Universitat Autònoma
de Barcelona

IceFire

videojuego online
para Xbox 360

Memoria del proyecto

de Ingeniería Técnica en

Informática de Gestión

Realizado por

Josep Rius López

Y dirigido por

Xavier Verge

Escola d'Enginyeria
Sabadell, Septiembre de 2011

El sotasignat, **Xavier Verge Mestre**, professor de l'Escola d'Enginyeria de la UAB,

CERTIFICA:

Que el treball al que correspon la present memòria ha estat realitzat sota la seva direcció per en **Josep Rius López**

I per a que consti firma la present.

Sabadell, Setembre de 2011

Signat: Xavier Verge Mestre

FULL DE RESUM – PROJECTE FI DE CARRERA DE L'ESCOLA D'ENGINYERIA

Títol del projecte: IceFire Videojuego para Xbox 360	
Autor: Josep Rius López	**Data:** Setembre 2011
Tutor: Xavier Verge Mestre	
Titulació: Enginyeria Tècnica d' Informàtica de Gestió	

Paraules clau

• **Català**:
Videojoc, XNA, Microsoft Xbox 360, MOBA, Internet, Xbox LIVE, usuari, jugador, host, clients, servidors.

• **Castellà:**
Videojuego, XNA, Microsoft, Xbox 360, MOBA, Internet, XBOX LIVE, usuario, jugador, host, clientes, servidores.

• **Anglès:**
Videogame, XNA, Microsoft Xbox 360, MOBA, Internet, Xbox LIVE, user, player, host, clients, servers.

Resum del projecte

• **Català:**
El projecte consisteix en la realització d' un joc online per la consola de Microsoft Xbox 360 titulat IceFire.
El videojoc pertany al génere dels MOBA i permetrà als usuaris jugar de forma offline al mode de práctica pero també en un mode online crear o unirse a salas de joc per començar partides on podran seleccionar entre dos personatges diferents amb les seves habilitats propies i enfrontar-se amb el seu rival.

• **Castellà:**
El proyecto consiste en la realización de un juego online para la consola de Microsoft Xbox 360 titulado IceFire.
El videojuego pertenece al género de los MOBA y permitirá a los usuarios jugar de forma offline en el modo práctica pero tambien en un modo online donde crear o unirse a salas de juego donde empezar partidas en las que podrán elegir entre dos personajes distintos con sus habilidades propias y enfrentarse con su rival.

• **Anglès:**
This project involves the creation of a online game to Microsofts's consola Xbox 360 called IceFire.
This videogame is a MOBA and the users will be available to play offline with the practice mode or they can use the online mode where they can create or join the lobby and start games with options to choice two diferent characters with their typical abilities and play versus other players.

Contenido

1 INTRODUCCIÓN 9
1.1 Introducción al Proyecto 9
1.2 Estructura de la Memoria 11
2 ESTUDIO DE VIABILIDAD 12
2.1 CAPACIDAD DEL SISTEMA 12
2.1.1 Tipología y palabras clave 12
2.1.2 Descripción 12
2.1.3 Objetivos del proyecto. 12
2.1.4 Prioridades de los objetivos del proyecto 13
2.1.5 Definiciones, acrónimos y abreviaciones 13
2.2 Requerimientos del proyecto 15
2.2.1 Requerimientos Funcionales 15
2.2.2 Requerimientos No Funcionales 16
2.2.3 Prioridades de los requisitos funcionales 16
2.2.4 Prioridades de los requisitos no funcionales 16
2.2.5 Relación entre los requisitos y objetivos 16
2.3 Partes interesadas 17
2.3.1 Stakeholders 17
2.3.2 Perfiles de usuarios 17
2.3.3 Equipo del proyecto 17
2.4 Restricciones del sistema 18
2.5 Alternativas del sistema y selección de la solución 18
2.5.1 Alternativa 0 Juego offline Xbox 360 18
2.5.2 Alternativa 1 Juego Online Xbox 360 con servidor principal y clientes 19
2.5.3 Alternativa 2 Juego Online PC cliente a cliente usando conexión Visual Studio 19
2.5.4 Alternativa 3 Juego Online cliente a cliente para PC y Xbox 360 usando LIVE 20
2.5.5 Alternativa 4 Juego Online server a cliente para PC y Xbox 360 usando LIVE 20
2.5.6 Alternativa 5 Juego Online cliente a cliente para PC y Xbox 360 LIVE y XNA 4.0 21
2.5.7 Solución propuesta 21
2.6 Riesgos 22
2.6.1 Evaluación de riesgos 22
2.6.2 Catalogación de riesgos 23
2.6.3 Plan de contingencia 23
2.7 Presupuesto 23
2.8 Planificación del proyecto 24
2.9 Conclusiones del estudio de Viabilidad 26

3 FUNDAMENTOS TEÓRICOS 28
3.1 XNA 28
3.2 NET FRAMEWORK 32
3.3 XNA Network Protocol Specication 33
3.4 Xbox 360 37
3.5 Otros fundamentos teóricos 38
4 ANÁLISIS DEL JUEGO 39
4.1 Mecánica del juego 39
4.2 DISEÑO GRÁFICO 41
4.2.1 Introducción 41
4.2.2 Sprites y Spritesheets 42
4.2.3 Mapeado 43
4.2.4 Interfaz gráfica 44
4.2.5 Diseño gráfico del conjunto 46
5 IMPLEMENTACIÓN DEL PROYECTO 47
5.1 Introducción 47
5.2 Clases generales 48
5.3 Clases del juego 57
5.4 Conexión Online 62
5.4.1 .Implementación 62
5.4.2 Algoritmos de predicción 69
5.5 Implementación de Gráficos 69
5.5.1 Introducción 69
5.5.2 TexturaAnimada 70
5.5.3 Camara2D 71
5.5.4 Scrolling Background 71
5.6 Implementación del Sonido 72
5.6.1 Introducción 72
5.6.2 MediaPlayer 72
5.6.3 XTC 73
5.6.4 Reproducción de sonido 73
5.6.5 Elección de música y efectos sonoros 73
5.7 Colisiones entre Sprites 73
5.7.1 Introducción 73
5.7.2 Intersects 74
5.7.3 Colisiones mediante Path 74
5.7.4 Opciones alternativas 75

5.8 Timers 76
6 PRUEBAS BETA TESTING 78
6.1 Introducción 78
6.2 Etapas 78
6.2.1 Funcionalidades Offline 78
6.2.2 Funcionalidades Online 78
6.2.3 Testeo exhaustivo Online 79
6.2.4 Testeo con latencias reales 79
7 CONCLUSIONES 80
7.1 Consecución de objetivos 80
7.2 Desviaciones sobre la planificación 81
7.3 Líneas de ampliación 84
7.4 Comercialización del videojuego 86
BIBLIOGRAFÍA Y LINKS: 88
ANEXO 1: GUÍA RÁPIDA DEL JUEGO 91

Índice de Figuras

Figura 1 Imágen del clásico DUNE 2 10
Figura 2 Imágen del MOBA de éxito League of Legends 10
Figura 3 Imágen de Bloodline Champions 10
Figura 4 Bucle de Conexión 35
Figura 5 Funcionamiento del In Game Update 36
Figura 6 Xbox 360 Arcade 38
Figura 7 Xbox 360 Elite 38
Figura 8 Firewind caminando 42
Figura 9 Firewind quieta 42
Figura 10 Firewind herida 42
Figura 11 Firewind cayendo 42
Figura 12 Iced caminando 43
Figura 13 Iced quieto 43
Figura 14 Iced herido 43
Figura 15 Iced cayendo 43
Figura 16 Torre Fire 43
Figura 17 Torre Ice 43
Figura 18 Base Fire 43
Figura 19 Base Ice 43
Figura 20 Mapeado 44
Figura 21 Portada IceFire 45
Figura 22 Selección de Firewind 45
Figura 23 Selección de Iced 45
Figura 24 Retrato Iced 46
Figura 25 Retrato Firewind 46
Figura 26 Icono de habilidad principal de Iced 46
Figura 27 Icono de habilidad secundaria de Iced 46
Figura 28 Icono de habilidad principal de Firewind 46
Figura 29 Icono de habilidad secundaria de Firewind 46
Figura 30 Pantalla de juego 46
Figura 31 Diagrama 1 de clases de IceFire 49
Figura 32 Diagrama 2 de clases de IceFire 49
Figura 33 Pantalla de Selección de Héroes 50
Figura 34 Pantalla de Lobby 50
Figura 35 Diagrama 3 de clases IceFire 51
Figura 36 Pantalla de sala creada 52
Figura 37 Seleccionamos Sala 52
Figura 38 Entramos en Sala 52
Figura 39 Pantalla de Selección Inicial 53
Figura 40 Pantalla de Selección de Opciones 53
Figura 41 Pantalla de Pausa 53
Figura 42 Pantalla de Victoria 54
Figura 43 Diagrama 4 de clases IceFire 54
Figura 44 Diagrama 5 de clases de IceFire 57
Figura 45 Mapeado externo de IceFire 75
Figura 46 Trazado Interno de IceFire 75

1 INTRODUCCIÓN

1.1 Introducción al Proyecto

Cuando un alumno decide realizar su proyecto final de carrera, una de las cosas que sin duda busca es que le atraiga el proyecto en algún sentido, todo el mundo sabemos que uno trabaja más y pone más énfasis en proyectos que le atraigan de alguna manera.

Los videojuegos es sin duda un área que atrae a bastantes estudiantes de Informática y realizar un proyecto de este estilo se plantea en muchas ocasiones como una opción, el problema es el desconocimiento en cómo empezar.

En ese sentido la aparición de XNA ha supuesto una entrada de aire fresco para los que quieran realizar un videojuego para diferentes plataformas como PC, Windows Phone, Zune o la propia Xbox 360. ¿Pero qué es XNA? Aunque parezca un acrónimo no lo es, XNA es un conjunto de herramientas con un entorno de ejecución administrado proporcionado por Microsoft que facilita el desarrollo de juegos de ordenador posibilitando también herramientas para trabajar en su consola XBOX 360. Básicamente estamos hablando de una API para desarrollo de juegos para Visual Studio.

Por tanto al inicio del proyecto existían diversas posibilidades, básicamente se podían reducir a desarrollar un juego online para PC, para PS3 o para Xbox 360. La opción para PS3 estaba en parte descartada como se explicará en el Estudio de Viabilidad pero la gran cantidad de información y facilidades dadas por Microsoft y por la propia comunidad de XNA hacía de esa propuesta sin duda la más atrayente.

Además al tener la posibilidad de conocer otro proyecto realizado para Xbox 360 daba la seguridad de que era un proyecto factible, ahora había que aportar una dificultad nueva y encarar nuevos retos. Prácticamente todos los juegos que aparecen actualmente son desarrollados con miras a ser jugados en red vía Internet, con lo cual el reto estaba ahí, realizar un juego para consola Online con todas las dificultades que ello entrañaba.

Ahora bien, las dificultades para realizar un juego Online son varias, inicialmente desconocimiento del lenguaje y de cómo realizar la aplicación, por tanto se tenía que plantear un proyecto en el aspecto jugable sencillo, siendo los juegos MOBA un tipo de juegos en expansión actualmente una de las opciones más firmes.

