggerExit (Collider collider).

Como ocurría con los Colliders los métodos OnTriggerEnter y OnTriggerExit se ejecutan una sola vez y OnTriggerStay se ejecutara mientras el objeto no abandone el trigger.

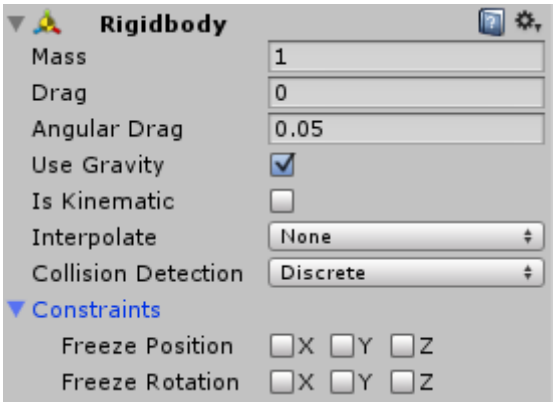
Es **importante** remarcar en el caso de los triggers, para que estos eventos se disparen uno de los dos objetos que colisionan debe tener un componente de **rigidbody**, esto le permite a Unity detectar que un objeto ha entrado en el trigger.

Rigidbody

Es un componente que hará que los objetos estén bajo el control del motor de física de Unity, añadir un componente rigidbody a un objeto hará que se vea afectado por la gravedad y colisione con los distintos objetos que posean un collider.

Unity además nos ofrece un conjunto de funciones para añadirle fuerzas al objeto y moverlo de forma realista.

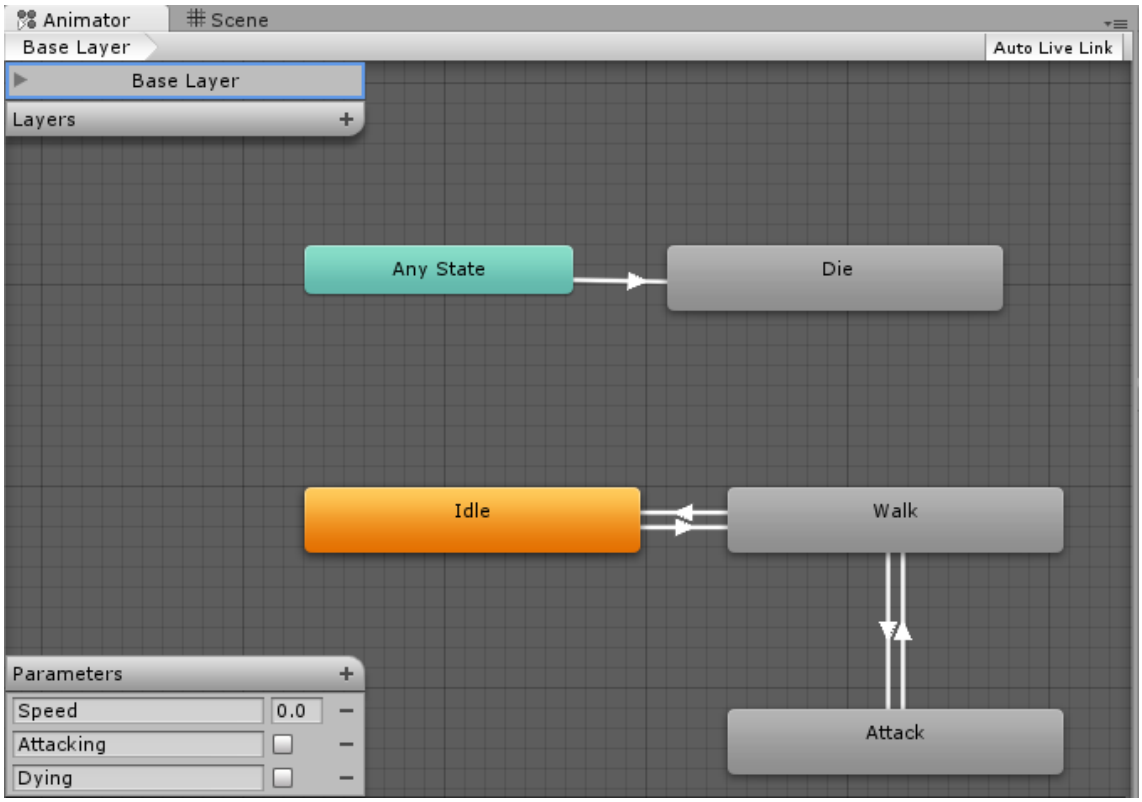
Si queremos que nuestro objeto posea un rigidbody pero no se vea afectado por el motor de física, debemos de marcar el checkbox **Is Kinematic**



Componente rigidbody

Mecanim, Animator, Animator Controller y Animation

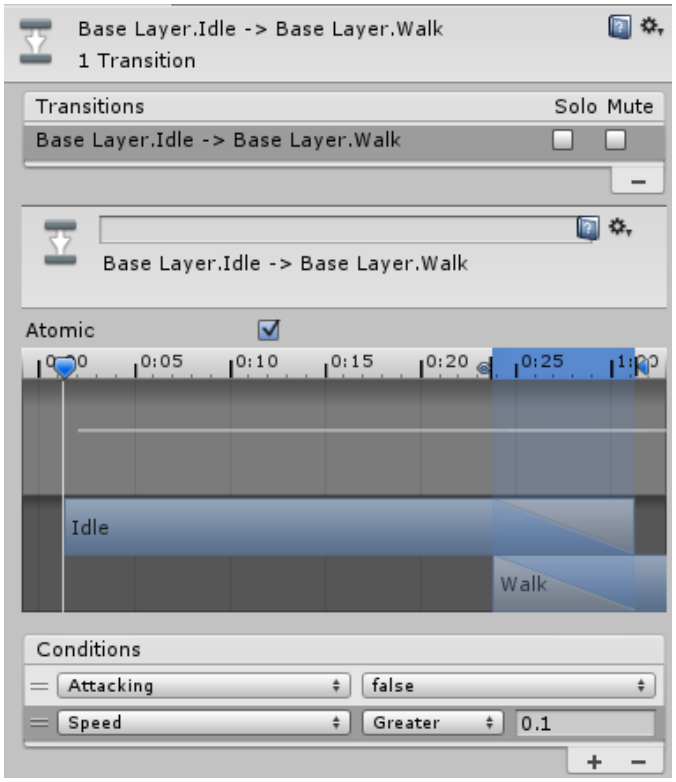
Mecanim es un nuevo, poderoso y flexible sistema de animación que posee Unity, nos permite crear máquinas de estado y gestionar la transición entre animaciones, asignando condiciones que nos llevaran de un estado o animación a otra.



Ventana animator de Mecanim

Cada uno de los estados representa una animación y las flechas son las transiciones, si seleccionamos una de estas en el inspector podremos editar la condición que hará que se pase de un estado a otro.

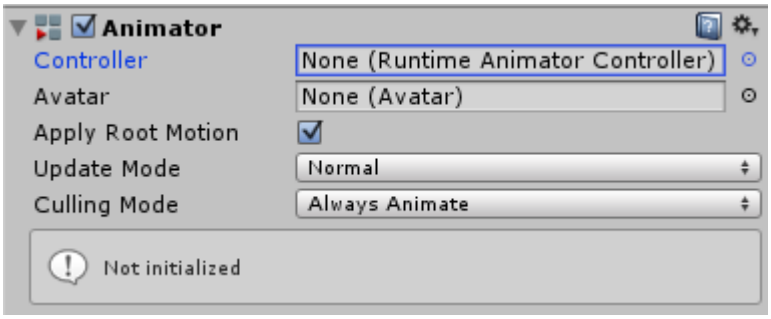
En la esquina inferior izquierda podremos crear parámetros, que serán las variables que se usaran para crear las condiciones de transición, estas variables serán accesibles a través de código y del componente **Animator Controller.**



Ventana inspector con transición seleccionada

Animator

Es el componente principal a través del cual se añaden animaciones a los objetos. Necesitaremos un Animator Controller donde se encontrara la máquina de estados con todas las transiciones y un Avatar sobre el que se aplicaran las animaciones, este último no es necesario en todos los casos.



Componente Animator

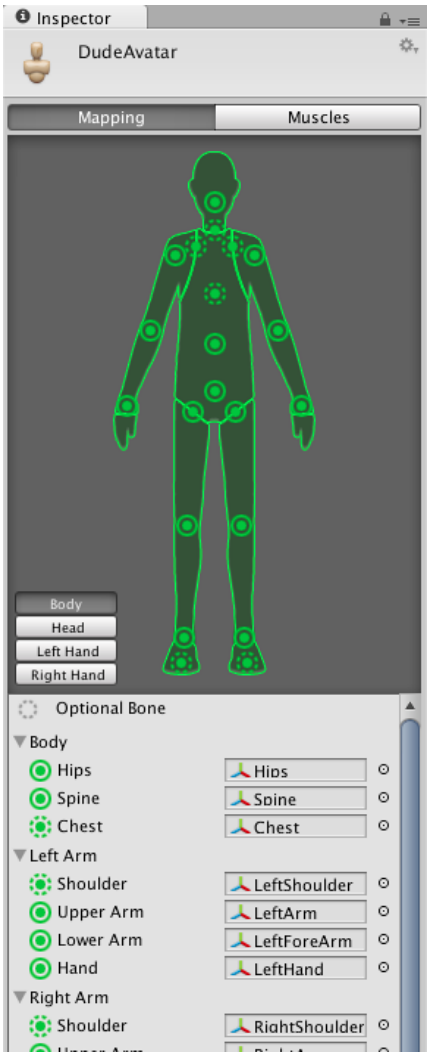
Animator Controller

El Animator Controller nos permitirá acceder a la ventana donde podremos añadir las distintas animaciones, transiciones y variables que deseemos. A través de este controlador podremos acceder a las variables declaradas y ejecutar las transiciones cuando lo consideremos adecuado.

