UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Videojuego Beast's Retreat en Unity con C# integrando RT-DESK: Interfaz y Editor

Proyecto Final de Carrera

Ingeniería Informática

Autor: Santiago Martínez Gómez

Director: Ramón Pascual Mollá Vayá

29 de septiembre de 2014

A Javier Baixauli Herraez,

por su ayuda incondicional que me ha

permitido llegar hasta aquí.

Resumen

Videojuego Beast's Retreat en Unity con C# integrando RT-DESK: Interfaz y Editor es un PFC que corresponde a una de las dos partes del desarrollo del juego *Beast's Retreat* creado con mi compañero Javier Baixauli.

Desarrollado con el motor *Unity,* el juego consiste en defender tu base de numerosas hordas de enemigos que irán llegando sucesivamente con el fin de destruirla. Basado en los conocidos *Tower Defense*, el jugador podrá disfrutar de una experiencia mejorada con escenarios y modelos 3D frente a los habituales juegos 2D que caracterizan el género.

Dejando de lado el habitual sistema de torres, esta parte del proyecto aporta al juego una gran innovación: la implementación de un sistema de compra y combinación de objetos, así como la edición de torres que el usuario deberá desbloquear y mejorar para defender su base. También abarca los menús de inicio del juego y en general toda la navegabilidad hasta los modos de juego desarrollados por mi compañero en su proyecto **Videojuego Beast's Retreat en Unity con C# integrando RT-DESK: Gameplay**.

Por último, la integración del *kernel* RT-DESK permitirá al jugador disfrutar de un *gameplay* con mayor rendimiento gracias a las herramientas que proporciona. Esta memoria documentará todo el proceso de desarrollo de la interfaz y el editor desde las primeras fases.

Palabras clave: videojuego, Unity, 3D, Interfaz, Editor, C#, RT-DESK, Tower Defense.

Tabla de contenidos

1. INTRODUCCIÓN

1.1. MOTIVACIÓN

Desde hace años nuestra principal motivación ha sido entrar en la industria del videojuego, lo que también se aplica a este proyecto. A pesar de que los informáticos cada vez son más necesitados y las empresas ofrecen puestos de trabajo, no parece estar pasando lo mismo en este ámbito. Por tanto es prácticamente necesario participar o desarrollar algún proyecto antes de poder poner un pie en este mundo del videojuego, y que mejor manera de hacerlo que con nuestro *PFC*.

Por otro lado, *Unity* está cogiendo nombre y ya se pueden ver muchos productos de gran nivel comercial creados con este motor de videojuegos multiplataforma. Su código fuente abierto y el abanico de opciones de desarrollo (*PS3*, Web, *Android*, *IOS*, entre otras), así como la cantidad de información, *plugins* y tutoriales disponibles en la red hace de *Unity* la mejor herramienta para crear nuestro primer proyecto, sin dejar de lado la calidad alcanzable.

Por último añadir que gracias a la guía de nuestro tutor encontramos una nueva motivación, crear algo diferente, es decir, algo innovador dentro de los juegos de este género que llame la atención. Tras informarnos decidimos que estableceríamos la diferencia en cuanto al rendimiento y racionamiento de potencia en comparación con otros juegos, y la manera de hacerlo era la integración de la *API RT-DESK*.

1.2. INFLUENCIAS

Por supuesto, nos hemos basado en otros juegos, tanto del mismo género como de otros diferentes, aunque siempre bajo la misma temática, la fantasía. A continuación se exponen aquellos con rasgos que nos hayan gustado para el desarrollo de *Beast's Retreat*:

- En concreto, para la parte del juego desarrollada en este proyecto nos basamos en **Doodle God**. Se trata de un ingenioso mezclador de elementos en el cual, a partir de lo más básico obtienes hasta un total de 115 elementos. Se puede encontrar en la web *Minijuegos*.

Ilustración 1: Doodle God de Minijuegos

- A pesar de que el juego es en 3D hemos querido mantener la esencia 2D para el lanzamiento de proyectiles gracias a la influencia de juegos como Bois D'Arc.

Ilustración 2: Bois D'Arc de Minijuegos

- Por último, y como mayor influencia de todas, los juegos del tipo *Tower Defense*. El juego que más nos ha impactado es sin duda **Kingdom Rush**, disponible en *Armor Games* y con un diseño sencillo pero muy atractivo.

Ilustración 3: Kingdom Rush de Armor Games

1.3. OBJETIVOS

El objetivo detrás de este desarrollo es ofrecer al usuario una experiencia *Tower Defense* en la plataforma *PC* diferente y más completa con respecto al resto de juegos de este género. *Beast's Retreat* contiene un sistema de interfaces que permitirán la compra, creación y edición de material de batalla que posteriormente será utilizada por el usuario en el *gameplay*.

Un objetivo paralelo es la integración de *RT-DESK* en el motor de videojuegos multiplataforma *Unity*, un simulador de eventos discretos orientado a simular sistemas mediante teoría de colas, que permitirá reducir el coste de computación del

dispositivo. Dado que la optimización que buscamos con *RT-DESK* va orientada a la física y control de trayectorias de los proyectiles en el *gameplay,* este proyecto solo abarca la integración y correcta adaptación del código fuente del kernel en *Unity*.

1.4. PROCEDIMIENTO UTILIZADO

El procedimiento utilizado para el desarrollo de *Beast's Retreat* ha sido similar al habitual ciclo de vida de un producto software, a pesar de tratarse de un videojuego. Podemos distinguir pues las diferentes fases de desarrollo mediante una pequeña ilustración:

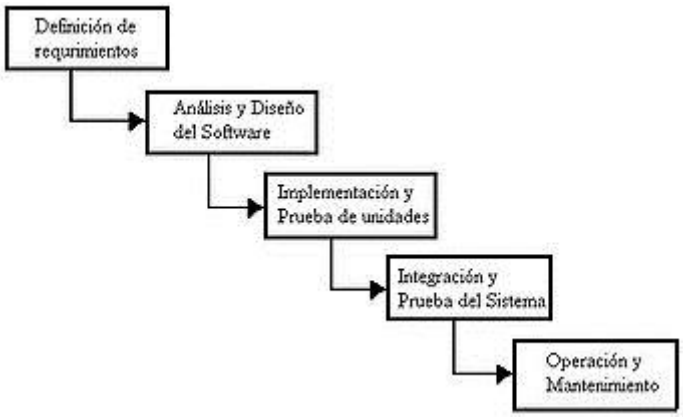


Ilustración 4: Ciclo de vida del software

La etapa de definición de requerimientos, en nuestro caso, puede considerarse el desarrollo del documento *GDD* (*Game Design Document*, véase en el Anexo A). En una fase temprana del proyecto describimos en éste todos y cada uno de los elementos que contendría el juego (música, escenarios, modos de juego…). Por otro lado el propio *GDD* también contiene elementos de análisis y diseño, ya que incluye bocetos, transiciones y descripciones de funcionalidad, tanto de interfaz como de *gameplay*.

Paralelo al análisis podríamos incluir una fase de planificación. El diagrama de Gantt fue de gran ayuda para ir controlando el avance del desarrollo y el tiempo restante hasta la fecha de entrega, así como la asignación de tareas y estimaciones en cuanto a costes temporales en función de los recursos utilizados.

A continuación la fase de implementación. Contiene en su totalidad el desarrollo del código fuente, diseño de los modelos, iconos y su funcionalidad. Es sin duda la fase más larga del proyecto y la que requiere mayor coordinación entre los programadores, ya que podría haber fallos de coherencia en el código. Va muy ligada a la fase de

integración y prueba del sistema, que engloba la unión en un mismo proyecto de ambas partes, la interfaz de usuario y el *gameplay* de *Beast's Retreat*.

Por último la fase de mantenimiento. Al ser un juego con una fuerte base en los elementos y su combinación el mantenimiento será extenso, ya que siempre podrán incluirse nuevos materiales o elementos combinables a partir de los cuales surgirán nuevas torres, armas, entre otros.

1.5. RIESGOS

Como estudiantes de Ingeniería Informática, y por tanto especializados en la programación, no disponíamos de un diseñador que colaborase con nosotros para el *Look&Feel* del juego. Este es un riesgo alto ya que nos arriesgábamos a una mala presentación visual del juego. Consideramos que para versiones tempranas del juego lo importante es mostrar una buena funcionalidad (como en las versiones alfa), pero sin duda un factor decisivo de cara a una posible comercialización.

Por otro lado, nunca habíamos desarrollado un videojuego a pesar de ser uno de nuestros hobbies y entender acerca del ámbito y los géneros. Por ello elegimos *Unity*, por ser una de las herramientas más extendidas y con mayor soporte para nuevos desarrolladores para minimizar el riesgo temporal, ya que antes de comenzar con el desarrollo hubo una fase de preparación y aprendizaje a través de tutoriales escritos y online.

1.6. IMPORTANCIA DEL SECTOR

Los videojuegos suponen hoy en día uno de los principales entretenimientos para usuarios de todas las edades y tipos, y crece al mismo ritmo que crece la tecnología y la variedad. Actualmente hay una amplia gama de dispositivos que soportan la simulación de juegos, dispositivos móviles, consolas (portátiles y no portátiles), *PC*… Esto aumenta las opciones de entrada de nuevos desarrolladores en el ámbito.

Dentro de la industria del videojuego, es sabido que las grandes empresas dominan el comercio (al menos las ventas para grandes plataformas como *PS4*, *Xbox One*, etc…), sin embargo puede observarse que la industria se está expandiendo. La prueba principal de este crecimiento son los juegos *indie,* que abundan plataformas de venta digitales, en portales web y dispositivos móviles, aunque también comienzan a ver la luz en algunas plataformas como *PS3* o *PS4*.

Datos obtenidos en la página de *CNN México* afirman que se espera que el sector cierre 2014 con ventas por arriba de los 74,000 millones de dólares. Pero no sólo se trata de los videojuegos, sino también de masas. Al cierre de 2013 se registró que en el mundo había más de 1,200 millones de personas que entraban dentro de la categoría de videojugador o *gamer*.

En concreto, en nuestro país, hoy el videojuego se posiciona como la principal opción de ocio para cada vez más segmentos poblacionales. Así, la a penetración social del videojuego continúa aumentando, y es que a día de hoy el 62% de los menores de edad y el 24% de los adultos españoles se declaran ya usuarios habituales , según el último estudio realizado por la consultora *Gfk.*

Por tanto podemos concluir que este sector tiene una gran visión de futuro dentro de la informática. Teniendo en cuenta el nivel gráfico y de computación que está alcanzando la industria es cuestión de tiempo que los desarrollos se vuelquen en las nuevas tecnologías de realidad virtual, como ya puede observar con *Oculus Rift*.

1.7. ASPECTO FINAL

Beast's Retreat tiene un aspecto basado en elementos del género fantasía y medieval, intentando que el usuario experimente la mayor inmersión posible dentro de la historia y el personaje. Los diseños y distribución de los elementos están simplificados para facilitar la interacción con las diferentes funcionalidades del juego. El resultado final se muestra con algunas ilustraciones:

Ilustración 5: Menú principal de Beast's Retreat

Ilustración 6: Gameplay de Beast's Retreat

2. DESCRIPCIÓN

Beast's Retreat es una nueva experiencia de *Tower Defense* creada con el fin de entretener a los usuarios mediante elementos de investigación y batalla:

- El enfoque de investigación en el juego permite la creación de torres de defensa de forma progresiva, es decir, combinando elementos básicos para obtener objetos más complejos. El objetivo final es obtener torres con las que defender tu base mediante la combinación de:
    - Una estructura base.
    - Un personaje o raza.
    - Un arma.
- El enfoque bélico permite utilizar las torres diseñadas en el editor para mantener a raya las hordas de enemigos que irán llegando a lo largo de cada escenario. Antes de cada batalla se podrá planificar la formación de las torres y sus posiciones para maximizar la eficiencia de cada una y asegurar la victoria.

La combinación de ambos elementos pretende proporcionar al usuario la sensación de asemejarse a un general de guerra que está en proceso de reconquista.

2.1. ARGUMENTO

El jugador se pondrá en la piel del mejor estratega de la Alianza de los Doce, una unión de doce reinos humanos, elfos y enanos que antaño se unieron para ganar la Guerra de los 100 Siglos que acabó con los conflictos entre la alianza y el ejército de las bestias provenientes de una isla más allá del mar que los humanos llaman Fin de La Costa, trayendo prosperidad y paz a los reinos que conformaban la alianza. Sin embargo con el tiempo, tras no obtener beneficios por la guerra, la corrupción y la codicia de algunos líderes empeoró la economía y las relaciones entre la unión.

En el momento más crítico, a punto de la disolución de la Alianza de los Doce y una posible guerra civil, un nuevo líder aparece entre las bestias y lidera un ataque a la Península con el fin de destruir todo a su paso y sumir en la desesperación a todas la razas. Debido al miedo y a la influencia del nuevo líder, algunos reinos de las diferentes razas caen rendidos a su poder y traicionan a la alianza, enfrentándose a aquellos que no han sucumbido a las tinieblas y que hacen todo lo posible por resistir. Cuando todo parece perdido, en el ataque a un pequeño pueblo con talento para la creación de armas, el jugador liderará la resistencia, comenzando con una campaña de reconquista

que le convertirá en el mejor general de la historia, consiguiendo de nuevo el apoyo de todos los reinos y devolviendo de nuevo la esperanza y la prosperidad a la Alianza de los Doce.

2.2. GÉNERO

Anteriormente hemos clasificado *Beast's Retreat* como un juego tipo *Tower Defense* pero, ¿qué implica pertenecer a este género?

Se trata básicamente en un tipo estrategia en el cuál el usuario debe fortificar mediante torres una base que tiene que defender de los enemigos que van llegando. Cada torre tiene un coste y habilidades propias, donde reside la verdadera esencia del juego. El éxito al superar el juego o no depende de la habilidad del jugador para posicionar las torres y maximizar su eficiencia.

Sin embargo, *Beast's Retreat* no es el típico *Tower Defense*. Debido a las influencias comentadas en el apartado 1.2 en la introducción, se ha combinado éste género con el tipo habilidad. A pesar de ser 3D se ha mantenido la jugabilidad 2D de los habituales simuladores de tiro con arco que se pueden encontrar sobre todo en algunas páginas web.

Por último, los modos de creación podrían situarlo también, dentro del tipo estrategia, en la categoría de lógica. El mezclador permite al jugador obtener nuevos objetos más complejos que pueden ser obtenidos mediante la combinación de objetos más simples.

2.3. JUGABILIDAD

La interacción y los controles del juego están orientados a la comodidad de usar únicamente el ratón. Antes de nada mostraremos cual es la apariencia del juego:

Ilustración 7: Gameplay 2 de Beast's Retreat

Para seleccionar una torre solo hay que pulsar en cualquiera de los botones de selección de torre que podemos ver en la siguiente imagen. Respetan el orden de las torres en el escenario y cada botón contiene imágenes que responden al tipo de arma que tiene cada una.

Ilustración 8: Botón de selección de torre

Como puede observarse en la siguiente ilustración, la formación de las torres está compuesta por 2 filas. Esto significa que los enemigos pueden venir por cualquiera de ellas y sin embargo no podemos hacer fuego cruzado. El botón de cambio de vista permite al usuario cambiar la orientación de la cámara para poder controlar las torres situadas en la parte contraria.

Ilustración 9: Botón de cambio de vista

El botón wave permite provocar a la siguiente horda de enemigos en caso de que el jugador haya acabado con la anterior antes de que el tiempo destinado a ello haya finalizado.

Ilustración 10: Botón wave

Podemos pausar el juego o controlar el sonido con los botones de sonido y pausa situados en la esquina superior derecha de la pantalla.

Ilustración 11: Botones mute y pausa

Ilustración 12: Juego en pausa

En la esquina superior izquierda tenemos la vida de nuestra base. Cada vez que un enemigo impacte contra ella morirá, pero nos restará puntos. Según el tipo de enemigo el impacto nos costará 1 o más puntos, haciendo que se acentúe la necesidad de habilidad del jugador para lograr acertar con los proyectiles a largas distancias.

Ilustración 13: Barra de vida

Para disparar debemos pulsar en cualquier punto del escenario que no sea un botón y la barra de carga empezará a cargar, cuando se suelte la torre disparará en función de la potencia dada al disparo.

Ilustración 14: Potencia de la torre seleccionada

Por último para manejar la cámara solo debemos pulsar con el botón derecho y arrastrar para movernos en los ejes x e y. Para hacer zoom debemos utilizar la rueda del ratón.

2.4. MODOS DE CREACIÓN

Como bien se ha explicado anteriormente, la parte de investigación del juego estaba enfocada a la compra y combinación de elementos para la obtención de objetos más complejos. Podemos definir 3 fases que coinciden con los modos de creación que proporciona *Beast's Retreat*:

2.4.1. TIENDA DE OBJETOS

La tienda de objetos permite al usuario comprar los elementos más básicos y de fácil obtención en función de los créditos disponibles. Al empezar el juego obtiene 4000 créditos que deberá utilizar con cabeza para comprar los elementos necesarios para desbloquear las primeras torres de defensa. Cada vez que supere un escenario el jugador obtendrá créditos en función del daño recibido y la manera de eliminar a las hordas enemigas, que le permitirán comprar nuevos elementos más caros y que darán lugar a torres más potentes.

Las siguientes ilustraciones mostrarán la jerarquía de objetos que se pueden comprar. Para los jugadores que busquen una compra rápida existen determinadas ofertas que les proporcionarán un conjunto de elementos de manera rápida. Estas ofertas irán actualizándose y ampliando la variedad más adelante.

Ilustración 15: Tienda (Panel de Ofertas)

También se pueden comprar los objetos por separado. Se pueden observar dos tipos:

- Los materiales, es decir, ítems no elementales. Dentro de éstos podemos encontrar:
    - Planos de las torres (para construir estructuras).
    - Alianzas (con las diferentes razas de la Península).
    - Armas

- Los elementos, que pueden ser combinados con los materiales para obtener nuevas razas, armas o estructuras elementales.

Ilustración 16: Tienda (Panel todo)

2.4.2. MEZCLADOR DE ELEMENTOS

Una vez comprados los materiales es hora de mezclarlos. Las combinaciones funcionan de la siguiente manera. Una combinación consiste en la unión de un material y un elemento. Hay un conjunto de combinaciones disponibles, como por ejemplo:

- Combinar un **Plano pequeño** y un elemento **Madera** da como resultado una **Estructura de madera pequeña**.

Con el tiempo se añadirán más combinaciones posibles. Las combinaciones disponibles se pueden encontrar en el *GDD* (Anexo A).

Ilustración 17: Mezclador de objetos

2.4.3. EDITOR DE TORRES

El Editor de torres permite mediante un sencillo asistente crearlas paso por paso (al cual se accede desde el botón Create Tower), orientando al usuario que objetos debe utilizar en cada momento. Al final del asistente se podrá comprobar que objeto se ha desbloqueado con la combinación seleccionada.

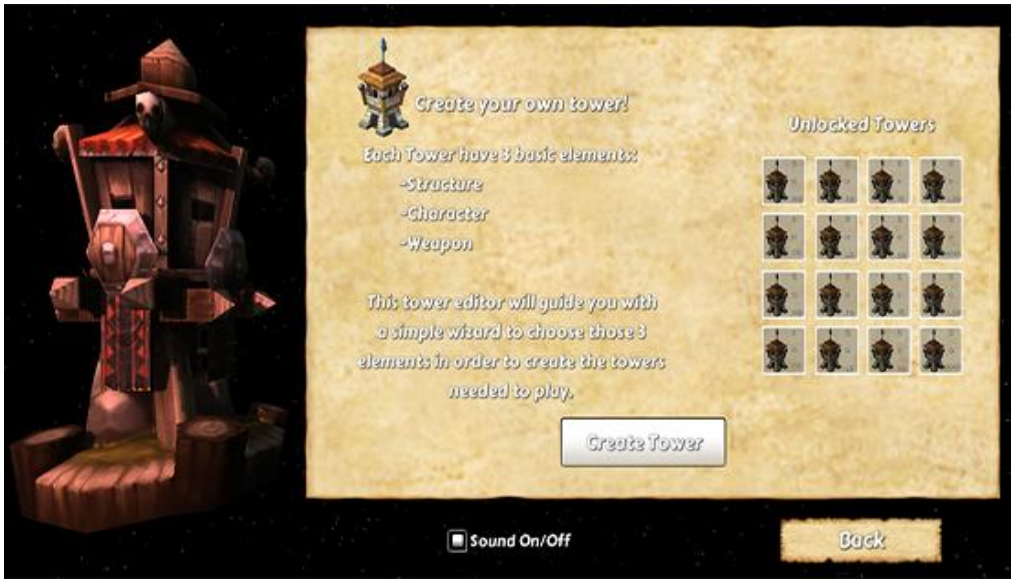


Ilustración 18: Editor de torres

2.5. *DETALLES TÉCNICOS*

Las funciones o características del juego pueden dividirse en función de los proyectos que conforman *Beast's Retreat*:

- Interfaz y Editor (este proyecto)
    - Menú principal con animación
    - Selección de escenarios
    - Navegabilidad entre modos de creación
    - Persistencia de datos
    - Gestión de créditos
    - Gestión de escenarios superados
    - Tienda y compra de objetos
    - Mezclador y combinación de elementos
    - Editor y desbloqueo de torres
    - Gestión de objetos
    - Formación y selección de torres por escenario

- Gameplay (**Videojuego Beast's Retreat en Unity con C# integrando RT-DESK: Gameplay** de Javier Baixauli)
    - Máquina de estados determinista para cada torre
    - Controlador de recargas
    - Física para proyectiles
    - Movimiento de cámara limitado
    - Cambio de perspectiva
    - Interfaz de juego
    - Controlador de hordas de enemigos
    - Generador de partículas para crear efectos elementales
    - Mostrar daño por impacto
    - Disparos críticos en la cabeza
    - Controlador de colisiones

3. ESTADO DEL ARTE

En este apartado haremos un repaso de las técnicas utilizadas y relacionadas con este proyecto. Es importante hacer un estudio de mercado y conocer la gama de opciones que hay disponibles para el desarrollo de un videojuego, de manera que podamos elegir aquella con menor coste y mayor eficiencia. Por otro lado, también es fundamental explicar, al menos brevemente y de manera introductoria, la tecnología utilizada y el porqué de nuestra elección.

3.1. MOTORES GRÁFICOS

Antes de decidirnos por el motor gráfico *Unity* hicimos un pequeño estudio acerca de los motores disponibles actualmente para comprobar que realmente era la mejor opción para nosotros. En este apartado daremos un repaso a unos cuantos motores gráficos, describiendo para qué tipo de juegos están pensados, así como su capacidad de portabilidad a otras plataformas.

- **Vision Engine By Trinigy** (**www.trinigy.net**). La empresa alemana **Trinigy** nos proporciona un más que interesante motor gráfico para *PC*, *Xbox 360* y *PlayStation* 3. Sus diversas herramientas nos ofrecen la oportunidad de crear varios tipos de juegos: rol, acción, estrategia… Muy bien documentado y con un precio asequible.

- **C4 Engine by Terathon** (**www.terathon.com/c4engine**). Sin duda una apuesta muy interesante, arriesgada y decidida del diseñador Eric Lengyel. Éste nos presenta un motor con muchas características técnicas, fácil de adaptar a varios tipos de juegos, con buena documentación y un precio fantástico. Funciona para *PC*, *PlayStation* 3 y *Mac OSX* (sí, en *Mac* también se juega). Su única pega es su calidad gráfica. En todo caso, se trata de una buena decisión para empezar a crear un proyecto de videojuego.

- **Gamebryo Engine by Emergent** (**www.emergent.net**). Un motor con sobrada experiencia y con títulos que avalan su calidad como **Civilization 4** u **Oblivion**. Funciona en *Xbox 360*, *PlayStation* 3, *PC* y, recientemente, han anunciado su

versión para *Wii*. Un motor bien documentado y que le daría a nuestro título una apariencia realmente buena.

- **Vicius Engine by Vicious Cycle Software** (www.viciousengine.com). Un motor gráfico sin grandes alardes pero que tiene la ventaja de que funciona en *PSP*. Utilizado en juegos como **300: March to Glory** es ideal para juegos de acción con la portátil de **Sony**. Su sistema de trabajo es muy fluido y está pensando para producciones rápidas.

- **Torque for Wii** (www.garagegames.com/products/torque/twii). Este popular motor de **GarageGames** tiene una versión más que interesante para **Wii**, con todas las posibilidades que ofrece la revolucionaria consola de **Nintendo** y, además, con un interesante coste.

- **SourceEngine by Valve** (www.valvesoftware.com). Poco podemos decir de este software utilizado en las producciones de **Half-Life**. Se trata de un motor gráfico sencillamente impresionante que lo tiene todo para que un proyecto se convierta en súper ventas. Es ideal si el proyecto es un shooter en primera persona; si no, hay opciones mejores. Ahora, su principal problema es el coste de licencia.

- **CryEngine2 by Crytek** (www.crytek.com). La empresa alemana **Crytek** sorprendió a todos con su motor para el juego **Far Cry**, y ahora con **Crysis** nos con unos gráficos y un despliegue técnico sencillamente maravilloso. Ideal para proyectos *PC* y *Xbox 360*, se ha anunciado ya el desarrollo *PS3*. Muy orientado a shooters en primer persona.