Para los profanos del mundo videojuego hablar de beat'em up, shoot'em up, MOBA o RTS seguramente les sonará extraño, básicamente estamos hablando de géneros de videojuegos.

Al realizar un juego online hace falta un planteamiento, un diseño inicial de cómo va a funcionar, al encararlo al modo online un aspecto básico es la competición entre jugador contra jugador, por tanto necesitamos un género de videojuego que potencie esto.

Los RTS acrónimo de Real Time Strategy o que es lo mismo estrategia en tiempo real son un tipo de juegos que suelen jugarse jugador contra jugador, ejemplos clásicos de este tipo de juegos son Dune 2, Command and Conquer, Warcraft y Age of Empires. En la mayoría de este tipo de juegos hablamos de control de tropas, creación de edificios, exploración de mapeado, etc. Con lo cual no son precisamente juegos sencillos de realizar con un tiempo limitado de un año.

Figura 1 Imágen del clásico DUNE 2

Pero últimamente a partir de un mod(un juego modificado por los jugadores) del Warcraft 3 conocido como Dota(Defense of the Ancients) ha aparecido un nuevo género de estrategia llamado MOBA, básicamente consiste en el control de un héroe que lucha contra otro héroe enemigo intentando destruir su base. Estos personajes controlados por el jugador tienen diferentes tipos de habilidades y el que hayan varios tipos de héroes disponibles posibilita nuevas estrategias o formas de jugar distintas.

Se trata de un genero actualmente en expansión y con un gran éxito sobretodo en la plataforma de PC, títulos como League of Legends o Heroes of NewEarth son ejemplos claros de dos videojuegos con éxito aparecidos recientemente.

Figura 2 Imágen del MOBA de éxito League of Legends

Caso aparte es el de un titulo llamado Bloodline Champions, realizado con XNA para PC que se ha acabado convirtiendo en un título comercial siendo en sus inicios un proyecto final de carrera.

El caso es que en consola este género aun no ha eclosionado, en parte porque se suele jugar con ratón y teclado y parece como que el mando no es tan cómodo para este tipo de juegos o quizás también por razones de target de público. Sea como sea el realizar un MOBA online para Xbox 360 no es algo habitual ahora mismo.

Figura 3 Imágen de Bloodline Champions

Por tanto la opción de los MOBA para un juego es muy buena debido a la aceptación actual de este género entre los jugadores y supone una novedad en el ámbito de consolas.

Estas son algunas de las principales razones que me han llevado a iniciar este proyecto de un videojuego Online para Xbox 360.

1.2 Estructura de la Memoria

La Memoria está dividida en varias partes, algunas de ellas más habituales y otras más inusuales debido a las particularidades del proyecto pero, en todo caso, necesarias para una explicación detallada.

En una primera parte tendremos el Estudio de Viabilidad donde se hablará de las alternativas estudiadas y las razones para elegir hacer un título para Xbox 360 con XNA. Mención especial al apartado de la planificación del proyecto basada en prototipos en las que se observará el recorrido desde las versiones Alpha hasta la llegada de la Beta.

A continuación tendremos la parte de conocimientos previos, debido a la amplitud de esta parte se tratan varios temas desde el protocolo de conexión a internet de XNA hasta las características de la máquina en la que trabajaremos.

El apartado de análisis del juego nos explicará el diseño de la mecánica del juego, las razones de algunas decisiones y cuál es el concepto del juego que se está desarrollando.

Seguidamente tendremos el apartado del diseño gráfico en el que se verá el trabajo realizado para la creación de sprites, sprite sheets, interfaz gráfica y todo lo relacionado con la parte artística del videojuego.

En la próxima etapa tendremos todo lo relacionado con el diseño de clases y la codificación e implementación de éstas, estará dividido en dos partes, una primera donde se hablará de ellas de forma general y se verá el diagrama de clases correspondiente, y una segunda parte donde tendremos diferentes apartados y se explicará de forma concreta partes como los gráficos, sonido y colisión de sprites.

La sección de pruebas de beta testing estará dedicada a como se realizaron los tests del juego tanto de forma offline como online y explicará las fases que tuvo.

Finalmente tendremos las conclusiones, líneas de ampliación del producto y el apartado de comercialización del videojuego donde nos hablará de cómo vender nuestro producto con las vías proporcionadas por XNA y la compañía Microsoft.

La bibliografía utilizada corresponde en su totalidad a una enlaces online que serán debidamente documentados.

Como anexos tendremos una guía rápida del manual de juego en el que se explicará cómo son los controles del videojuego tanto en su versión de PC como de consola.

Son siete Capítulos diferenciados más los anexos para poder explicar desde la creación del concepto al desarrollo del proyecto.

2 ESTUDIO DE VIABILIDAD

A continuación se presenta el Estudio de Viabilidad del proyecto IceFire videojuego para Xbox 360.

2.1 CAPACIDAD DEL SISTEMA

2.1.1 Tipología y palabras clave

Tipología: Proyecto de Desarrollo.
Palabras clave: videojuego, XNA, Microsoft, Xbox 360, MOBA, internet, XBOX LIVE, usuario, jugador, host, clientes, servidores.

2.1.2 Descripción

Análisis, implementación y diseño de un videojuego online en Xbox 360.

La aplicación tendrá que ser realizada en Visual Studio C# y utilizando los protocolos y librerías establecidos por Microsoft para usarse en su máquina.

2.1.3 Objetivos del proyecto.

O1- Realizar juego offline de un jugador y dos jugadores que funcione correctamente.
O2- Realizar la conexión online y gestionarla correctamente.
O3- Realizar interfaz gráfica externa al juego, menús de juego, opciones, selección de salas para los jugadores.
O4- Creación de más de un personaje distinto: apariencia gráfica, habilidades y características.
O5- Añadir efectos sonoros al videojuego: música adecuada y efectos sonoros a las habilidades y eventos del juego.
O6- Creación de un mapeado adecuado para 1 vs 1 balanceado para ambos jugadores.
O7- Objetivo secundario: Añadir mecánicas extras al proyecto, añadir NPCs de cada bando,
O8- Objetivo secundario: Realización de la interfaz gráfica del juego con barras de vida, retratos de avatar, mini mapa, stats de los jugadores.
O9- Objetivo secundario: creación de contenido extra: más de un mapa disponible, más de un personaje elegible, diferentes modos 1vs1, 2vs2, 3vs3.
O10- Eliminar todos los posibles bugs del programa y que funcione correctamente.

2.1.4 Prioridades de los objetivos del proyecto

Objetivo	Crítico	Prioritario	Secundario
O1	X		
O2	X		
O3	X		
O4		X	
O5		X	
O6		X	
O7			X
O8			X
O9			X
O10		X	

2.1.5 Definiciones, acrónimos y abreviaciones

UAB:	Universitat Autònoma de Barcelona.
Usuario:	En sentido general, un usuario es un conjunto de permisos y de recursos (o dispositivos) a los cuales se tiene acceso. Es decir, un usuario puede ser tanto una persona como una máquina, un programa, etc. En nuestro caso se trata de cualquier persona con acceso a nuestro programa, para acceder a la forma offline lo podrá hacer de forma anónima pero para acceder al online lo tendrá que hacer de forma registrada.
Videojuego:	Un videojuego o juego de vídeo es un software creado para el entretenimiento en general y basado en la interacción entre una o varias personas y un aparato electrónico que ejecuta dicho videojuego; este dispositivo electrónico puede ser un ordenador, una máquina arcade, una videoconsola, un dispositivo handheld (un teléfono móvil, por ejemplo) los cuales son conocidos como "plataformas. En nuestro caso la plataforma donde desarrollaremos el videojuego será la consola Xbox 360.
Internet:	La red de redes. Una definición formal sería: conjunto descentralizado de redes de comunicación interconectadas que utilizan la familia de protocolos TCP/IP.
Online:	Literalmente significaría en línea. Entre varios significados, en el mundo de los videojuegos se utiliza cuando el producto utiliza los servicios de internet para poder conectar dos usuarios.
Microsoft:	Compañía creadora del sistema operativo Windows. En el caso del proyecto que nos ocupa es importante por ser la creadora de la consola que vamos a utilizar la Xbox 360 y por crear la red de servicios LIVE.
XNA:	No es un acrónimo, se trata de una serie de librerías y de servicios creados por Microsoft para facilitar la creación de videojuegos en su consola Xbox 360. También permite la creación en otras plataformas como PC, Windows phone y Zune.

appHUB:	Página web oficial en la actualidad de Microsoft para dar soporta a XNA, se subdivide en dos secciones, la sección para móviles y la sección para consolas.
Xbox 360:	Consola creada por Microsoft con capacidad para manejar tanto gráficos 2D como 3D además de tener dispositivos multimedia como reproductor de películas y música con posibilidad de conexión a internet para los servicios LIVE.
Network LIVE:	Es el servicio de videojuegos en línea de Microsoft que da soporte a los videojuegos multijugador de las videoconsolas Xbox 360 y Xbox, además de las plataformas para el sistema operativo Microsoft Windows (Games for Windows - Live). El servicio "Silver" es gratuito y el "Gold" tiene un costo de suscripción. Los contenidos son pagados aparte. El servicio fue lanzado el 14 de marzo de 2002, obteniendo un gran éxito en su estreno. Actualmente agrupa cerca de 20 millones de usuarios inscriptos y está disponible en varios idiomas.
Host:	El término Host se usa en informática para referirse a los ordenadores conectados a una red que proveen y utilizan servicios de ella. En general, los hosts son máquinas que ofrecen servicios de transferencia de archivos, conexión remota, servidores de base de datos, servidores web, etc. En nuestro caso tendremos dos usuarios donde uno de ellos hará de Host al otro creando la sala y partida donde jugarán.
Cliente:	Un cliente de videojuego, es un programa usado para jugar un videojuego en línea y que se conecta a un servidor de videojuego.
Servidor:	Una aplicación informática o programa que realiza algunas tareas en beneficio de otras aplicaciones llamadas clientes. Algunos servicios habituales son los servicios de archivos, que permiten a los usuarios almacenar y acceder a los archivos de una computadora y los servicios de aplicaciones, que realizan tareas en beneficio directo del usuario final. Este es el significado original del término. Es posible que un ordenador cumpla simultáneamente las funciones de cliente y de servidor. En nuestro caso los servidores vendrán proporcionados por el servicio LIVE.
MOBA (Multiplayer Online Battle Arena):	Tipo de juego donde varios jugadores compiten entre sí en una zona de juego (arena) a partir de conexiones online entre ellos.
RTS (Real Time Strategy):	Palabra utilizada en los videojuegos para definir a los juegos de estrategia en tiempo real a diferencia de otros tipos de juegos estrategia como los juegos por turnos.
Shot'em up:	Palabra utilizada para definir a los videojuegos que se basan en disparos, normalmente suelen ser juegos de naves donde se dispara a los enemigos, un ejemplo muy conocido es el Space Invaders.
NPC (Non Player Character):	Literalmente "personaje no jugador", hace referencia en los videojuegos a todo personaje controlado por la máquina y donde ningún jugador tiene control.