Avatar

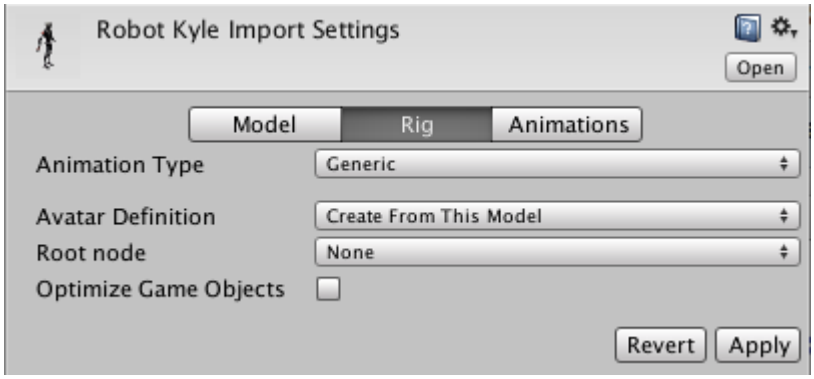
El Avatar guardará información sobre el esqueleto del objeto al que se le está aplicando la animación, de esta manera podrá realizar la transición entre ellas de manera que el cambio no sea demasiado cortante.

Para la creación de los Avatares, Unity dispone de una herramienta que nos permite asignar los distintos huesos o partes del rigging de nuestro modelo 3D que se deben mover, en caso de que la importación por defecto no haya podido configurarlo correctamente.



Ventana Configuración Avatar

Aunque Mecanim nos muestra todo su potencial con humanoides, no todos los personajes de nuestro juego lo son, para ello Unity cuenta con la integración de animaciones genéricas

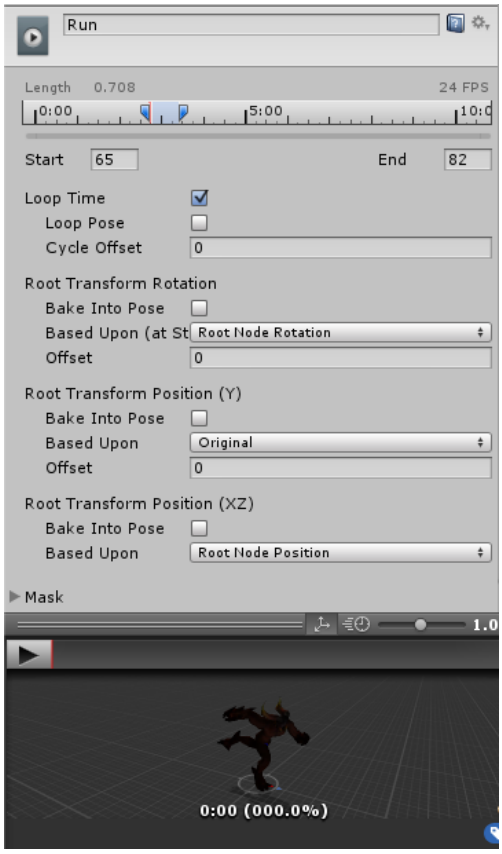


Ventana Configuración de importación de modelo 3D

Muchos de los modelos 3D tienen su propio esqueleto y Unity nos permite crear un Avatar a partir de la información del modelo importado.

Animation

Las animaciones son la parte final de este conjunto de componentes que nos permitirán que nuestros personajes y otros elementos cobren vida dentro del juego.



Ventana Clip de animación

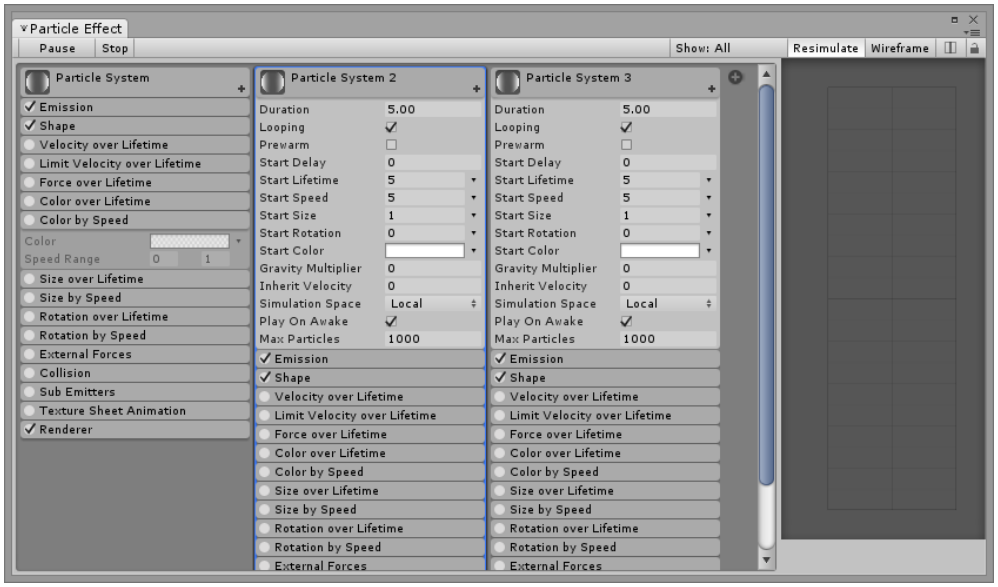
En esta ventana podremos configurar múltiples opciones, pero en especial me interesa resaltar la casilla que se encuentra marcada **Loop Time** esto permitirá que la animación se ejecute nuevamente al terminar entrando en un bucle, no nos interesa que si nuestro personaje aún se encuentra en un estado de "Corriendo" la animación se paralice en el último fotograma,

causando un efecto un tanto extraño donde el personaje se seguiría moviendo pero sin ejecutar la animación.

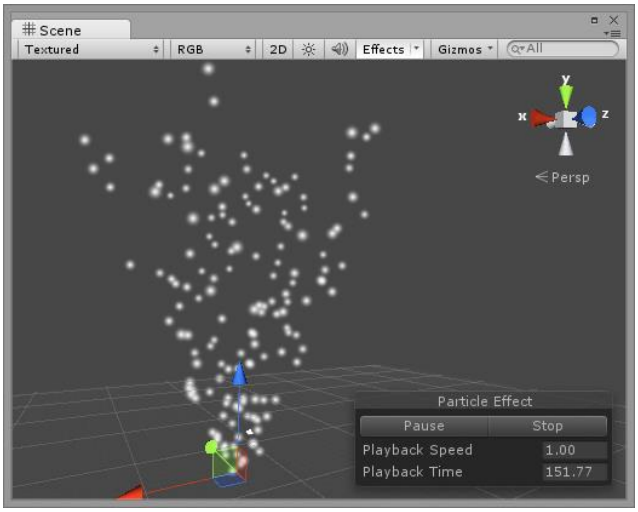
Sistemas de partículas

Las partículas no son más que pequeñas imágenes que se muestran y que son movidas por un sistema. Cada partícula representa una diminuta parte de un fluido o una entidad amorfa que al estar juntas en gran número crean la impresión de una entidad.

Unity posee un sistema de partículas llamado Shuriken que nos permite crear esas explosiones que tanto nos llaman la atención. Con Shuriken podemos modificar una gran cantidad de valores como, color, forma, velocidad, numero de partículas a emitir, etc., que nos ayudaran a crear impresionantes y únicos sistemas de partículas, pero que requiere de un gran trabajo artístico y dedicación.

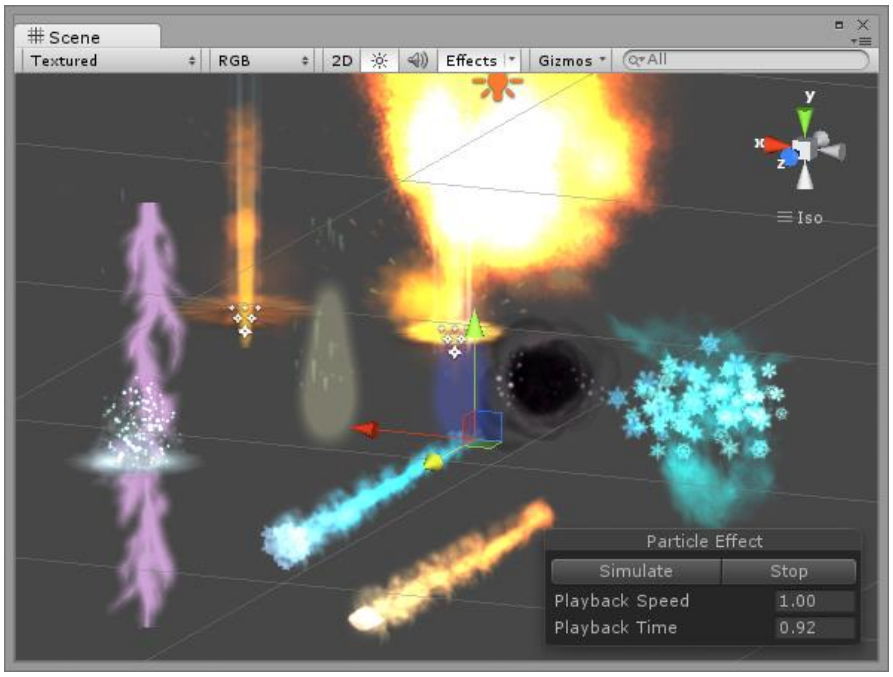


Ventana Edición de sistemas de partículas



Ventana Scene con un sistema de partículas en ejecución

Haciendo uso de estos sistemas de partículas se crearon las magias que usaran los enemigos, jugador y torres del videojuego creado en este proyecto, estos sistemas de partículas se obtuvieron desde la Asset Store.



Ventana Scene con sistemas de partículas usados en el juego

GUI

La implementación de la GUI dentro de Unity fue uno de los dolores de cabeza con los que se tuvo que lidiar durante el desarrollo del proyecto. En versiones anteriores de Unity solo se disponía del método **OnGUI** y dentro de este definíamos los botones, texto e imágenes que queríamos mostrar junto con su tamaño, color y posición en la pantalla.

Hacer esto sin poder ver cómo está quedando el diseño resulta una tarea bastante compleja y la modificación de uno de estos elementos podía hacernos perder una gran cantidad de tiempo.

Para aliviar este trabajo, varios grupos de desarrolladores han creado herramientas o plugins que facilitan en gran medida esta tarea.

Dentro de las opciones que se barajaron estaban **Daikon forge** y **NGUI** que en su momento tenían un precio que rondaba los 90€, por suerte **NGUI** disponía también de una versión gratuita, que no recibe soporte y que no se encuentra en su versión más actualizada, pero que aun así cumplía con los requisitos y funcionalidades que necesitábamos, **Daikon forge** actualmente ha retirado su plugin de la Asset Store ya que Unity en su versión 4.6 dispone de un nuevo y mejorado sistema de UI, similar a los plugins mencionados. De hecho el creador de **NGUI** participó en el desarrollo de este nuevo sistema.