- **Unreal Engine by Epic Game** (www.unrealtechnology.com). Tiene un coste desproporcionado, pero la fama de ser el mejor motor gráfico de videojuegos. Es multiplataforma y tremendamente potente como se puede ver en creaciones del calibre **Gear of Wars**. Se ha licenciado para cientos de títulos que nos inundarán el mercado en los próximos años.

A pesar de que nos podría convenir más de un motor gráfico de los expuestos anteriormente, podemos ver que prácticamente no hay ninguno gratuito. Tal vez **Gamebryo** nos habría venido bien para realizar nuestro proyecto. Sin embargo, *Unity*

presenta una increíble comunidad de desarrolladores e información en la red, que te permiten desarrollar prácticamente cualquier tipo de juego además de las prestaciones que tiene, más que suficientes para desarrollar nuestro proyecto:

- Exportación Multiplataforma gratuita (móvil, web, PC y Mac, PS3).
- Scripting en Java Script, C# ó Boo. Contiene un editor de código.
- Soporta gran cantidad de paquetes 3D y texturas de múltiples extensiones.
- Soporte para creación de redes y juego en línea.
- Editor de terrenos y vegetación incorporada.
- Creación de Videojuegos 2D y 3D.
- Gestor de animaciones, arboles de mezcla y máquinas de estado.

3.2. EL MOTOR UNITY 3D

Una vez elegido el motor gráfico debemos escoger el lenguaje de programación e informarnos acerca de su relación con el editor de *Unity*.

Ofrece tres lenguajes de programación diferentes, estos son *UnityScript* (un *Javascript* con ligeras modificaciones), *Boo* y C#, siendo todos combinables en un mismo proyecto.

Decidimos de entre estas opciones la que mejor conocíamos, C#. Las principales razones fueron:

- Conocimiento del lenguaje con una buena profundidad gracias a diversas asignaturas cursadas en la carrera.
- Mayor facilidad de depuración frente al lenguaje UnityScript, ya que éste último tiene un fuerte tipado dinámico.
- Inmensa diferencia en cuanto a documentación y aprendizaje en la red, mucho mayor en C# que en Boo.

C# es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET. Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma .NET, similar al de Java, aunque incluye mejoras derivadas de otros lenguajes.

Una vez conocido el lenguaje a utilizar y las posibilidades, nos disponemos a aprender sobre el ciclo de vida de Unity de los objetos y el funcionamiento del editor.

Los elementos básicos en *Unity* se llaman *GameObject*. La mayoría de elementos del juego dependen de estos objetos o están enlazados a uno. Cualquier tipo de elemento, ya sea físico, de interfaz o scripts, debe estar asociado a un *GameObject* por código o en el inspector para que *Unity* pueda compilarlos e incluirlos en la escena.

Todos estos *GameObjects* se agrupan dentro de una *Scene*. Una *Scene* o escena es un entorno donde se desarrolla una parte determinada del juego. Aunque todo un juego podría desarrollarse dentro de una misma *Scene*, conviene separar el contenido en varias diferentes para mejorar la eficiencia y tiempo de computación, ya que aunque los elementos no sean visibles en la cámara continúan renderizándose.

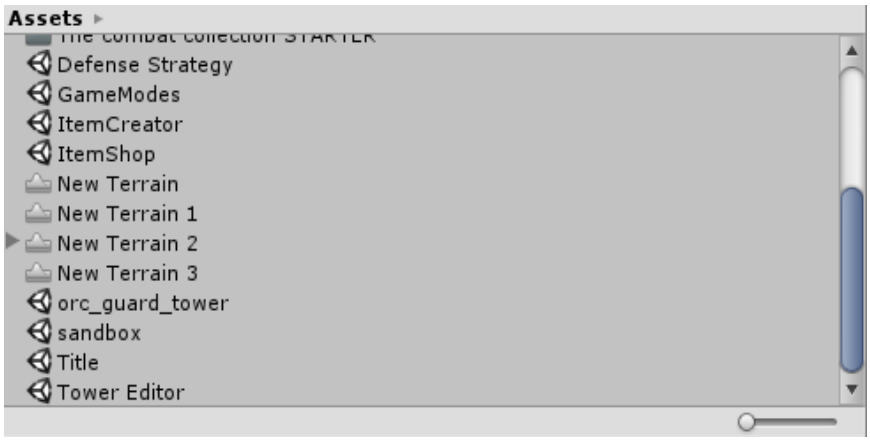


Ilustración 19: Una scene por cada modo

Hemos comentado anteriormente los *GameObject* y la importancia de que cada elemento esté asociado a uno de ellos. Algunos de los componentes más importantes son el *Renderer*, que se encarga de que el objeto sea visible, dándole una forma y un color o textura, el *Rigidbody* y el *Collider*, que gestionan las colisiones con otros elementos y las características de la física simulada por *Unity*, la *Camera*, que es, como indica su nombre, el encargado de renderizar la escena de acuerdo a su configuración y el componente *Script* que permite asociar el comportamiento de un script con un *GameObject*.

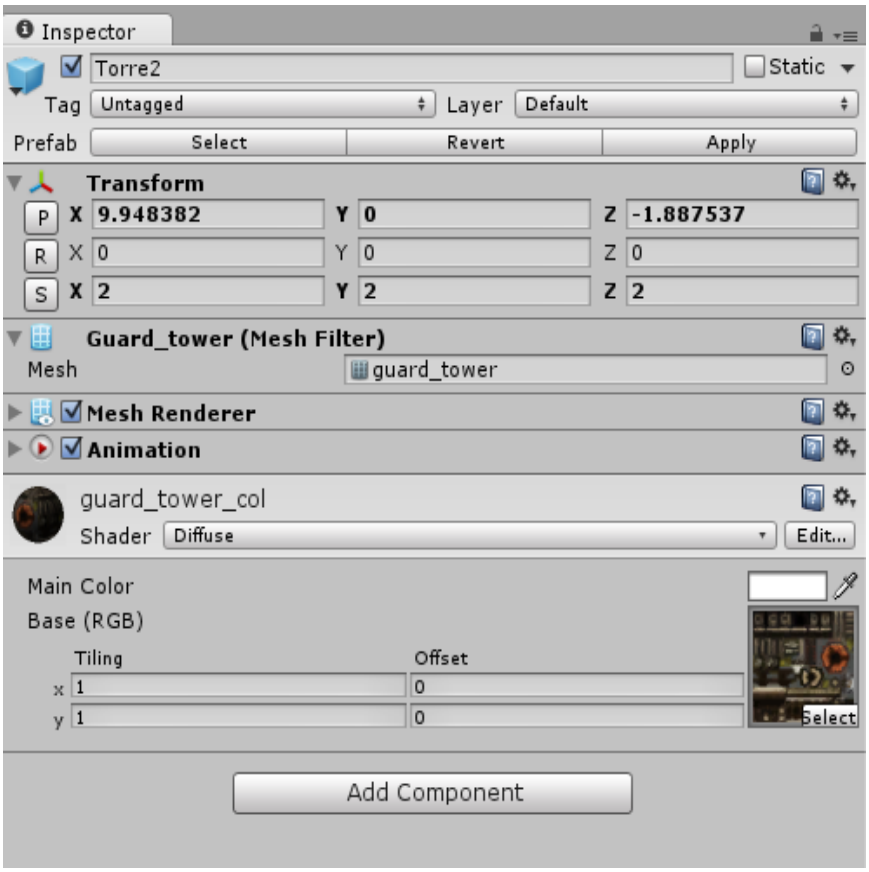


Ilustración 20: Ejemplo de componentes en un GameObject

Todos los atributos de los componentes se pueden editar tanto por código como desde el editor en tiempo de ejecución. Esto es especialmente interesante en los atributos de los scripts, los cuales si se declaran públicos pueden editarse para testear de forma sencilla sin tener que editar el código cada vez. Por otro lado también hay que destacar la facilidad del motor para asociar objetos de la escena con los scripts, ya que declarando una sencilla variable y haciendo el correspondiente *attach* ya puedes acceder a todas sus propiedades sin ningún tipo de problema.

Ilustración 21: Ejemplo de script adjunto con variables asociadas a otros GameObject

Como podemos ver en la ilustración 20, todos los componentes pueden habilitarse o deshabilitarse excepto uno. El componente *Transform* viene por defecto en todos los *GameObject* y decide la posición, rotación y escala que tendrá el objeto. Hay que tener en cuenta que esta componente depende de la jerarquía de los objetos, es decir, la distribución de éstos en la escena. Si un componente es hijo de otro (por ejemplo, un botón que tiene dentro un *label* o etiqueta), aunque el componente hijo no se encuentre en el origen de coordenadas, sus coordenadas serán (0, 0, 0), debido a que su sistema de referencia es el componente padre.

Unity ofrece unos métodos que son ejecutados para cada etapa del ciclo de vida de un *GameObject*. De esta forma reescribiendo los métodos manejadores somos capaces de controlar los objetos durante todo su ciclo. Para poder acceder a estos métodos debemos hacer uso de *MonoBehaviour*.

Esta clase es la base del motor. Cualquier clase que quiera acceder a ciertas funciones básicas del motor debe heredar de ésta. Esto permite acceder a métodos como *Awake()*, que es la primera instancia de un script en la *Scene*, el *Start()* que se ejecuta inmediatamente después y permite inicializar todas las variables, o el *Update()*, que se ejecuta continuamente mientras el *script* esté activo, una vez por cada *frame*. Al crear un nuevo script *Unity* genera automáticamente código para estos dos últimos métodos. Se debe tener cuidado con el método *Update()* ya que, en caso de no querer sobreescribirlo, si se deja el método vacío volverá a llamar a *Update()* una vez más por cada frame, lo que nos haría perder eficiencia y ganar coste de computación.

En la siguiente imágen podemos ver el ciclo completo de vida de un *GameObject*:

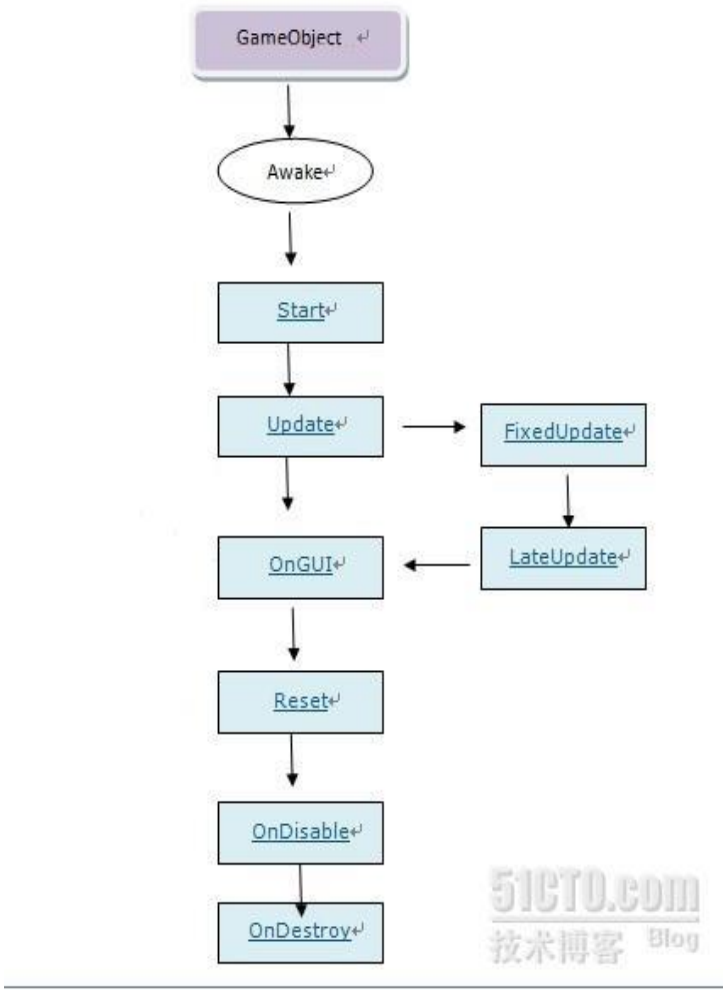


Ilustración 22: Ciclo de vida de un GameObject

3.3. NGUI

A nivel de interfaz, el método clave es el método *OnGUI()* de la clase *MonoBehaviour*. *OnGUI()* es un método similar a *Update()* que actualiza cada *frame* en caso de estar utilizando objetos de interfaz. Estos objetos los podemos obtener a través de la clase *GUI*, a partir de la cual descienden cualquier tipo de elemento de interfaz como *labels, sprites,* botones, entre otros.

Sin embargo, como hemos dicho en apartados anteriores, *Beast's Retreat* aspira a tener una interfaz de usuario rica y con una gran funcionalidad orientada a la investigación y obtención de nuevos objetos. Los elementos de la clase *GUI* requieren de inicializaciones y código escrito con el fin de situarlos en la pantalla por medio de coordenadas. Esto supondría una implementación tediosa y de gran coste temporal.

Buscando una mejor opción para este problema dimos con **NGUI**. Se trata de un paquete de *Unity* orientado exclusivamente a facilitar el diseño de interfaces. Cualquier tipo de elemento obtenido por medio de la clase *GUI* tiene en *NGUI* una clase propia (por ejemplo, una *label* equivale a la clase *UILabel*), con diversas variables y objetos por defecto que permiten posicionarlos en la pantalla manualmente y personalizar el estilo, generando el código automáticamente en *background*.

NGUI dispone de diferentes ventanas de ayuda en el editor para generar tus propias interfaces:

- En primer lugar, el **UI Wizard**. Esta es una ventana sencilla que permite añadir a la escena los elementos necesarios para comenzar a desarrollar la interfaz. Puede elegirse entre 2D y 3D. En este caso hemos elegido siempre 3D ya que en algunas escenas se combinan con escenarios 3D que actúan a modo de fondo. La opción 3D añade un objeto *UIRoot,* que contiene una *Camera* y un *Panel.* La cámara permite visualizar **únicamente** los objetos de la capa UI, es decir, los objetos de interfaz añadidos, mientras que el panel actúa como contenedor de elementos de interfaz menores, aunque también pueden situarse fuera siempre y cuando estén dentro del *UI Root*.

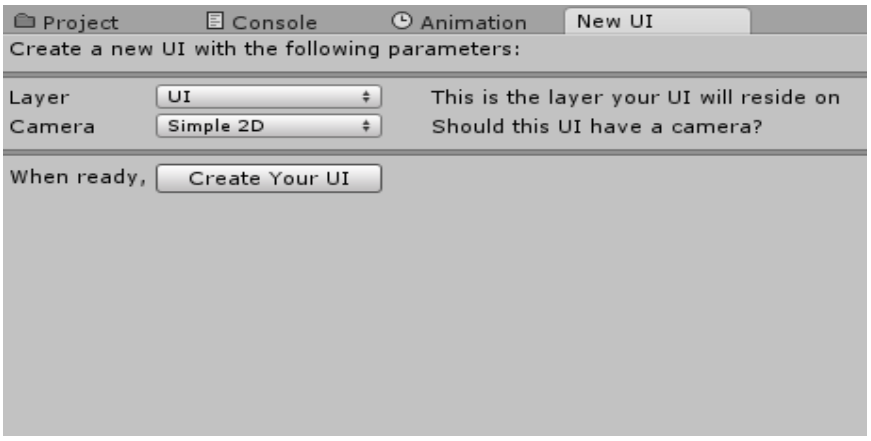


Ilustración 23: UI Wizard

- Tanto los paneles como las cámaras tienen una ventana de soporte para organizarlas, activarlas y situarlas en la cámara correspondiente. Se trata de las ventanas **Camera Tool** y **Panel Tool**.

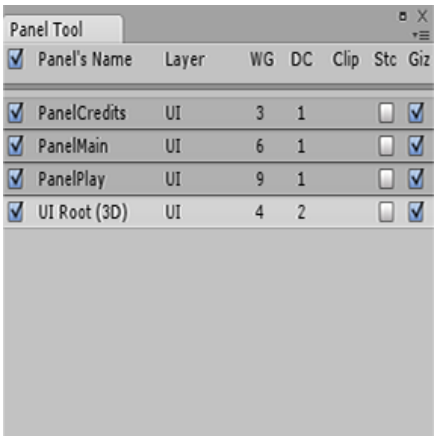


Ilustración 24: Panel Tool

Ilustración 25: Camera Tool

- A continuación, el **Widget Wizard**. Esta es la ventana que más se utiliza, ya que es la que permite incorporar a la escena los elementos de interfaz. Es tan simple como seleccionar el elemento, los *sprites* que quieras añadirle para cada estado (normal, pulsado, bloqueado en el caso de los botones, por ejemplo) y pulsar "*Add to*", que lo añadirá al panel seleccionado.

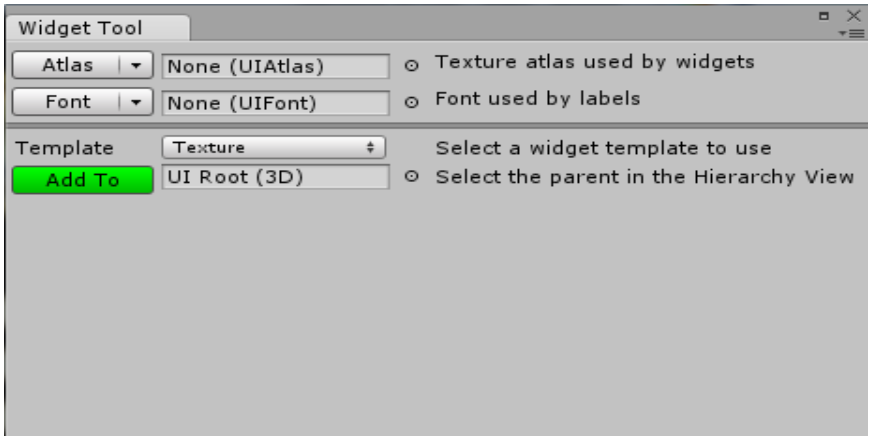


Ilustración 26: Widget Wizard

- Como se puede observar en la parte superior del *Widget Wizard*, hay dos elementos que tienen que seleccionarse para poder crear los elementos, *labels*, etc. Se trata de los *Atlas* y las *Fonts*. Estos elementos son los que almacenan todos los *sprites* utilizados en el proyecto y los tipos de letra para los *labels*. Para cada uno de ellos existe su propia herramienta de creación. En primer lugar el **Atlas Maker**. Esta herramienta permite crear nuevos *atlas* o actualizar los ya creados mediante el añadido de nuevos *sprites*. La propia herramienta almacena todos los *sprites*, creados por separado, en una única imagen

automáticamente al añadirlos. Y por último el **Font Maker**, que consiste simplemente en añadir a *Unity* una *font* creada externamente al motor.

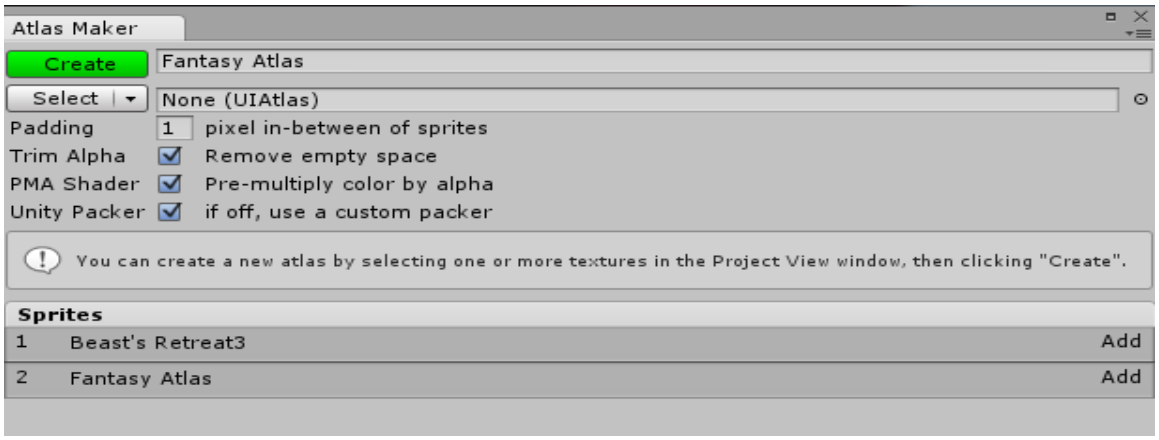


Ilustración 27: Atlas Maker

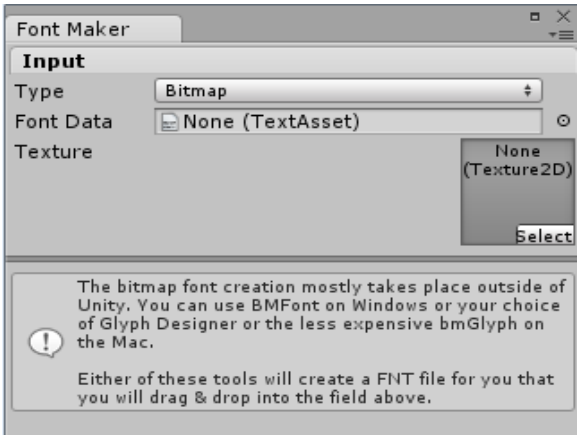


Ilustración 28: Font Maker

A pesar de las facilidades que proporcionaba NGUI, se trata de un paquete que sólo puede utilizarse en su mayor versión mediante la compra de éste en el *Asset Store*. Por tanto la versión utilizada, aunque era gratuita, es muy anterior a la actual y por tanto en algunos apartados está obsoleta (por ejemplo, la funcionalidad para arrastrar objetos por la pantalla está mucho más desarrollada ahora que en la versión utilizada). La versión de NGUI utilizada en este proyecto es la 2.7.0, mientras que actualmente NGUI se encuentra en la versión 3.7.3 con diferentes precios:

- La versión estándar se puede obtener en el *Asset Store* de *Unity* por 95$.
- La versión profesional puede obtenerse por 200$ y proporciona acceso a un repositorio propio de *NGUI* con las últimas actualizaciones y pruebas beta.

- Si una compañía necesita múltiples copias de *NGUI* para un estudio, puede escoger la oferta de *NGUI Site License* por 2000$.

Para acceder a cualquier tipo de información acerca del *NGUI*, descargas u otros puedes acceder a su espacio en la página oficial.

4. PLANIFICACIÓN

La planificación de un proyecto es una etapa que puede perfectamente integrarse, combinarse o realizar paralelamente con la etapa de análisis. Sin embargo, debido a ciertos puntos que queríamos especificar y que habrían hecho el apartado de análisis muy extenso, se ha decidido dividirlos en dos apartados diferentes.

4.1. DESCOMPOSICIÓN DEL PROYECTO

En el apartado 1.4 se ha hablado a grandes rasgos de las fases del proyecto basadas en el ciclo de vida del software habitual. Este apartado tiene como fin detallar las tareas o actividades que se iban realizando en cada fase del proyecto:

- **Fase de investigación**: En esta fase el proyecto todavía no se había comenzado a desarrollar. Dado que no habíamos utilizado *Unity* en profundidad hasta ahora, en este periodo nos dedicamos explícitamente a documentarnos acerca del motor por medio de tutoriales (tanto libros como tutoriales online en vídeo). Por otro lado, la implementación de *RT-DESK* implicaba el uso de funcionalidades que tan solo la versión PRO de *Unity* contiene (importación de *DLL*'s), así que tuvimos que investigar la manera de hacerlo sin esta funcionalidad por medio del *FAQ* de *Unity* e información de la red (el foro oficial de *Unity* y otros).

- **Fase de análisis**: En la fase de análisis se puede englobar tanto la especificación de requisitos, como el diseño, como el propio análisis de *Beast's Retreat*. En esta etapa nos dedicamos a desarrollar el documento *GDD* disponible en el Anexo A de este documento, que contiene toda la información acerca del juego a desarrollar.

- **Fase de instalación de software**: Tras tener claro lo que teníamos que desarrollar pasamos a la etapa de instalación. La instalación de *Unity* en sí no es muy difícil, pero posteriormente requería la importación de numerosos paquetes y extensiones (como *NGUI* por ejemplo) necesarias para la implementación. También hemos utilizado software como *Adobe Photoshop* para el diseño de los elementos de interfaz.

- **Fase de implementación**: Este período consiste únicamente en el desarrollo del código fuente y creación del juego, desarrollo de los diseños, imágenes, modelos, etc.

- **Fase de pruebas**: Tras implementar todo el juego, realizamos una serie de pruebas para comprobar que la navegabilidad (explicada en el apartado de Análisis), visualización y funcionalidad implementada en el juego era correcta en función de lo planeado.