Power Up:	Palabra utilizada en los videojuegos para definir los objetos que cogidos de alguna manera (normalmente pasando encima de ellos) mejoran las características de los personajes.
Stats:	Término utilizado cuando hablamos de las características de los personajes de los videojuegos, sobretodo en juegos de tipo de aventura-rol o RTS aunque en la actualidad prácticamente en todos los videojuegos tenemos stats de los elementos del videojuego.
Sprites:	Los sprites se trata de un tipo de mapa de bits dibujados mediante hardware gráfico sin cálculos adicionales de la CPU. A menudo son pequeños y parcialmente transparentes. Típicamente, los sprites son usados en videojuegos para crear los gráficos. Con el paso del tiempo el uso de este término se extendió a cualquier pequeño mapa de bits que se dibuje en la pantalla, incluso si tiene que moverlo todo el procesador central y no cuenta con hardware especializado.
Sprite Sheets:	Literalmente lo podríamos traducir como hoja de sprites, se trata de imágenes que carga un programa donde se encuentran animaciones de los sprites, posteriormente el programa se encarga de que en pantalla solo se muestre el frame correspondiente.
Frame:	Se denomina frame en inglés, a un fotograma o cuadro, una imagen particular dentro de una sucesión de imágenes que componen una animación. La continua sucesión de estos fotogramas producen a la vista la sensación de movimiento, fenómeno dado por las pequeñas diferencias que hay entre cada uno de ellos. En ese sentido en las animaciones cada dibujo de una animación completa sería un frame.

2.2 Requerimientos del proyecto

2.2.1 Requerimientos Funcionales

1. Mantenimiento del juego en XNA(inicialmente en XNA 3.1 posibles ampliaciones en XNA 4.0 o posteriores).
2. Creación de salas de juego.
3. Control de acceso de los usuarios a las salas de juego.
4. Mantenimiento de la conexión durante la partida.
5. Creación de Sprite Sheets para animaciones de personajes.
6. Creación de Sprites para habilidades de personajes y torres.
7. Creación de interfaz gráfica.
8. Creación de Sprites para torres.
9. Creación de Sprites para bases.
10. Creación/Asignación de banda sonora y efectos sonoros.
11. Buena interactividad en la partida por usuarios en ella (posibles problemas con lag).
12. Creación de personajes diferentes (aspecto físico, características, habilidades...).
13. Copias de seguridad del proyecto.

2.2.2 Requerimientos No Funcionales

1. Normas de LIVE de Microsoft
2. Normas de uso XNA
3. Normas de uso de recursos
4. Cumplimiento de la LOPD en lo referente a los datos y derechos de futuros usuarios/clientes del programa.

2.2.3 Prioridades de los requisitos funcionales

	RF 1	RF 2	RF 3	RF 4	RF 5	RF 6	RF 7	RF 8	RF 9	RF 10	RF 11	RF 12	RF 13
Esencial	X	X	X	X	X	X					X		X
Condicional								X	X	X			
Opcional							X					X	

2.2.4 Prioridades de los requisitos no funcionales

	RNF 1	RNF 2	RNF 3	RNF 4
Esencial	X	X	X	X
Condicional				
Opcional				

2.2.5 Relación entre los requisitos y objetivos

	RF 1	RF 2	RF 3	RF 4	RF 5	RF 6	RF 7	RF 8	RF 9	RF 10	RF 11	RF 12	RF 13	RNF 1	RNF 2	RNF 3	RNF 4
O1	X				X	X	X	X	X	X			X		X	X	X
O2	X	X	X	X							X		X	X	X	X	X
O3	X						X						X		X	X	X
O4	X				X	X						X	X		X	X	X
O5	X									X			X		X	X	X
O6	X							X	X				X		X	X	X
O7	X				X	X		X	X		X	X	X		X	X	X
O8	X						X						X		X	X	X
O9	X				X	X					X	X	X		X	X	X
O10	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

2.3 Partes interesadas

2.3.1 Stakeholders

Nombre	Descripción	Responsabilidad
Xavier Verge	Director del proyecto	Supervisa el trabajo del alumno.
Josep Rius López	Responsable del proyecto	Realiza el proyecto.

2.3.2 Perfiles de usuarios

Nombre	Perfil	Responsabilidad
Josep Rius	Administrador del sistema, programador principal. Usuario Beta Tester interno	Dirige y realiza las actividades de Tests offline y online del proyecto en versión de PC i X 360.
Ferran Deitx "Asmodeu"	Usuario Beta Tester externo	Ayuda en actividades de Tests offline y online del proyecto en versión de PC i X 360.
Jordi Sánchez "Yarek"	Usuario Beta Tester externo	Ayuda en actividades de Tests offline y online del proyecto en versión de PC i X 360.

2.3.3 Equipo del proyecto

Nombre	Descripción	Responsabilidad
Xavier Verge	Tutor del proyecto	Ayuda en la Definición, gestión, planificación del proyecto. Supervisa el trabajo del alumno.
Josep Rius	Analista/programador/técnico de pruebas	Realiza el programa: Diseño, metodología, especificaciones, elementos gráficos y sonoros... Desarrolla el videojuego con el análisis y planificación prevista. Participa en el diseño de las pruebas internas y externas. Realiza las pruebas y participa en el proceso de Beta test.

2.4 Restricciones del sistema

a) La aplicación se ha de implementar en un entorno Windows.
b) La aplicación se ha de implementar con Visual Studio C#.
c) La aplicación ha de estar preparada para usarse en Xbox 360 y PC.
d) El proyecto se ha de finalizar antes del día 30 de mayo de 2010.
e) El videojuego se ha de implementar usando los protocolos de Xbox Live. Su uso tiene sus ventajas y desventajas pero también sus limitaciones como se ve en el siguiente cuadro:

TABLA 1: Usos de Live	Xbox 360 consola	Desarrollo de juego para Windows	Windows Phone
Ejecutar un juego con XNA Framework.	LIVE Silver membership + Premium XNA Creators Club membership	No requiere nada.	No requiere nada.
Usar un System Link para conexión de Área Local (LAN).	LIVE Silver membership + Premium XNA Creators Club membership	No requiere nada.	No requiere nada.
Utilizar el Xbox LIVE y Games for Windows – Servidores Live	LIVE Silver membership + Premium XNA Creators Club membership	LIVE Silver membership + Premium XNA Creators Club membership	No Disponible en Windows Phone
Usar LIVE para conectarse a otras máquinas usando conexión de Internet mientras el juego está en desarrollo.	LIVE Gold membership + Premium XNA Creators Club membership	LIVE Silver membership + Premium XNA Creators Club membership	No Disponible en Windows Phone

2.5 Alternativas del sistema y selección de la solución

Para la realización de las alternativas hay que tener muy en cuenta algunas limitaciones, sobre todo las de la tabla 1(usos de Live) descrita anteriormente en la sección de restricciones.

2.5.1 Alternativa 0 Juego offline Xbox 360

Un juego Offline del tipo que queremos desarrollar tendría las funcionalidades básicas.

Funcionalidades:
Modo de Practica de 1 jugador.
Modo Versus de jugador contra jugador en misma pantalla (con las problemas que ello conlleva: zoom out y zoom in o Split Screen).
Opciones de dificultad y sonido.

Costes:
1x PC (gama media) 500 €
1x Xbox 360 Elite 199 €
1x XNA creators club premium Account 0 € (por msdn de UAB)
1x Visual Studio 2008 0 € (por MSDN de UAB)
1x XNA 3.1 0 € (Gratuito)
1x Gimp (programa de dibujo) 0 € (Gratuito)

2.5.2 Alternativa 1 Juego Online Xbox 360 con servidor principal y clientes

Esta sería la mejor alternativa de cara a un buen servicio entre clientes, ya que habría un server intermediario general al que todos se conectarían y enviaría los paquetes. La desventaja de usar este tipo desarrollo es que es más cara ya que necesitas un servidor dedicado mientras que con salas simplemente conectándose los clientes entre si ya basta.

Funcionalidades:
Modo de Practica de 1 jugador.
Modo Versus de jugador contra jugador mediante conexión a servidor.
Opciones de dificultad y sonido.

Costes:
2x PC (gama media) 1000 €
1x servidor 500 €
1x Xbox 360 Elite 199 €
1x XNA creators club premium Account 0 € (por msdn de UAB)
1x Visual Studio 2008 0 € (por MSDN de UAB)
1x XNA 3.1 0 € (Gratuito)
1x Gimp (programa de dibujo) 0 € (Gratuito)

2.5.3 Alternativa 2 Juego Online para PC cliente a cliente usando conexión de Visual Studio

Otra posibilidad es que XNA son unas librerías de Visual Studio C#, pero también se pueden usar las que dispone Visual Studio. La desventaja en este sentido es usar eso implicaría que sería una versión para PC ya que en consola Xbox 360 solo se puede usar los protocolos de envíos de paquetes de XNA.

Funcionalidades:
Modo de Practica de 1 jugador.
Modo Versus de jugador contra jugador mediante conexión entre clientes creando salas.
Opciones de dificultad y sonido.

Costes:
2x PC (gama media) 1000 €
1x Visual Studio 2008 0 € (por MSDN de UAB)
1x XNA 3.1 0 € (Gratuito)
1x Gimp (programa de dibujo) 0 € (Gratuito)

2.5.4 *Alternativa 3 Juego Online cliente a cliente para PC y Xbox 360 usando servicio LIVE*

Esta alternativa consistiría en usar protocolos del servicio Xbox Live donde se conectaría un cliente con otro. Las ventajas es que no nos hace falta servidor y hay bastante información para realizar este tipo de conexiones. La desventajas es que al ser de cliente a cliente, habrá un Host y otro que conecta con lo cual el anfitrión siempre será el que mejor latencia tendrá.

Funcionalidades:
Modo de Practica de 1 jugador.
Modo Versus de jugador contra jugador mediante conexión entre clientes creando salas.
Opciones de dificultad y sonido.

Costes:
2x PC (gama baja-media) 1000 €
1x Xbox 360 Elite 199 €
2x XNA creators club Premium Account 0 € (por msdn de UAB)
1x Visual Studio 2008 0 € (por MSDN de UAB)
1x XNA 3.1 0 € (Gratuito)
1x Gimp (programa de dibujo) 0 € (Gratuito)
1x Xbox Live Gold Account 1 año 59 €
1x Xbox Live Silver Account 0 € (Gratuito)

2.5.5 *Alternativa 4 Juego Online server a cliente para PC y Xbox 360 usando servicio LIVE*

Esta alternativa consistiría en usar protocolos del servicio Xbox Live donde se conectarían los clientes con otro que haría de servidor. El problema es el aumento de precio al necesitar un servidor permanente y el aumento de complejidad al usar una arquitectura de cliente a servidor.

Funcionalidades:
Modo de Practica de 1 jugador.
Modo Versus de jugador contra jugador mediante conexión entre clientes creando salas.
Opciones de dificultad y sonido.