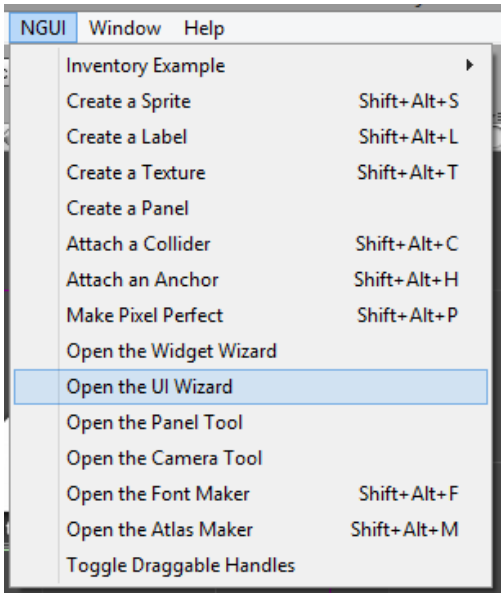
NGUI

Durante el desarrollo del proyecto se usó el plugin NGUI en su versión 2.7.0 que es gratuita pero que no posee todas las funcionalidades de su versión más actualizada.

A continuación describiremos un poco el funcionamiento de este plugin.

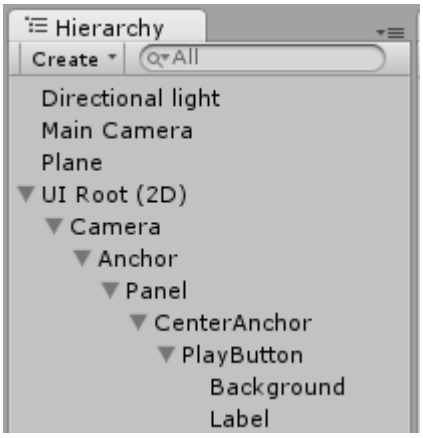
Para empezar, NGUI necesita de una cámara extra para poder dibujar todos los elementos que queremos añadir. Esta cámara será similar a la Main Camera que ya se encuentra en la escena pero con una pequeña diferencia, utiliza una proyección ortográfica, y al ser elementos 2D los que queremos mostrar como botones, etiquetas, barras de progreso que representen la vida o energía del jugador, no es necesario que se tenga una profundidad.

Para poder crear esta cámara añadimos un UI Root a la escena, que se crea con los nuevos menús que estarán disponibles desde que se importa el plugin a Unity.



Menú NGUI en Unity

A partir de aquí se nos generara un nuevo elemento en la escena que contendrá una cámara y un panel sobre el que empezaremos a crear los distintos elementos de los que dispondrá nuestra interfaz.



UI Root en la vista Hirerarchy de la escena

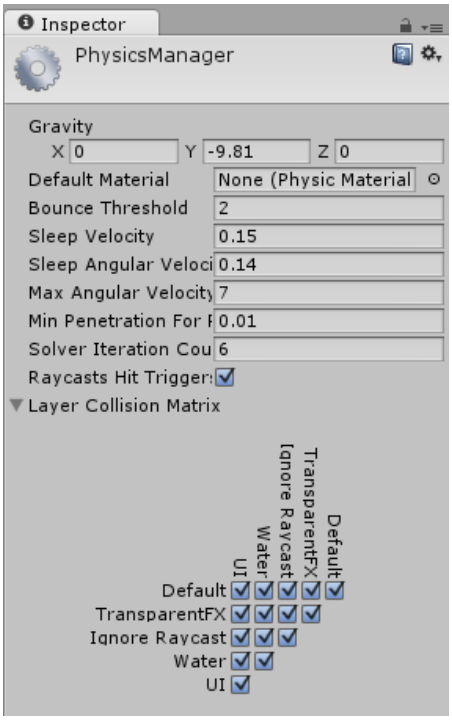
Para facilitar la creación de los distintos elementos como botones y barras de progreso, NGUI nos permite la creación de atlas de texturas, estos atlas son imágenes donde se "empaquetan" todas las texturas que queremos usar, de esta manera reducimos los drawcalls

del juego. Si no tenemos todas las texturas en un atlas se hará un drawcall por cada textura en la escena y como consecuencia el rendimiento empeorará.

Layers

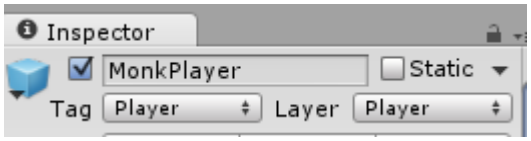
Los layers o capas de física son una herramienta muy interesante, donde con ayuda de una matriz de colisiones podemos definir que objetos podrán interactuar entre sí.

Por defecto Unity ya nos ofrece un conjunto de Layers, pero podemos crear nuevos para ser usados en nuestro juego.



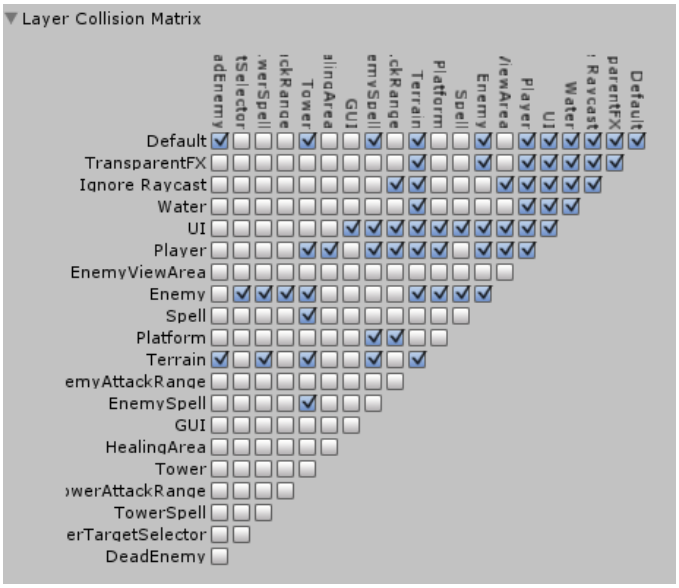
Matriz de colisiones

Estos Layers y los nuevos que creemos se podrán asignar a los distintos GameObjects desde la ventana de inspector, permitiendo que estos GameObjects se vean afectados por la matriz de colisiones que gestiona el motor de física de Unity.



Ventana inspector, Layer y Tag del GameObject MonkPlayer

Durante el desarrollo de este proyecto se generó la siguiente matriz con nuevos layers que se asignaron a distintos objetos para que interactuaran entre sí.

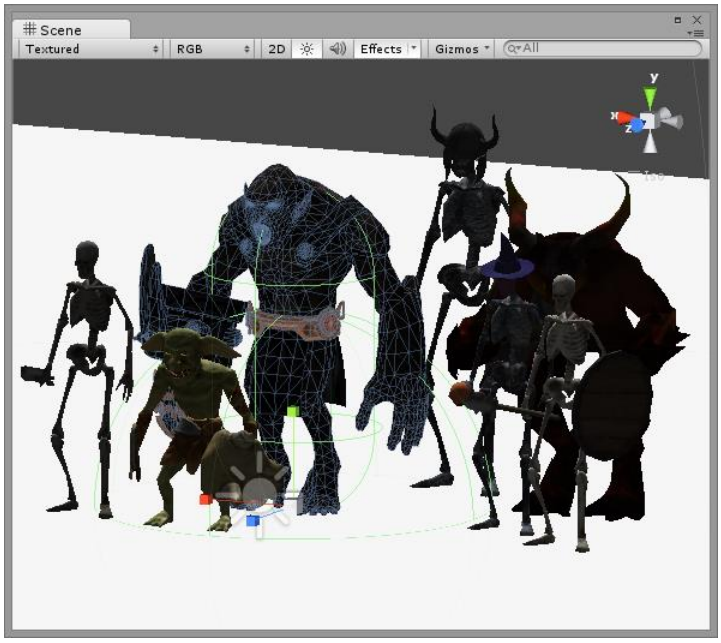


Matriz de colisiones Darkest Nights

Para comprender mejor como funciona esta matriz de colisiones imaginemos dos GameObjects con su respectivo collider, uno bajo el layer Player y otro bajo el layer Enemy, si en la matriz de colisiones buscamos la intersección de estos dos layers y desactivamos el checkbox, estos dos objetos no colisionaran entre sí, ya que le estamos diciendo al motor de física que ignore las colisiones entre estas dos capas.

Implementación y creación de enemigos

Como primer paso en la creación de los enemigos nos fijamos en que tipos de enemigos queremos, llegando a la decisión de tener enemigos caster o ataque a distancia y enemigos melee o ataque cercano.



Modelos 3D de los enemigos

Como comentamos anteriormente los modelos 3D y sus animaciones las conseguimos a través de la Asset Store.

Para la creación de su comportamiento, se implementó una máquina de estados con los siguientes estados:

Perseguir objetivo: Si el jugador se encuentra dentro del rango de visión del enemigo este le perseguirá, pero si el jugador sale de su rango de visión, el enemigo cambiará su objetivo, tomando como nuevo objetivo la plataforma.

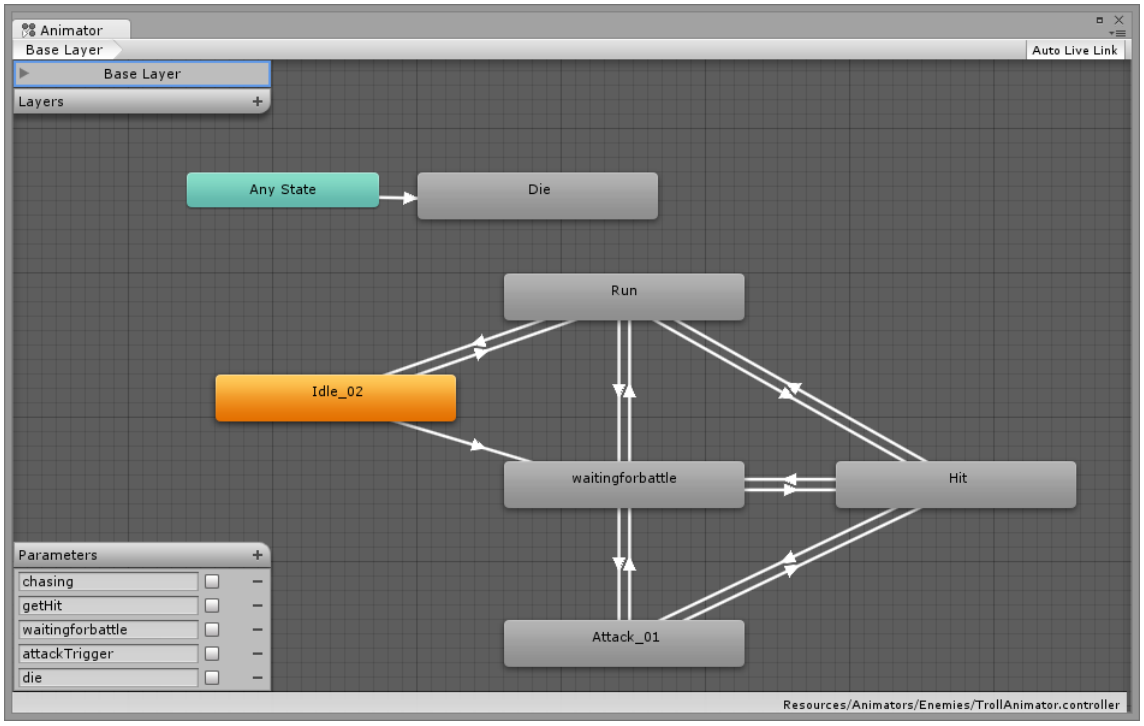
Atacar: Una vez el enemigo alcance a su objetivo le atacara hasta derrotarlo o hasta que este salga de su alcance de ataque.