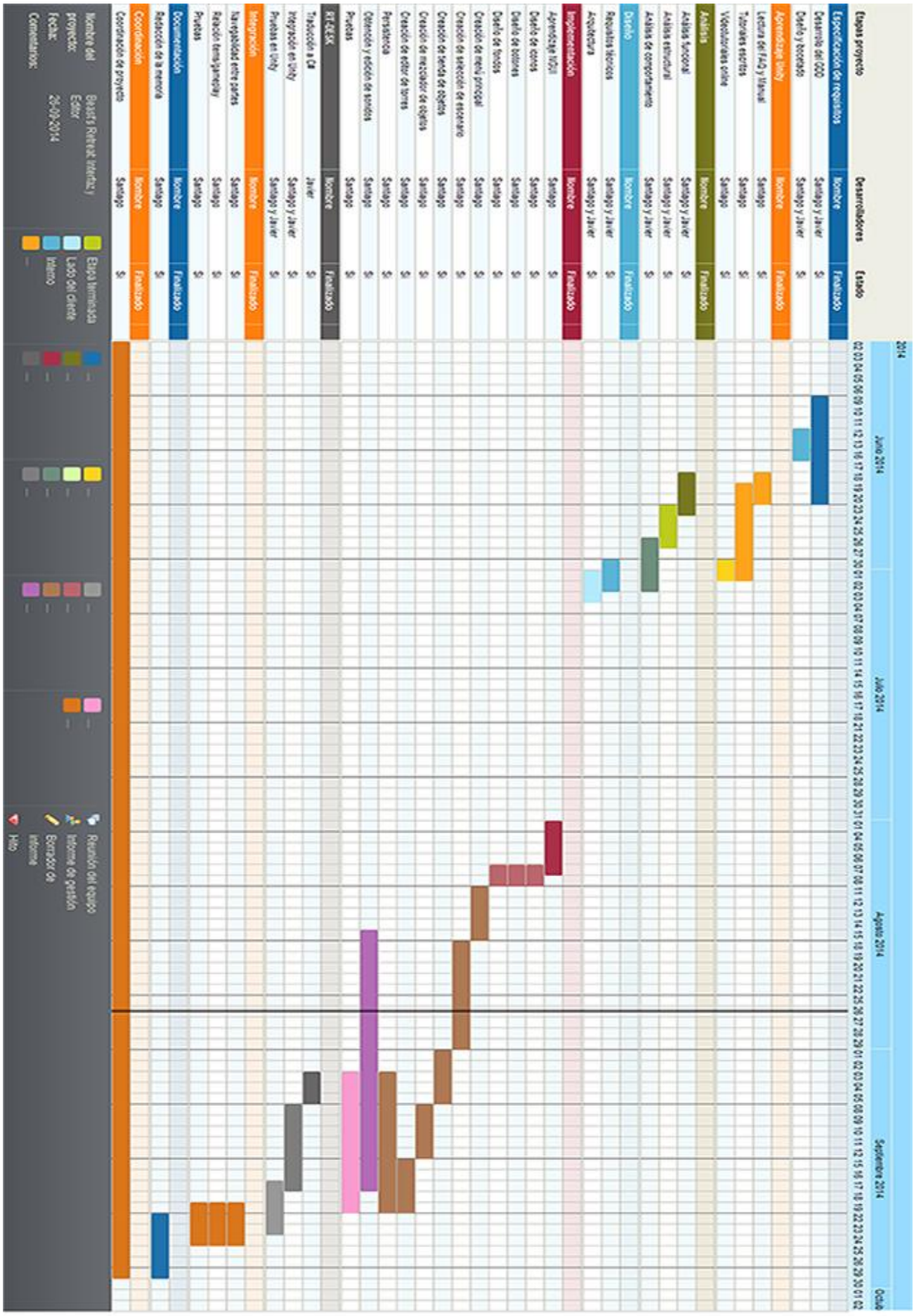
- **Fase de integración**: Esta fase implica unificar ambos proyectos, tanto la interfaz y el editor como el *gameplay* de *Beast's Retreat*. Se basa en la combinación de los recursos de ambos proyectos y la coordinación por medio de un repositorio para implementar la experiencia de juego completa.

- **Fase de pruebas de integración**: Consiste en comprobar que ambos proyectos se han unificado correctamente y el flujo de ejecución es correcto.

- **Fase de documentación y entrega**: Esta fase corresponde, como parte final del proyecto, a la redacción de la memoria y documentación del juego *Beast's Retreat* para su entrega como *PFC*.

4.2. DIAGRAMA DE GANTT

Planificar un proyecto proporciona una duración teórica del desarrollo, sin embargo la duración práctica puede variar con respecto a ésta. En este apartado presentaremos el diagrama de Gantt que representa la verdadera duración del proyecto y sus fases, aunque aclarando varios factores del diagrama:


Etapa proyecto
Desarrolladores
Estado
2014
Junio 2014
Julio 2014
Agosto 2014
Septiembre 2014
Especificación de requisitos
Nombre
Finalizado
Desarrollo del GDD
Santiago y Javier
Diseño y bocetado
Aprendizaje Unity
Análisis
Lectura del FAQ y Manual
Tutoriales escritos
Videotutoriales online
Análisis funcional
Análisis estructural
Análisis de comportamiento
Diseño
Revisión técnica
Arquitectura
Implementación
Aprendizaje NGUI
Diseño de clases
Diseño de botones
Diseño de fondos
Creación del menú principal
Creación de selección de escenario
Creación de lectura de objetos
Creación de recolector de objetos
Creación de editor de torres
Persistencia
Obtención y edición de tensión
Pruebas
RT-DESK
Traducción a C#
Integración en Unity
Pruebas en Unity
Integración
Interoperabilidad entre partes
Relación demo-gameplay
Pruebas
Documentación
Redacción de la memoria
Coordinación
Coordinación del proyecto
Nombre del proyecto:
Beast's Retreat: Interfaz y Editor
Fecha:
28-08-2014
Comentarios:
Etapa terminada
Hito
Reunión del equipo
Informe de gestión
Borrador de informe

Empezaremos desde el principio:

- En primer lugar, en la fase de especificación de requisitos, es verdad que se mejoró en gran medida y finalizó el documento *GDD* (Anexo A) entre el 2 y el 21 de junio, pero la primera versión, aunque no completa, se fue realizando a lo largo de los meses de marzo y de abril debido al curso académico.
- La fase de aprendizaje fue intensa en las fechas puestas en el diagrama de Gantt, aunque podría extenderse hasta prácticamente finales de julio o principios de agosto.
- Las fases de análisis y diseño podrían perfectamente integrarse dentro del desarrollo del *GDD*, ya que incluye tanto análisis funcional, como de comportamiento, como requisitos técnicos y/o arquitectura.
- Podemos observar que entre el diseño y la implementación hay un período de inactividad. Este período se debe a los exámenes de recuperación y a la finalización de las prácticas en empresa.
- Debido al comienzo tardío de la implementación, las últimas etapas se solapan, mezclando desarrollo con integración y con documentación en cierta medida.
- La fase de coordinación se extiende durante todo el proyecto debido a que en todo momento hubo comunicación acerca de los cambios, avances y otros.

Así pues, podemos observar a partir de este diagrama que el tiempo de desarrollo del proyecto se aproxima a las **12 semanas**, donde 7 son de implementación, dejando 4 para la planificación y 1 para la documentación del proyecto.

4.3. PLANIFICACIÓN DE RECURSOS

Para el desarrollo de este proyecto, es decir, la interfaz y el editor, se ha utilizado el mismo software que para la parte del *gameplay*, aunque distinto hardware. Este apartado tiene como fin mostrar ambos tipos de recursos utilizados con alto nivel de detalle:

- **Hardware**: Respecto al hardware, las prestaciones de la máquina utilizada para el desarrollo son:
    - Intel Core 2 Duo P8700, con 2,53 GHz de velocidad, es el procesador de la máquina que se ha utilizado en la totalidad del desarrollo.
    - Memoria RAM de 4 GB, así como un disco duro con 2 unidades de 232 GB cada uno.

- **Software**: A continuación, el software utilizado para desarrollar *Beast's Retreat*:
    - Sistema operativo *Windows 7 Home Premium* de 64 bits en la máquina utilizada.
    - *Adobe Photoshop CC* de 64 bits para el diseño de las imágenes, iconos, botones, y cualquier elemento artístico 2D.
    - *Autodesk 3D Studio Max 2014* para el desarrollo de modelos 3D y animaciones de modelos.
    - *Microsoft Word 2013* para el desarrollo del documento *GDD* y la memoria del *PFC*.
    - *Star UML* para el desarrollo de los diagramas.
    - *Tom's Planner* para generar el diagrama de Gantt en línea.
    - *Audacity* para edición de algunos sonidos y música de fondo.

4.4. ESTIMACIÓN DE COSTES

Por último, en las medidas utilizadas para establecer la planificación del proyecto, se encuentra la estimación de costes, en las que incluiremos también la de riesgos.

4.4.1. PRESUPUESTOS

La estimación de costes de *Beast's Retreat* se ha realizado sin hacer uso de ningún modelo preestablecido para su cálculo, como podría ser *COCOMO* o *Delfi*. De esta manera nos centraremos en los costes teniendo en cuenta sobre todo el tiempo invertido en el desarrollo, la extensión y los recursos utilizados.

4.5.1.1. EXTENSIÓN

Cuando hablamos de extensión del proyecto hablamos de líneas de código. Una de las técnicas de estimación de costes es determinar el tamaño del proyecto en función del total de líneas de código, obteniendo a partir de ellas un determinado grado de extensión, y por tanto el número de desarrolladores correspondiente al proyecto.

Un parámetro como este puede determinar además el tipo de diseño e implementación que se utilizará en el proyecto. Hay que tener en cuenta que las líneas en blanco o comentarios no se incluyen en este parámetro.

Observemos la tabla siguiente:

Dimensión del proyecto	Nivel de complejidad
0-1.000 líneas de código fuente	Trivial
1.000-10.000 líneas de código fuente	Simple
10.000-100.000 líneas de código fuente	Difícil
100.000-1.000.000 líneas de código fuente	Complejo
>1.000.000 líneas de código fuente	Casi imposible

Ilustración 30: Clasificación del proyecto por líneas de código

Basándonos en la clasificación de E.N.Yourdon, que determina la complejidad de un proyecto en función de las líneas de código, la interfaz y el editor de *Beast's Retreat* como proyecto se clasifica en el nivel simple, ya que se encuentra en las 3800 líneas de código aproximadamente. Sin embargo hay que tener en cuenta dos factores:

- *Beast's Retreat* en su plenitud también contiene el *gameplay*, que podría elevar las líneas de código a más de 7000.
- No se trata de una aplicación estándar, sino de un videojuego. El trabajo visual y de diseño no se aprecia en esta escala, por tanto no es la dificultad real del proyecto.

Teniendo en cuenta estos factores consideramos que *Beast's Retreat* es un proyecto difícil, que requeriría idealmente entre 3 y 4 desarrolladores y 1 o 2 diseñadores. Se han utilizado estas escalas debido a que no se ha encontrado información acerca de la clasificación de un proyecto cuya temática sea un videojuego.

4.5.1.2. COSTES DE PERSONAL

Disponemos pues de los datos necesarios para calcular el coste del desarrollo del personal que ha trabajado en el proyecto. Como se ha dicho anteriormente, la duración total del proyecto ha sido de 12 semanas.

A pesar de lo dicho en el apartado anterior acerca de los desarrolladores y diseñadores requeridos, en este proyecto tan solo hemos trabajado 2 programadores. Por lo tanto, sabiendo que ambos desarrolladores tienen un título universitario, se estima un sueldo de unos 1200 euros/mes, basándonos en la situación actual del mercado y en el sueldo que debería tener un ingeniero técnico informático.

De esta manera, sabiendo que el proyecto ha durado unos 3 meses, el coste de personal sería aproximadamente:

- 3 x 1200 = 3600 € por programador.
- 3600 x 2 = **7200 €** en total.

4.5.1.3. COSTES DE RECURSOS

A nivel de hardware no debería haber gastos ya que la máquina ya estaba en posesión del desarrollador. Sin embargo, a nivel de software podemos establecer los costes en función de las licencias:

- Sistema operativo *Windows 7 Home Premium*: 73 euros.
- Motor de juegos *Unity:* 0 euros.
- *Adobe Photoshop CC*: Suscripción mensual de 12,29€/mes.
- *Star UML*: 0 euros.
- *Microsoft Word 2013*: 99€ todo el *Microsoft Office*.
- *Autodesk 3D Studio Max 2014*: 0 euros (versión de prueba).
- *Tom's Planner*: 0 euros.
- *Audacity:* 0 euros.

De esta manera, al haber durado 3 meses, los costes de recursos ascienden a:

- 73 + (12,29 x 3) + 99 = **208,87 €**

4.5.1.4. OTROS COSTES

Podría haber costes extra si durante el desarrollo nos hubiésemos visto obligados a comprar paquetes de *Unity* en el *Asset Store* o licencias. Sin embargo, *Unity* dispone de versión gratuita para el desarrollo (suficiente para realizar este proyecto) y paquetes subidos a la tienda por desarrolladores que pueden ser descargados gratuitamente. Por tanto, se ha decidido que no se estudiarán más costes que los ya nombrados.

4.5.1.5. RESULTADOS

Si sumamos todos los costes calculados a lo largo de este apartado, es decir, los de personal y los de recursos, obtenemos:

- 7200 + 208,87 = 7408,87 € aproximadamente en 3 meses de desarrollo.

4.4.2. RIESGOS

Por último, y dentro del apartado de estimación de costes, haremos un breve análisis de los riesgos del proyecto. Los riesgos pueden clasificarse de manera general por categorías, o por el nivel de predicción, además de basarse en 2 atributos principales; la incertidumbre y la pérdida. Basándose en estos 2 últimos, un proyecto podría abandonar su desarrollo en una temprana etapa de planificación, por ello es tan importante una buena planificación, análisis y diseño, para reducir los costes al mínimo.

Principalmente podemos hablar de 3 categorías distintas de riesgos, siendo cada una influyente en una característica diferente:

- -**Riesgos de proyecto**: Son aquellos factores que amenazan el cumplimiento puntual del proyecto. Los riesgos del proyecto identifican los posibles problemas de presupuesto, planificación temporal, personal (asignación y organización), recursos, cliente y requisitos y su impacto en un proyecto de software. En este caso, el principal riesgo ha sido el comienzo tardío del

desarrollo debido al curso académico y a unas prácticas en empresa realizadas en la empresa *Everis* este mismo año.

- **Riesgos técnicos**: Son los que ponen en peligro la calidad y la planificación temporal del software. Si un riesgo técnico se hace realidad, la implementación se complica hasta volverse imposible. El principal riesgo técnico de este proyecto era las condiciones de la máquina de desarrollo, ya que al ejecutar procesos con alto coste computacional la refrigeración interior no es suficiente y tiende a recalentarse con facilidad, haciendo posible la pérdida de datos y el retraso en la implementación.

- **Riesgos de negocio**: Son los que amenazan a la viabilidad del software que se pretende desarrollar. Se pueden distinguir 5 tipos de riesgos de negocio:

  - Riesgo de mercado: La aplicación tiene un buen funcionamiento pero sin demanda. En este caso la comercialización del juego es todavía una duda y el principal objetivo es poner un pie en la industria del videojuego, por lo tanto el riesgo no se considera alto.
  - Riesgo estratégico: El proyecto no entra en el perfil de la estrategia de la empresa. Los clientes han ido evaluando y comprobando cada uno de los avances, asegurándose de que está hecha según sus requisitos. Este tipo de riesgo se podrá evaluar cuando se presente a alguna empresa y determinen el nivel del desarrollo a la hora de la contratación.
  - Riesgo de marketing: El producto final no se sabe vender. Se trata de un juego que, aunque está dentro de un género muy visto, presenta innovaciones visuales y de rendimiento, por tanto no se considera un gran riesgo.
  - Riesgo de presupuesto: En cuanto a presupuestos, las continuas mejoras y cambios propuestas para la aplicación a lo largo del desarrollo podrían suponer un aumento del presupuesto. Es un riesgo medio ya que en una hipotética comercialización, el mantenimiento del juego podría suponer cambios en los diseños, modelos y otros elementos, suponiendo un mayor coste.
  - Riesgo de dirección: En este caso, el director es un profesor de la Universidad Politécnica de Valencia. Es posible un repentino cambio en el puesto, aunque es poco probable, al menos hasta que acabe el curso académico.

5. ANÁLISIS

La sección de análisis del sistema, posterior al apartado de planificación y especificación de requisitos, tiene como objetivo mostrar los diferentes diagramas realizados, así como la navegabilidad entre escenas del juego *Beast's Retreat* y la manera en la que los programadores se han coordinado para realizar el desarrollo.

5.1. ANÁLISIS FUNCIONAL

El análisis funcional tiene por objetivo mostrar la navegabilidad entre escenas del proyecto de *Beast's Retreat*:

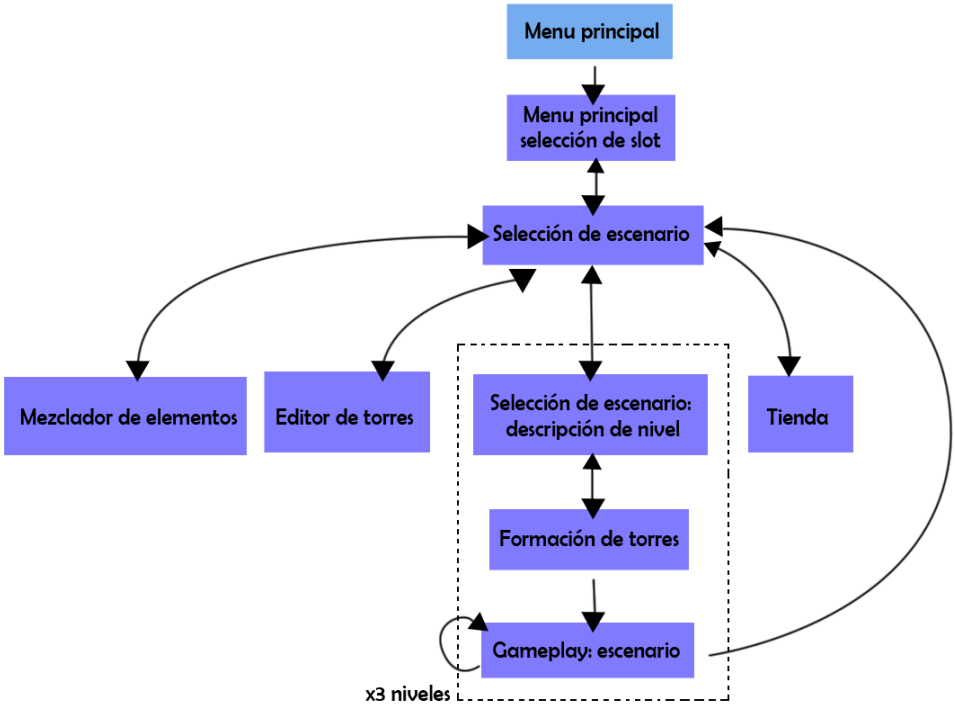


Ilustración 31: Diagrama de flujo

Podemos ver en la ilustración anterior que el punto de entrada al juego es el menú principal. Desde esta escena podemos ver la intro del juego y la selección de partida por slot, los créditos o salir del mismo.

Ilustración 32: Menú principal

Los slots nos permitirán almacenar 3 partidas en la misma máquina gracias a la persistencia de datos implementada en *background* dentro de *Unity*.

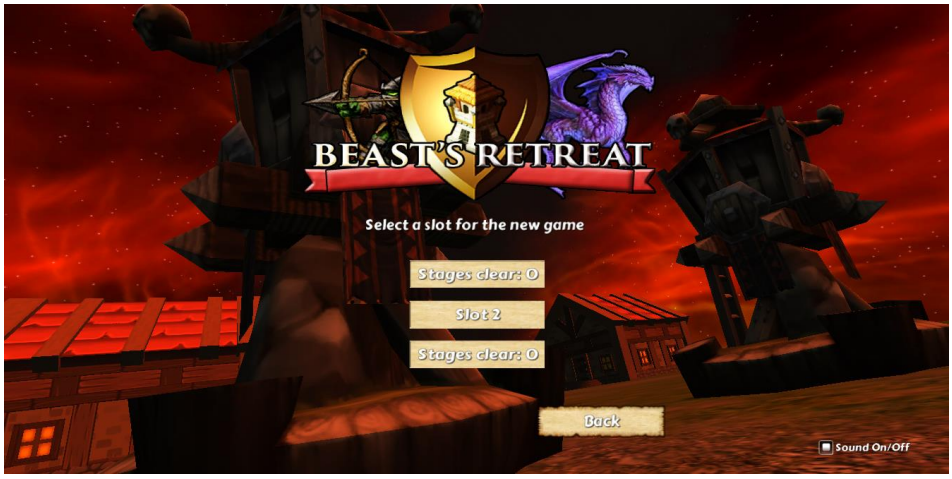


Ilustración 33: Selección de slot

Una vez seleccionado el slot entramos en la selección de escenario. Este es el punto central del juego, a partir de aquí se puede acceder a todas las partes del juego, incluido los modos de creación.

Ilustración 34: Selección de escenario

Es hora de gastar nuestros primeros créditos en la tienda, a la que podemos acceder mediante el correspondiente botón y comprar objetos que nos servirán para montar nuestras torres o incluso fabricar otros objetos. Hay tanto objetos individuales como ofertas especiales.

Ilustración 35: Tienda

Tras comprar los objetos podemos combinarlos para obtener objetos más complejos que nos servirán para potenciar nuestras torres. Esto se hace desde el mezclador de objetos, otro modo de creación accesible desde la selección de escenarios.

Ilustración 36: Mezclador

Por último tenemos el editor de torres. Es el último de los modos de creación implementados y permite crear torres a partir de los objetos obtenidos en los modos anteriores. Se accede también desde la selección de escenario e incluye un asistente que guía al usuario para la creación, lo que le ayudará a desbloquear diferentes torres para el *gameplay*.

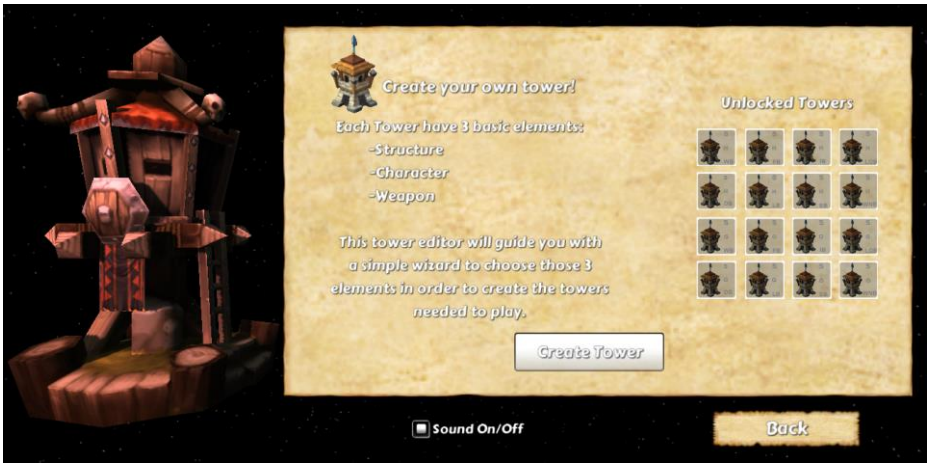


Ilustración 37: Editor de torres

Pasamos a la selección de los escenarios, que se irán desbloqueando a modo de historia conforme el usuario vaya superándolos. Al seleccionar un escenario, una descripción junto con la ilustración correspondiente se abre paso desde arriba para que elijamos el modo de juego.

Ilustración 38: Descripción de nivel

Como podemos ver, hay dos modos de juego: Campaña y Survival. Tras la selección del modo el botón *Play* es habilitado, permitiendo navegar a la pantalla de Formación de Torres.

Ilustración 39: Formación de torres

Tras seleccionar las torres con las cuales queremos defender nuestra base en el escenario a jugar, podemos empezar el juego mediante el botón *Play*, que nos llevará a dicho escenario con la formación de torres que hayamos elegido.

Ilustración 40: Gameplay

No superar el escenario redirigiría el juego al menú principal. Sin embargo, superarlo implica volver a la selección de escenarios.

5.2. ANÁLISIS ESTRUCTURAL

La parte de interfaz y editor de *Beast's Retreat* se ha basado exclusivamente en el diseño visual y aplicarle funcionalidad a cada uno de los botones y elementos disponibles para la interacción. Por tanto, al no tener un profundo sistema de objetos y relaciones entre ellos, el análisis funcional del apartado anterior es suficiente.

A pesar de ello sí se puede hacer un poco de hincapié en la funcionalidad de persistencia para objetos. Para ésta hemos creado dos clases:

- La clase **Item**, que permite almacenar el nombre y el tipo del objeto. Necesaria para la creación y almacenamiento de nuevos objetos en la base de datos, que sirven para mezclar con otros o para crear las torres para el *gameplay*.

- La clase **ItemDatabaseManager**, que permite la inserción, actualización y borrado de objetos en la base de datos. Es la clase utilizada en todos los modos de creación para el almacenamiento y recepción de objetos.

5.3. ANÁLISIS DE COMPORTAMIENTO

Este apartado del proyecto de interfaz y editor de *Beast's Retreat* está dedicado a su pilar maestro, la persistencia de datos. Podríamos decir que hay dos tipos de persistencia:

- Por un lado la persistencia de datos de usuario. Esto es, los créditos de los que dispone, y una serie de datos que permiten conocer el slot actual utilizado y la cantidad de escenarios superados.

- Por otro lado la persistencia de objetos, de la que ya hemos hablado en el apartado anterior.

Sin embargo, hemos hablado de ella estructuralmente. Mediante un conjunto de diagramas de secuencia, nos disponemos a explicar el comportamiento de las diferentes escenas a la hora de cargar estos datos de ambos tipos. También podría considerarse análisis de comportamiento a la transición de las interfaces por medio de los botones y los scripts que utilizan, sin embargo haremos referencia a este tema en el apartado de arquitectura y diseño.