Costes:
2x PC (gama baja-media) 1000 €
1x servidor 500 €
1x Xbox 360 Elite 199 €
2x XNA creators club premium Account 0 € (por msdn de UAB)
1x Visual Studio 2008 0 € (por MSDN de UAB)
1x XNA 3.1 0 € (Gratuito)
1x Gimp (programa de dibujo) 0 € (Gratuito)
1x Xbox Live Gold Account 1 año 59 €
1x Xbox Live Silver Account 0 € (Gratuito)

2.5.6 *Alternativa 5 Juego Online cliente a cliente para PC y Xbox 360 usando LIVE con XNA 4.0*

Necesitaremos una 360 para compilar la versión de PC en la máquina, 2 cuentas Premium y dos cuentas Silver para poder realizar los tests entre los 2 PC. Ya que queremos hacer las pruebas con Live necesitaremos la cuenta Gold.

Funcionalidades:
Modo de Practica de 1 jugador.
Modo Versus de jugador contra jugador mediante conexión entre clientes creando salas.
Opciones de dificultad y sonido.

Costes:
2x PC (gama baja-media) 1000 €
1x Xbox 360 Elite 199 €
2x XNA creators club premium Account 0 € (por msdn de UAB)
1x Visual Studio 2010 P.E. 900 €
1x XNA 4 0 € (Gratuito)
1x Gimp (programa de dibujo) 0 € (Gratuito)
1x Xbox Live Gold Account 1 año 59 €
1x Xbox Live Silver Account 0 € (Gratuito)

2.5.7 *Solución propuesta*

	Costes adquisición	**Costes adaptación**	**Nuevos Recursos**	**Soporte**	**Complejidad**
Alternativa 0	699	Bajos	Adaptable		Media
Alternativa 1	1699	Altos	Adaptable	Microsoft	Muy Alta
Alternativa 2	1000	Medianos	Adaptable	Microsoft	Alta
Alternativa 3	1258	Medianos	Adaptable	Microsoft	Alta
Alternativa 4	1758	Altos	Adaptable	Microsoft	Muy Alta
Alternativa 5	2158	Medianos	No lo necesita	Microsoft	Alta

A la vista de las características y el cuadro comparativo, la versión Offline sería la más económica. Las versiones para PC también nos proporcionarían un abaratamiento pero ya que el proyecto nace como una versión para consola la alternativa tres es la más adecuada. La alternativa uno sería parecida a la tres pero la necesidad de un servidor subiría el precio aún más al necesitar otra máquina.

La alternativa cinco por otra parte seria la ideal, ya que trabajaríamos con las versiones más actuales de Visual Studio y de XNA pero ya que el encarecimiento sería excesivo la más adecuada para el proyecto es la tres.

2.6 Riesgos

2.6.1 Evaluación de riesgos

Los riesgos desarrollando el proyecto son varios, y difieren desde riesgos por planificación a efectos imprevistos por actualizaciones en el servicio de XNA por parte de Microsoft.

R1. Planificación temporal optimista: Este riesgo nos lo provocaría un Estudio de Viabilidad demasiado optimista. Esto podría provocar que no se acabase en la fecha prevista, aumentan los recursos.

R2. Falta de alguna tarea necesaria: Este inconveniente nos lo provocaría un Estudio de Viabilidad donde faltase alguna tarea importante para el proyecto. Esto podría provocar que no se cumpliesen los objetivos del proyecto.

R3. Presupuesto poco ajustado: Este riesgo nos lo provocaría un Estudio de Viabilidad donde el presupuesto no fuese el adecuado (por ejemplo más bajo que el real). Esto podría provocar menos calidad, pérdidas económicas.

R4. Cambio de requisitos: Esta dificultad aparecería si en el Estudio de Viabilidad en el apartado del análisis de requisitos tengamos que hacer cambios. Esto podría provocar un retraso en el desarrollo y resultado.

R5. Equipo del proyecto demasiado reducido: Este riesgo aparecería si el equipo para el proyecto fuese insuficiente para mantener el ritmo planteado en el Estudio de Viabilidad. Esto podría provocar un retraso en la finalización del proyecto o que no se cumpliesen los objetivos del proyecto.

R6. Herramientas de desarrollo inadecuadas: Este contratiempo aparecería en la implementación donde veríamos si el estudio inicial y las herramientas para su desarrollo eran las adecuadas. Esto podría provocar un retraso en la finalización del proyecto, menos calidad,...

R7. Fase de test incorrecta: Este obstáculo podía aparecer en la fase de desarrollo e implementación. Esto podría provocar una falta de calidad, insatisfacción de los usuarios del producto y por tanto pérdidas económicas en el desarrollador.

R8. Incumplimiento de alguna norma, reglamento o legislación: Este problema puede aparecer en cualquier fase. Esto podría provocar que no se cumpliesen los objetivos y podría tener repercusiones legales.

R9. Abandonamiento del proyecto antes de la finalización: Este riesgo puede aparecer en cualquier fase. Esto podría provocar pérdidas económicas y el desencanto de haber abandonado un proyecto.

2.6.2 Catalogación de riesgos

Riesgos	Probabilidad	Impacto
R1	Alta	Crítica
R2	Alta	Crítico
R3	Alta	Crítico
R4	Alta	Marginal
R5	Alta	Crítico
R6	Baja	Crítico
R7	Baja	Crítico
R8	Alta	Crítico
R9	Alta	Crítico

2.6.3 Plan de contingencia

Si alguno de los riesgos previos apareciese las posibles medidas son las siguientes:

Riesgos	Solución que hay que adoptar
R1	Retrasar alguna funcionalidad, afrontar posibles pérdidas.
R2	Revisar el estudio de viabilidad, modificar la planificación.
R3	Afrontar las posibles pérdidas intentando reducirlas al máximo.
R4	Retrasar la funcionalidad, modificar la planificación y presupuesto.
R5	Pedir un retraso, afrontar las pérdidas.
R6	Proveer con herramientas alternativas, mejorar la calidad.
R7	Realizar un calendario de betatests y realizar el máximo de tests posibles.
R8	Diseñar los tests con antelación, realizar tests automáticos, afrontar pérdidas económicas.
R9	Revisar la seguridad en cada fase, aplicar políticas de seguridad.

2.7 Presupuesto

2x PC (gama baja-media) 1000 €
1x Xbox 360 Elite 199 €
2x XNA creators club premium Account   0 € (por msdn de UAB)
1x Visual Studio 2008 0 € (por msdn de UAB)
1x XNA 3.1 0 € (Gratuito)
1x Gimp (programa de dibujo) 0 € (Gratuito)
1x Xbox Live Gold Account 1 año 59 €

1x Xbox Live Silver Account 0 € (Gratuito)

	Coste Amortización	Coste Unitario	Periodo de Amortización	Periodo de Utilización
Amortización PC principal programador	13 €	500 €	36 meses	1 año
Amortización Xbox 360	5 €	199 €	36 meses	1 año
Amortización XNA Premium creators club	0 €	0 € (licencia UAB)	12 meses	1 año
Amortización PC secundario programador	13 €	500 €	36 meses	6 meses
Amortización Visual Studio 2008	0 €	0 € (licencia UAB)	36 meses	1 año
Amortización GIMP	0 €	0 €(gratuito)	36 meses	2 meses
Amortización XNA 3.1	0 €	0 €(gratuito)	36 meses	1 año
Amortización LIVE Gold	4 €	59 €	12 meses	1 año
Amortización LIVE Silver	0 €	0 €(gratuito)	36 meses	1 año

2.8 Planificación del proyecto

Para desarrollar un proyecto es necesario realizar una programación detallada una vez se han determinado los objetivos. Dentro de esta planificación es conveniente marcar hitos, es decir llegar a ciertos niveles de desarrollo en unas fechas determinadas. En nuestro caso, las numerosas incertidumbres incrementan sobremanera la complejidad de la planificación, ya que inicialmente el alcance del proyecto es indeterminado en algunos puntos y dependiendo de la resolución de estos y el tiempo restante acabarían determinando el alcance del proyecto final. Al acabar el hito de la versión offline hay dos caminos posibles, empezar la programación del cliente online o proseguir con la programación del juego complicándolo más.

En este sentido se plantea como en el desarrollo de un videojuego comercial donde tenemos diferentes prototipos pasando diferentes fases: preAlpha, Alpha, Beta,... hasta llegar a la versión final. El último prototipo tendrá que acabarse por mayo del 2011.

Las dificultades que se pueden encontrar es inicialmente un lenguaje de programación no utilizado previamente(Visual Studio C# 2008) y unas API como las de XNA que requerirán un aprendizaje. Por otra parte ya inicialmente se prevé que otro problema añadido al ser un título para X 360 es los conflictos de incompatibilidades entre librerías con respecto al PC, es decir, para la compilación del programa en la consola. En relación a esto otro contratiempo es que al usar la versión XNA 3.1 se corre el riesgo de que en una nueva actualización ya no haya soporte en el programa.

Por todo ello, la metodología que se quiere llevar a cabo consiste en tres horas cada día de lunes a viernes por semana con posibilidad de poder añadir horas extra el sábado y domingo.

Los prototipos iníciales con sus tiempos estimados con objetivo a la entrega en la primera convocatoria de proyectos final de carrera en 2011 son los siguientes:

Prototipo Cero Pre Alpha

Características: Personaje en un escenario con perspectiva aérea que se pueda mover. Utilizaríamos escenarios y sprites no propios de cara a testear movimientos y bugs. Implementación de menús iníciales.
Fechas previstas: 11 Octubre 2010-24 Octubre 2010
Duración total en horas: 30 horas

Prototipo Offline Alpha 1

Características: modo offline, 2 jugadores en misma pantalla con dos mandos, pudiéndose mover. El escenario ya tiene bases y torres gráficamente (sin implementar su comportamiento). Implementación de menús correspondientes.
Fechas previstas: 25 Octubre 2010-7 Noviembre 2010
Duración total en horas: 30 horas

Prototipo Offline Alpha 2

Características: Inicio de programación de torres, bases (básicamente que se puedas eliminar). Eliminación de jugadores y respawn (resurrección al pasar un tiempo). Implementación inicial de Interfaz.
Fechas previstas: 8 Noviembre 2010-21 Noviembre 2010
Duración total en horas: 30 horas

Prototipo Offline Alpha 3

Características: Añadimos dos tipos de "héroes" disponibles para el jugador con sus correspondientes habilidades. Creación gráfica propia, interfaz, bases, torres, héroes y desarrollo inicial de animaciones. Corrección de bugs y buen funcionamiento general del juego.
Fechas previstas: 22 Noviembre 2010-12 Diciembre 2010
Duración total en horas: 45 horas

***HITO: Final versión Offline*:** Programación videojuego Offline finalizada. En este momento acabaríamos la etapa offline e intentaríamos realizar una primera versión online y dependiendo de los resultados haríamos la variante online o offline. El siguiente prototipo aún es común a las dos variantes pero los siguientes son exclusivos de cada una.