En combate: Cada vez que el enemigo ataca deberá esperar un determinado tiempo para realizar nuevamente un ataque durante ese periodo de tiempo se encuentra en este estado.

Recibir daño: Cuando el enemigo es golpeado su ataque se ve interrumpido y recibe daño reduciendo sus puntos de salud.

Morir: Si los puntos de salud del enemigo llegan a 0 el enemigo morirá y desaparecerá.

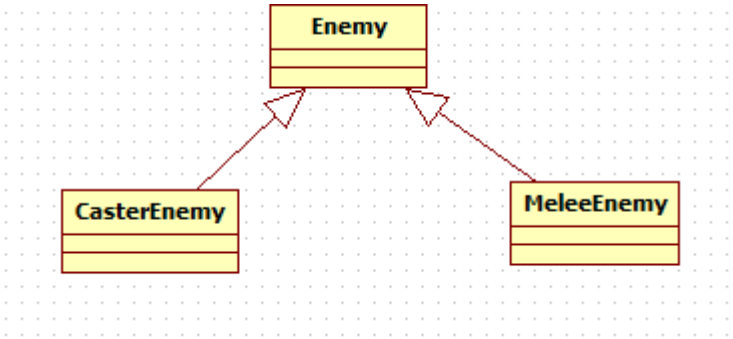
Teniendo en cuenta esta máquina de estados se creó un Animator Controller con las distintas animaciones para cada uno de ellos.



Máquina de estados de los enemigos

Una vez creado el Animator Controller empezamos a escribir el código de la clase Enemigo.

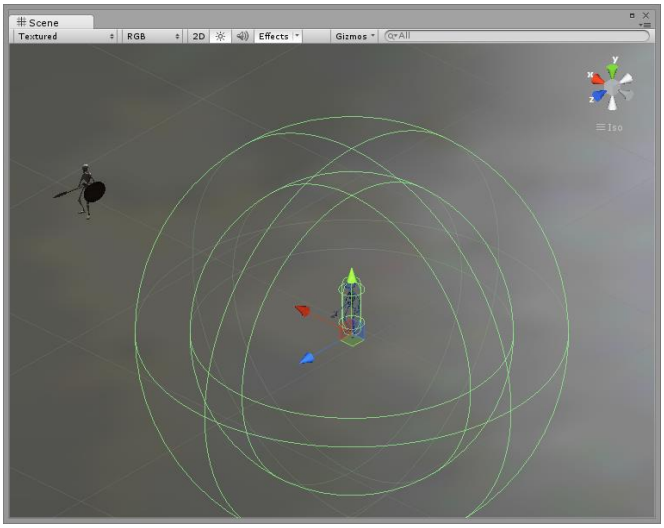
Como vamos a implementar dos tipos de enemigos que realizan exactamente lo mismo pero atacan de forma distinta creamos una clase abstracta Enemy de la que heredarán e implementaran cada uno su distinta forma de atacar.



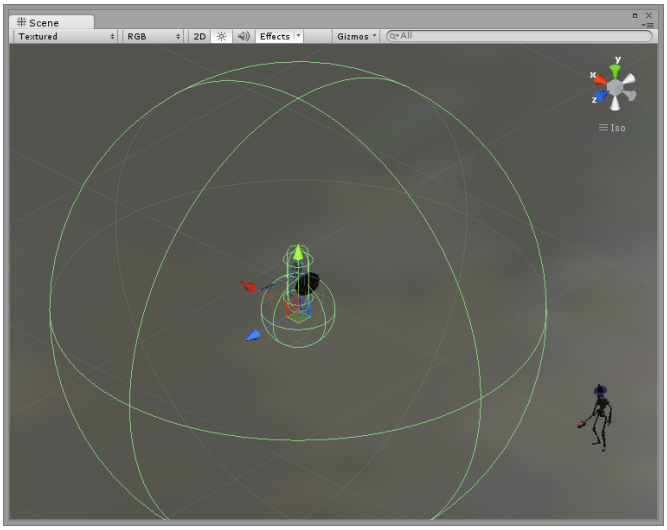
Herencias enemigo

Los enemigos de tipo caster instanciaran un sistema de partículas y atacaran desde una distancia mayor que los enemigos de tipo melee.

Para los campos de visión y rango de ataque de los enemigos, se hizo uso de los triggers.



Triggers enemigo caster



Triggers enemigo melee

Si nos fijamos en las imágenes **Triggers enemigo caster** y **Triggers enemigo melee** podremos identificar 3 colliders, en el centro de los personajes y con forma de capsula un collider que no permitirá que se atraviese a nuestro enemigo, el siguiente collider que funcionara como trigger será el rango de ataque, en el caso de los enemigos melee es una esfera pequeña y en el de los caster una esfera bastante más grande, y por último en la parte más exterior, el campo de visión de los enemigos que detectará la presencia del jugador causando que el enemigo le persiga mientras se encuentre dentro de este.

Cada uno de estos triggers se puso en un GameObject distinto ya que un GameObject solo puede tener un collider de cada tipo y era necesario usar varios colliders de forma esférica. Para ello se usaron empty GameObjects anidados como hijos del enemigo a los que se asignaron distintos layers de física, **EnemyAttackRange** y **EnemyViewArea**. Estos layers colisionarían únicamente con el jugador y la plataforma.

Jerarquía de enemigo

Una vez configurado el enemigo en la escena, con su modelo 3D, triggers, colliders, animator controller, y sus respectivos layers de física, era cuestión de añadir el script que contendría la clase enemigo y le ayudaría a cobrar vida.

La lógica del enemigo es bastante sencilla y se puede resumir en el método Update que se muestra a continuación.

```
protected void Update ()
{
    if (!isDying) {
        if (!isInAttackRange) {
            Chase ();
        } else {
            AttackTarget ();
        }
    }
}
```

Update de la clase Enemy

Además de los métodos que se encuentran dentro del Update, recordemos que se ejecuta cada fotograma, también posee métodos para recibir daño y morir.

```
public void ReceiveDamage (float damage)
{
    if (health <= 0) {
        return;
    }
    anim.SetTrigger ("getHit");
    health -= damage;
    if (health <= 0) {
        StartCoroutine (Die ());
    }
}
```

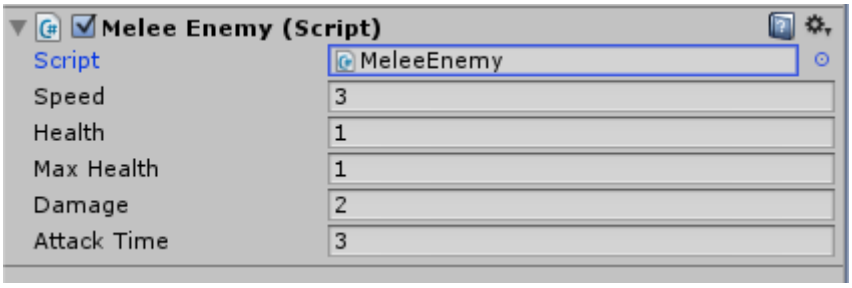
En este método de recibir daño podemos ver como se restan los puntos de salud del enemigo y como se llama al Animator Controller para que ejecute la animación correspondiente activando el disparador **getHit** definido anteriormente, este disparador permitirá la transición en la máquina de estados de Mecanim. En caso de que los puntos del enemigo lleguen a 0 se ejecutará la corrutina Die.

```
protected IEnumerator Die ()
{
    isDying = true;
    anim.SetTrigger ("die");
    yield return new WaitForSeconds (anim.GetCurrentAnimatorStateInfo (0).length + 1);
    alive = false;

}
```

Dentro de la corrutina Die se activa el trigger del Animator Controller que permitirá ejecutar la animación de morir y esperaremos a que esta animación se ejecute por completo para declarar al enemigo como muerto.

Estos enemigos poseen atributos que algunos de ellos se han ido mencionando, como puntos de salud, velocidad de ataque, daño y velocidad. Para configurar estos atributos de manera rápida, Unity nos permite al declarar una variable como publica y añadir el script a un GameObject modificarla directamente desde el editor.

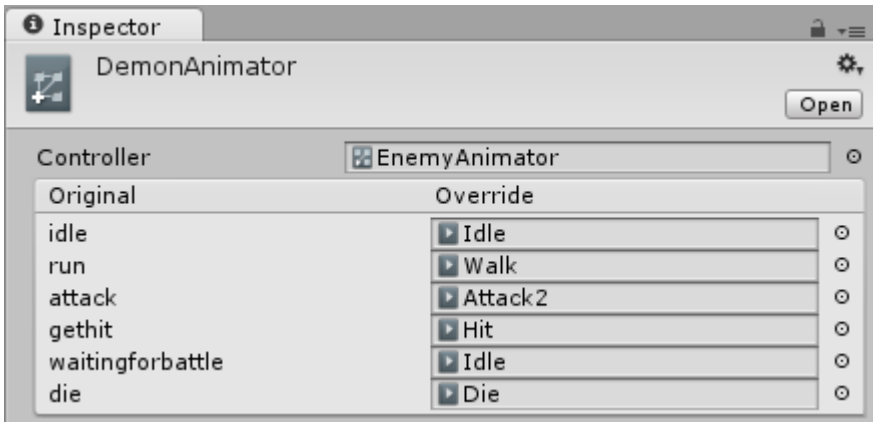


Variables en editor enemigo melee

Esto nos permite crear nuevos enemigos y configurar sus características de manera rápida y directa sin necesidad de modificar el script de la clase.