En primer lugar tenemos el menú principal. Bajo el nombre de escena **Title**, es la primera escena cargada en el juego y por tanto la que debe comprobar si hay partidas guardadas, para que el usuario pueda empezar una nueva o retomar una ya creada. Podemos observar que hay una entidad llamada **PlayerPrefs**. Esta clase es la encargada en todo el juego de almacenar los datos del usuario en *background*, incluso cuando el juego se haya cerrado.

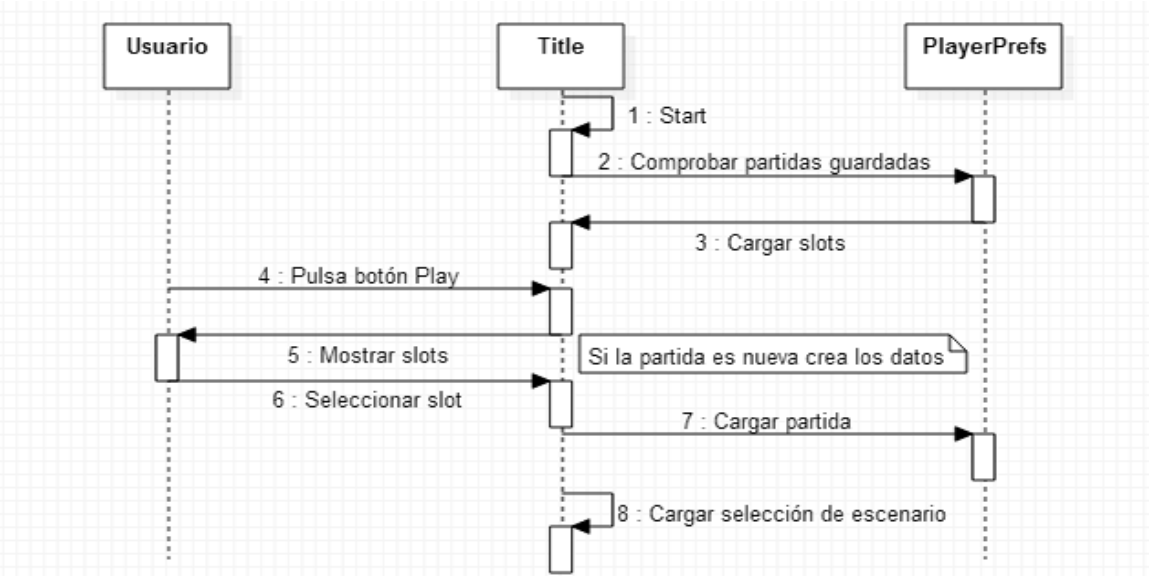


Ilustración 41: Secuencia de Title

Como podemos ver, tras crear o cargar los datos transita a la pantalla de selección de escenario o **Game Modes**. Esta pantalla tampoco utiliza la persistencia de objetos, solamente utiliza **PlayerPrefs** para conocer el número de escenarios superados y desbloquearlos en función de éste parámetro.

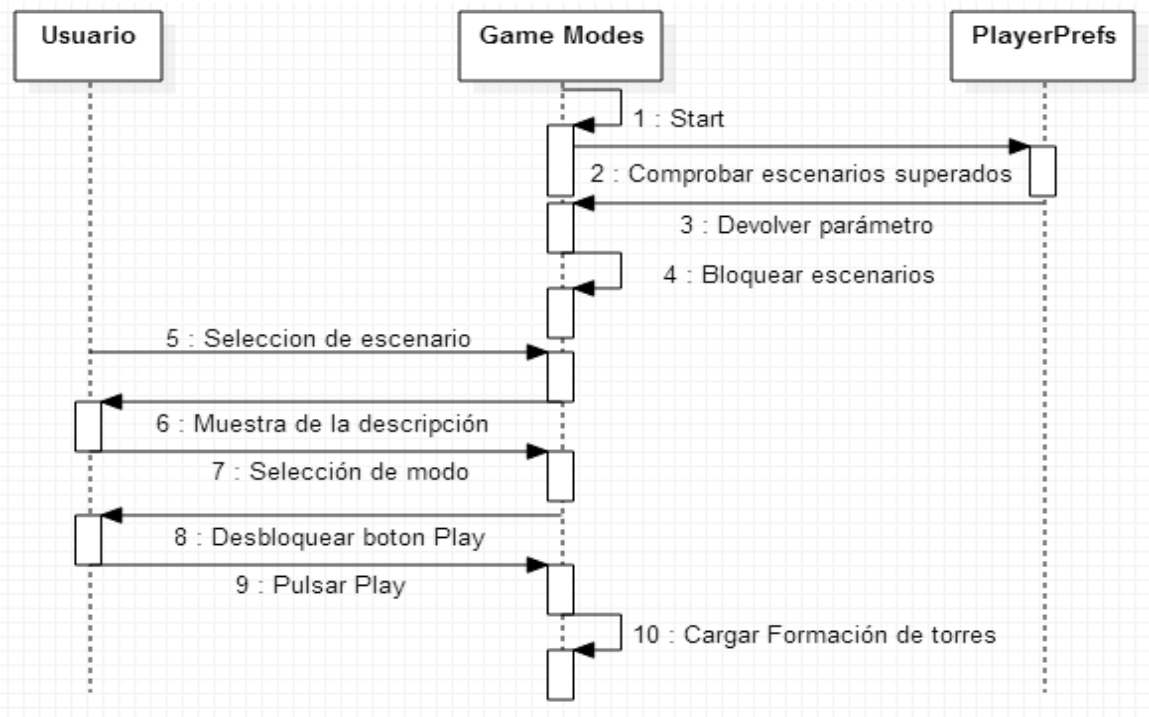


Ilustración 42: Secuencia de Game Modes

Al pasar a la formación de torres ya no hay vuelta atrás, se debe jugar el escenario seleccionado. A partir de aquí entra en acción la persistencia de objetos, ya que en esta pantalla el usuario debe elegir las torres con las que quiere defender su base. Al comienzo de la escena **Defense Strategy** se cargan los objetos de la base de datos mediante la entidad **ItemDatabaseManager** y se comprueba que torres tiene desbloqueadas el usuario. Más tarde, tras haber superado el escenario de juego, en función de la eficiencia y la puntuación conseguida el usuario obtendría más o menos créditos y se actualizaría el parámetro de escenarios superados, desbloqueándolos en el modo de selección.

Hay que destacar que en caso de no superar el escenario el juego se redirigiría al menú principal, por lo tanto los pasos 10, 11 y 12 no se ejecutarían.

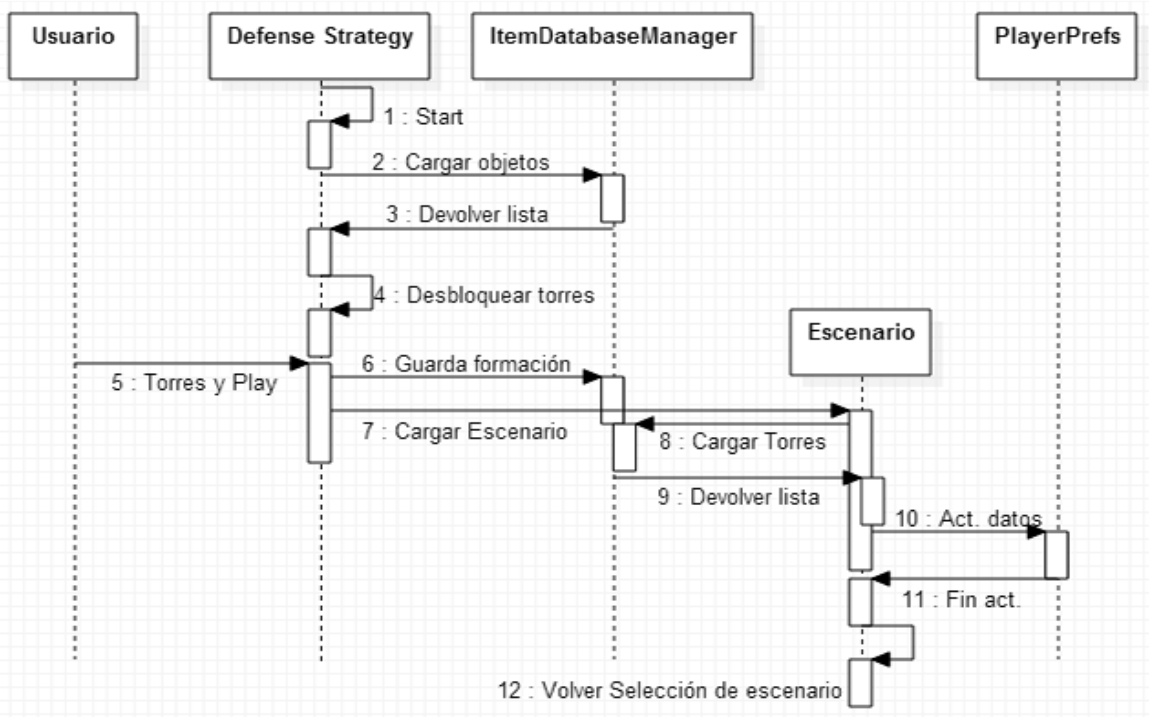


Ilustración 43: Secuencia de Defense Strategy

Anteriormente hemos comentado que la el modo de selección de escenarios era la puerta a todos los modos, tanto de creación como de juego. Por tanto, desde él se accede también a la tienda, el mezclador de objetos y el editor de torres.

Todos siguen una plantilla de comportamiento similar: se cargan los objetos de la base de datos (el fichero XML correspondiente al slot cargado) que se reciben en forma de

lista de objetos. Cada escena los gestiona como debe, y posteriormente se ejecuta su funcionalidad. Para finalizar una transacción, los objetos comprados u obtenidos se vuelven a almacenar en una lista y se insertan. Tras la inserción, al haber cambiado la base de datos, se vuelve a actualizar la interfaz.

La tienda de objetos es la única excepción con respecto a este procedimiento ya que no carga datos, sino que solo los guarda. Además, es el único modo de creación que interactúa con **PlayerPrefs** ya que necesita obtener y actualizar los créditos disponibles del usuario para realizar las compras. Bajo el nombre de **Item Shop**, actúa de esta manera:

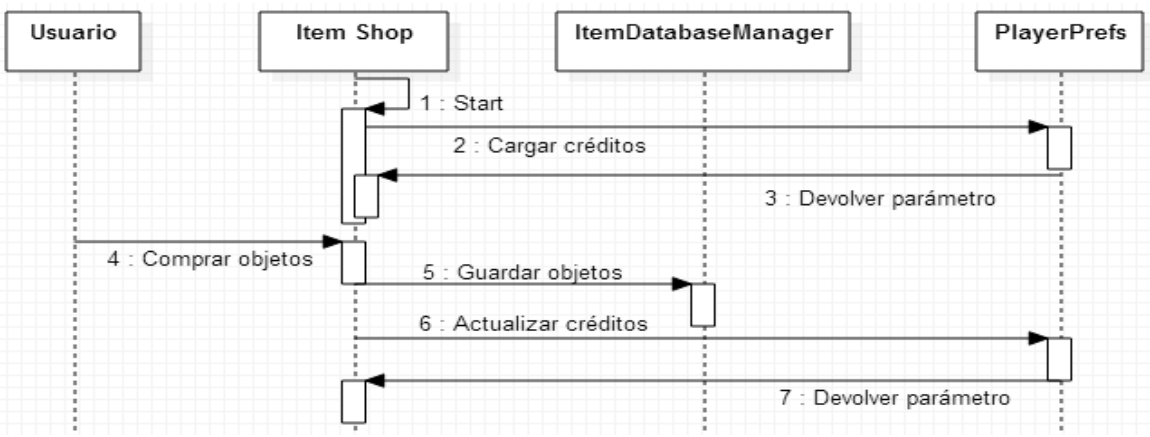


Ilustración 44: Secuencia de Item Shop

Los dos diagramas restantes son prácticamente idénticos. Uno para la escena **Item Creator** y otro para la escena **Tower Editor**, que como ya se ha dicho antes, tienen una persistencia prácticamente idéntica. La única diferencia es la forma de gestionar los objetos que se obtienen de la base de datos. Ambas cargan los objetos y definen el número de cada uno de ellos para poder utilizarlos para la mezcla o edición, sin embargo el editor de torres contiene una funcionalidad extra que le permite desbloquear o no las torres en función de si el usuario las ha conseguido. Como estamos mostrando el comportamiento de persistencia, haremos un único diagrama para ambas escenas:

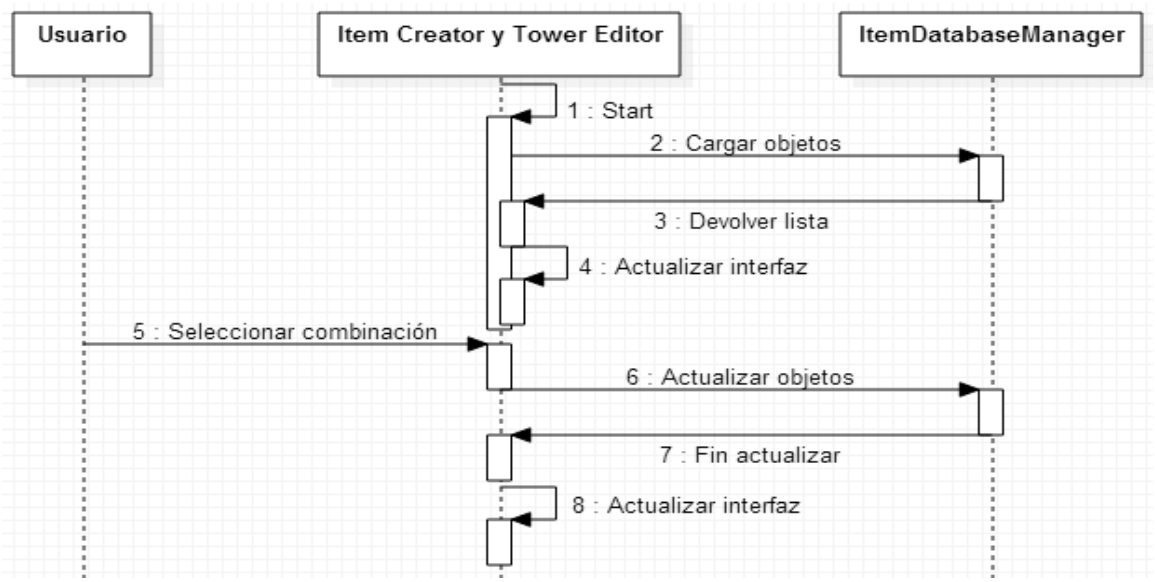


Ilustración 45: Secuencia de Item Creator y Tower Editor

5.4. COORDINACIÓN

A lo largo de todo el desarrollo nos hemos mantenido en comunicación para avisar de cambios, avances, innovaciones, entre otros. En concreto, como bien se ha mostrado en el diagrama de Gantt, hubo un gran número de reuniones en las etapas de desarrollo del documento *GDD* y de integración de *RT-DESK* en *Unity*.

Inicialmente, nada más comenzar el desarrollo utilizábamos como medio de coordinación o repositorio una carpeta compartida en *Dropbox.* Tras darnos cuenta de la importancia de hacer una buena división del desarrollo del juego para poder llegar a las fechas de entrega, decidimos separarlo en dos partes: **Videojuego Beast's Retreat en Unity con C# integrando RT-DESK: Interfaz y Editor**, desarrollado por mí, y **Videojuego Beast's Retreat en Unity con C# integrando RT-DESK: Gameplay**, desarrollado por mi compañero Javier Baixauli Herraez.

Obviamente, a pesar de ser partes independientes, se trata del mismo juego y por tanto nos hemos aconsejado y/o ayudado mutuamente. Debido además a una falta de espacio notable en *Dropbox*, decidimos crear un repositorio en *GitHub* para el proyecto, donde hemos ido subiendo las últimas versiones cada vez que obteníamos avances importantes.

Todas las fases han requerido de gran coordinación a lo largo del desarrollo, pero sin duda la mayor de ellas ha sido la de integración de *RT-DESK*. Después de que mi compañero tradujese a C# la API, estuvimos varios días quedando tanto entre nosotros

como con nuestro tutor de proyecto, ya que nadie había intentado integrarla en *Unity* hasta ahora y era algo totalmente nuevo.

También hemos necesitado una gran comunicación para integrar ambas partes. El paso de la selección de escenarios a un escenario de juego requiere enviar datos y modificar algunos elementos como el generador de hordas, la disposición y tipo de las torres, entre otros. También entra en juego la persistencia de los datos del usuario, por lo tanto era necesario comunicarnos para saber que parámetros debíamos modificar y coordinar.

Podemos concluir pues que la coordinación ha sido alta a nivel de información y comunicación, ya que ambas partes estaban muy bien separadas y no hemos necesitado fragmentos de desarrollo del compañero. Situamos pues un coste temporal de unas 2 horas semanales de coordinación para la etapa de implementación, variando en las otras fases de desarrollo, ya que en las fases comentadas anteriormente (desarrollo del GDD y fases tempranas, y posteriormente en la integración de *RT-DESK*) la coordinación fue superior, pudiendo ascender hasta las 6 horas semanales aproximadamente.

6. DISEÑO

No es la primera vez en esta memoria que hemos hablado de la posible fusión de las etapas de especificación de requisitos, análisis y diseño dentro de la etapa de redacción del documento *GDD* (Anexo A). Dado que la etapa de análisis tiene un completo estudio del comportamiento y navegabilidad de *Beast's Retreat* y los requisitos técnicos están especificados en gran medida en el documento *GDD*, la fase de diseño podría considerarse la más corta en este documento. A continuación se hablará de la arquitectura utilizada en *Unity* y la relación entre los objetos utilizados y scripts.

6.1. ARQUITECTURA DE SOFTWARE

La interfaz y editor no incluyen el uso de las físicas de *Unity* ni de muchos *GameObject* especializados en el *gameplay*, ya que se trata tan solo de la distribución visual de los menús, los modos de creación y la persistencia entre ellos (descrita en el apartado anterior con detalle). Sin embargo sí que puede hablarse del sistema de mensajes a través de los botones, los scripts adjuntos a los objetos, y la jerarquía de éstos.

Lo primero es hablar de la jerarquía de objetos en *Unity*, ya que es un poco particular para interfaces desarrolladas con *NGUI*. En el apartado 3.4 del documento correspondiente al estado del arte, se ha hecho una pequeña introducción al paquete de desarrollo de interfaces en el que comentamos la creación de los objetos *UI Root*, esenciales para todas ellas. Cualquier tipo de objeto 2D creado con *NGUI* debe figurar obligatoriamente como elemento hijo en la jerarquía, ya que sino la *Camera* no lo renderizará en la escena. Por otro lado, para facilitar la movilidad de elementos existe el objeto *Panel*, que permite situar un sistema de referencia para todos los objetos de interfaz que se sitúen dentro de éste. Ésta ha sido la manera para implementar las diferentes pantallas que transitarán en cada escena.

Ilustración 46: Jerarquía de objetos

Para hacer la transición entre pantallas entra en juego el sistema de mensajes y los movimientos. Ambas funciones tienen su clase específica en *NGUI*:

- En primer lugar, los **Button Message**. Esta clase permite fijar el nombre del método que va a ejecutar al pulsar el botón, pasar el ratón por encima, dejarlo encima sin pulsar, entre otros.

- En segundo lugar, los **Tweens**. El más utilizado ha sido el **Tween Position**, que aporta a un objeto la capacidad de moverse por la pantalla mediante código con una determinada duración. Este script se asignaba a los paneles de manera que, al pulsar el botón correspondiente, el panel actual se movía a otra posición y el panel destino se movía a la vista de la cámara.

Ambos son scripts que facilita *NGUI* y que han sido muy reutilizados en toda la parte de interfaz y editor. En concreto, el *Button Message* envía un mensaje según la acción a los llamados **Event Receiver**. Se trata del objeto destino del mensaje, que contiene el script donde se ha implementado el método a ejecutar. Aquí entra en juego los scripts adjuntos a los objetos. Habitualmente los *event receiver* son los objetos *UI Root*. Hay uno de estos objetos en cada escena, que contiene un script adjunto con los métodos y variables necesarios para la funcionalidad de cada interfaz.

Todo script que quiera adjuntarse a un objeto físico de la escena debe por necesidad descender de la clase *Monobehaviour*. Esta clase contiene los métodos necesarios para inicializar un objeto en la escena y actualizarlo *frame* a *frame*, así como algunos métodos de interfaz. Salvo los scripts destinados a la persistencia de objetos, el resto todos descienden de *Monobehaviour* ya que todos tienen que estar asociados al objeto *UI Root* correspondiente.

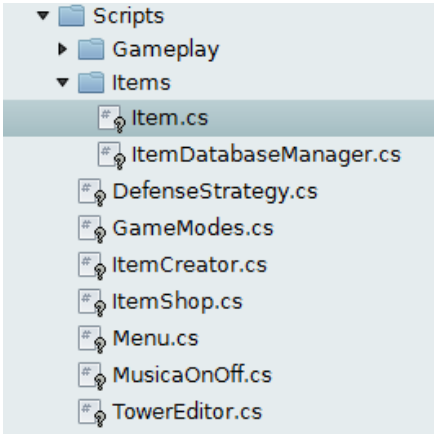


Ilustración 47: Scripts

7. IMPLEMENTACIÓN

Ya tenemos el juego completamente planificado y listo para empezar la implementación. Esta fase es la más técnica y larga del desarrollo. En este apartado documentaremos cada uno de los pasos que fueron necesarios para la primera versión del juego *Beast's Retreat* y las funcionalidades descritas en la descripción del mismo, abarcando desde la importación del material hasta la edición o ajuste de cualquiera de ellos y el código esencial que le da uso.

7.1. IMPORTACIÓN

Cuando *Unity* crea un nuevo proyecto permite la importación de numerosos paquetes, material y scripts que ofrece el motor de juego por defecto. Sin embargo cualquier material propio, ya sean imágenes, vídeo, sonido, scripts, elementos 3D…debe ser editado fuera del motor y posteriormente importado en alguno de los directorios que contiene el proyecto. Para ello simplemente hay que hacer click derecho dentro del editor y seleccionar *Import new asset*, como bien se muestra en la siguiente ilustración.

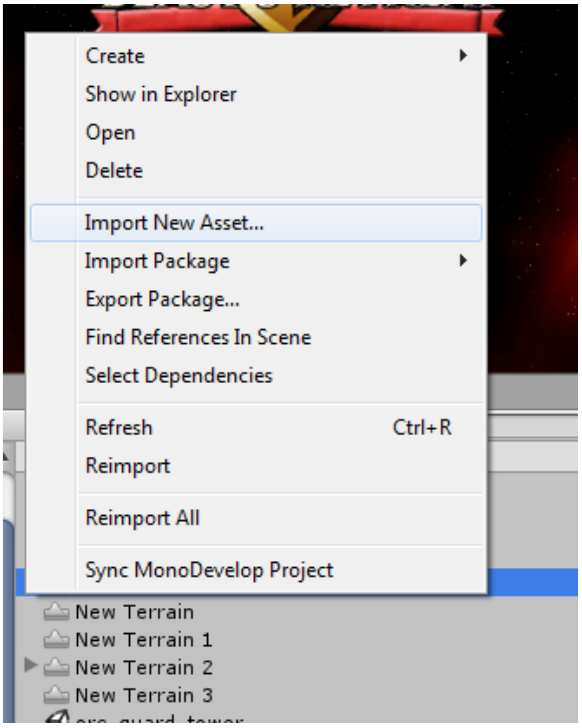


Ilustración 48: Importando recursos

Tras seleccionarlo aparece el típico examinador de *Windows* (o puede diferir según el sistema operativo) que permite seleccionar el fichero en el directorio donde se encuentre. Este proceso debe ser seguido para importar cualquier tipo de elemento, y ha sido muy utilizado en esta implementación ya que el pilar principal de mi parte de este proyecto eran las interfaces y modos de creación, que requerían una gran variedad de botones y *sprites* diferentes.

Otra manera de importar ficheros es desde el *Asset Store* de *Unity*. Consiste en una tienda donde los desarrolladores cuelgan sus propios desarrollos a un determinado precio. Ha sido muy útil para descargar paquetes de vegetación de terrenos, modelos 3D como torres, sonidos, entre otros. Una vez descargado el propio motor gráfico te ofrece importar los paquetes, por lo que no hace falta ni siquiera salir de la propia interfaz de *Unity*.

Ilustración 49: Asset Store

7.1.1. NGUI

En el apartado 3.3 del estado del arte de este documento se ha explicado con bastante profundidad las funcionalidades que aporta el paquete de desarrollo de interfaces *NGUI*. En este apartado nos dedicamos únicamente a explicar la importación del paquete.

NGUI actualmente se encuentra en la versión 3.7.3 Esta versión, como ya se ha explicado anteriormente, es software de pago con diferentes licencias en función del uso. Sin embargo, para desarrolladores no profesionales existe una versión gratuita, la 2.7.0. Se trata de una versión muy anterior y con algunas clases obsoletas, pero aun así contiene una gran cantidad de funciones que facilitan mucho el diseño de la interfaz de cualquier juego.

Frente a la versión de pago disponible en el *Asset Store*, esta versión gratuita se obtiene externamente al motor gráfico y se importa como un paquete de la misma forma que se importa un nuevo recurso, explicado en el apartado anterior. Una vez importado, aparece en el menú superior de *Unity* una pestaña con las diferentes opciones para comenzar la implementación.