Prototipo Online Alpha 4 (Común a ambas variantes)

Características: Iniciar la programación de cara a desarrollar el juego Online, determinar la viabilidad de la opción online y finalizar desarrollo de animaciones.
Fechas previstas: 13 Diciembre 2010-9 Enero 2011
Duración total en horas: 60 horas

Variante Online

Prototipo Online Beta 1

Características: Testear a fondo modo online, creación de escenario más grande. Testear héroes y funcionamiento de sus habilidades. Intentar habilitar más de 2 jugadores (2 vs 2 o 3 vs 3).
Fechas previstas: 10 Enero 2011-6 Febrero 2011
Duración total en horas: 60 horas

Prototipo Online Beta 2

Características: Implementación de mejora de funcionamiento de ataque de torres de defensa, creación de NPC (personajes llevados por la máquina): enemigos matables, tropas aliadas que intentan ayudar a eliminar la torre.
Fechas previstas: 7 Febrero 2011-6 Marzo 2011
Duración total en horas: 60 horas

Prototipo Online Beta 3

Características: Creación de más héroes. Complicar el escenario, creación de más animaciones y sprites propios etc. Depuración de todos los bugs encontrados.
Fechas previstas: 7 Marzo 2011-1 Mayo 2011
Duración total en horas: 120 horas

<u>Variante Offline</u>

Prototipo Online Beta 1

Características: Implementación de mejora de funcionamiento de ataque de torres de defensa, creación de NPC (personajes llevados por la máquina): enemigos matables, tropas aliadas que intentan ayudar a eliminar la torre.
Fechas previstas: 10 Enero 2011-6 Febrero 2011
Duración total en horas: 60 horas

Prototipo Online Beta 2

Características: Creación de más héroes. Complicar el escenario, creación de más animaciones y sprites propios etc.
Fechas previstas: 7 Febrero 2011-6 Marzo 2011
Duración total en horas: 60 horas

Prototipo Online Beta 3

Características: Realizar Split Screen posibilitando más de dos jugadores en misma consola, hasta cuatro. Depuración de todos los bugs encontrados.
Fechas previstas: 7 Marzo 2011-1 Mayo 2011
Duración total en horas: 120 horas

Información Global

Fecha Inicio: 11 Octubre
Fecha Finalización: 1 de Mayo
Total de horas: 435 horas.

Esta planificación nos posibilitará todo el mes de Mayo disponible para los imprevistos y la realización de la memoria del proyecto.

2.9 Conclusiones del estudio de Viabilidad

Viendo por tanto todos los datos queda claro que no es un proyecto fácil ni barato, requiere de unos ciertos prerrequisitos que en algunos puntos son bastante caros.

Hay bastantes alternativas para su realización, en el apartado correspondiente se ha explicado por qué nos hemos decantado por la tercera, por el coste y resultados obtenidos es seguramente la más adecuada. Hay otras con versiones más actualizadas como la alternativa usando XNA 4.0 o con un tratamiento del usuario más equitativo utilizando un servidor de intermediario.

El tener que trabajar de forma online, fuerza a que necesitemos dos máquinas para poder conectarse entre ellas ya que la conexión entre PC y consola no se puede realizar, por tanto o necesitamos dos consolas o dos PC para poder realizar los tests adecuados para el perfecto funcionamiento del producto.

Por otra parte las herramientas que utilizamos son en su mayoría programas de Microsoft y requieren de licencia. Por suerte en este sentido para la realización del proyecto disponemos de las licencias proporcionadas a los alumnos de la UAB lo cual nos ahorra gran parte de los costes del proyecto, esto a la vez nos limita también a lo que nos proporcionen estas licencias pero sin duda nos da herramientas adecuadas para la realización del videojuego.

Otro elemento importante es la gran cantidad de documentación y soporte por parte de la propia Microsoft y de la comunidad de usuarios de XNA.

En principio viendo los resultados del Estudio de Viabilidad es un proyecto factible y realizable en el tiempo estipulado.

3 FUNDAMENTOS TEÓRICOS

3.1 XNA

XNA es una herramienta que se anunció el 24 de marzo de 2004, en la Game Developers Conference en San José, California. La primera comunidad Technology Preview de XNA Build fue lanzada el 14 de marzo de 2006. XNA Game Studio 2.0 fue lanzado en diciembre de 2007, seguida de XNA Game Studio 3.0 el 30 de octubre de 2008. Posteriormente apareció una versión 3.1 y actualmente va por la 4.0.

XNA está incluido dentro de las Microsoft Game Development Sections, componiéndose de un estándar Kit de desarrollo para Xbox 360 y el propio XNA Game Studio siendo las últimas versiones más encaradas al desarrollo de títulos para su plataforma de Windows phone. La propia página de XNA ahora mismo ha sido dividida en App Hub para el desarrollo en Zune y móviles y en Xbox 360 con el XNA creator's Club.

Esta herramienta nos proporciona una serie de métodos a través de las librerías XNA Framework que nos facilitan la creación para videojuegos para PC y Xbox 360.

En el aspecto teórico tenemos las librerías siguientes:

Microsoft.Xna.Framework: Proporciona las clases game así como los timers y game loops.

Microsoft.Xna.Framework.Audio: Contiene APIs (aplicaciones de interfaz de programación) de bajo nivel que pueden cargar y manipular XACT y así como usar archivos de audio.

Microsoft.Xna.Framework.Content: Contiene las herramientas para usar los componentes del Content Pipeline.

Microsoft.Xna.Framework.Design: Proporciona un camino unificado para convertir tipos de variables a otros tipos.

Microsoft.Xna.Framework.GamerServices: Contiene clases para implementar varios servicios relacionados con los jugadores. Estos servicios pueden ser desde comunicarse directamente con el jugador, información del jugador, o reflejar elecciones realizadas.

Microsoft.Xna.Framework.Graphics: Contiene métodos API que toman ventaja de las capacidades de manejo de objetos en 3D de la consola.

Microsoft.Xna.Framework.Graphics.PackedVector: Representa tipos de datos con componentes que no son múltiplos de 8 bits.

Microsoft.Xna.Framework.Input: Contiene clases para recibir información sobre el teclado, ratón, y mando de 360.

Microsoft.Xna.Framework.Media: Contiene clases para enumerar, escuchar canciones, álbumes, playlists e imágenes.

Microsoft.Xna.Framework.Net: Contiene clases para implementar soporte para Xbox LIVE,

multiplayer, y networking para XNA Framework games.

Microsoft.Xna.Framework.Storage: Contiene clases que permiten leer y escribir en archivos.

En el aspecto práctico, para ello necesitamos Visual Studio C#, en el caso que nos ocupa, el 2008 ya que trabajaremos con la versión XNA 3.1. Existen versiones más actuales como XNA 4.0 pero para ello necesitamos el Visual Studio C# 2010 y están más encaradas a la producción de videojuegos para Windows Phone.

Una vez instalado el Visual Studio 2008 junto con XNA ya tendremos el entorno preparado para trabajar en un juego de PC.

Básicamente consistiría en crear un nuevo proyecto del tipo Windows Game y ya nos proporcionaría el esqueleto para programar nuestro juego.

Para trabajar con Xbox 360 es un poco más complicado y hay algún requisito más.

Para empezar tenemos que compilar nuestro juego en la consola y para ello consola y ordenador tienen que estar comunicados de alguna forma, la manera más usual es pertenecer a la misma red. Si este prerrequisito se cumple necesitaremos el XNA Game Studio Connect. Este programa nos permitirá realizar un enlace entre la consola y el ordenador y siguiendo los protocolos de conexión que nos piden al instalarlo podremos compilar con nuestro PC en la consola. Para poder descargar este programa necesitamos tener una cuenta Premium de XNA Creator's Club.

En lo referente a la realización del código para la consola no nos tenemos que preocupar mucho ya que el propio Visual Studio creará un programa espejo si lo deseamos y tendremos una versión para PC y otra para Xbox 360.

Si previamente hemos configurado correctamente todo podremos compilar de tres formas distintas: directamente en PC, en consola, o la versión mixta que consiste en compilar a la vez para PC y consola.

Existen otros complementos como el XACT o el Microsoft Cross-platform Audio que son programas para trabajar con el audio de Xbox 360 pero realmente lo mínimo necesario para poder trabajar en un videojuego en la consola de Microsoft es lo anteriormente citado.

Cuando creamos un proyecto en Visual Studio 2008 del tipo Windows Game nos aparece lo siguiente:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace WindowsGame1
{
    /// <summary>
    /// This is the main type for your game
```

```
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        /// <summary>
        /// Allows the game to perform any initialization it needs to before starting to run.
        /// This is where it can query for any required services and load any non-graphic
        /// related content.  Calling base.Initialize will enumerate through any components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }

        /// <summary>
        /// LoadContent will be called once per game and is the place to load
        /// all of your content.
        /// </summary>
        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);

            // TODO: use this.Content to load your game content here
        }

        /// <summary>
        /// UnloadContent will be called once per game and is the place to unload
        /// all content.
        /// </summary>
        protected override void UnloadContent()
        {
            // TODO: Unload any non ContentManager content here
        }

        /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input, and playing audio.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }
```

```
        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            base.Draw(gameTime);
        }
    }
}
```

Éste es el esqueleto más básico de un juego para Windows realizado con XNA, a continuación vamos a explicar de qué elementos se compone y su funcionamiento.
Los primeros componentes importantes que podemos ver son:

```
         GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
```

Éstos dos componentes son los encargados de la parte gráfica siendo el objeto spriteBatch del tipo SpriteBatch el elemento al que iremos pasando con los datos gráficos, es decir si necesitamos crear un método que haga algo con la parte gráfica tenemos que pasarle ese objeto.

Por tanto el GraphicsDeviceManager es el encargado de crear la parte gráfica, existe otro device importante que no aparece en este esqueleto básico, el AudioDeviceManager, que como se entenderá es el encargado del sonido.

A partir de esos dos devices tenemos 4 métodos básicos: Initialize, Load, Update y Draw.

El Initialize es el método básico donde inicializamos las variables y objetos que necesitemos posteriormente.

El método Load será el encargado de cargar todo el material que necesitemos, un ejemplo serían las texturas de los sprites y sonidos guardándose como content en el device correspondiente gráfico, audio o genérico. Ejemplos prácticos sería:

```
public override void Load(ContentManager content)
        {
            soundEffect1 = content.Load<SoundEffect>(soundName1);
            Textura = content.Load<Texture2D>("Textures/firewind");//textura de tamaño modelo
         }
```

En este ejemplo cargaríamos un efecto de sonido y una textura 2D en las dos variables correspondientes.
El siguiente método es el Update, éste es el método que va actualizándose a cada segundo a partir de que se genera nuestro contenido, aquí introduciremos el código que necesitemos que se vaya actualizando segundo a segundo, por ejemplo el control del sprite a través del mando, colisiones de sprites o también algún cambio en las animaciones. Un ejemplo práctico sería:

```
public override void Update(HeroOnline hero, TowerOnline tower, BaseHero baseHero, float elapsed, Texture2D field2) {
            //actualizamos las texturas animadas
            TexturaFTrampa.UpdateFrame(elapsed);
            TexturaRespirando.UpdateFrame(elapsed);
            TexturaCaminando.UpdateFrame(elapsed);
            TexturaDisparando.UpdateFrame(elapsed);
            TexturaHerido.UpdateFrame(elapsed);

            if ((Is_Firing2 == true)||(Is_Firing == true))
```

```
            {
               soundEffect3.Play();
               disparoVivo = true;
               Main_Skill = false;
               Is_Firing = false;
               Is_Firing2 = false;
            }
}
```

En este ejemplo de **Update**, actualizaríamos a cada segundo mediante su método correspondiente las animaciones y en el caso de que hubiese un disparo, actualizaríamos las variables correspondientes.

Finalmente en este esqueleto básico que estamos analizando tenemos el método **Draw**. Éste método será el encargado de dibujar el contenido que hemos definido previamente por pantalla y tiene interiormente ciertos métodos muy potentes para el tratamiento de objetos, deformaciones gráficas, rotaciones u otros métodos que necesitemos. Un ejemplo práctico sería:

```
public override void Draw(SpriteBatch batch)
        {
            if (State == true)
            {//está vivo
                if (estadoAnime == EstadoAnime.disparando)
                    TexturaDisparando.DrawFrame(batch, Posicion, IsRight);
                if (estadoAnime == EstadoAnime.trampa)
                    TexturaFTrampa.DrawFrame(batch, Posicion, IsRight);
                if (estadoAnime == EstadoAnime.quieto)
                    TexturaRespirando.DrawFrame(batch, Posicion, IsRight);
                if (estadoAnime == EstadoAnime.caminando)
                    TexturaCaminando.DrawFrame(batch, Posicion, IsRight);
                if (estadoAnime == EstadoAnime.herido)
                    TexturaHerido.DrawFrame(batch, Posicion, IsRight);
            }
            if (State == false)
            {//esta muerto
                TexturaMuerto.DrawFrame(batch, Posicion, IsRight);
            }
            //Pintamos el disparo fuego
            if (DisparoVivo == true)
            {
                batch.Draw(TexturaFire, PosicionDisparo, Color.White);
            }
}
```

En este ejemplo nos dibujaría dependiendo del estado de animación un sprite o otro o dependiendo del estado del disparo, si aparece o no.

3.2 NET FRAMEWORK

Dentro de esta API tenemos las clases y métodos necesarios para trabajar en la conectividad entre nuestros jugadores.

Nombre Clases	Descripción
AvailableNetworkSession	Describe una sesión multijugador a la que se puede unir.
AvailableNetworkSessionCollection	Representa una colección de sesiones disponibles para unirse.
GameEndedEventArgs	Representa los argumentos pasados por el evento de

	GameEnded .
GamerJoinedEventArgs	Representa los argumentos pasados por el evento GamerJoined .
GamerLeftEventArgs	Representa los argumentos pasados por el evento GamerLeft.
GameStartedEventArgs	Representa los argumentos pasados por el evento GameStarted .
HostChangedEventArgs	Representa los argumentos pasados por el evento HostChanged.
InviteAcceptedEventArgs	Representa los argumentos pasados por el evento InviteAccepted .
LocalNetworkGamer	Representa un jugador local en una sesión network.
NetworkException	Da error si hay un error de comunicación de red.
NetworkGamer	Representa un jugador en una sesión de network.
NetworkMachine	Representa una máquina física (como una Xbox 360 o un ordenador con Windows) que está participando en una sesión de multijugador. Puede ser usado para detectar cuando más de un NetworkGamer está jugando con la misma máquina.
NetworkNotAvailableException	Excepción que aparece si no está disponible la red.
NetworkSession	Representa una sesión de juego multiplayer.
NetworkSessionEndedEventArgs	Representa los argumentos pasados por el evento SessionEnded . Estos argumentos son pasados a los controladores de eventos cuando la sesión finaliza.
NetworkSessionJoinException	Nos da error si encuentra un problema al unirnos a una sesión.
NetworkSessionProperties	Describe información específica del tipo de juego sobre el objeto de NetworkSession .
PacketReader	Proporciona la funcionalidad común para leer paquetes de la red.
PacketWriter	Proporciona la funcionalidad común para escribir información en paquetes enviados por la red.
QualityOfService	Describe la calidad de la conexión de la red entre la máquina y host de la sesión multiplayer que es encontrado con un pedido de juego.

3.3 XNA Network Protocol Specication

De serie podemos utilizar las API que nos da el Visual Studio para la realización de juegos online para una plataforma de PC pero para poder realizar las conexiones online con la consola y el PC tenemos otra opción que es la más recomendable de cara a las consolas y en nuestro caso el juego que nos ocupa, los servidores LIVE de Microsoft.

Para poder usar esta red tenemos que usar los protocolos de especificación de red de XNA. Estos protocolos de red sirven tanto para conexiones mediante cable entre consola y consola como conexiones mediante internet.

Estaríamos hablando de dos tipos de conexión LAN (o system link como lo denomina Microsoft) y Peer to Peer. La conexión LAN no se ha realizado en la versión final del videojuego, pero su código en parte sería parecido a la versión de red de internet, la diferencia principalmente está en los problemas físicos añadidos que tenemos al realizar una conexión de punto a punto.

Los juegos en red tienen tres enemigos mortales:

Latencia: ocasiona que la información llegue tarde.

Perdida de paquetes: Ocasiona que alguna información no llegue nunca.

Ancho de banda: limita la cantidad de información que se puede enviar.

La latencia es causada por problemas físicos, ya que no se puede superar la velocidad de la luz por lo que nuestro envío de información siempre estará limitado.

Además la información de red no viaja en el vacío, la velocidad de la luz normalmente es medida ahí, pero cuando la información viaja a través de cables fibra óptica o de cobre se ve ralentizada a un 60%.

Por otra parte hay que tener en cuenta que viajará de modem a modem, uno en cada extremo, cada modem añade unos 10 milisegundos de latencia, si además el receptor no usa el mismo ISP en el camino habrán varios router y cada router añade unos 5 a 50 milisegundos de latencia.

Para hacernos una idea, tenemos el siguiente cuadro teniendo como origen Berlín:

Ciudad	Distancia (km)	Tiempo Calculado (ms)	Tiempo medido (ms) (ping 32byte)
Londres	910	4.67	137 (www.proteusinvestigations.co.uk)
Jerusalén	2,900	14.87	-
Pretoria	8,660	44.41	272 (mybroadband.co.za)
Méjico	9,710	49.79	671 (mexicocity.gob.mx)
Sídney	15,960	81.85	432 (cityofsydney.nsw.gov.au)

Nos podemos preguntar hasta qué punto nos pueden perjudicar estos problemas en la conexión, bien los juegos de Xbox funcionan con latencias de hasta 200 milisegundos siendo éste un límite aproximado.

En la referencia a la pérdida de paquetes es algo que se quiere evitar pero inevitable, en general los programadores prefieren alguna perdida de paquetes que una mala latencia aunque esto también varía dependiendo del protocolo usado. Para la mayoría de juegos se usa **UDP**, pero el **UDP** no es fiable, envía los paquetes a la red y los olvida, no sabemos si el paquete llega ni en qué orden, y si el paquete se pierde no es recuperable. Otro problema es que la información en los paquetes se puede corromper. En lo referente al orden usando un protocolo de TCP se puede solucionar pero solo en juegos de estrategia por turnos o videojuegos parecidos es recomendable ya que es demasiado lento para responder con un juego en el que la latencia tome partido.

Ahora pensemos que tenemos 20 bytes del encabezado de IP, 8 bytes para el encabezado UDP y 22

bytes para el servicio LIVE y XNA framework(usado para NAT trasversal, encriptación, y envío fiable/ordenado).

Por tanto 50 bytes de encabezado de información por paquete.

Para no sobrepasar estos límites hay métodos de compresión muy potentes proporcionados por XNA/C#:

Si una variable float no puede ser mayor que 1, podemos convertirla en Alpa8 (75% de reducción)

Si un valor float puede ser mayor que 1 podemos convertirla en Halfsingle (50% de reducción)

- Vector2 a HalfVector2
- Vector4 a HalfVector4
- Si un Vector3 es normalized a Normalized101010
- Si no está normalized a 3 HalfSingle
- Quaternion a NormalizedByte4

Podemos usar también Color.packedValue

En la siguientes imágenes extraídas de XNA Network Protocol Specication, Draft realizado por Brady Dial y Charles McGarvey podemos ver un esquema del tipo de conexión que realiza la consola.

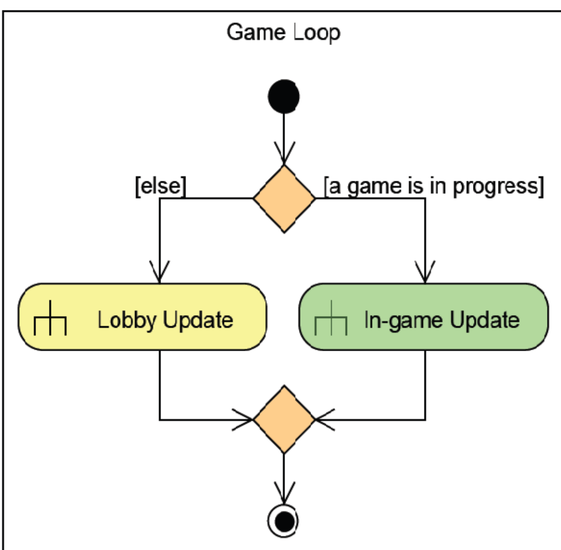

Figura 4 Bucle de Conexión

Realiza un bucle en donde dependiendo si estamos en partida o no utiliza dos métodos distintos, el Lobby Update o el In-game Update, es decir las actualizaciones de la sala principal(donde creamos las salas de juego de cada jugador) o de las actualizaciones dentro del juego.

El funcionamiento del In-game Update es el siguiente que se muestra en el esquema.

Figura 5 Funcionamiento del In Game Update

A cada instante miramos si hemos recibido los eventos de cada cliente conectado a nuestra partida, en el caso de que no recibiésemos nada entraríamos en la zona de pérdida de conexión pero en el caso de que los recibamos el In-game Update intenta ajustar la latencia de los clientes y envía los eventos del siguiente instante a los otros clientes, aplicamos los datos que hemos recibido en nuestra partida actualizando el estado del juego e incrementamos el contador de instantes.