Una vez añadido el script al GameObject con el resto de componentes tenemos nuestro enemigo completamente funcional.

OverrideAnimator



OverrideAnimator en Inspector

Gracias a este componente la creación de enemigos con distintos modelos 3D y nuevas animaciones se convierte en una tarea casi trivial, ya que podemos sobrescribir un Animator Controller que previamente hemos creado y sustituir las distintas animaciones de los estados, permitiéndonos de esta manera usar las transiciones existentes y que no debamos modificar ni crear ningún componente nuevo. Solo nos debemos asegurar de asignar este OverrideAnimator con su respectivo Avatar al componente de animación del GameObject para que la magia suceda.

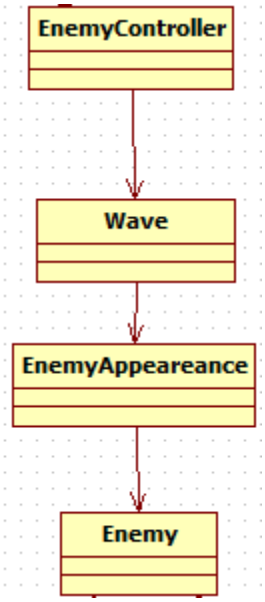
A partir de aquí empezamos a generar los prefabs de los distintos tipos de enemigos con todos sus componentes y características que posteriormente nos servirán para crear las oleadas.

Generación de oleadas de enemigos

Para generar las distintas oleadas de enemigos se implementó un EnemyController que se encargará de gestionar cada cuanto tiempo aparecerán nuevos enemigos, el tipo y porcentaje de enemigos que aparecerán, la población total de enemigos en la escena y de instanciarlos y eliminarlos una vez hayan muerto.

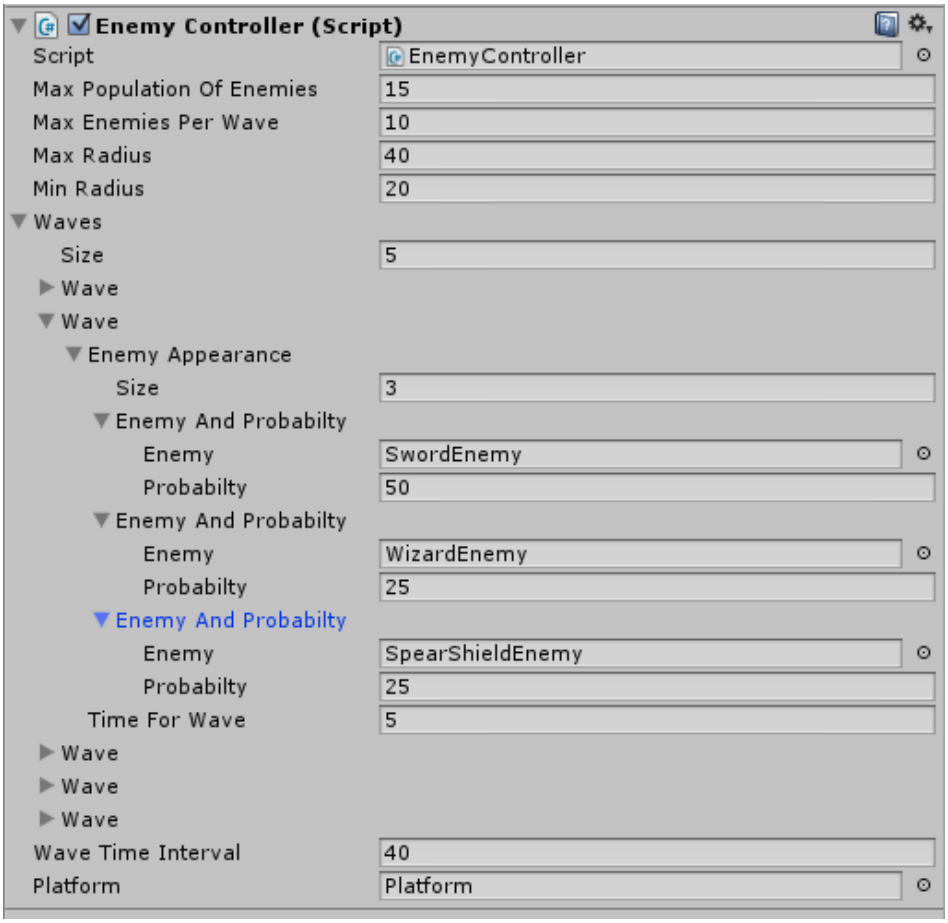
Identificamos entonces una clase o entidad más, las oleadas (Wave), pero también era necesario definir una estructura que nos permitiera modificar el porcentaje de aparición de cada enemigo para las distintas oleadas, se creó entonces las EnemyAppeareance. Así, las oleadas están compuestas de EnemyAppeareance y estas a su vez definían que enemigo y con qué probabilidad aparecería.

Quedando entonces la siguiente estructura de clases.



Estructura para la generación de enemigos

Para la creación de las oleadas se hizo uso de variables públicas que nos permiten modificar de manera rápida la configuración de cada una. Teniendo entonces el siguiente componente, en el cual se indica un punto central, la plataforma, alrededor del cual aparecerán los distintos enemigos en un radio máximo y mínimo.

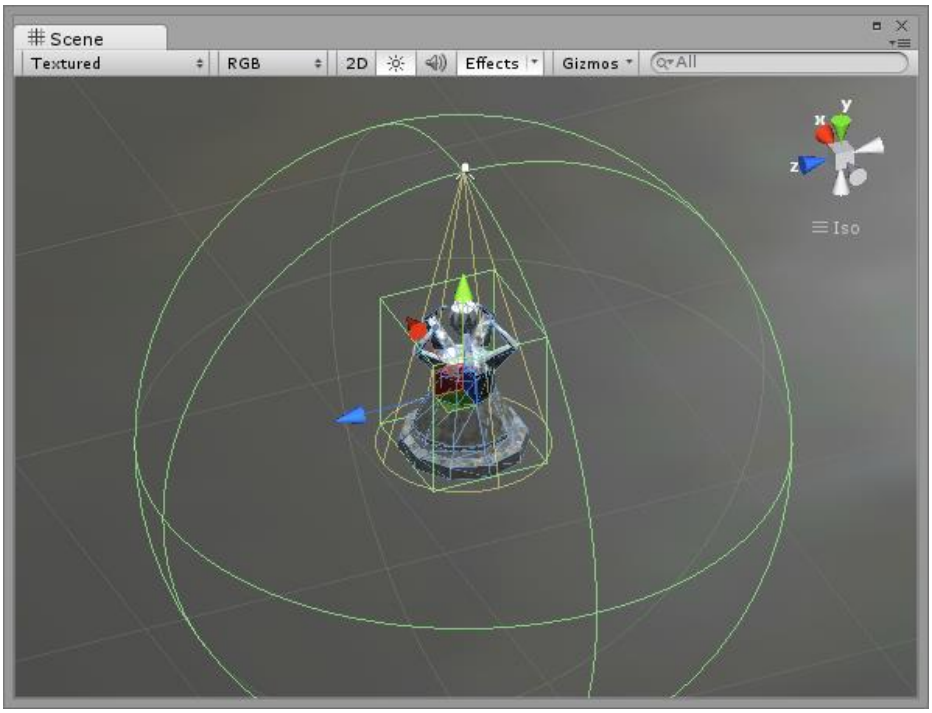


Componente Enemy Controller

Dentro podemos indicar la cantidad de oleadas que deseamos y que tipo de enemigo vamos a instanciar, SwordEnemy, WizardEnemy y SpearShieldEnemy, son algunos de los prefabs de enemigo que se han creado y que este controlador generará con la probabilidad indicada. Además podemos establecer cuanto tiempo pasará desde la última oleada para que se instancie la siguiente.

Implementación de torres

Las torres, son elementos del juego que nos facilitarán el trabajo y nos ayudaran a defender la plataforma. Al igual que los enemigos hacen uso de triggers para la detección de su objetivo.



Triggers de torre

Para activar las torres el jugador deberá atacarla con alguno de sus hechizos, para identificar una torre activa de una que no lo está usamos un pequeño punto de luz y una "gema" que cambia de color.

Estas torres al igual que los enemigos, tienen un tiempo de recarga en el cual no pueden atacar, y al igual que los enemigos de tipo caster instancian un sistema de partículas para atacar.

Propiedades de la torre

Como en un mismo nivel podemos tener varias torres, y cada torre posee una lista de los enemigos que están en su área de ataque, siendo posible que un mismo enemigo se

encontrara en varias listas a la vez, era necesario gestionar estas listas correctamente ya que un enemigo que era derrotado por otra torre, no desaparecía por arte de magia de las demás, simplemente dejaba un hueco donde se encontraba su referencia causando errores de acceso y generando posiciones nulas.

Para ello se implementó el patrón de **Observador y Observables** donde los enemigos eran los objetos observados y las torres los observadores. Así cada vez que un enemigo era eliminado, todas las torres u otros observadores eran notificados y eliminaban correctamente este objeto de sus respectivas listas.