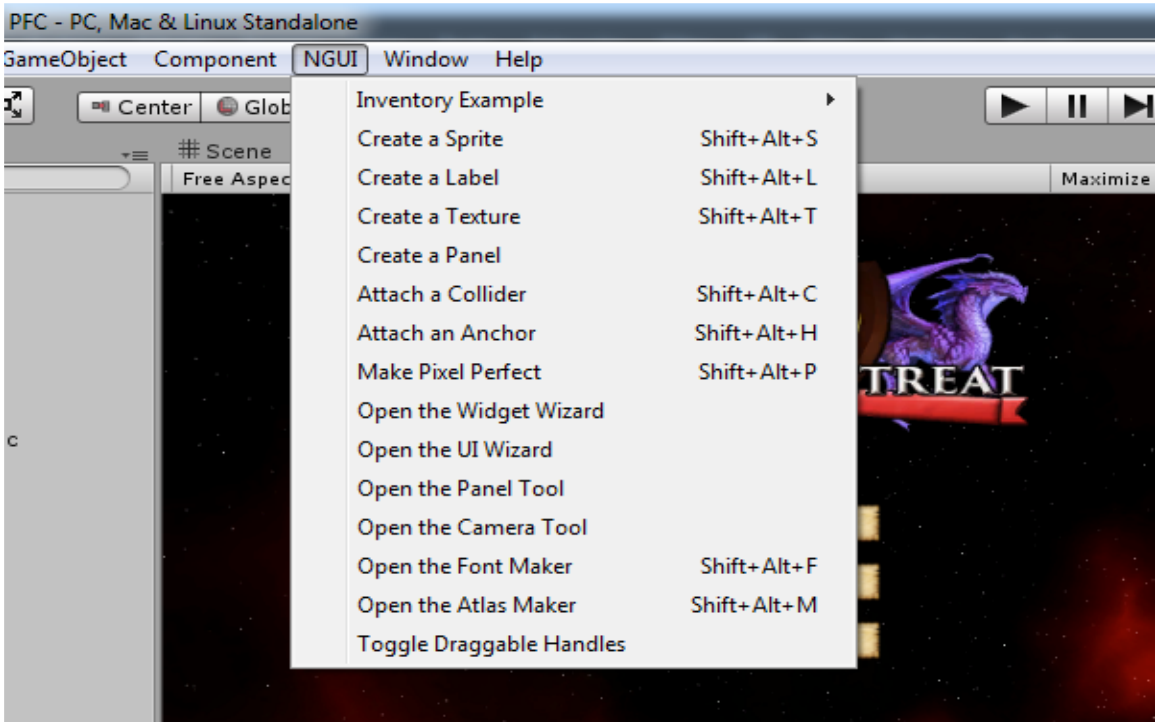


Ilustración 50: Menú NGUI

7.1.2. ELEMENTOS 3D

Aunque la gran mayoría de la interfaz está compuesta por elementos 2D, hay escenas pertenecientes a este proyecto donde entran en juego elementos 3D:

- La primera de ellas es el menú principal. Para el desarrollo de esta escena se han utilizado los siguientes elementos:

    - Una torre obtenida en el *Asset Store* bajo el nombre de *Orc Guard Tower*.
    - Una casa medieval con toque *cartoon* bajo el nombre de *Medieval House*, también del *Asset Store*.
    - Un terreno pintado con llanuras de césped y caminos de tierra, para el cual se ha descargado un paquete de vegetación del *Asset Store.*

  Hay que destacar también elementos como el cielo o la iluminación, que se tratarán más adelante.

- La segunda es el editor de torres. Se ha utilizado la misma torre que en el menú principal, asignándole una rotación constante mientras dure la ejecución de la escena.

7.1.3. ELEMENTOS 2D

Estos elementos son propios del *NGUI* y predominan en todos los modos de creación del juego. Son el elemento principal debido a que corresponden a botones con los que se implementa la transición entre interfaces, escenas, compra de objetos, y prácticamente cualquier tipo de funcionalidad.

En general, lo que más se ha diseñado y posteriormente importado al proyecto son botones y fondos. *NGUI* permite crear unos elementos llamados *atlas* que almacenan todos los *sprites* de todos los botones y los fondos y permiten colocarlos manualmente en el editor. Este tema se tratará más adelante.

7.2. ESPECIFICACIÓN DE LAS CÁMARAS

En el apartado anterior hemos hablado de los elementos 2D y 3D. Este apartado está orientado a explicar una pequeña distinción en el renderizado de ambos tipos de objetos.

A pesar de que ambos se colocan en la escena de manera similar y se visualizan en el editor, a la hora de renderizar hay una pequeña diferencia: la cámara. *NGUI* coloca cuando crea la interfaz una cámara que solo visualiza los objetos 2D, por tanto los objetos 3D tienen que ser tratados por separado con la *Main Camera*.

En los primeros momentos de la implementación esta distinción fue motivo de problemas y retrasos, ya que no se conocía. Sin embargo con el tiempo los diseños fueron mejorando y aunque no se volvieron a realizar muchas combinaciones de ambos tipos de objetos en una escena, aprendí mucho acerca de la posición de los objetos en ambas cámaras, distancias, capas, entre otros. Estos temas se tratarán en el siguiente apartado.

7.3. DISEÑO DE INTERFACES CON NGUI

En el apartado 3.3 del estado del arte se habló de la existencia de un gran número de ventanas, asistentes disponibles en el menú y el orden seguido para el correcto desarrollo de las interfaces con *NGUI*. En este apartado se hablará de los detalles técnicos a la hora de añadir los objetos a la escena, su colocación y los tipos utilizados.

1. En primer lugar, creamos la base de la interfaz. La herramienta *UI Tool* añade a la escena el objeto *UI Root*, que contiene el sistema de referencia de la *Camera* 2D y un *Panel* por defecto.
2. Una vez situado el *UI Root* ya podemos empezar a colocar elementos. Con el *Widget Tool* podemos seleccionar un *atlas* y una *font*, necesarios para incluir cualquier tipo de objeto 2D en la escena. Una vez pulsado el botón *Add* to el elemento se incluye dentro del panel seleccionado en el origen de coordenadas.

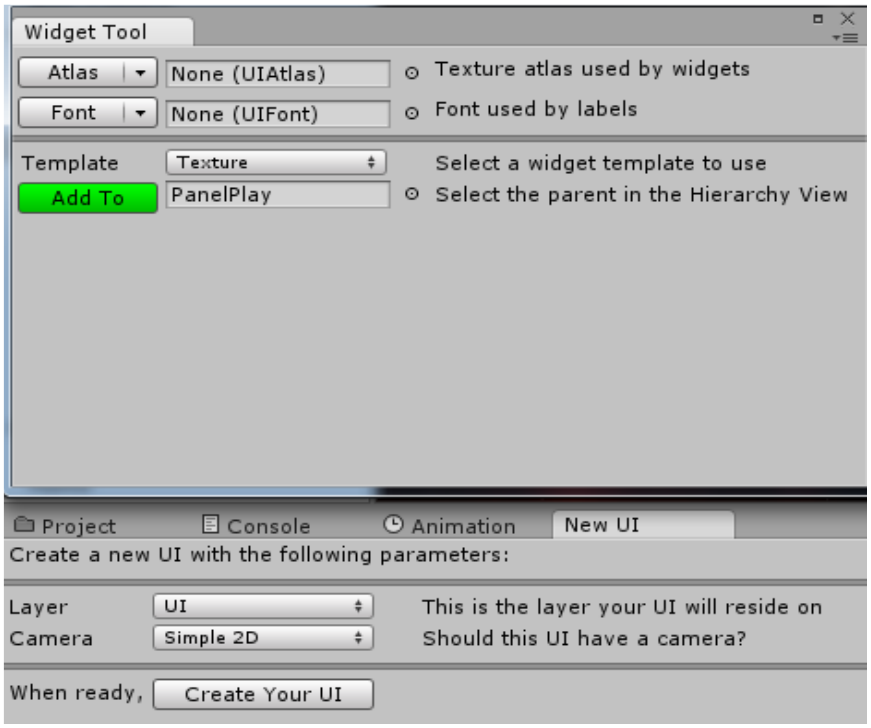


Ilustración 51: Widget Tool y UI Tool

Este segundo paso se repite un gran número de veces hasta tener en pantalla todos los elementos de interfaz deseados en cada escena. Haciendo *click* en el objeto podemos moverlo por el sistema de coordenadas y posicionarlo en función de la imagen que nos muestra la cámara en la pestaña *Game*, que contiene en todo momento una vista preliminar de cómo se vería el juego en ese momento.

Ilustración 52: Objeto en escena y vista preliminar

Los objetos que más se han utilizado para la implementación son descritos a continuación:

- **Botones**: Son el principal medio de transición entre interfaces y ejecución de métodos. El tipo de botón más utilizado ha sido el llamado **Image Button**. Cada botón requería del diseño de 3 *sprites* distintos (4 en caso de querer un estado bloqueado):
  - Normal: Estado normal del botón, sin ningún tipo de acción o efecto.
  - *Hover*: Estado en el que el puntero del ratón se sitúa encima.
  - *Pressed*: El botón se encuentra pulsado.

Por tanto para cada botón personalizado se requerían realizar 3 diseños diferentes en base a una plantilla creada. Todos ellos disponían de un fondo transparente que en una primera importación de *Unity* no servían, es decir: tras importarlas era necesario acceder a las propiedades de la imagen mediante el inspector y activar el *checkbox Alpha is transparent*, que las renderizaba correctamente. Las imágenes correspondientes a cada botón puede encontrarse en el último apartado del Anexo A de este documento.

Ilustración 53: Botón elemento Aire

- **Fondos**: Sitúan al usuario en la escena mediante imágenes relacionadas con el modo accedido. Para situar los fondos se utilizaban objetos **Texture**. Este objeto permitía utilizar un material o imagen importado al proyecto y situarlo en el editor, mostrándolo en la cámara. Hay que destacar que tras la importación había que cambiar el modo de redimensionado según el tamaño de la imagen, ya que si no era el adecuado la calidad podía llegar a ser mala. En general lo mejor era no redimensionarla, aunque para imágenes no excesivamente grandes se usó con frecuencia el modo *To Nearest*, que la redimensionaba al tamaño de estándar de *Unity* más cercano. También se usan en las pantallas de carga de cada escena, junto con iconos personalizados, importados y diseñados de la misma manera que los fondos.

Ilustración 54: Fondo carga Item Shop

- **Label**: Los *labels* o etiquetas son básicamente texto. Para crear una etiqueta es necesario utilizar una *font.* En este caso se ha utilizado siempre la *font Fantasy*, una fuente que viene por defecto con *NGUI* junto a uno de sus atlas de ejemplo.

- **Checkbox**: Utilizados únicamente para darle la opción al jugador de eliminar el sonido de fondo y los efectos en caso de que sea una molestia para él.

Ilustración 55: Checkbox sound on/off

Hay que tener cuidado con los objetos que contienen subelementos, por ejemplo, los *Image Button* están compuestos de un *background* (fondo o *sprite)* y un *label*. A la hora de redimensionar los objetos por escalado hay que hacerlo mediante el objeto padre y su componente *Transform*. En caso contrario, si por ejemplo modificamos sólo el *background*, aparecerá inicialmente redimensionado pero a la hora de la simulación volverá al tamaño del objeto padre.

Uno de los principales problemas al situar los objetos en la escena eran las capas. A la hora de colocar unos objetos por delante de otros, obviamente es importante su posición en el eje correspondiente (habitualmente el eje Z). Sin embargo, hay veces que a pesar de estar por delante de otro objeto no se visualiza éste, sino el que se encuentra más hacia atrás. Esto es debido a que cada vez que situamos un objeto se coloca en una capa posterior para poder ser visualizado en el editor, y al cambiarla en cualquier otro objeto trastoca las posiciones. Los *labels* son los que más problemas han dado habitualmente, hasta el punto de buscar nuevas soluciones referentes al movimiento de paneles en direcciones verticales u horizontales.

Ilustración 56: Capas y profundidad

Esta imagen puede ayudar a comprender mejor el concepto. Podemos observar que, a pesar de estar colocado en una posición delantera, el objeto que se visualiza es el objeto de detrás.

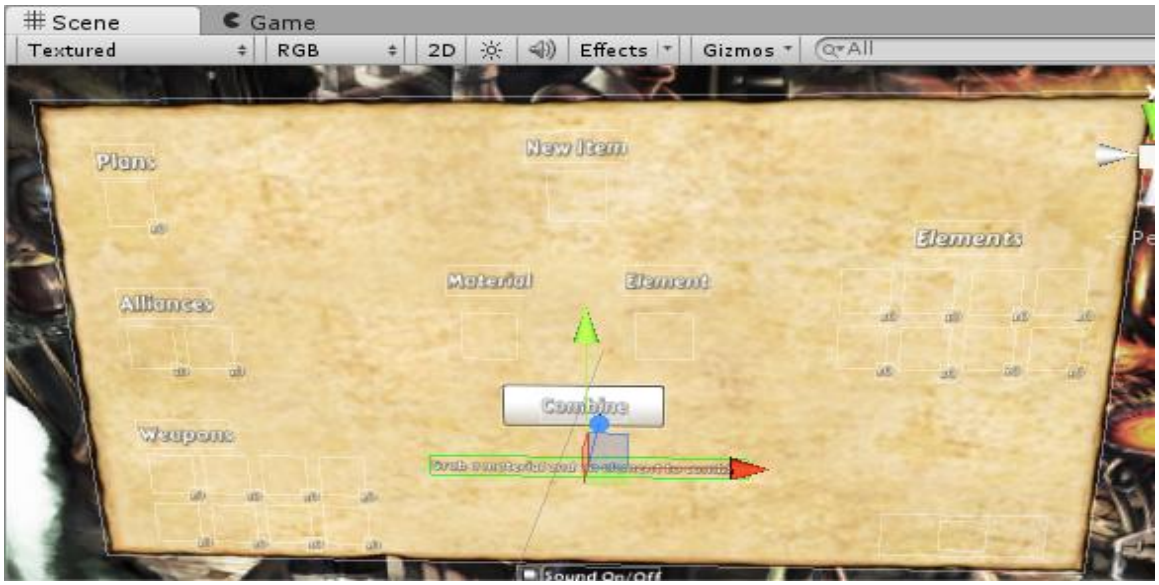


Ilustración 57: Objetos delanteros que no se ven

Otro de los temas de confusión ha sido el tema de las distancias de cada elemento a la cámara. La relación cámara/panel y la llegada de los eventos de ratón a cualquier elemento de interacción en ocasiones ha creado problemas. Por ejemplo, en la escena *GameModes*, inicialmente surgieron algunos debido a que al estar a una gran distancia de la cámara no llegaban los eventos a pesar de colocar el ratón encima o pulsarlos.

Podemos concluir pues que el diseño de interfaces con *NGUI* es cómodo pero requiere del conocimiento de algunos conceptos clave para evitar conflictos y agilizar el desarrollo.

7.4. TRANSICIÓN ENTRE INTERFACES

Anteriormente se ha dicho que los elementos principales de transición han sido los botones y que éstos se situaban dentro de paneles. Por tanto es fácil deducir que, si pudiésemos mover esos paneles, podríamos cambiar fácilmente de pantalla e implementar nuevas funcionalidades necesarias.

La manera de mover estos paneles son los **Tween.** En concreto, asociar un **Tween Position** a un objeto permite el movimiento de éste por código, por tanto asociarlos a los paneles nos daría la funcionalidad que buscamos. Es sencillo: tras asociarlo al objeto hay que desactivarlo, ya que por defecto lo mueve al origen de coordenadas.

Ahora es el momento de moverlo cuando nosotros lo necesitemos por medio de código.

Posteriormente explicaremos como conseguimos mover el panel mediante un mensaje enviado por cualquier botón al script que contiene el método, pero por ahora hablaremos sólo de dicho script. El primer paso es declarar un objeto tipo *Tween Position* que nos permita adjuntar uno de nuestros paneles. Tras hacer esto hay que tener en cuenta 3 propiedades fundamentales:

- From: La propiedad From especifica la posición desde la cual se va a mover el objeto. Debemos declarar un objeto **Vector3** con las coordenadas x, y, z de la posición origen.
- To: La posición destino. Al igual que en el anterior, declaramos un **Vector3** con las coordenadas destino en la escena.
- Duration: Determina el tiempo que tardará el movimiento entre ambas posiciones.

En la siguiente ilustración observamos un ejemplo doble de *Tween Position*: al mismo tiempo que pinchamos sobre la pestaña, se intercambian los paneles en el eje Z y se cambia la posición de los botones que actúan como pestaña.

Ilustración 58: Tween en las pestañas y cambio de paneles

Una vez completada las interfaces de una escena, podríamos requerir navegar a otra diferente. La clase **Application** proporciona el método **LoadLevel** que permite cargar la escena escrita en un *string* que se envía como parámetro.

7.5. ANIMACIONES

Para darle un toque más trabajado al juego, en algunas escenas se han implementado animaciones. Un ejemplo es la animación del menú principal para mostrar por primera vez los botones de *play, credits* y *exit*, o las animaciones de carga de cada modo de creación.

Podemos afirmar con seguridad que la mayoría de animaciones consisten en la modificación del componente *transform*, salvo algunas que modifican el *alpha* o transparencia de los objetos. Posteriormente se hará una clasificación de las existentes. De cualquier manera, todas las animaciones creadas en esta parte del proyecto se han creado con la herramienta *Animation.*

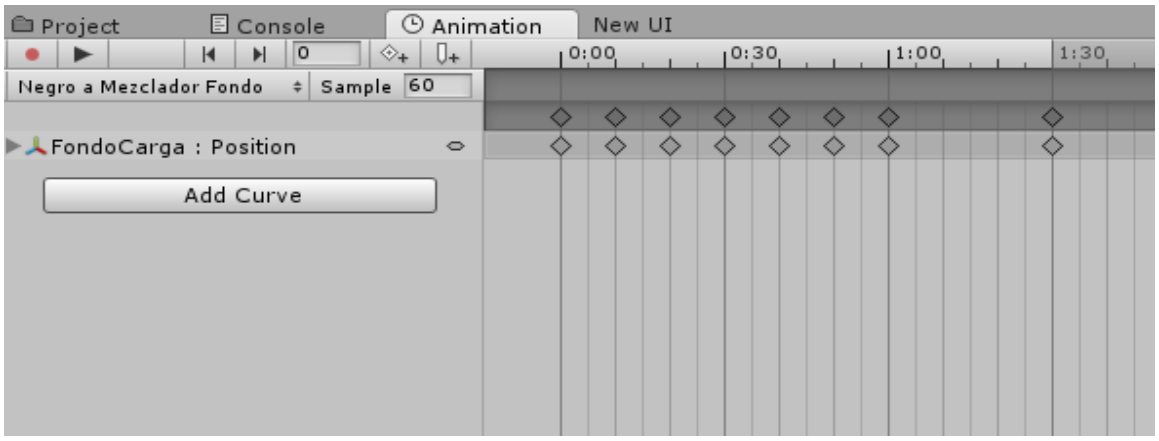


Ilustración 59: Herramienta animation

La forma de operar es la siguiente: se escoge la propiedad a modificar de entre los componentes del objeto, y se crea la animación mediante **key frames**. Se trata de crear un nuevo elemento en la línea de tiempo y cambiar los valores respecto del anterior. Automáticamente la herramienta generará todos los *frames* desde el *key frame* anterior hasta el creado, y así con el siguiente hasta acabar la animación. Cabe destacar el color de los botones, es decir, cuando una animación se está generando, los botones cambian al color rojo. Para terminar de editar una animación tan sólo hace falta volver a pinchar el botón de grabar.

Ilustración 60: Grabando animación

Si un objeto de la escena tiene una sola animación, ejecutará esta animación al principio de la escena. Si no se desea que la animación se repita, hay que examinar sus propiedades con el inspector y desactivar la propiedad ***Loop***. Para las animaciones de carga esta funcionalidad por defecto es muy útil, sin embargo puede requerirse alguna animación posterior lanzada por algún evento. Para ello debemos realizarlas desde código.

Aplicar una animación desde código requiere obtener la propiedad **animation** y ejecutar el método **Play**. Este método especifica el nombre de la animación y el modo de ejecución como parámetros. Hay que destacar que, si quiere realizarse una animación y detener todas las demás, es necesario utilizar el modo **PlayMode.StopAll**. Por otro lado, para ejecutar una función cuando la animación acabe hay que utilizar la expresión **yield return new WaitForSeconds(seconds)**. Esta expresión permite esperar el número de segundos especificados como parámetro hasta ejecutar la siguiente operación. Recordar que para utilizar esta línea de código el método que lo contenga debe devolver un **IEnumerator,** sino el compilador da problemas. La única pega es que posteriormente ese método no se puede depurar.

Por último hablar del componente **Animator.** Si un objeto no tiene este componente no se podrán adjuntar animaciones y por tanto no se podrán reproducir. Tiene que estar activado en todos los objetos que reproduzcan animaciones en el editor o activarse posteriormente por código.

7.6. INTERACCIÓN Y SISTEMA DE MENSAJES

Para desencadenar una acción después de pulsar un botón es obligatorio enviar un mensaje que ejecute un método existente en un script. La clase **UI Button Message** aporta esta funcionalidad.

UI Button Message es un script de tipo **Interaction** que proporciona *NGUI* y que se añade a los botones como componente. Tiene dos elementos clave:

- Trigger: Es el disparador, es decir, el evento que producirá el envío del mensaje. Existen algunos como *OnClick*(), *OnMouseOver*(), *OnMouseDown*()…
- Function Name: Es el nombre del método que va a ejecutar cuando se envíe el mensaje.
- Event Receiver: Es el objeto que va a recibir el mensaje, y por tanto el que debe contener el script adjunto con dicho método.

Por lo general los scripts que suelen contener estos métodos suelen ser los objetos *UI Root*. Estos scripts contienen el 98% de la funcionalidad de las interfaces de cada escena.

7.7. PERSISTENCIA DE DATOS

Probablemente el pilar fundamental de esta parte del proyecto. Los modos de creación diseñados y la propia temática y estructura de *Beast's Retreat* requieren que se almacene datos de usuario y objetos para poder avanzar en la historia. Tras hacer una pequeña investigación, se exponen las maneras de almacenar datos y cuales convienen más para este proyecto:

- Variables estáticas: Consiste en crear variables en los scripts que permanezcan sin destruirse a lo largo de la ejecución. Dicho script tiene que estar asociado a un objeto que al ejecutar el método *Awake()* utilice la función **DontDestroyOnLoad**() para no destruirse al cambiar de escena. No es conveniente ya que solo almacena datos en ejecución y nosotros queremos que se almacenen incluso cuando no se está ejecutando.
- PlayerPrefs: Almacena datos por medio de **keys** que define el usuario. Estos datos permanecen en *Unity* incluso cuando es cerrado, así que es muy conveniente para los datos de usuario.
- *XML*: Los *XML* permiten almacenar información a modo de pequeñas bases de datos. Ideales para juegos que no requieran un almacenamiento grande, como Beast's Retreat, ya que está desarrollado de manera que solo necesite conocer qué objetos posee.
- Bases de datos externas: La manera de almacenar grandes cantidades de datos. No nos conviene en este proyecto.

7.7.1. ENTRE ESCENAS

Entre escenas los datos más importantes a almacenar son los siguientes:

- El *slot*: Es fundamental conocer que slot ha seleccionado el jugador para poder almacenar los datos correctamente y no corromper los datos de otros slots. Hay un total de 3 slots soportados.
- Los escenarios superados: Inicialmente el usuario no ha superado ningún escenario. Es importante que conforme se vayan superando se vaya actualizando el valor para poder avanzar en el juego.

- Los créditos: Obligatorios para comprar material, elementos y combinarlos para obtener torres.

El funcionamiento global es el siguiente: cada vez que se ejecuta el juego se comprueba por medio del parámetro con *key* **slotConPartida** si hay partida guardada (en función de si está a 1 o a 0). Si la hay, cambia el slot y pone el número de escenarios superados. Una vez seleccionado un slot, si hay partida entra directamente. Si no hay partida comienza una nueva cambiando el valor **slotConPartida** a 1, y cambia las *keys* **creditsSlotX** (x es el slot correspondiente) a 4000 como valor inicial y los **stagesSlotX** a 0 (valor que determina el número de escenarios superados). Tanto si hay partida como si no, cambia el valor de la k*ey* **slot** al seleccionado.

Una vez hecho esto consiste únicamente en cargar los valores en otras escenas. Debemos utilizar la clase estática **PlayerPrefs** y los métodos **GetInt(key)** para cargar, o **SetInt(key,value)** para guardar datos. Cualquier tipo de transacción con *PlayerPrefs* requiere que se ejecute al final el método **Save()**.

Por último añadir la funcionalidad de borrar partidas guardadas. Existe un botón en la escena **GameModes** que permite borrar los datos de un *slot*. Esto es básicamente poner todos los elementos de *PlayerPrefs* referentes a un slot a sus valores nulos y borrar el contenido el fichero que contiene los objetos, explicado en el próximo apartado.

7.7.2. OBJETOS

En la introducción de la persistencia de datos se ha hablado del por qué utilizar *XML* era conveniente para este proyecto. La persistencia de objetos se ha realizado mediante la lectura y escritura en ficheros *XML* llamados **ItemDatabaseSlotX**, que almacenan por unidad los objetos.