El protocolo funciona y permite una buena versatilidad a la hora de realizar las conexiones correctamente.

Ahora bien tiene una serie de limitaciones que ahora pasaremos a comentar:

- No hay ningún mecanismo que permita a un jugador conectarse a una partida ya empezada, esto que en principio simplemente implicaría que cuando empezamos la partida tenemos que estar seguros de que este balanceada (por ejemplo que fuesen 2 jugadores contra otros dos) implica más cosas, ya que impide las reconexiones en el caso de que perdiésemos la conexión.
- Las partidas prosiguen incluso en el caso de que se caigan jugadores, solo en el caso de que no existiese el anfitrión se pararía.
- La latencia la determina la peor conexión entre los clientes.
- Cada cliente tiene su estado de juego, esto que a priori facilita la interacción entre los clientes también podría ocasionar posibles hacks de visibilidad.
- Protocolos similares fueron usados en los primeros juegos de disparo que en su época eran recordados por su pésimo servicio de soporte de red. En la actualidad se utiliza normalmente un sistema más fiable de topología de cliente-servidor, sin embargo hay que comentar que muchos RTS han utilizado protocolos parecidos al XNA con muy buen resultado con lo cual lo convierte en un protocolo adecuado para este tipo de juegos.

Por tanto al trabajar con este protocolo ya conocemos las ventajas y desventajas que nos encontraremos. Sin duda en la actualidad hay protocolos mejores o topologías como la de cliente servidor más adecuadas para un juego con miles de usuarios y varios conectados por partida.
Para el juego que nos ocupa limitado a dos jugadores en duelo conectándose vía salas **XNA** puede ser una gran opción, al estilo de muchos juegos de estrategia que han usado sistemas similares.

3.4 Xbox 360

La principal herramienta para el proyecto es la consola de Microsoft, por tanto tendremos que conocer algunas de sus características técnicas:

CPU	3.2 GHz(PowerPC) Tri-Core Xenon
GPU	ATI Xenos 512 MB de RAM GDDR3 a 500 MHz
Soporte	DVD,Compact Disc. Adicional: HD DVD (descontinuado)
Almacenamiento	Disco duro extraíble de 20, 60, 120 ó 250 GB, Tarjetas de memoria de 64, 256 y 512 MB (descontinuadas)
Controles	Control Xbox 360
Conectividad	Modelo S 2.4 GHz wireless, 5 puertos USB 2.0, Wifi 802.11a/b/g/n, receptor IR, 100 Mbit Ethernet, Puerto auxiliar.

Xbox 360 es la segunda videoconsola de sobremesa producida por Microsoft, fue desarrollada en colaboración con IBM y ATI.

Fue lanzada en Norteamérica, Japón, Europa y Australia entre 2005 y 2006. Su servicio Xbox Live permite a los jugadores competir vía online y descargar contenidos como juegos arcade, demos, trailers, programa de televisión y películas. La Xbox 360 es la sucesora directa de la Xbox, y compite actualmente contra la PlayStation 3 de Sony y la Wii de Nintendo como parte de las videoconsolas de séptima generación.

Sus principales características, es su CPU basado en un IBM PowerPC y su GPU que soporta la tecnología de Shaders Unificados. El sistema incorpora un puerto especial para agregar un disco duro externo y es compatible con la mayoría de los aparatos con conector USB gracias a sus puertos USB 2.0. Los accesorios de este sistema pueden ser utilizados en un PC como son los mandos y el volante Xbox 360.

Microsoft apostó por la tecnología HD DVD mientras que su competencia por parte de Sony por el Blue Disc. En términos prácticos en la realización de nuestro proyecto eso no nos influye mucho ya que instalaremos el juego en el disco interno de la consola.

Desde su lanzamiento en 2005, existen cuatro modelos de los cuales tres están a la venta el modelo Elite y Arcade y el modelo de 250 Gb. Actualmente existe un nuevo modelo disponible, el Slim, con una arquitectura nuevamente diseñada y entre sus principales nuevas características están la inclusión de Wi-Fi y que sea más silenciosa. Algunas de estas diferentes versiones de la consola se pueden apreciar en la imagen:

Figura 6 Xbox 360 Arcade

Figura 7 Xbox 360 Elite

Siendo la principal diferencia los accesorios que incorporan en el paquete básico, discos duros, número de mandos y como no el tema estético.

Cosas que si se deberán tener en cuenta, la potencia de la CPU y GPU, la consola tiene preparadas unas potentes rutinas para controlar y manejar gráficos tanto en 2D como en 3D pero al igual que cuando en MATLAB el hacer operaciones con código es más lento que el realizar operaciones con los propios métodos del programa, en la consola ocurre lo mismo con lo cual aprender los métodos que utiliza será muy importante en la finalización del juego.

Como datos comerciales, destacar que el 29 de octubre de 2010 Microsoft ha actualizado sus datos de ventas de Xbox 360, y la consola suma ya en todo el mundo 42.5 millones de unidades vendidas. El ratio de juegos por consola es de 5,9.

3.5 Otros fundamentos teóricos

Para desarrollar este proyecto son necesarios conocimientos en informática sobre Metodología de programación e Ingeniería de Software.

El lenguaje de programación utilizado es el C# contenido en Visual Studio 2008. Es un lenguaje muy similar hasta cierto punto a otros estudiados en la carrera de informática en la EUI de Sabadell como Java o Python, está orientado a la programación de objetos donde podremos definir las clases

4 ANÁLISIS DEL JUEGO

4.1 Mecánica del juego

Antes de empezar el diseño de las clases que necesitaremos y su codificación e implementación en nuestro programa necesitamos tener claras una serie de cosas.

Diseñar el aspecto jugable de los juegos no es tarea fácil, los juegos pueden ser muy complejos con miles de reglas o muy intuitivos y sencillos al estilo tetris que con muy pocos elementos pueden hacer disfrutar al jugador.

Ahora mismo el estilo MOBA lo podemos subdividir en dos estilos de juego. Los juegos donde estarían DOTA, League of Legends, Heroes of Newearth que serían juegos con elementos de subir de nivel, golpes críticos, NPCs... básicamente elementos externos a los jugadores y otros como Bloodlines que se encaran con solo el contenido de los jugadores, no hay golpes críticos, ni NPCs, ni se sube de nivel.

IceFire en este sentido se coloca en un punto intermedio, la idea es hacer un MOBA lo máximo de balanceado posible encarado al e-Sport con lo cual se asemeja al Bloodlines Champions pero también quiere introducir algunos elementos de los otros MOBA en su esquema básico.

Los elementos de la idea inicial con la que se presentó el proyecto eran los siguientes: tendremos dos bases para cada jugador, dos torres defensivas que defenderán al jugador de alguna manera (en aquella etapa temprana del proyecto aun no estaba claro como lo defenderían), los personajes que controlarán cada jugador y unos puntos de mejora o Power Ups que estarían por el mapeado aleatoriamente.

La mecánica sería de la siguiente manera, cada jugador tendrá que eliminar la torre del otro, en su elección tendrá el disparar al jugador contrario o a su torre, si el jugador contrario es eliminado, estará un tiempo inactivo, que a más número de muertes significará mayor inactividad. Por el mapeado habrán Power ups que podrán ser recogidos subiendo las características de los héroes. Una vez destruida la torre se podrá destruir la base.

Inicialmente se pensó que ya que los Power ups aparecían de forma aleatoria por el mapa, los disparos de las torres podían ser de la misma manera, es decir al azar.

El problema es que unas torres de este estilo no eran operativas y aunque su tamaño de tiro fuese mayor que el de los jugadores el número de aciertos contra los enemigos era muy bajo.

Por tanto aquí se barajaron varias posibilidades:

- Una era dejar las torres y los Power ups como estaban y desarrollar nuevos elementos como NPCs que ayudasen a cada jugador. Añadir nuevos NPCs hubiese sido parecido a añadir niveles en los jugadores o introducir golpes críticos, si intentamos realizar un juego válido para el eSport se trata de minimizar el factor suerte y maximizar el factor habilidad con lo cual si se busca la introducción del mínimo número de elementos externos a los jugadores.