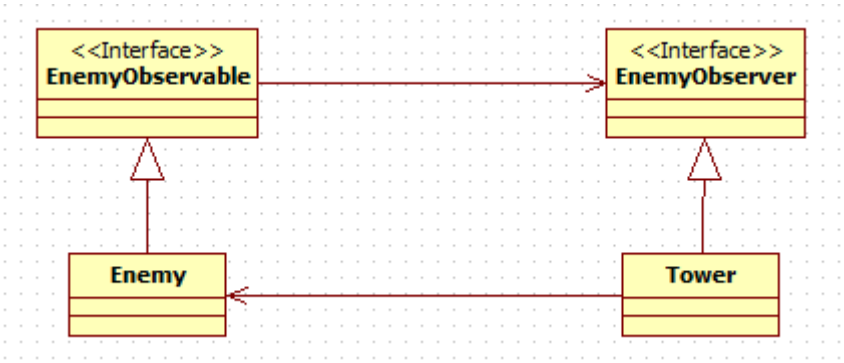


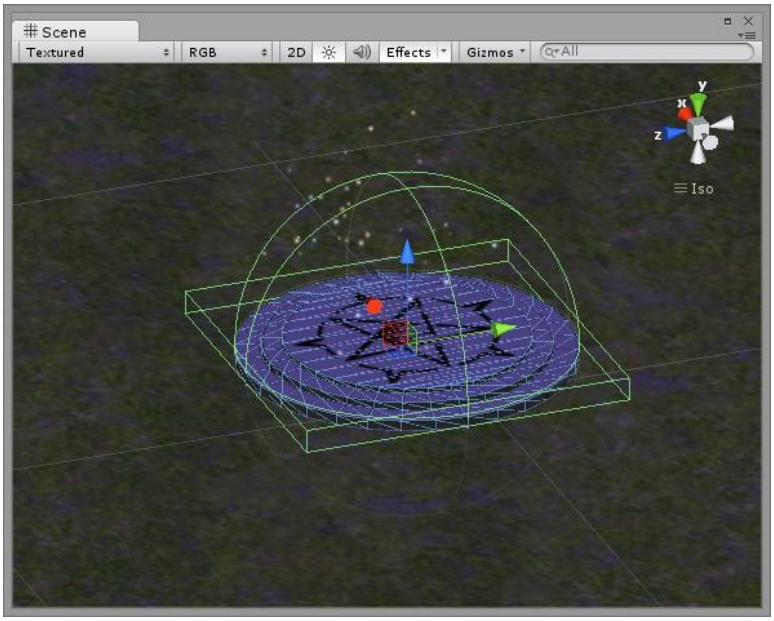
Diagrama observador y observable

Implementación de la plataforma

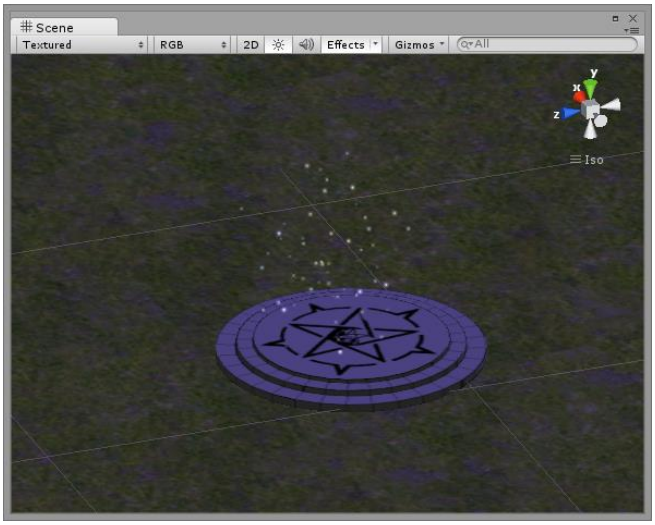
Para crear la plataforma, primero creamos un modelo 3D básico con Blender, podíamos haber escogido alguna de la Asset Store pero mis ganas de aprender y de conocer cada aspecto del desarrollo de un videojuego me llevo a hacerlo por mis propios medios.

Al igual que los enemigos, la plataforma también posee unos puntos de salud, será el objetivo de muchos de los enemigos siempre que no nos ataquen y deberemos defenderla para poder completar con éxito el juego.

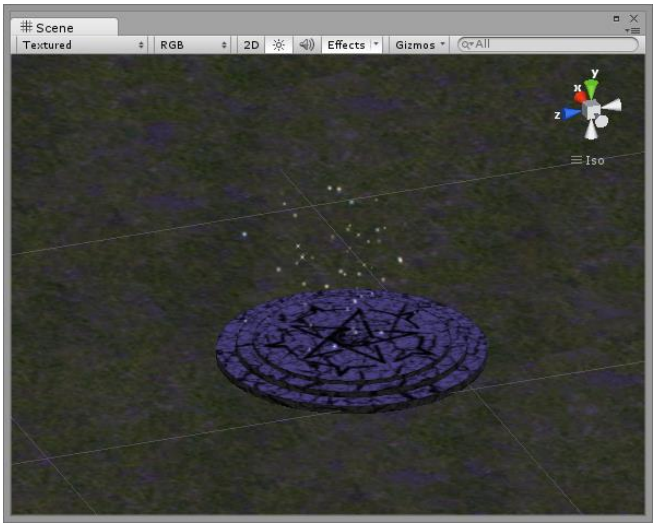
También se generaron distintas texturas para la plataforma que irán cambiando a medida que pierda puntos de salud, mostrándola cada vez más deteriorada.



Trigger y collider plataforma



Plataforma normal



Plataforma deteriorada

La plataforma dentro del juego también proporciona beneficios al jugador, ya que si permanece sobre ella recuperara vida y energía o mana con el tiempo.

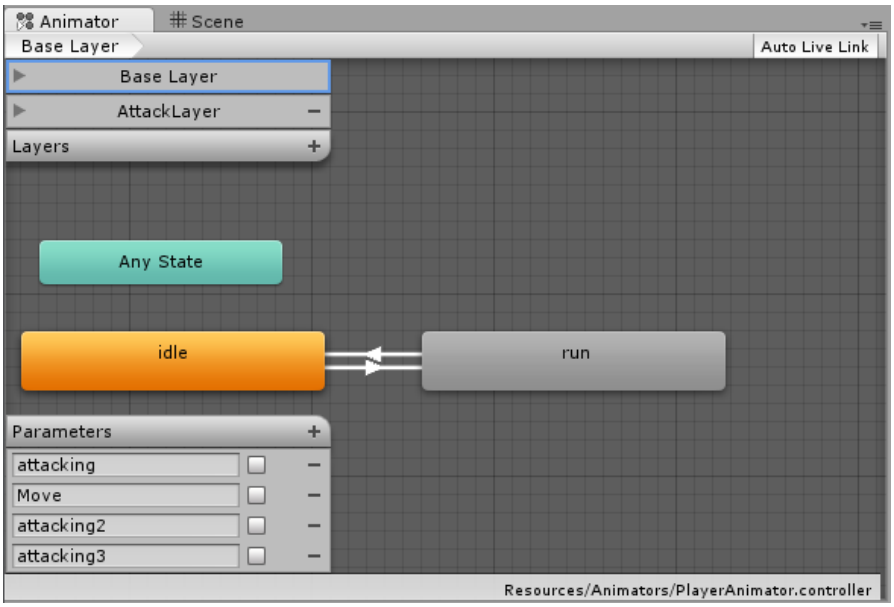
Implementación del personaje

Para el personaje en su primera versión usamos uno de los esqueletos con una variación en su color, pero para identificarlo mejor se decidió cambiar por completo el modelo 3D y usar un monje al que se añadieron los distintos componentes necesarios para su total funcionamiento.

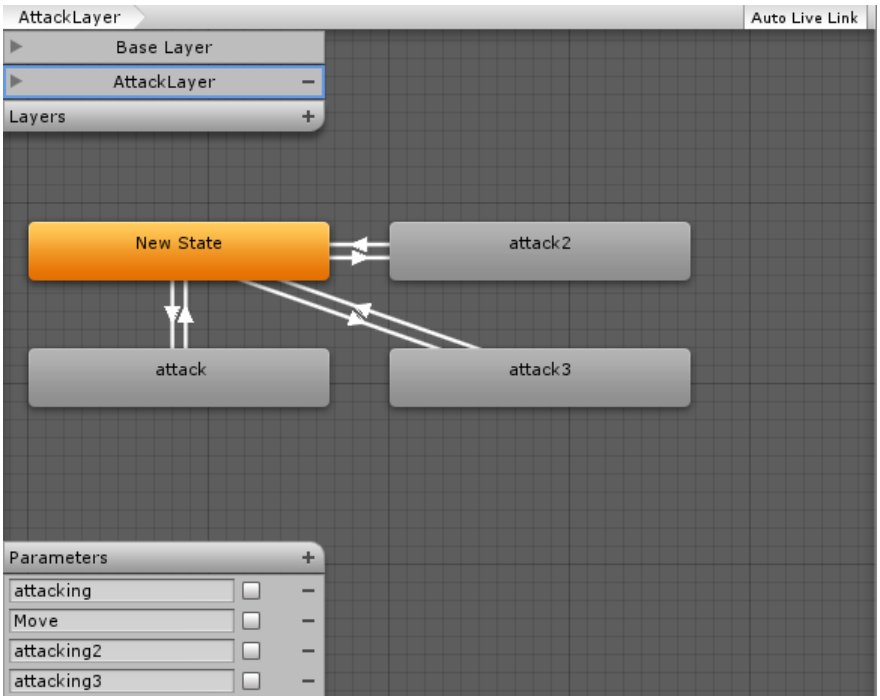
Personaje principal sobre la plataforma

Al igual que ocurrió con los enemigos, el personaje posee un Animation Controller con su correspondiente máquina de estados.

Esta vez el Animation Controller posee 2 Layers que permitirán la ejecución de distintos conjuntos de animaciones. Se creó un layer separado para las distintas animaciones de ataque.



Base Layer



Attack Layer

Como podemos ver en las imágenes del Animation Controller anteriores, nuestro personaje solo tendrá 3 estados

Ocioso (idle): Mientras se encuentre quieto.

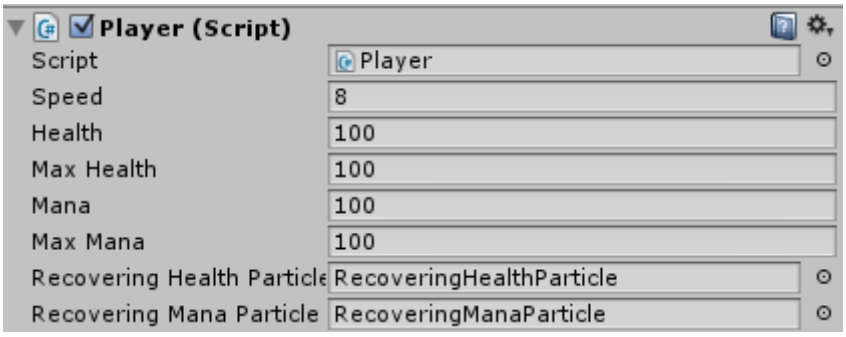
Correr (run): Al moverse de un punto a otro.

Atacar (attack): Al lanzar un hechizo.

He de mencionar que el Attack Layer se añadió en la implementación del último control que fue el que mejores resultados dio. Anteriormente solo existía una animación de ataque que se encontraba en el Base Layer.