Para poder gestionar estos objetos necesitamos crear la clase **Item**. Esta clase contiene los atributos **ItemName** e **ItemType**, a partir de los cuales más adelante se pueden gestionar las operaciones de base de datos. La clase que los gestiona es la clase **ItemDatabaseManager**. Contiene los métodos de lectura y escritura a los ficheros *XML* que se utilizan en los diferentes modos de creación.

Básicamente gestiona los objetos con **listas.** Una vez añadidos a la lista los evalúa por nombre, realizando las operaciones oportunas según el método. Existe el método **guardarEnDBXMLDOCUMENT(listaItems)**, que escribe en el *XML* los objetos que obtenga como parámetro y se usa en todos los modos de creación. Por el lado contrario existe **BorrarDeDB()**, utilizado cuando se limpian los datos de un slot. Para

los modos de combinación y creación de torres existen los métodos **CombinarEnDb.** Este método tiene 2 sobrecargas, eliminando los objetos combinados según el modo (3 en el editor de torres, 2 en el mezclador) e insertando el nuevo mediante el primer método explicado.

Mediante el uso de estos métodos es posible la carga y almacenamiento de los objetos en los diferentes modos de creación. Todos los modos siguen este esquema de carga: se crea para cada elemento o material una variable tipo **int** que se actualiza en función de cada elemento que lea al recorrer la lista de objetos. Por otro lado, para desbloquear las torres en el editor se utilizan variables **bool**. Ambas permiten más tarde la comprobación de la existencia de cada objeto para utilizarlos según la escena en la que se encuentren y la funcionalidad requerida. También en función de estas variables se actualizan elementos de interfaz como los *labels* de cantidad o determinados *sprites*.

7.8. INTEGRACIÓN CON EL GAMEPLAY

El punto de conexión de la interfaz con el *gameplay* es la escena **Defense Strategy**. Esta escena te permite seleccionar la formación de torres que quieres utilizar para el *gameplay*. La forma de que esas torres carguen correctamente en el escenario es la siguiente:

- Existe una variable almacenada en *PlayerPrefs* llamada **selectedStage**. Esta variable permite conocer el escenario seleccionado y cargarlo correctamente.
- La formación se almacena en variables de *PlayerPrefs* llamadas **TorreX**, que mediante un *string* define el tipo de torre almacenado en esa posición.
- Al comenzar el escenario se cargan las torres en función de la formación almacenada. Según la variable *selectedStage,* se modifican los generadores de hordas para simular el aumento de dificultad.
- Una vez acabada la ejecución del escenario, en caso de victoria por parte del jugador se cambia el valor *stagesSlotX* y se aumentan los créditos en función de la vida que le reste al jugador.

7.9. ESCENAS

En este apartado se pretende describir como se han implementado las funcionalidades clave de cada escena y los detalles técnicos como la iluminación, fondos o sonido.

En primer lugar, las escenas. Todas las que vayan a estar en el proyecto y sean ejecutadas deben estar declaradas en el apartado **Build Settings**, ya que sino *Unity* no las compila.

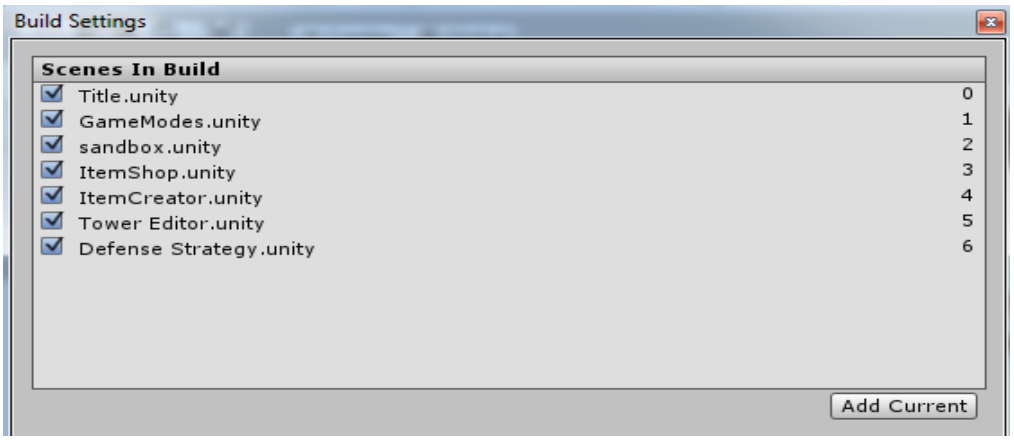


Ilustración 50: Build Settings y escenas añadidas

Cada una de ellas tiene diferentes funcionalidades, aunque en algunos casos han sido reutilizadas:

- Title: Contiene operaciones de transición y carga de datos que ya han sido comentadas anteriormente.
- GameModes: Su rasgo más importante a nivel de implementación es la selección de modo. La activación de **Campaña** o **Survival** se realizará mediante el correspondiente botón, que activará una variable **modo** comprobada más adelante.
- ItemShop: Tan sólo tiene funciones de guardado. Como información de los objetos que compra el jugador, cada vez que selecciona uno lo agrega a la lista y a un *string* que se mostrará en la pantalla en la parte derecha. El botón limpiar lista permite borrar todos los elementos seleccionados hasta ahora. Pulsar el botón **Buy** comprobará los **creditosAGastar** que se han ido sumando con cada elemento seleccionado, y si es menor que los créditos almacenados en *PlayerPrefs* entonces podrá realizar la compra. Cualquier operación no permitida será mostrada en el *label* inferior de la pantalla.
- ItemCreator: Carga los objetos de la manera explicada en el apartado 7.7.2 y opera con ellos con las combinaciones. Cada vez que un objeto es seleccionado, si lo posee, se carga en un *sprite* y actualiza una variable tipo **string**. El método **comprobarCombinaciones()** permite asegurar que la combinación es válida y entonces la realiza, ejecutando el método de *ItemDatabaseManager*.
- TowerEditor: Funciona exactamente igual que la escena *ItemCreator*.

- DefenseStrategy: La única diferencia con respecto a las anteriores es la carga de una textura de fondo en función del escenario. Al iniciar la escena comprueba la variable *selectedStage* y mueve unas texturas u otras.

7.9.1. ILUMINACIÓN Y FONDOS

Este apartado solo se aplica a escenas que tengan elementos 3D colocados, ya que no afecta a los elementos 2D insertados mediante *NGUI*. Éstas son **Title** y **Tower Editor**. Para ambas se han utilizado **Directional Light**, un tipo de luces que apuntan desde el infinito hacia la dirección a la que está orientada, independientemente de la posición del propio objeto.

Acerca de los fondos hay que nombrar el **SkyBox**. Es una propiedad ubicada en los **RenderSettings** que permite colocar materiales como cielo en las escenas. Al fin y al cabo, cada escena está contenida dentro de un cubo con texturas o con un color seleccionado en estas opciones.

7.9.2. MÚSICA

La música de fondo de cada escena ha sido adjuntada a los elementos **MainCamera** de cada escena mediante componentes **AudioSource**. Estos componentes permiten adjuntar un fichero de audio y que se reproducido por el objeto. Pueden seleccionarse propiedades como **Play on Awake** para reproducirlo al principio de la escena y **Loop** para repetirlo cada vez que finalice.

Para comodidad del usuario se han implementado los llamados **Sound On/Off**, *checkbox* destinados a silenciar la música de fondo y los efectos de sonido de los botones. Estos *checkbox* contienen un script adjunto como componente que permite, al igual que los *Button Message*, asignar un *Event Receiver* y un nombre de función a ejecutar. En este caso el *Event Receiver* es la cámara principal, que contiene el script **Musica On Off**, encargada de silenciar los sonidos. Recoge todos los componentes *AudioSource* de la cámara y baja el volumen a 0, o lo sube en caso de activarlo.

Por otro lado están los **Button Sound**. Estos scripts permiten asignar un sonido diferente por cada evento de ratón. Por lo general se ha utilizado el sonido **Tap**, que venía como ejemplo en una carpeta del paquete *NGUI*.

8. RT-DESK

Bajo el acrónimo de Real Time Discrete Event Simulation Kernel, se trata de un gestor de eventos discretos orientado a simular sistemas mediante teoría de colas, que permitirá reducir el coste de computación del dispositivo. Actúa como núcleo de gestión de eventos por medio de un sistema de mensajes definidos por el desarrollador en función del sistema en el que se vaya a integrar y la funcionalidad requerida.

Decimos que es discreto porque la ocurrencia de un evento provoca la respuesta del sistema y el lanzamiento de los mensajes oportunos frente a la constante pregunta por dicho evento por parte de un gestor continuo. Por lo tanto hablamos de un gran ahorro y aumento en el rendimiento en cuanto al uso de CPU gracias a este racionamiento de potencia.

Como hemos mencionado anteriormente, el desarrollador creará mediante *RT-DESK* su propia gama de mensajes que el gestor utilizará, asegurando su llegada en el instante de tiempo deseado por el programador y estableciendo la comunicación entre los objetos seleccionados. Se trata pues de una *API* con clases, funciones y métodos orientados a obtener esta funcionalidad, que debe integrarse en el software con el que se desarrolla cada producto.

En este caso se trata de *Unity*, ya que estamos desarrollando un videojuego. La *API* se proporciona en el lenguaje C++, que no es soportado por el motor de juego, por lo tanto tuvimos que traducirla al lenguaje *C#*. Por motivos temporales mi compañero Javier Baixauli la tradujo completamente, delegándome la primera integración y adaptación a *Unity*, de la que hablaré en el siguiente apartado.

8.1. INTEGRACIÓN A UNITY

Tras la traducción del código de la *API* a C# llegó el momento de la integración. Aquí nos encontramos frente a un problema serio. *Unity* dispone de dos versiones, la versión estándar, gratuita y libre de uso para todo el mundo, y la versión Pro, con funcionalidades extra para desarrolladores profesionales. Ésta última dispone de la integración de *DLL's* propias al motor de juego que la primera versión no tiene.

Por tanto integrar *RT-DESK* como *DLL* queda descartado, teniendo que buscar una segunda manera. La opción escogida fue crear un script *C#* por cada clase y fichero del *RT-DESK*. A partir de aquí llegó el segundo problema principal. *Unity* dispone de un editor propio con el que gestionar el código de tus proyectos llamado MonoDevelop.

Cualquier detalle o configuración acerca del código de un proyecto debe solucionarse desde ahí, lo que crea cierta independencia entre el editor gráfico y el editor de código.

De esta forma *Unity* está configurado para cada trabajar con el *framework .NET* 3.5 por defecto. Algunas de las funciones utilizadas en el código requieren del uso de las versiones 4.5 y 4.0. Aunque ésta última sí que está soportada por *Unity,* la independencia de la que hablamos anteriormente hacía que con cualquier mínimo cambio en el editor gráfico se volviese a restaurar la configuración por defecto en el editor de código, es decir, volver a la versión 3.5. Estuve mirando la manera de conseguir que ésta configuración no cambiase pero parece que la versión gratuita de *Unity* no permite conservar estos cambios.

La única opción fue adaptar el código a la versión 3.5. Tras adaptar toda la *API* pude observar que la función más afectada fue la de *Agressive Inlining*, que se situaba encima de cada cabecera de cada método o función y proporcionaba la optimización de omitir la lectura de su código excepto cuando fuese utilizado. Por otro lado también afectó a varias librerías importadas, sobre todo en la librería *Thread* y las *tasks*.

Una vez que conseguí la compilación del código era hora de hacer las primeras pruebas. Nuestra primera prueba consistió en el cambio de color de un cubo en un proyecto aparte creado únicamente con el fin de comprobar el correcto funcionamiento de la *API*. Para ello tuvimos que crear diferentes clases:

- La clase CuboRTDESKEntity, extiende de RTDESKEntity y sobreescribe el método ReceiveMessage. Este método es clave ya que recibe e interpreta el mensaje enviado. Esta clase no hace falta que extienda de *Monobehaviour*.
- La clase Cubo, en cuyo interior se define un objeto de tipo CuboRTDESKEntity para enviarle el mensaje del cambio de color.

Sin embargo esta funcionalidad implementada era muy vaga y para nada utiliza el verdadero potencial de *RT-DESK,* por ejemplo, no utiliza *Timers* ni se han declarado más mensajes. Esta primera prueba fue realizada sin una verdadera comprensión de la *API* en su totalidad y por tanto no llegó a funcionar. A partir de este punto se encargó mi compañero Javier Baixauli de completar el funcionamiento y obtener una escena en la que dos esferas se movían. Puede verse el resultado final en su proyecto **Videojuego Beast's Retreat en Unity con C# integrando RT-DESK: Gameplay**.

9. CONOCIMIENTOS APLICADOS EN EL PROYECTO

Desarrollar un juego implica conocer una amplia variedad de conocimientos informáticos. Muchos de ellos se aprenden en la carrera que hemos cursado, aunque algunos de ellos no son el caso. A continuación se exponen los conocimientos fundamentales que se han requerido para la realización de este proyecto y las asignaturas que nos los han facilitado por orden de necesidad según ha ido avanzando el desarrollo.

- **Matemáticas**: Tanto para realizar operaciones en código como para los sistemas de referencia. Se han necesitado buenos conocimientos matemáticos, tanto básicos como cálculos en sistemas 3D para colocar los objetos en las coordenadas correspondientes, operar con sus tamaños, traslaciones, saber ubicarse en la escena, iluminación, sombras y reflejos. También se han necesitado algunos conocimientos estadísticos para la realización de esta memoria. Asignaturas: *Análisis Matemático, Estadística, Matemática Discreta y Álgebra, Gráficos por Computador*.

- **Gráficos por computador y diseño 3D**: El modelado 3D de objetos y la creación de escenarios y escenas en *Unity* han requerido de estos conocimientos, que obtuvimos en su gran mayoría en el último año de carrera gracias a la especialidad elegida. Lo mismo se aplica a las animaciones. Asignaturas: *Gráficos por computador, Diseño Asistido por Computador, Fabricación Asistida por Computador, Producción de Imagen Digital.*

- **Programación**: La programación ha sido un conocimiento fundamental para la realización de *Beast's Retreat*. Prácticamente cualquier tipo de funcionalidad interna debía implementarse con código. Incluímos dentro de este conocimiento la comprensión de la propia programación (es decir, compiladores, lenguajes de bajo nivel, otros lenguajes relacionados con *C#*) y cualquier tipo de concepto de programación optimizada: usabilidad, reutilización de código, programación orientada a objetos, entre otros. Asignaturas: *Programación, Estructura y Tecnología de Computadores, Estructura de Datos y Algoritmos, Estructura y Tecnología de Computadores II, Metodología y Tecnología de la Programación, Introducción a la teoría de autómatas y lenguajes formales, Desarrollo de Software Basado en Componentes, Ingeniería del Software de Sistemas, Laboratorio de Desarrollo de Sistemas de Información, Concurso de Algoritmos ACM, Ingeniería de la*

Programación, Inteligencia Artificial, Procesadores de Lenguajes y Gráficos por Computador.

- **Lógica:** Tanto para el código como para la implementación de algunas animaciones se requerían algunos conocimientos de lógica. Hemos dado un buen número de asignaturas a lo largo de la carrera que nos han facilitado estos conocimientos. Asignaturas: *Estructura y Tecnología de Computadores, Matemática Discreta y Álgebra, Diseño Lógico, Estructura y Tecnología de Computadores II, Arquitectura e Ingeniería de Computadores.*

- **Edición multimedia:** Con multimedia nos referimos en este caso a la edición de imagen y sonido. Los conocimientos enseñados en la carrera son escasos, aunque fueron ampliados gracias a la selección de algunas asignaturas optativas de la especialidad Multimedia de Ingeniería Técnica en Informática de Sistemas. Asignaturas: *Introducción a la Síntesis, Edición y Postproducción de Audio, Introducción a la Edición y Postproducción de Imágenes y Vídeo, Fundamentos de Sistemas Multimedia, Producción de Imagen Digital, Tratamiento de Imagen Digital.*

- **Bases de datos:** El almacenamiento de datos ha sido uno de los factores necesarios para implementar la persistencia de objetos. A lo largo de la carrera se han dado un par de asignaturas con esta temática, aunque en algunas otras se ha trabajado indirectamente con almacenes de datos. Asignaturas: *Bases de Datos, Arquitectura de Sistemas de Bases de Datos, Bibliotecas Digitales, Metodología y Tecnología de la Programación, Laboratorio de Desarrollo de Sistemas de Información, Fundamentos de Sistemas Multimedia e Ingeniería de la Programación.*

- **Ingeniería del software:** Con Ingeniería del Software abarcamos los criterios a seguir del buen desarrollador, así como la gestión y planificación de proyectos (ciclo de vida), trabajo en equipo, control de versiones de proyecto en repositorios, coordinación entre desarrolladores, etc. Ha sido fundamental para escribir código eficiente, evitar costes extra por fallos de coordinación y planificar el proyecto desde el primer día. Asignaturas: *Desarrollo de Software Basado en Componentes, El Proceso del Software, Ingeniería del Software de Sistemas, Laboratorio de Desarrollo de Sistemas de Información, Ingeniería de la Programación, Ingeniería de Requerimientos.*

- **Medición de prestaciones:** Para documentar el proyecto se han requerido algunos conocimientos matemáticos y de cálculo de prestaciones y costes tanto de las máquinas como del propio software desarrollado. Asignaturas:

Computabilidad y Complejidad, Estadística, Evaluación de Sistemas Informáticos y Arquitectura e Ingeniería de Computadores.

- **Idiomas:** El idioma principal del juego es el inglés. A nivel de programación, hemos dado alguna asignatura que nos ha ayudado a mejorar nuestra gama de vocabulario. Asignaturas: *Inglés Técnico*.

10. CONCLUSIONES

Desde que decidí que quería entrar en la industria del videojuego, supe que la mejor manera era hacer un juego como *PFC*. Era la oportunidad perfecta para aprender a utilizar un motor de juego, comprender el funcionamiento y comenzar con mi primer desarrollo.

Sin embargo al principio no estaba nada seguro. Por mi cuenta ya había utilizado el motor de juegos *Unity* para ir aprendiendo un poco y trasteando, así que ya sabía lo complejo que podía ser desarrollar un videojuego completo. De la misma manera, este año ha sido un año muy intenso en cuanto a curso académico debido al gran número de asignaturas (aparte de que mi especialidad tenía un buen número de ellas, tuve que escoger varias de libre elección y aún cargaba con dos del año pasado) y a las prácticas en empresa que me dejaban con muy poco tiempo libre para dedicarle a este proyecto. Estos factores provocaron que nos volcásemos en el desarrollo muy tarde, tras haber finalizado los exámenes de junio.

A pesar de ello nos sobrepusimos a la falta de tiempo y a la dificultad de aprendizaje, algo de lo que me siento orgulloso. El miedo de no acabar pasó de pronto a las ganas de hacer más contenido del que había propuesto para la entrega de mi proyecto, ya que al ser aficionado a los videojuegos, ir viendo el resultado final conforme el desarrollo avanzaba fue alentador. Ha sido sin duda la mejor experiencia de la carrera y la más innovadora, ya que hasta ahora solo había desarrollado proyectos como páginas web o aplicaciones de escritorio.

Por otro lado está la integración del *RT-DESK*. El objetivo principal era integrarlo para mejorar el rendimiento en el *gameplay*, por lo que no tiene demasiado peso en este proyecto debido a la separación del trabajo a realizar. En cambio he colaborado con mi compañero Javier Baixauli en su proyecto **Videojuego Beast's Retreat en Unity con C# integrando RT-DESK: Gameplay** para la integración de éste, adaptando las diferentes clases traducidas por él al motor de juego, y con ayuda del tutor de proyecto me di

cuenta de que podría ser una opción interesante para implementar el sistema de mensajes de los botones.

Puedo concluir pues que este proyecto ha sido divertido pero también el más difícil, ya que hice otro *PFC* en el último año de Ingeniería Técnica Informática de Sistemas. Es un proyecto que volvería a repetir si tuviese más tiempo para no ir tan apurado, ya que hemos sido muy bien guiados por nuestro tutor Ramón Mollá y hemos aprendido mucho.

11. BIBLIOGRAFÍA

Preparación pre-proyecto:

- Libro Unity 3D Game Development by Example: Beginner's Guide by Ryan Henson Creighton.

- https://unity3d.com/es/learn/tutorials/modules

Implementación:

- https://cgcookie.com/unity/cgc-courses/getting-started-with-ngui-for-unity/
- http://www.tasharen.com/forum/
- http://answers.unity3d.com/questions/
- https://www.safaribooksonline.com/library/view/ngui-for-unity/
- www.youtube.com (hay todo tipo de tutoriales y canals de *Unity)*
- *Asset Store* de *Unity*

Imagen:

- www.**deviantart**.com/
- free**fantasymaps**.org/
- http://www.tor.com/blogs/2013/12/how-to-make-a-fantasy-world-map-emperors-blades
- www.google.es (buscador de imágenes)

Sonido:

- www.youtube.com (búsqueda de efectos de sonido y música de fondo)

Redacción de la memoria:

- http://mexico.cnn.com/tecnologia/2014/08/29/15-datos-para-entender-la-industria-de-los-videojuegos
- http://www.aevi.org.es/la-industria-del-videojuego/en-espana
- riunet.upv.es
- www.minijuegos.es
- www.armorgames.com
- www.wikipedia.org

12. GLOSARIO

Plugins: Un complemento o plugin es una aplicación que se relaciona con otra para aportarle una función nueva y generalmente muy específica. Esta aplicación adicional es ejecutada por la aplicación principal e interactúan por medio de la *API*.

API: Interfaz de programación de aplicaciones (IPA) o API (del inglés *Application Programming Interface*) es el conjunto de funciones y procedimientos (o métodos, en laprogramación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

DLL: Una biblioteca de enlace dinámico o más comúnmente DLL (sigla en inglés de *dynamic-link library*) es el término con el que se refiere a los archivos con código ejecutable que se cargan bajo demanda de un programa por parte del sistema operativo.

Kernel: En informática, un núcleo o kernel (de la raíz germánica *Kern*, núcleo, hueso) es un software que constituye una parte fundamental del sistema operativo, y se define como la parte que se ejecuta en modo privilegiado (conocido también como modo núcleo). En este proyecto se refiere al gestor *RT-DESK*.

Gameplay: Interacción del usuario con la simulación de juego.

GDD: Game Design Document. Este documento contiene todos los requisitos funcionales y técnicos de un juego y se realiza antes de la implementación. Disponible en el Anexo A.

PFC: Proyecto Final de Carrera.

Background: Hace referencia a procesos en segundo plano.

Sprite: Se trata de un tipo de mapa de bits dibujados en la pantalla de ordenador por hardware gráfico especializado sin cálculos adicionales de la CPU.

13. ANEXO A: GDD DEL JUEGO

Documento de diseño para:

BEAST'S RETREAT

The Ultimate Tower Defense Game

Version # 1.00

TABLA DE CONTENIDOS

VISIÓN GENERAL DEL JUEGO

FILOSOFÍA

PUNTO FILOSÓFICO #1

Plantear retos a los usuarios que les atraigan los juegos de estrategia/habilidad y poner a prueba su capacidad para superar complicados niveles.

PUNTO FILOSÓFICO #2

Entretener al usuario y aislarlo de la realidad y los problemas que conlleva.

PUNTO FILOSÓFICO #3

Despertar interés en el usuario por conseguir todos los elementos y descubrir todas las combinaciones posibles.

PREGUNTAS FRECUENTES

¿QUÉ ES EL JUEGO?

Consiste en defender tu base de los diferentes enemigos que irán apareciendo, mediante el uso de diferentes torretas que el jugador podrá manejar. Utilizando estas torres deberemos apuntar y disparar a los enemigos para acabar con ellos y evitar que destrocen la base.

¿POR QUÉ SE HA CREADO EL JUEGO?

Este tipo de juegos ha dado resultados en el pasado, pero actualmente no hay ningún juego sobresaliente en este ámbito. Si se gestiona bien podría llegar a ser un juego adictivo que guste a un amplio sector del mercado. Juntando esto con la capacidad para expandir el juego mediante nuevas torres y elementos para mezclar podemos

crear un efecto adictivo en el usuario para conseguir todos los elementos y combinaciones.

¿DÓNDE TOMA LUGAR EL JUEGO?

Transcurre en un mundo de fantasía que combina diferentes elementos medievales, mitológicos y heroicos. El escenario principal es La Península, y contiene paisajes tan variados como llanuras con castillos, bosques, desiertos, montañas de esmeralda, lagos y ciudades costeras.

¿QUÉ PUEDO CONTROLAR?

El control del juego está basado en las diferentes torres que podemos utilizar. En el juego tendremos una base con 6 torres distintas, las cuales utilizaremos para disparar y acabar con los enemigos.

Estas torres deberán crearse a partir de diferentes elementos que el usuario podrá combinar para obtener nuevos objetos que posteriormente se utilizarán en los mapas.

Podemos elegir que torres utilizar para cada misión, siendo de vital importancia elegir las torres adecuadas para mejorar la eficacia de los disparos según los tipos de enemigos que aparecerán.

¿CUÁNTOS PERSONAJES PUEDO CONTROLAR?

El jugador podrá manejar diferentes tipos de torretas dependiendo de las combinaciones que haya realizado y el número de éstas que permita el nivel en el que se encuentre. Independientemente del número de torretas disponibles, solo podrá controlarse una a la vez.