Ahora nos queda la implementación de la lógica del jugador, esta fue la parte en la que más se invirtió tiempo en el desarrollo del proyecto, ya que se implementaron distintos tipos de controles que afectaban la forma de moverse y de atacar del jugador.

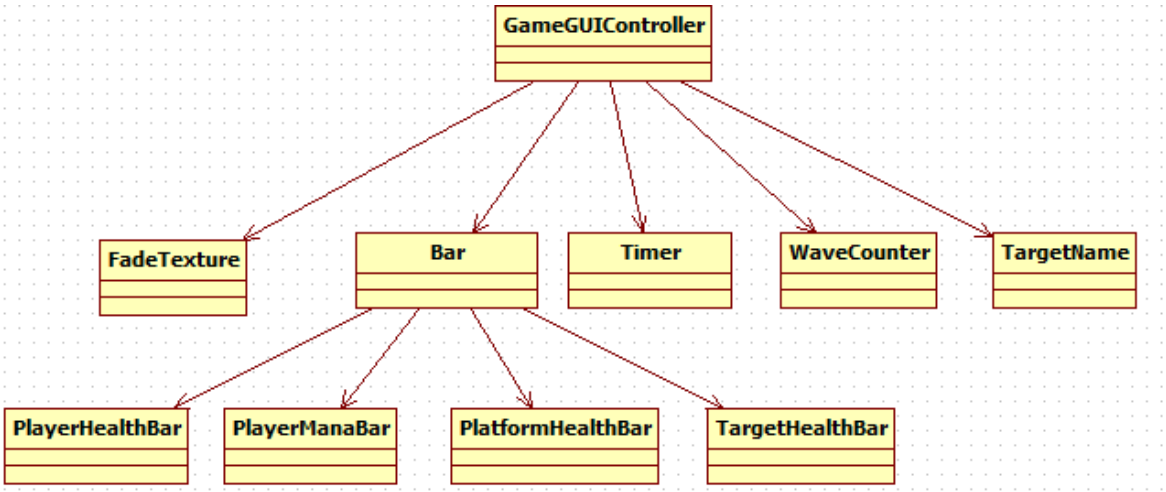
Pero como podemos esperar, la implementación del jugador también hacia uso de las variables públicas que nos permiten configurar el jugador de manera rápida.



Implementación de GUI

Como ya se comentó anteriormente la implementación de la GUI se realizó usando el plugin NGUI 2.7.0. Y Fue una de las partes que más varió a lo largo del desarrollo del proyecto junto con los controles que se implementaron.

Pero finalmente se obtuvo la siguiente estructura.



Cada uno de los elementos de esta estructura los podemos ver reflejados en la pantalla de juego

GUI del juego

En la parte superior nos encontramos de izquierda a derecha, barras de vida y mana del jugador, barra de vida que representa la del objetivo actual junto con su nombre y barra de vida de la plataforma.

En la parte inferior de izquierda a derecha, contador que indica cuando tiempo falta para la siguiente oleada, contador de número de oleadas, botones de magias disponibles.

En los bordes de la pantalla tenemos un bloodSplat que aparecerá y desaparecerá de manera continua indicándonos que estamos recibiendo daño.

Además se implementó un sistema de mensajes, para alertar al jugador de que las oleadas empezaran pronto y avisarle que su objetivo es defender la plataforma, para avisarle que una nueva oleada aparece, que se ha quedado sin energía para lanzar un hechizo, o para advertirle de que sus hechizos aún no están listos.

Mensajes de alerta

Es el GameGUIController el encargado de actualizar la información de esta vista para mantener al usuario siempre informado de lo que está ocurriendo a su alrededor.

Controles

La implementación de los controles es a mi parecer una de las partes más difíciles de establecer en un juego ya que esto definirá el comportamiento de los jugadores, nos limitara o nos indicara

de qué manera podemos implementar las distintas mecánicas, es una parte esencial que puede llevar a que un jugador disfrute o se frustre. Es por ello que en el desarrollo de este proyecto se crearon distintas versiones de controles cambiando totalmente la manera de interactuar de los jugadores con el personaje y los hechizos de los que disponía, teniendo como intermediario una pantalla táctil.

Tipo 1

La primera versión de los controles y quizá la más sencilla, implementaba un **click to move** donde el personaje se movía hacia la dirección donde se había realizado un click o tap, esta posición seleccionada se marcaba con una flecha roja.

Control tipo 1 click to move

Para lanzar hechizos, bastaba con hacer tap o click sobre el enemigo instanciando un sistema de partículas que al entrar en contacto con el enemigo le harían daño.

Control tipo 1 lanzando hechizo

Uno de los principales inconvenientes con este tipo de control es la precisión que podemos tener a la hora de apuntar en una pantalla táctil, en un ordenador y con un puntero de ratón resulta bastante sencillo ya que podemos ajustar el sitio donde queremos lanzar nuestro hechizo seleccionando con mayor facilidad a los enemigos, pero en una pantalla táctil no resulta tan fácil

acertar en un punto, si tenemos en cuenta que los enemigos pueden ser demasiado pequeños o si se encuentran muy lejos.

El fallar en seleccionar a un enemigo causaba que el jugador se moviera, ya que habíamos hecho un tap sobre el mapa indicándole una nueva posición a la que dirigirse, esto resultaba en una muerte o derrota la mayoría de las veces ya que si se acumulaba una oleada o una cantidad bastante grande de enemigos los puntos de vida del jugador bajarían considerablemente.

El hecho de fallar sin querer causaba una sensación de pérdida de control, y llevaba por consiguiente a la frustración, haciendo que el jugador no quisiera seguir.

Por este motivo nos replanteamos cambiar los controles del jugador llevándonos a los de tipo 2.

Tipo 2

En esta segunda versión de los controles manteníamos el **click to move** pero se implementó un sistema distinto con las magias, ahora el jugador debería activar una magia pulsando sobre el botón correspondiente.

Controles tipo 2 click to move + selección de hechizo

Si se había pulsado uno de los botones de magia, la siguiente vez que el jugador realizara un click o tap se lanzaría un hechizo sobre el punto seleccionado instanciando el sistema de partículas correspondiente y si se mantenía el dedo sobre la pantalla se seguirían lanzando hechizos hasta que el usuario decidiera levantarlo lo que causaba una desactivación de la magia, mientras el jugador mantuviera el dedo sobre la pantalla podía realizar un drag para lanzar hechizos en distintos puntos causando mucho daño a los enemigos que fueran alcanzados.

Aunque resultaba bastante sencillo de jugar, le quitaba dificultad al juego y no es lo que queremos. Además el jugador debería estar alternando constantemente levantando el dedo para moverse y pulsando de nuevo una magia para volverla a lanzar, lo que le restaba tiempo de reacción, le obligaba a pararse a pensar, ¿Está activada la magia o puedo moverme? Porque en ocasiones olvidaba si había pulsado uno de los botones. Una de las posibles soluciones para

este problema habría sido simplemente resaltar la magia que estuviera seleccionada en ese momento.

Aun así, no era un tipo de control que terminara de gustar y el **click to move** no nos terminaba de convencer.

En este tipo de control se empezó a crear el sistema de selección de magias, llevándonos a la creación de nuevas clases.

En primer lugar los distintos tipos de magias elementales de los que el jugador podrá disponer, cada uno de estos instanciará un sistema de partículas distinto y causara diferentes cantidades de daño.

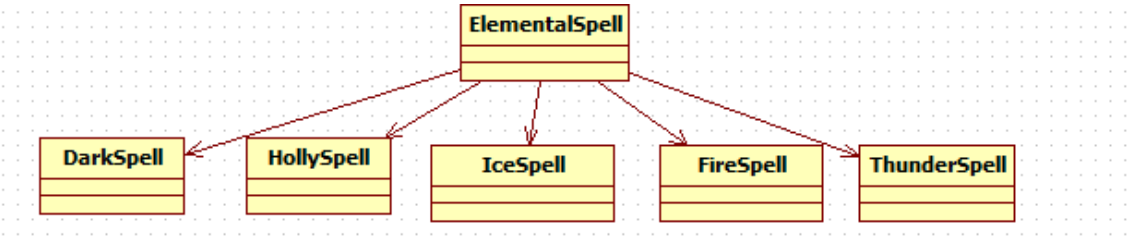
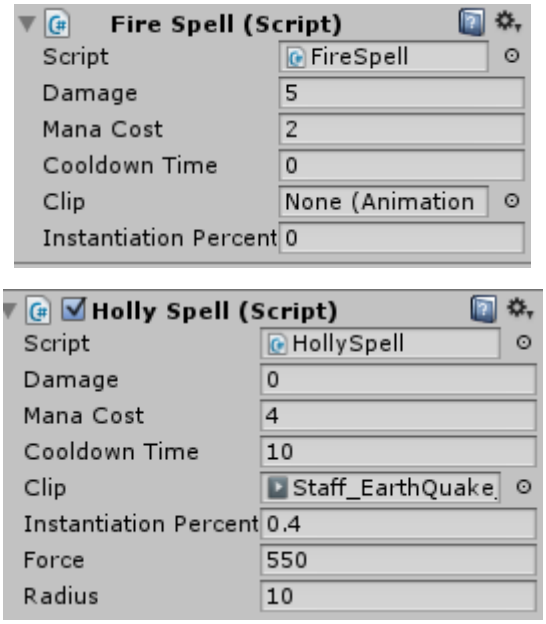


Diagrama con los tipos de magias



Vistas de los scripts de FireSpell y HollySpell

Lo que nos llevó a crear el SpellSelector, como su nombre lo indica, es un mecanismo más que usaremos para saber que hechizo seleccionó el jugador.

Tipo 3

En la implementación de este tercer tipo de control, se cambió el **click to move** por uno donde el personaje seguía el dedo del jugador por la pantalla, es decir, si el jugador mantenía su dedo sobre una esquina superior de la pantalla el personaje caminaría en esa dirección.

En cuanto a los hechizos, se realizó una pequeña variación, ahora el jugador debería mantener el botón de la magia pulsado, mientras este botón se encontrara pulsado el segundo toque en la pantalla detectado seria el punto donde se lanzaría el hechizo, si el jugador levantaba el dedo de la magia el personaje se empezaría a mover.

Cabe mencionar que este fue el primer control que hizo uso de la capacidad multitáctil de las pantallas.

Aunque no se ha mencionado anteriormente, para capturar el Input del usuario Unity nos ofrece distintos métodos dependiendo de la plataforma. En los controles anteriores el Input que se capturaba era el de un ratón con los métodos de la clase **Input**, **GetMouseButton**, **GetMouseButtonDown** y **GetMouseButtonUp**, que son transformados a sus equivalentes en toques al compilar en una plataforma móvil, pero hacer uso de estos métodos nos limitaba a un solo toque, ya que por lo general solo disponemos de un puntero de ratón, se realizó de esta manera para poder probar los tipos de controles a medida que se desarrollaban sin necesidad de crear builds específicas para las plataformas móviles ya que el input del ratón se puede capturar directamente desde el editor.

Unity también nos ofrece a través de su clase **Input**, métodos para capturar los toques en una pantalla táctil, GetTouch (int touch), en el cual deberemos indicar a que toque queremos acceder.

Una vez hemos accedido al toque deseado podremos consultar distinta información, como por ejemplo en qué posición de la pantalla se realizó o en qué fase se encuentra, los toques de Unity poseen las siguientes fases:

Began: Se realizó un toque sobre la pantalla.

Moved: Se movió el dedo que se encontraba en un punto de la pantalla a otro.

Stationary: Se ha realizado un toque pero no ha cambiado su posición.

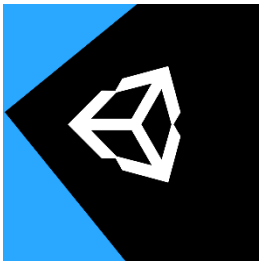
Ended: Se dejó de realizar el toque sobre la pantalla.

Canceled: El sistema canceló el seguimiento del toque.

```
void Update() {
    if (Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Moved) {
        Vector2 touchDeltaPosition = Input.GetTouch(0).deltaPosition;
        transform.Translate(-touchDeltaPosition.x * speed, -touchDeltaPosition.y * speed, 0);
    }
```

Ejemplo de uso de toques en Unity

Para poder probar de manera rápida los controles que se implementaron usando los métodos de **Input. GetTouch**, se hizo uso de la herramienta Unity Remote 4, que nos permite capturar los eventos de los toques sin necesidad de crear una build. Se configura de manera que al conectar el móvil al ordenador y ejecutar el juego en el editor, la ventana game de Unity aparece directamente en el dispositivo y envía los eventos de los toques generados en la pantalla del móvil.



Unity Remote logo

Unity Remote 4 app

Como la lógica de los controles se iba complicando cada vez más al añadir la de los toques, se vio la necesidad de crear un manejador de estos eventos que notificara a los controladores correspondientes. Se creó entonces el **TouchEventHandler** que nos daría información sobre la cantidad de toques que había en la pantalla, si estos toques se habían realizado sobre uno de los botones, si se había levantado uno de los dedos y que consecuencias tenía si esto se realizaba sobre un botón o sobre el terreno donde se movía el jugador.

Volviendo al funcionamiento de este tipo de controles, nos dimos cuenta de que resultaba divertido poder levantar grandes muros de fuego y derrotar a varios enemigos y rápidamente cambiar a otro hechizo o moverte con solo levantar un dedo.

Lanzando hechizo control tipo 3

Pero de nuevo aparecían los inconvenientes de hacerlo en una pantalla táctil, nuestras manos no son invisibles y con este tipo de control en muchas ocasiones cubríamos la pantalla

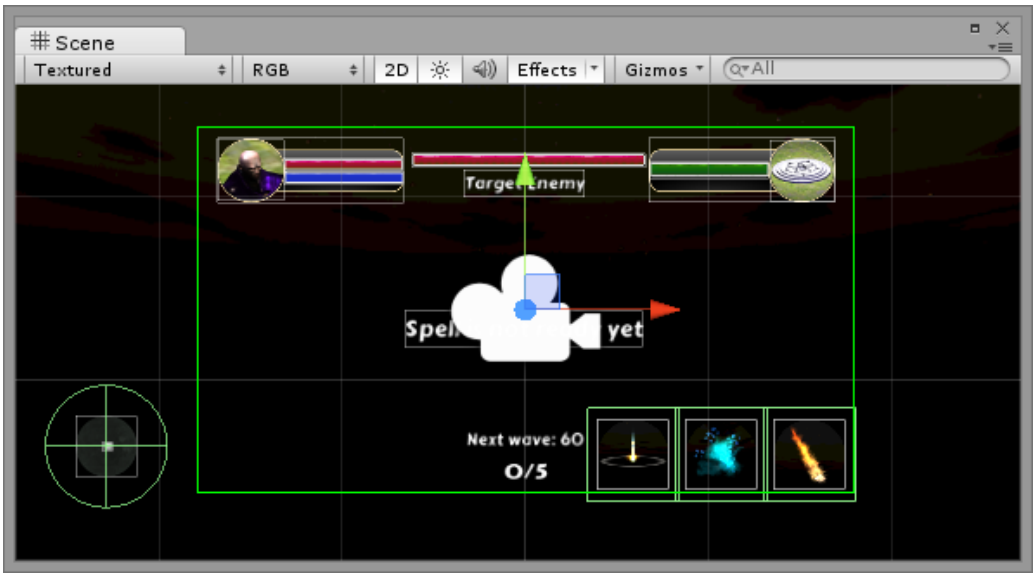
bloqueándonos a nosotros mismos y restándonos visibilidad de lo que sucedía, esto nos llevaba a una pérdida de información y a su vez de control. Se acumulaban enemigos o aparecían repentinamente cuando retirábamos nuestra mano de la pantalla.

Por esto se decidió replantear nuevamente el control del jugador.

Tipo 4

En este último tipo de control decidimos tomar inspiración de otros juegos e implementar algo de lo que veníamos huyendo, tener un joystick en la pantalla, también se modificó la forma en que se lanzaban los hechizos, para que el efecto de pulsar una magia fuera inmediato.

En la implementación de este Joystick se quiso hacer algo diferente, un joystick que apareciera en cualquier sitio de la pantalla y que desapareciera al dejar de tocarla permitiendo tener más espacio para apreciar el entorno, y no el típico joystick anclado a una esquina que ocupa siempre una porción de la pantalla, de esta manera el jugador tendría más flexibilidad a la hora de tomar el control del jugador.



Cámara de la interfaz gráfica control tipo 4

Como podemos observar, el joystick se encuentra fuera del recuadro verde que nos indica los límites de la cámara que dibuja la interfaz gráfica. Este joystick se moverá dentro del recuadro cuando el jugador pulse sobre la pantalla del dispositivo, causando que el joystick se ubique bajo su dedo y pueda empezar a mover el personaje inmediatamente.

El hecho de que el joystick se pueda reposicionar fácilmente permite al jugador reaccionar más rápido a distintos eventos y situaciones.

También se rediseñaron los hechizos para permitir al jugador más variedad de acciones, un hechizo de luz que empujara a los enemigos generando una onda en un radio alrededor del personaje.

Hechizo de luz

Un hechizo de hielo que se lanza en forma de cono frontal causando daño a múltiples enemigos.

Hechizo de hielo

Y por último un hechizo de fuego que dispara un proyectil, en un principio este proyectil era disparado siempre en la dirección hacia donde el jugador estaba mirando, pero resultaba casi imposible apuntar de tal manera que el proyectil impactara con el enemigo deseado, el simple hecho de que este hechizo no funcionara de adecuadamente o que no hiciera lo que el jugador esperaba le daba una sensación de frustración al no poder vencer a sus oponentes.

Para solucionar este problema se implementó un disparo magnético, que permite que los hechizos se dirijan al enemigo más cercano.

Para la detección de enemigos el personaje hace uso del mismo patrón observador-observer implementado con las torres, donde tiene un trigger que le indica que enemigos se encuentran cerca de él, para que no resultara en un efecto extraño este trigger se encuentra delante del jugador, de esta manera solo atacará a enemigos que se encuentren enfrente y no detrás.

Hechizo de fuego

Conclusiones y trabajo futuro

El definir el tipo de controles que deseamos en un videojuego es una parte crucial, que requiere de una gran cantidad de tiempo, planificación, creación de prototipos y testeo para verificar su usabilidad.

Los controles de un juego son la herramienta principal a través de la cual podemos hacer llegar o transmitir a los jugadores un conjunto de sensaciones, que deberán procurar hacer la experiencia del juego única y en cualquier caso hacerle sentir cómodo y en total control en todo momento.

Con la implementación de los distintos tipos de controles, hemos podido observar y experimentar como un mecanismo que permite la comunicación del usuario con el juego, en este caso las pantallas táctiles, se puede convertir en un fuerte aliado si sabemos aprovechar todas sus capacidades o un gran enemigo si no tenemos en cuenta esos pequeños detalles que pueden resultar cruciales.

Aunque no está del todo mal que nos aventuremos a probar cosas nuevas a querer innovar y crear algo diferente a lo que ya existe, es recomendable fijarnos en como lo han hecho los demás, nos puede ayudar a hacernos una idea y a evaluar una solución frente a otra ahorrándonos horas y horas de trabajo.

El análisis, diseño e implementación de un videojuego es un trabajo que requiere de capacidades tanto artísticas como técnicas, de ahí que resulte conveniente que se haga con un equipo interdisciplinar donde cada uno pueda aportar su granito de arena.

La implementación de un videojuego puede resultar en una tarea sumamente divertida y en una experiencia muy enriquecedora que lleva horas y horas de dedicación pero que al final, cuando se ven los resultados, se obtiene una sensación altamente gratificante.

Este proyecto pretendía ser un punto de partida para el videojuego Darkest Nights permitiendo realizar pruebas de distintos controles y jugabilidad de las cuales se pudiera obtener información que faciliten la toma de decisiones en fases posteriores del desarrollo.

A partir de aquí se puede empezar a planificar la continuación de dicho proyecto, enriqueciendo el contenido desde el punto de concepto de juego, diseño de jugabilidad, diseño artístico y nuevas funcionalidades que se quieran integrar.

Bibliografía

Unity Scripting API

http://docs.unity3d.com/ScriptReference/

Jesse Schell's, **The Art of Game Design**

Jeff W. Murray, **C# Game Programming Cookbook for Unity 3D**

Michael Doherty, **A Software Architecture for Games**

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.2884&rep=rep1&type=pdf

Unity: Now You're Thinking With Components

http://gamedevelopment.tutsplus.com/articles/unity-now-youre-thinking-with-components--gamedev-12492

Refactoring Game Entities with Components

http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy

Animate anything with Mecanim

http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/animate-anything

Gamification model canvas

http://www.gameonlab.com/canvas/