¿CUÁL ES EL OBJETIVO?

El objetivo final es defender la base de las hordas enemigas y lograr eliminar a todos los enemigos sin que nuestra vida llegue a cero.

¿QUÉ TIENE DIFERENTE?

Este juego se diferencia del resto por combinar elementos de varios géneros, como es la elaboración de torretas a partir de elementos que podremos obtener de diferentes formas.

CARACTERÍSTICAS

CARACTERÍSTICAS GENERALES

Modelos 3D.
Vista 2D.
Cambio de cámara.
32-bit color.
Diferentes niveles.

EDITOR

Fácil uso.
Integrado en el juego.
Crea nuevas torretas mediante combinación de elementos.

GAMEPLAY

Jugabilidad 2D.

Selección de trayectorias de proyectiles en base a altura y potencia.

Modo de juego survival por oleadas.

EL MUNDO DEL JUEGO

VISIÓN GENERAL

El mundo en el que se sitúa el juego será de fantasía con influencias medievales, mágicas y referencias a otros juegos.

Este mundo contendrá escenarios y lugares muy diversos como: praderas verdes en colinas, bosques élficos, cuevas infestadas de murciélagos , minas enanas, poblados humanos, llanuras de orcos, desiertos, tierras volcánicas, montañas de esmeralda...

CARACTERÍSTICAS DEL MUNDO

Los diferentes escenarios tendrán efectos especiales según el clima de la zona. Esto puede variar desde una niebla que cubre unas laderas, a lluvia o incluso tormentas de arena.

EL MUNDO FÍSICO

VISIÓN GENERAL

El mundo físico estará muy limitado, de forma que solo podremos ver el escenario en el cual estemos actualmente. Solo influirán en la jugabilidad la pendiente o silueta del escenario y el viento. De forma que el mundo simplemente servirá de ambientación.

LUGARES CLAVE

Las diferentes localizaciones clave serán los distintos niveles del cuál conste el juego. Habrá una serie de niveles los cuales deberemos ir superando para seguir al siguiente. El orden de estos niveles será crucial pues si no conseguimos superar el primero no podremos acceder al segundo y así sucesivamente.

VIAJES

Los viajes por el mundo se realizarán a través de un mapa de selección de nivel. En este mapa podremos ver todo el mundo creado y seleccionar el nivel que queremos jugar y por tanto viajar a través del mundo. Como ya se ha comentado para poder ir a los escenarios más lejanos deberemos superar primero los niveles que nos llevan a estos.

ESCALA

La escala a la que se representará el mundo será siempre la misma. Obviamente será una escala reducida para poder observar el escenario con perspectiva y así atacar al enemigo siguiendo una estrategia.

SISTEMA DE RENDERIZADO

VISIÓN GENERAL

El juego se renderizará utilizando una perspectiva horizontal situando la cámara en el eje z y apuntando a x e y. Será una vista típica 2D utilizando un sistema de renderizado 3D anclado en dichos ejes.

RENDERIZADO 2D/3D

Se utilizará el motor gráfico que ofrece *Unity*. De esta forma al utilizar un modelo 3D luego podremos realizar diferentes movimientos alrededor del mundo para cambiar entre diferentes vistas.

CÁMARA

VISIÓN GENERAL

Básicamente podremos variar entre dos puntos de cámara distintos (z y -z). Este movimiento de cámara permitirá situarnos a ambos lados de un camino pudiendo así tener más torres a elegir.

DETALLE DE CÁMARA #1

Movimiento en el eje x. Podremos movernos alrededor del eje x de una forma limitada. Es decir podremos movernos desde el punto de inicio de los enemigos hasta nuestra base.

Movimiento en el eje y. Podremos movernos alrededor del eje y de una forma limitada. Es decir podremos movernos desde el suelo hasta la altura de la torre más alta.

Movimiento en el eje z. Podremos movernos alrededor del eje z de una forma limitada. Es decir podremos movernos desde el punto más cercano a la fila de torres hasta un lugar más alejado que nos permita ver el máximo rango posible.

DETALLE DE CÁMARA #2

El segundo tipo de movimiento nos permitirá realizar una translación en la cámara para intercambiar la posición z de esta. Es decir que solo podrá estar en dos posibles posiciones z opuestas y su punto de mira siempre será el mismo. Será como mirar una calle desde un lado o desde el otro. Esto nos permitirá tener dos filas de torres disponibles, pero solo podremos utilizar una de las filas según en qué posición estemos.

MOTOR DEL JUEGO

VISIÓN GENERAL

El motor físico del juego tendrá que tener en cuenta varias cosas en mente. Deberá controlar a los enemigos, los proyectiles y límites del escenario.

DETALLES DEL MOTOR

Se deberá poner especial cuidado en las trayectorias de los proyectiles. Estos proyectiles dependiendo de sus características físicas realizarán diferentes trayectorias para una misma altura y fuerza. El peso y la forma de los proyectiles afectarán a la trayectoria haciendo que no solo importe la altura y la potencia con la que se lancen.

DETECCIÓN DE COLISIONES

Será de vital importancia hacer un buen uso de las colisiones de los proyectiles con los enemigos. Para asegurarnos de no perdernos ninguna colisión se hará un seguimiento especial de los proyectiles teniendo en cuanta que van a llevar una velocidad importante y que se debe comprobar adecuadamente su colisión.

También se deberán comprobar las colisiones de los enemigos con el límite de nuestra base y así eliminar su aparición en el mundo y evitar sobrecargas así como hacer lo mismo con el lugar de inicio de los enemigos y los proyectiles.

MODELOS DE ILUMINACIÓN

VISIÓN GENERAL

El modelo de iluminación que utilizaremos será un modelo ambiental que dependerá de cada escenario. Se utilizará una iluminación basada en luces direccionales basándonos en la posición del sol en cada uno de nuestros escenarios. A parte de estas luces también deberemos tener en cuenta ciertas luces puntuales que emiten algunas partes de los escenarios como podrían ser hogueras o esferas de energía.

DISPOSICIÓN DEL MUNDO

VISIÓN GENERAL

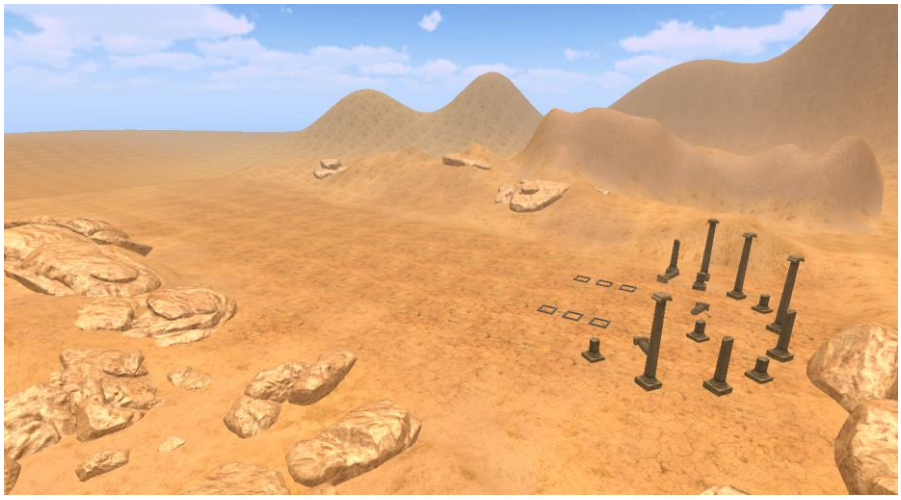
Los diferentes escenarios serán explorables a través de un mapa donde podremos seleccionar el nivel. A continuación se mostrará un mapa del mundo así como diferentes escenarios de este.

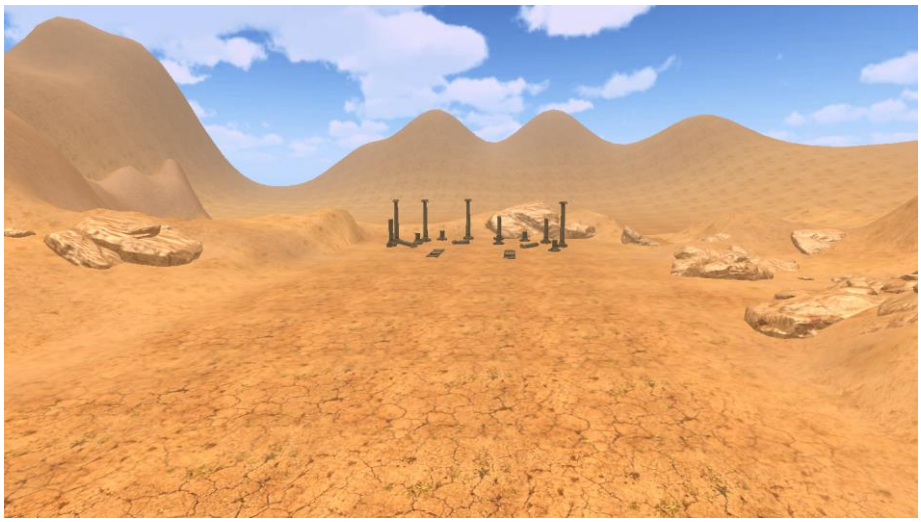
DETALLES DEL MUNDO #1

Mapa del mundo

DETALLES DEL MUNDO #2

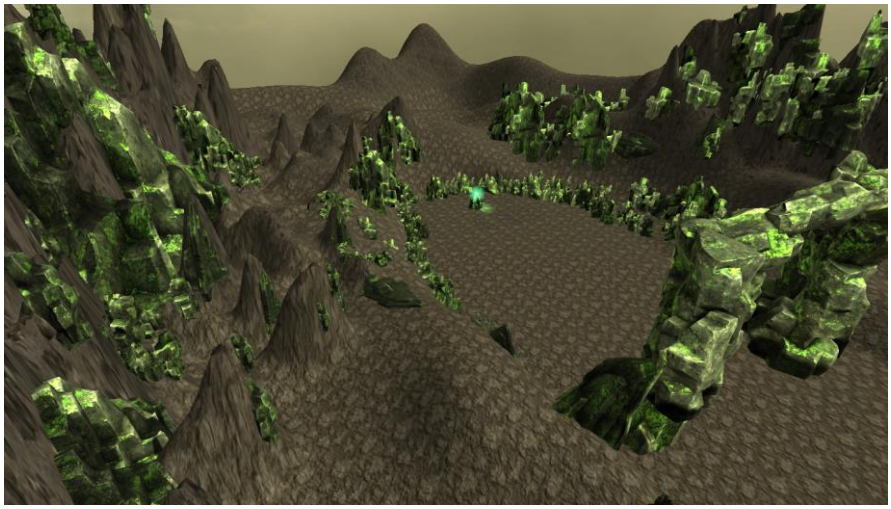
Desierto de Janna

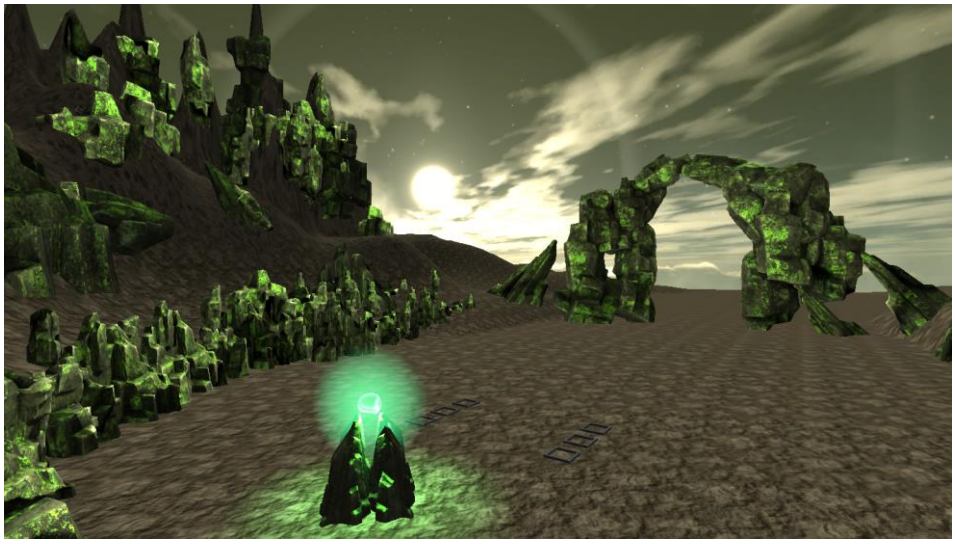


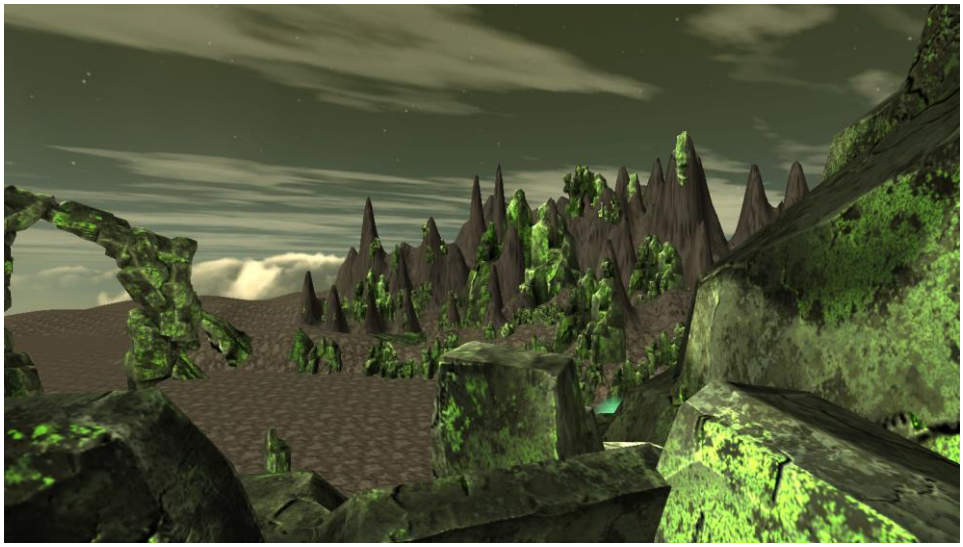


DETALLES DEL MUNDO #3

Asper Mountains

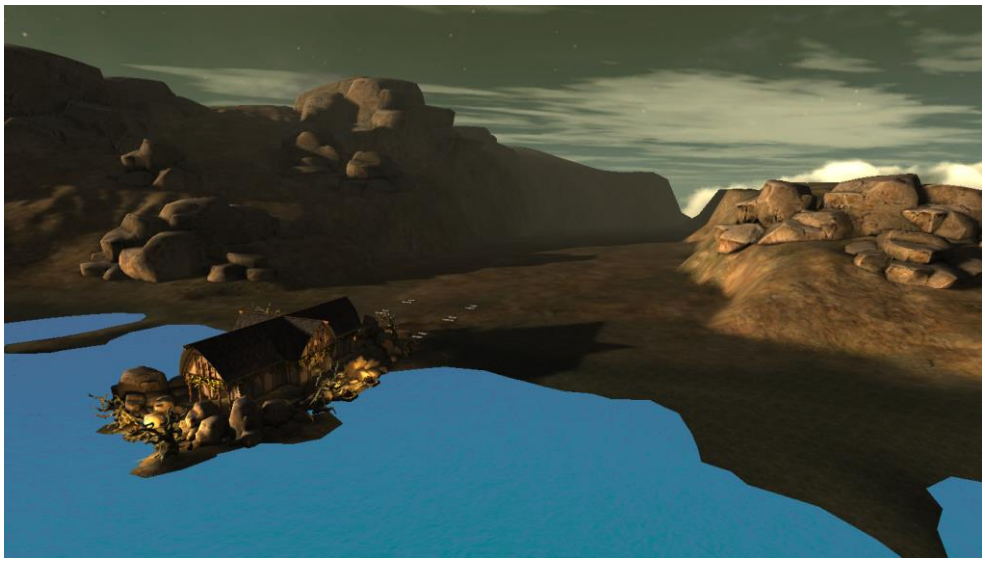


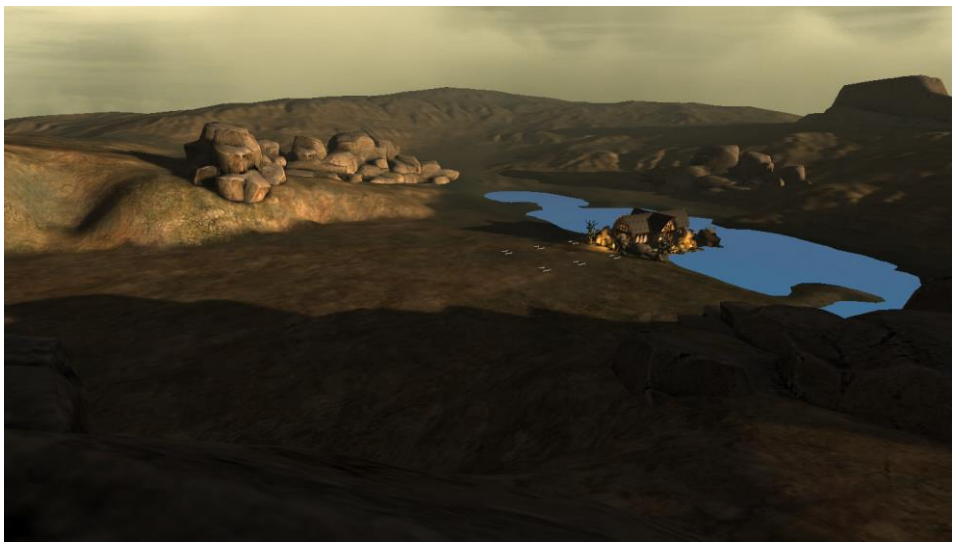




DETALLES DEL MUNDO #4

Castillo de Mereno





PERSONAJES DEL JUEGO

VISIÓN GENERAL

El personaje principal en este juego será un estratega. Es decir, nosotros mismos al elegir los distintos tipos de torres a utilizar en cada mapa. También seremos capaces de crear distintas torres y artefactos que utilizar luego.

Nuestro personaje carece de relevancia en el juego pues se enfoca más a las diferentes torres que trataremos como armas y a los personajes enemigos a abatir.

ENEMIGOS Y MONSTRUOS

Los enemigos o monstruos serán variados y de diferentes clases. Tendremos diferentes clases de monstruos dependiendo de su raza y tipo.

Los diferentes tipos de enemigos serán los siguientes:

- Básico: este será el enemigo básico, sin más resistencia que el resto ni habilidades especiales. Sus características dependerán únicamente de su raza.
- Ágil: este será un enemigo más rápido que el resto y para compensar tendrá una resistencia reducida respecto al tipo básico.
- Tanque: este enemigo será un poco más grande que el resto y con una resistencia superior a la media. Tendrá más debilidad a los daños por magia que a los físicos. También será más lento que el resto de unidades.
- Mago: este enemigo tendrá una vida y resistencia básica excepto para los daños especiales, a los cuales será más resistente. También podrá utilizar hechizos varios dependiendo de su raza para mejorar a sus compañeros.
- Jefe: este enemigo tendrá una resistencia superior a la media y podrá inspirar a sus compañeros haciendo que mejore su vida y velocidad. También hará que se generen enemigos extra mientras esté en el campo de batalla.

Las distintas razas de los enemigos serán las siguientes y pueden ofrecer distintas habilidades.

- Bárbaros: tendrán una vida y una resistencia media para todos los daños. La habilidad de sus magos curará a las criaturas cercanas.

- Orcos: tendrán una vida y resistencia superior. Tendrán una resistencia un poco más reducida en daño ligero que en pesado. La habilidad de sus magos aumentará todas sus resistencias durante un tiempo limitado.
- Elfos: tendrán una vida muy superior a la media pero una resistencia normal. Tienen una ligera posibilidad de esquivar daños físicos. La habilidad de sus magos los volverá invisibles durante un tiempo limitado (pero se les puede dañar igualmente).
- Bestias aladas: tendrán una vida y resistencia menor a la media. Tienen la habilidad de volar lo que evita posibles daños por explosiones. La habilidad de sus magos las hace más resistentes a las magias.
- Híbridos humanos/bestia: tendrán una resistencia superior a la media y serán más rápidos. Cuando su vida sea baja correrán aún más rápido. La habilidad de sus magos hace que su vida se regenere durante un tiempo limitado.
- Enanos: tendrán una resistencia superior a la media pero serán más lento. Cuando sean dañados por primera vez se volverán inmunes a todo daño y excavaran para avanzar por debajo del escenario durante un tiempo limitado. La habilidad de sus magos los hace más resistentes a la magia.
- Muertos: tendrán una vida menor a la media. Cuando mueren tienen una posibilidad de volver a la vida de nuevo. La habilidad de sus magos hace que todos los muertos cercanos resuciten, sean muertos o no.
- Elementales: tendrán una vida normal y unas resistencias más extremas a los elementos basados en su tipo de elemento. La habilidad de sus magos hará que sean totalmente inmunes al daño elemental en el que estén basados.

INTERFAZ DE USUARIO

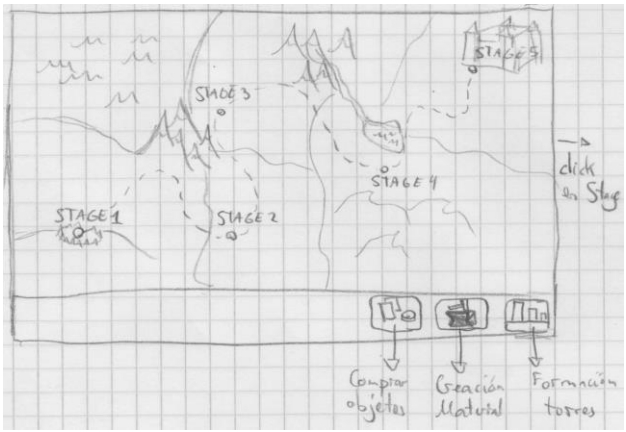
VISIÓN GENERAL

A continuación se muestran diferentes bocetos de la interfaz de usuario y las funcionalidades aportadas por el juego:

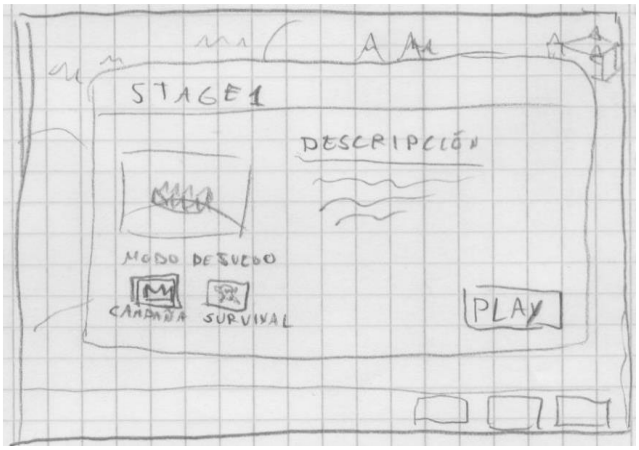
DETALLE DE INTERFAZ DE USUARIO #1 – PANTALLA DE INICIO

DETALLE DE INTERFAZ DE USUARIO #2 – SELECCIONAR PARTIDA

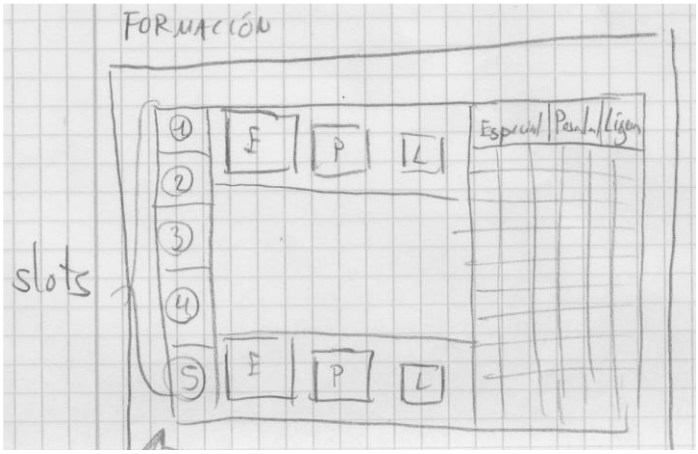
DETALLE DE INTERFAZ DE USUARIO #3 – SELECCIONAR ESCENARIO



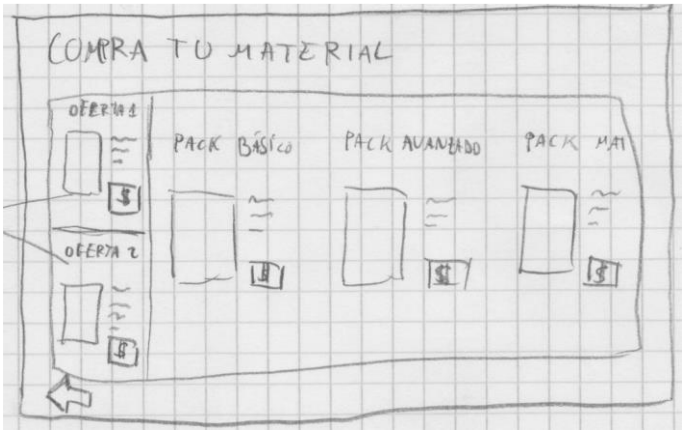
DETALLE DE INTERFAZ DE USUARIO #4 – DESCRIPCIÓN Y SELECCIÓN DE MODO



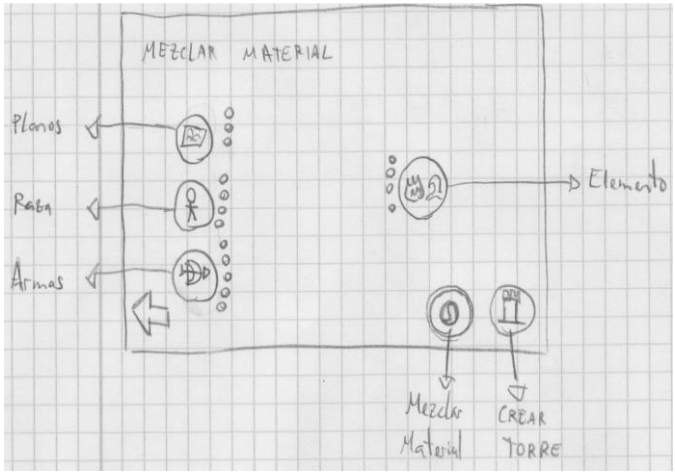
DETALLE DE INTERFAZ DE USUARIO #5 – SELECCIÓN DE LA FORMACIÓN DE DEFENSA



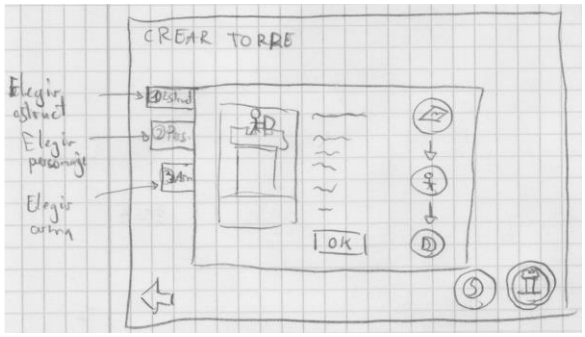
DETALLE DE INTERFAZ DE USUARIO #6 – COMPRAS DE MATERIAL Y OFERTAS



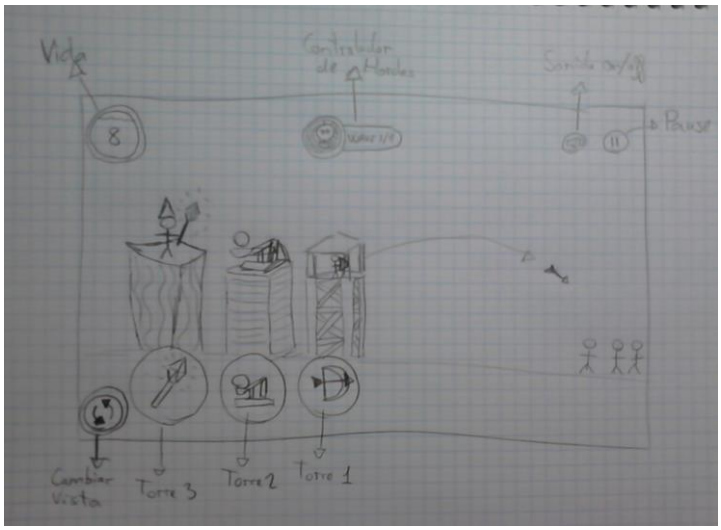
DETALLE DE INTERFAZ DE USUARIO #7 – EDITOR DE ARMAS



DETALLE DE INTERFAZ DE USUARIO #8 – CREACIÓN Y MEZCLA DE TORRES POR PASOS



DETALLE DE INTERFAZ DE USUARIO #9 – INTERFAZ GAMEPLAY



DETALLE DE INTERFAZ DE USUARIO #10 – INTERFAZ TORRE

ARMAS

VISIÓN GENERAL

Nuestras armas serán las diferentes torres que manejemos. Y podrán ser de 3 tipos: especial, pesadas y ligeras.

Las torres especiales tendrán el mayor tiempo de carga entre disparos de las tres. Serán las torres más grandes y utilizarán proyectiles especiales con diferentes efectos. Estás serán las torres que más diferencias tendrán entre ellas mismas. Sus trayectorias serán muy dependientes de la torre y podrán ser desde un barrido por toda la pantalla a una zona concreta. Por ejemplo en esta categoría podrían entrar dragones o magos que lancen como proyectil fuego o hechizos focalizados.

Las torres pesadas tendrán un tiempo de recarga menos que las especiales y serán un poco más pequeñas. Los proyectiles utilizados por estas torres tendrán un mayor peso y mucha más resistencia al viento con lo que tendrán trayectorias físicas más robustas respecto al viento. Por ejemplo en esta categoría entrarían catapultas, trabucos o cañones.

Las torres ligeras tendrán el tiempo de recarga más corto y serán más pequeñas que las anteriores. Los proyectiles utilizados por estas torres tendrán menor peso y serán muy susceptibles al viento. Por ejemplo en esta categoría entrarían arqueros o lanceros.

DETALLES DE ARMAS #1

Cada torre se divide en 3 partes:

- Estructura: La estructura definirá el tipo de torre, siguiendo los 3 tipos explicados anteriormente.
- Personaje: Según el tipo de torre, podremos elegir un personaje u otro, que se encargarán de disparar los proyectiles.
- Arma: Por medio del arma se definirá el tipo de trayectoria, proyectil, etc.

Cada parte podrá combinarse con los diferentes elementos para crear muchos y variados tipos de torres únicos, añadiendo distintos porcentajes y tipos de daños.

Como ejemplo se describirán algunas posibles combinaciones:

Planos	Elemento	Resultado
Grande	Madera	Torreón de madera
Mediano	Metal	Torre de metal
Pequeño	Madera	Torreta de madera pequeña

Raza	Elemento	Resultado
Humano	Energia	Mago
Humano	Tierra	Enano
Enano	Metal	Golem

Armas	Elemento	Resultado
Arco	Fuego	Arco de fuego
Catapulta	Hielo	Catapulta de hielo
Trabuco	Veneno	Trabuco envenenado

PARTITURAS MUSICALES Y EFECTOS DE SONIDO

VISIÓN GENERAL

La banda sonora del juego y los efectos de sonido serán sencillos, ya que en este juego queremos hacer un especial enfoque a mejorar la jugabilidad con respecto a otros juegos del mismo tipo. Cada parte del juego (ejemplo, UI de selección de modos de juego, niveles, etc…) tendrá una banda sonora diferente, así como cada acción dentro del modo de juego tendrá un efecto de sonido que haga que el usuario pueda meterse de pleno en el *gameplay*.

DISEÑO DEL SONIDO

Salvo necesidad, se utilizarán principalmente bancos de sonidos ya creados y bandas sonoras compuestas y disponibles para su uso. Se utilizará las bandas sonoras y música de fondo de esta manera:

- Una específica para la pantalla de inicio y selección de modos de juego.
- Cada nivel tendrá su propia música de fondo, asociada a la batalla. Si la duración es muy larga podrían incluirse hasta dos piezas musicales diferentes en el mismo nivel.
- El editor de armas también tendrá una pista musical específica. Al poderse editar armas tras obtener las puntuaciones, compartirá música de fondo con la pantalla de puntuaciones tras vencer un nivel.
- Las introducciones históricas también tendrán su propio fondo musical.

Con respecto a los efectos de sonido:

- Los monstruos y enemigos irán agrupados por tipos, y cada tipo tendrá su propio efecto sonoro cuando sea golpeado.
- Al inicio de cada ronda, efectos sonoros propios de la llegada de un ejército y el comienzo de una batalla tendrán lugar.
- Durante una batalla, se reproducirán sonidos de fondo comunes en una batalla (gritos, choques de espadas, etc…)
- Cada disparo de un arma del usuario tendrá también su propio sonido, agrupados como los enemigos.
- Al vencer un nivel, se reproducirá un efecto sonoro correspondiente.

Con respecto a la interfaz:

- Todos los botones tendrán un sonido leve que sonara cada vez que se pase por este. De esta forma sabremos que toda interfaz que haga sonido al pasar será un elemento interactivo.
- Los elementos del editor tendrán sonidos para señalar que la mezcla ha salido bien o mal. Estos sonidos serán distintos e identificativos.
- Cada vez que se seleccione un elemento a mezclar se generará un sonido distinto para dejar claro que se ha seleccionado un elemento.

JUEGO PARA UN JUGADOR

VISIÓN GENERAL

El juego está totalmente enfocado al disfrute de un solo jugador, ya que es el principal potencial de este tipo de juegos. La experiencia del usuario se basará en jugar por niveles, donde cada nivel tendrá un escenario en el cual habrá un elemento a defender, diferente en cada uno.

Cada nivel tendrá diferentes rondas. Después de cada ronda el usuario podrá, mediante la puntuación conseguida, retocar sus defensas para prepararse contra la siguiente oleada de enemigos que aparecerá en la ronda próxima hasta que llegue la última ronda, si la hay.

Así pues, el usuario tendrá que situar los diferentes elementos de defensa antes de cada ronda de entre las diferentes opciones que tiene disponibles para ese escenario, y colocarlos en función de las posiciones para cada tipo de elemento y como quiera defender su territorio.

DETALLES DEL JUEGO PARA UN JUGADOR #1

El principal modo de juego será el de campaña. Tras seleccionar la opción de jugar, los diferentes escenarios irán sucediéndose como una historia. Conforme avance en la campaña se irán desbloqueando los diferentes escenarios de los que dispone el juego, pero será necesario que el jugador los supere completamente.

Se le proporcionará al usuario un conjunto de armas básicas de inicio con las cuales irá interactuando para superar los diferentes niveles, consiguiendo nuevas a lo largo del modo de juego de diferentes formas.

DETALLES DEL JUEGO PARA UN JUGADOR #2

Tras haber jugado la campaña y desbloqueado los escenarios, el jugador podrá elegir jugarlos por separado en un modo survival. Este modo consistirá en ir superando rondas sin un límite, en la que cada ronda será más difícil que el anterior hasta que sea derrotado. De este modo podrá jugar hasta la saciedad su escenario favorito sin necesidad de pasar antes por el resto de escenarios anteriores.

DETALLES DEL JUEGO PARA UN JUGADOR #3

En los dos modos anteriores habrá otra característica en común, el editor de armas. El jugador puede conseguir armas completas mediante compra o bien mediante la combinación de elementos separados que también podrán conseguirse y comprarse. De este modo puede disfrutar de una gran combinación de armas diferentes, utilizando las más convenientes en cada escenario. Dichas armas y sus combinaciones están explicadas en el apartado de armas de este mismo documento.

HISTORIA

El jugador se pondrá en la piel del mejor estratega de la Alianza de los Doce, una unión de doce reinos humanos, elfos y enanos que antaño se unieron para ganar la Guerra de los 100 Siglos que acabó con los conflictos entre la alianza y el ejército de las bestias provenientes de una isla más allá del mar que los humanos llaman Fin de La Costa, trayendo prosperidad y paz a los reinos que conformaban la alianza. Sin embargo con el tiempo, tras no obtener beneficios por la guerra, la corrupción y la codicia de algunos líderes empeoró la economía y las relaciones entre la unión.

En el momento más crítico, a punto de la disolución de la Alianza de los Doce y una posible guerra civil, un nuevo líder aparece entre las bestias y lidera un ataque a la Península con el fin de destruir todo a su paso y sumir en la desesperación a todas la razas. Debido al miedo y a la influencia del nuevo líder, algunos reinos de las diferentes razas caen rendidos a su poder y traicionan a la alianza, enfrentándose a aquellos que no han sucumbido a las tinieblas y que hacen todo lo posible por resistir. Cuando todo parece perdido, en el ataque a un pequeño pueblo con talento para la creación de armas, el jugador liderará la resistencia, comenzando con una campaña de reconquista que le convertirá en el mejor general de la historia, consiguiendo de nuevo el apoyo de todos los reinos y devolviendo de nuevo la esperanza y la prosperidad a la Alianza de los Doce.

HORAS DE JUEGO

El modo campaña durará aproximadamente 3 horas, teniendo en cuenta el número de escenarios y la duración de cada ronda y las oleadas de enemigos de cada una. Puede alargarse o acortarse según el dominio del usuario, su conocimiento acerca de las armas y el tiempo que dedique a obtener nuevas combinaciones.

Sin embargo puede dedicarle muchas más horas de juego mediante el modo survival. En concreto, las que el usuario quiera.

CONDICIONES DE VICTORIA

El jugador obtendrá la victoria cuando haya completado todos los niveles del modo de campaña. Una vez obtenida la victoria tendrá total acceso a los escenarios y las armas disponibles, pudiendo repetir el modo campaña o jugar cada nivel por separado en el modo survival.

RENDERIZADO DE PERSONAJES

VISIÓN GENERAL

A continuación se mostrarán los modelos elegidos para representar las clases y enemigos descritos anteriormente.

RENDERIZADO: HUMANO

RENDERIZADO: GOBLIN

RENDERIZADO: TROL

RENDERIZADO: TORRE DE MADERA

RENDERIZADO: ZOMBIE

RENDERIZADO: DEMONIO DE HIELO

RENDERIZADO: DEMONIO DE FUEGO

RENDERIZADO: DEMONIO DE TIERRA

RENDERIZADO: DEMONIO OSCURO

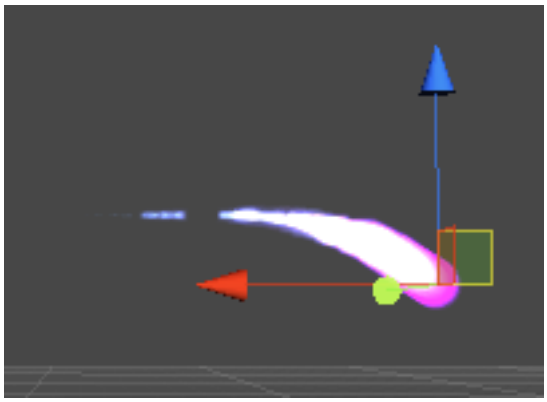
RENDERIZADO: GOLEM DE HIELO

EFECTOS ELEMENTALES

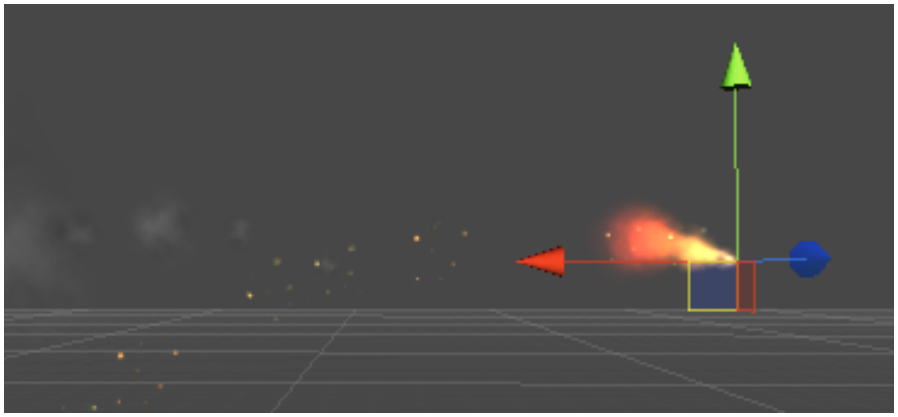
VISIÓN GENERAL

Aquí se podrán ver los diferentes efectos que tiene un arma al ser embebida con un elemento.

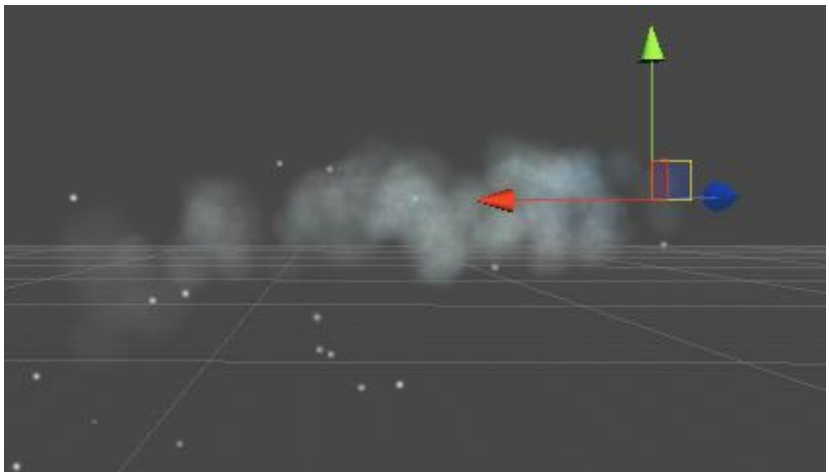
EFECTO POR DEFECTO



EFECTO DE FUEGO

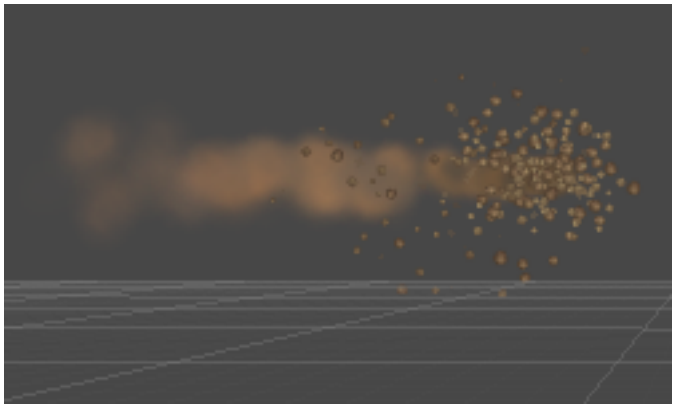


EFECTO DE HIELO

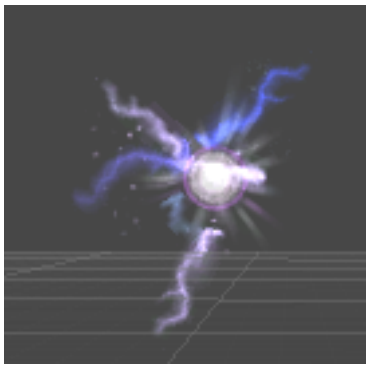


EFECTO DE VIENTO

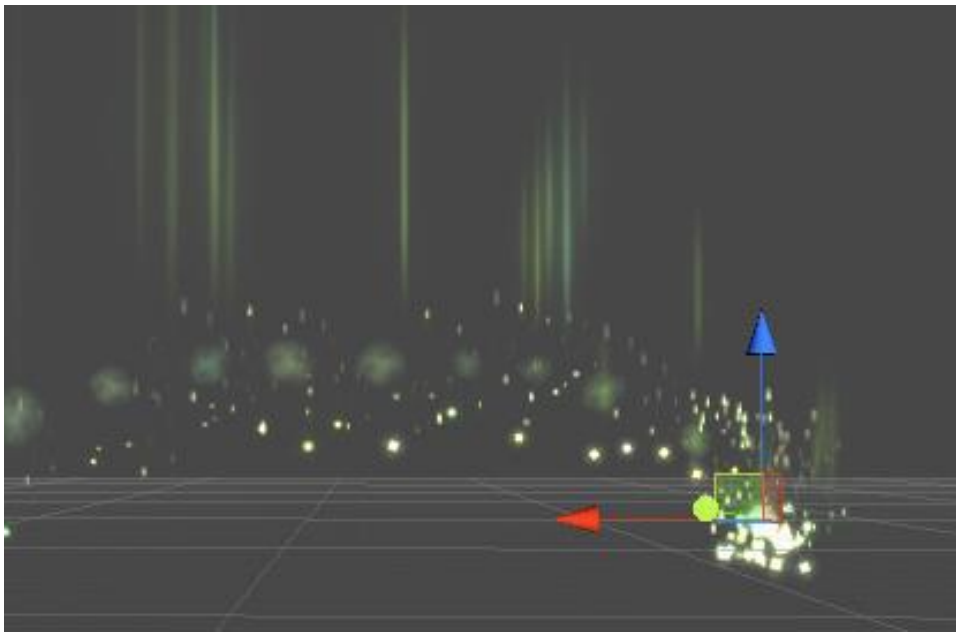
EFECTO DE TIERRA



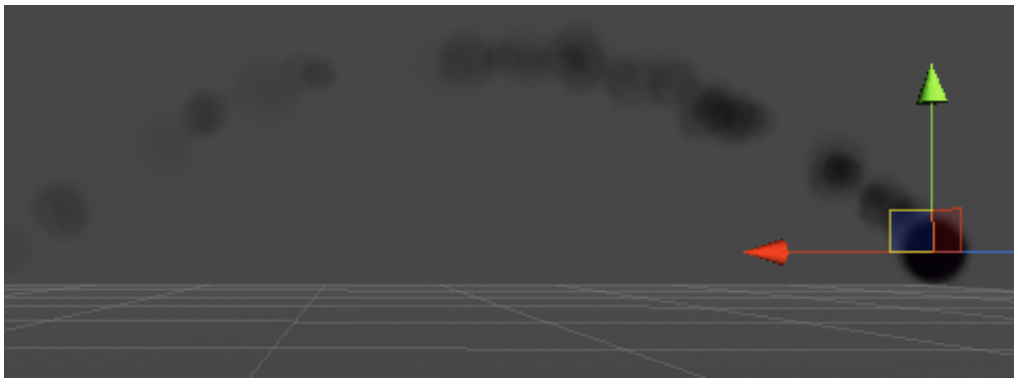
EFECTO DE ELECTRICIDAD



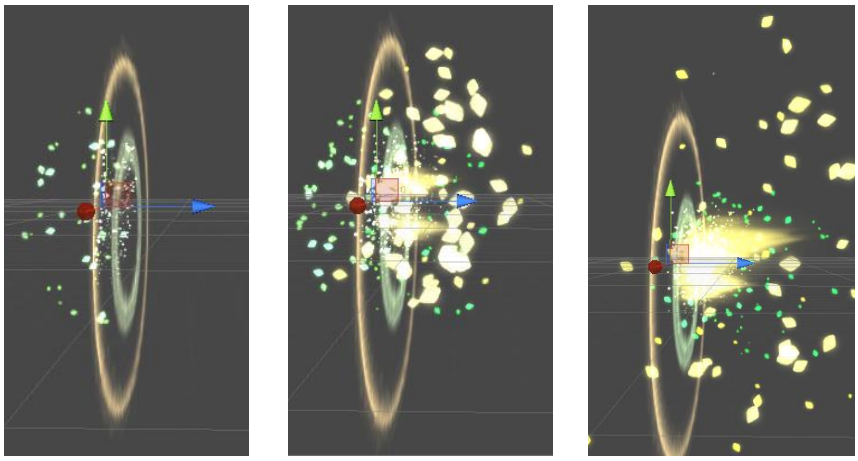
EFECTO DE LUZ



EFECTO DE OSCURIDAD



EFECTO DE BARRERA



COMBINACIÓN DE ELEMENTOS

Humano + Oscuridad = Goblin (disponible en esta versión)
Humano + Viento = Elfo
Humano + Tierra = Enano
Humano + Luz = Angel
Humano + Electricidad = Mago humano
Elfo + Viento = Mago Elfo
Enano + Tierra = Mago enano

Arco + Fuego = Arco de fuego (disponible en esta versión)
Arco + Hielo = Arco de hielo (disponible en esta versión)
Arco + Viento = Arco de viento (disponible en esta versión)
Arco + Tierra = Arco de tierra (disponible en esta versión)
Arco + Electricidad = Arco de electricidad (disponible en esta versión)
Arco + Luz = Arco de luz (disponible en esta versión)
Arco + Oscuridad = Arco de oscuridad (disponible en esta versión)
Lanza + Fuego = Lanza de fuego
Lanza + Hielo = Lanza de hielo
Lanza + Viento = Lanza de viento
Lanza + Tierra = Lanza de tierra
Lanza + Electricidad = Lanza de electricidad
Lanza + Luz = Lanza de luz
Lanza + Oscuridad = Lanza de oscuridad

Plano pequeño + Madera = Estructura de madera pequeña (disponible en esta versión)
Plano pequeño + Metal = Estructura pequeña de metal
Plano pequeño + Piedra = Estructura pequeña de Piedra
Plano mediano + Madera = Estructura mediana de madera
Plano mediano + Metal = Estructura mediana de metal
Plano mediano + Piedra = Estructura mediana de Piedra
Plano grande + Madera = Estructura grande de madera
Plano grande + Metal = Estructura grande de metal
Plano grande + Piedra = Estructura grande de Piedra

RENDERIZADO DE SPRITES

ARMAS

 Arco de madera

 Arco de fuego

 Arco de hielo

 Arco de luz

 Arco de oscuridad

 Arco de tierra

 Arco de viento

 Arco eléctrico

RAZAS


Goblin


Humano

PLANOS


Plano pequeño

ELEMENTOS


Electricidad


Fuego


Hielo


Luz

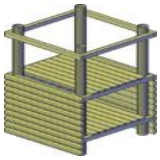

Madera


Oscuridad


Tierra


Aire

ESTRUCTURAS


Estructura de madera pequeña

BOTONES Y ELEMENTOS DE MENÚ


Volver al menú principal


Tienda de objetos


Mezclador de objetos

 Editor de torres

 Borrar partida

 Créditos

TORRES

 Torre de madera

 Plantilla de torre. Mediante *labels* se especificaba tamaño, raza y arma.