- La otra posibilidad de mecánica de las torres era más obvia, las torres apuntarían a los jugadores. De esta forma por un lado se induce a los jugadores a que se muevan para

esquivar esos disparos haciendo que su daño no sea insignificante, pero por otra parte como tampoco queremos que sea simplemente un juego de esquiva de disparos de torres hacemos que esos disparos sean más lentos.

En este sentido ya tenemos el campo de juego, una arena con dos torres que disparan a los héroes, sus dos bases respectivas y los héroes.

Restaba ahora acabar de definir los personajes elegibles por los jugadores. Normalmente en los MOBA los personajes tienen skills definidas y diferentes entre sí que les permite diferenciarse de otros.

Habilidades que se pensaron para los personajes fueron disparos, curaciones, invisibilidad, mejoras de velocidad de movimiento, velocidad de disparo, trampas que ralenticen y trampas que paralicen en el sitio. El problema de algunas de estas skills es que en un juego 1 vs 1 pueden desequilibrar demasiado y hay que ir probando para ajustar los números de daño, duración de las habilidades o tiempos de refresco, es un proceso inacabable. Es lo que se denomina balanceo de clases y es algo que en los juegos no termina y constantemente aparecen nuevos parches para mejorar e igualar las diferentes habilidades.

En esta etapa del proyecto nos decantamos por hacer dos personajes diferenciados pero que estuviesen igualados. El concepto básico era un personaje con habilidades de fuego y otros con habilidades de hielo. En sus stats iníciales ya habían diferencias, uno rápido y con menos vida y otro más lento y más vida.

A partir de aquí al personaje de fuego se le añadieron dos habilidades, un disparo con una velocidad y potencia mayores que el de hielo y una trampa que si el personaje enemigo caía en ella lo paralizaba permitiendo que los tiros de las torre le diesen o los del propio jugador.

El personaje de hielo con más vida pero más lento tenía un disparo menos veloz y débil que el del personaje de fuego por tanto necesitaba un plus para poder igualar los dos personajes. Este plus fue darle una habilidad de invisibilidad, el personaje estaría invisible durante unos segundos para el otro jugador permitiéndole recolocarse acercarse a él o realizar tiros más imprevistos y certeros.

Este era el set básico de los dos personajes que tras varios testeos resultó ser más adecuado, se probó con otro tipos de skills como habilidades de curación o de aceleración de movimiento pero la combinación que comentamos antes fue el más balanceado para los Beta Testers y el propio desarrollador.

En resumen, el concepto del juego quedó de la siguiente manera, dos personajes disponibles llamados Firewind e Iced, cada uno con dos skills diferenciadas. Cada jugador dispone de la torre defensiva y su base, para eliminar la base hay que eliminar primero la torre y ésta se dedica a dispararnos. Los disparos de las torres son más fuertes que los de los personajes para forzar que los usuarios los esquiven pero son más lentos.

A más muertes acumula un jugador más tiempo de resurrección tiene con lo cual tarda más en revivir, no se puede ir a lo loco a simplemente destruir las torres, hay que jugar con habilidad.

No hay ni golpes críticos, ni NPC, ni niveles, básicamente se trata de un juego de habilidad en el que se intenta que gane el que mejor estrategia de ataque y defensa utiliza.

Una vez destruida la torre y la base de un jugador el juego se termina y gana el jugador que disponga de base.

5 DISEÑO GRÁFICO

5.1.1 Introducción

Los juegos de la actualidad tienen un componente gráfico muy importante, ya desde sus inicios en los primeros videojuegos como el Pong aun teniendo una gran simplicidad comparado en los gráficos de hoy en día el grafismo tenía un gran peso en su diseño.

A día de hoy podemos hablar de que hay dos tipos de gráficos en los videojuego, gráficos 2D y gráficos 3D, inicialmente también indicaban si eran juegos en dos dimensiones o tridimensionales pero a día de hoy la distinción no está clara e incluso podemos hablar de juegos con gráficos en 2.5D, a continuación haremos una breve introducción a cada uno.

Gráficos 2D son los primeros gráficos, en general cuando hablamos de ellos nos referimos a sprites, tuvieron su época de esplendor en las consolas de 8 bits y 16 bits y aún hoy son utilizados en juegos de tipo flash en gran medida y muchos otros juegos de mayor relevancia como juegos de lucha o juegos RTS.

Los gráficos 3D se refieren a los gráficos poligonales, recayendo el peso en la capacidad matemática de las consolas que los respaldan, capaces de realizar en milésimas de segundos complicados cálculos de perspectiva, en sus inicios gráficamente hablando eran bastante simplistas a día de hoy, sin texturas y prácticamente poligonales en su totalidad. La progresión tecnológica ha posibilitado el recubrimiento de esos polígonos con impactantes texturas y que cada vez rocen el realismo más extremo. Generalmente los videojuegos más potentes apuestan por este tipo de gráficos porque son los más espectaculares y realistas.

Los gráficos 2.5D se refieren generalmente a la combinación entre gráficos 2D y 3D de alguna manera, por ejemplo un sprite con fondo poligonal, permitiendo rotar fácilmente el escenario y con 4 posiciones de sprites distintos que vaya rotando. Algunos RTS antiguos también lo han usado y algunos juegos online en la actualidad como el Ragnarok Online fueron de los primeros que apostaron por este tipo de juego.

Realizada la introducción para la realización de los gráficos, en IceFire solo se disponía de un grafista que es el propio desarrollador del título. Para realizar gráficos en 3D se requieren altos conocimientos de Maya o otras herramientas para tratar gráficos poligonales además de un engine que luego sea capaz de mover correctamente esos gráficos.

Apostar por ese tipo de gráficos no hubiese sido viable en el tiempo establecido. Por tanto la opción más viable y por la que se apostó fue por los gráficos 2D.

Los gráficos 2D o sprites como también se les denomina tienen sus ventajas y desventajas, queda claro que su tratamiento a priori es más fácil en lo referente a moverlo por pantalla, pero sin embargo en muchos otros aspectos dan más trabajo, sobretodo en el tema de la animación.

Hay que realizar Sprite Sheets, básicamente frames de animación y hacer que el programa actualice adecuadamente los movimientos con las decisiones del usuario. Cuanto más frames tienen estas Sprite Sheets más detallados son los movimientos de los personajes.

Un ejemplo sería que al pulsar adelante nos ponga el primer frame de la animación y mientras esté

pulsado se vaya incrementado el contador de frames mostrando la animación completa.

En el caso de nuestro proyecto hay que tener en cuenta que tendremos dos personajes con dos habilidades cada uno que además podrán moverse por la pantalla. Otro aspecto son las animaciones automáticas como es el caso de que alguno muera.

Por otra parte observaremos que el mapeado será mayor que el de la propia pantalla por qué daremos posibilidad de zoom out y zoom in.

Y finalmente quedarán elementos como las torres y las bases que tendrán que ser diferentes para cada bando.

Para la realización de estos gráficos se uso el programa Gimp de uso gratuito y con una capacidad parecida a otros más conocidos pero que requieren licencia como Photoshop.

5.1.2 Sprites y Spritesheets

Para la realización del personaje de fuego **Firewin**d tenemos algunas de las siguientes sprite sheets:

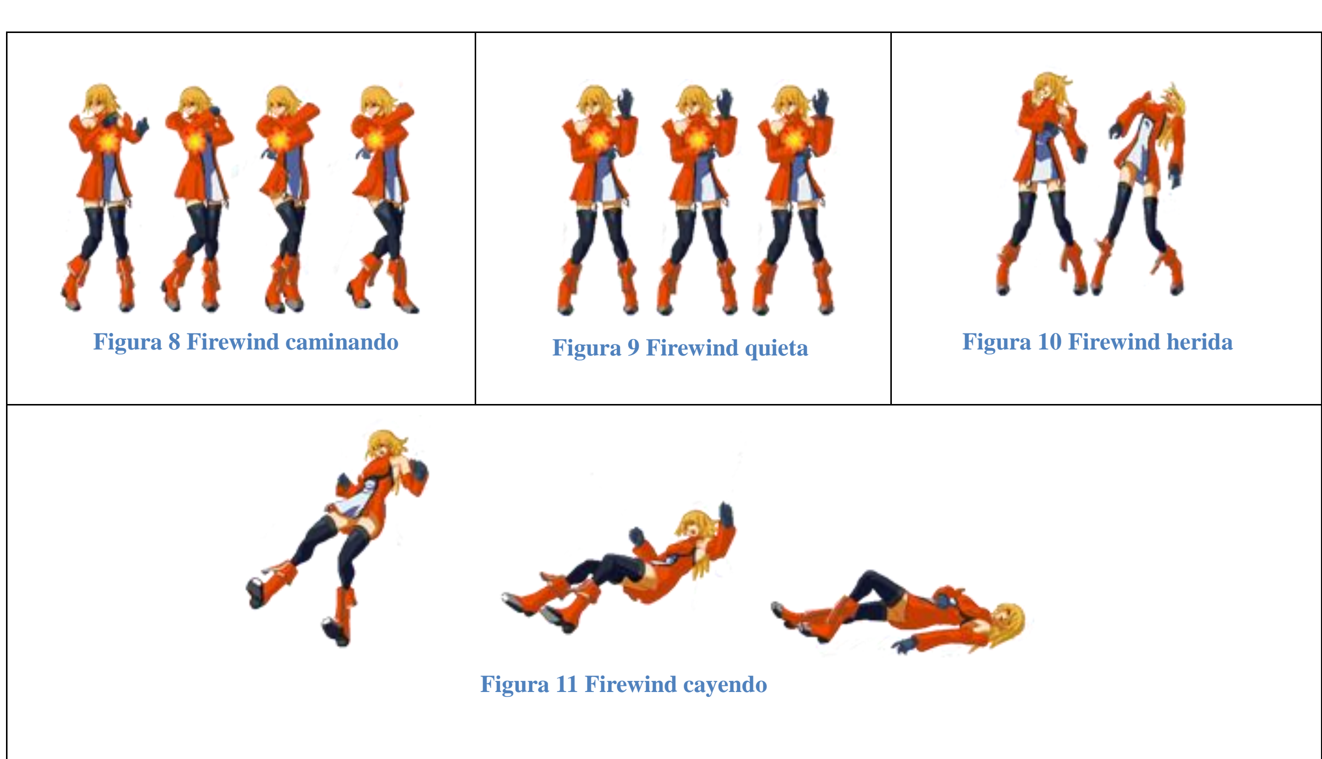

Figura 8 Firewind caminando

Figura 9 Firewind quieta

Figura 10 Firewind herida

Figura 11 Firewind cayendo

Como ejemplo del personaje de hielo Iced tenemos las siguientes sprite sheets:

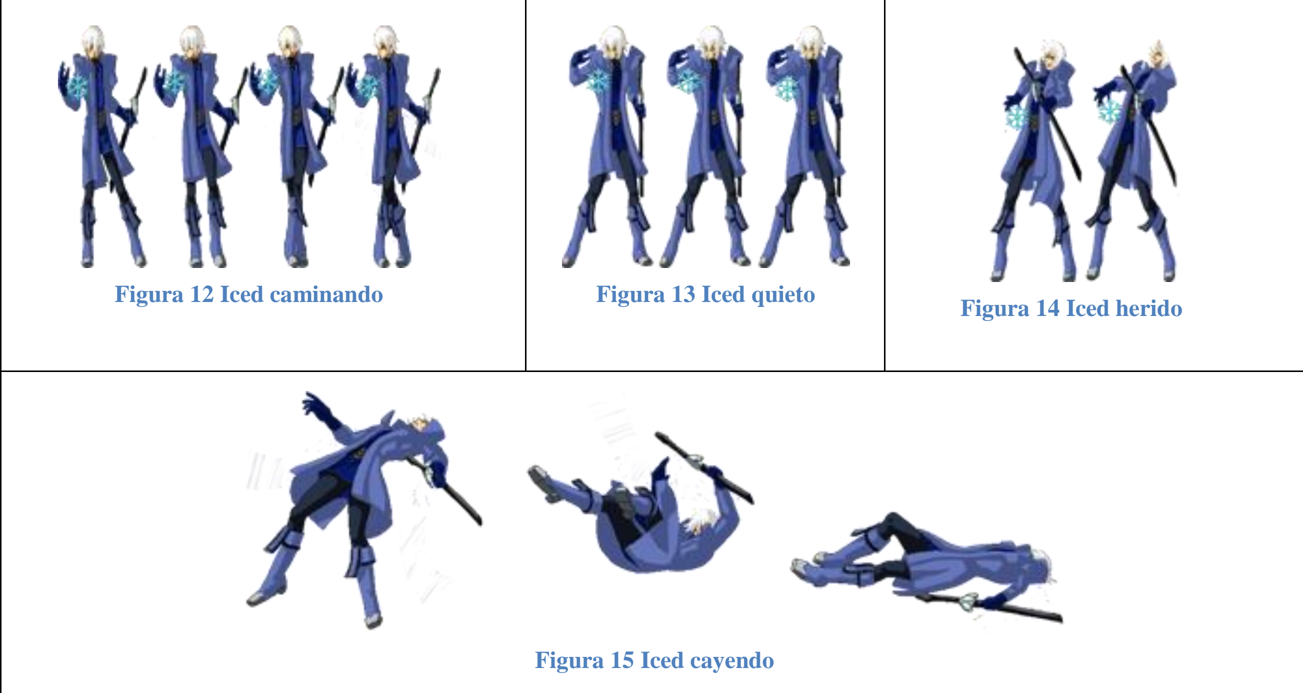

Figura 12 Iced caminando

Figura 13 Iced quieto

Figura 14 Iced herido

Figura 15 Iced cayendo

En el caso de la torre de fuego y hielo tenemos los siguientes sprites:

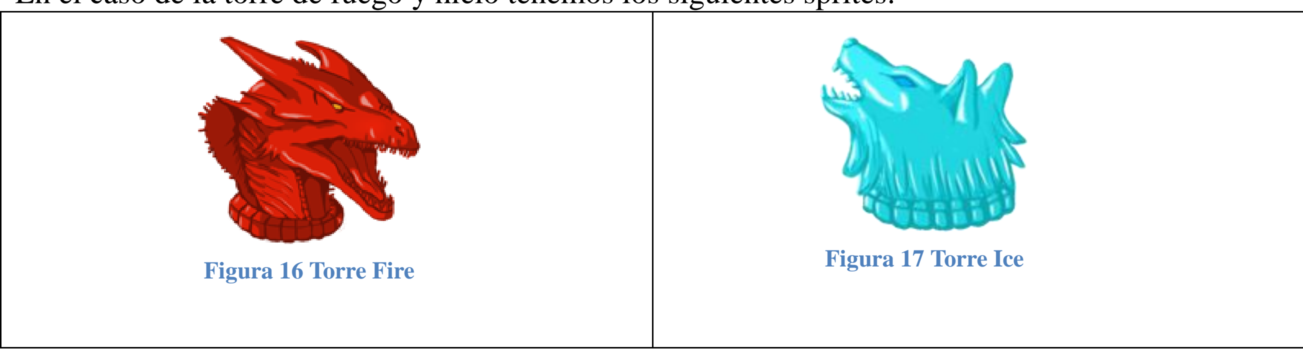

Figura 16 Torre Fire

Figura 17 Torre Ice

Y finalmente para las bases de fuego y hielo los elementos usados son los siguientes:

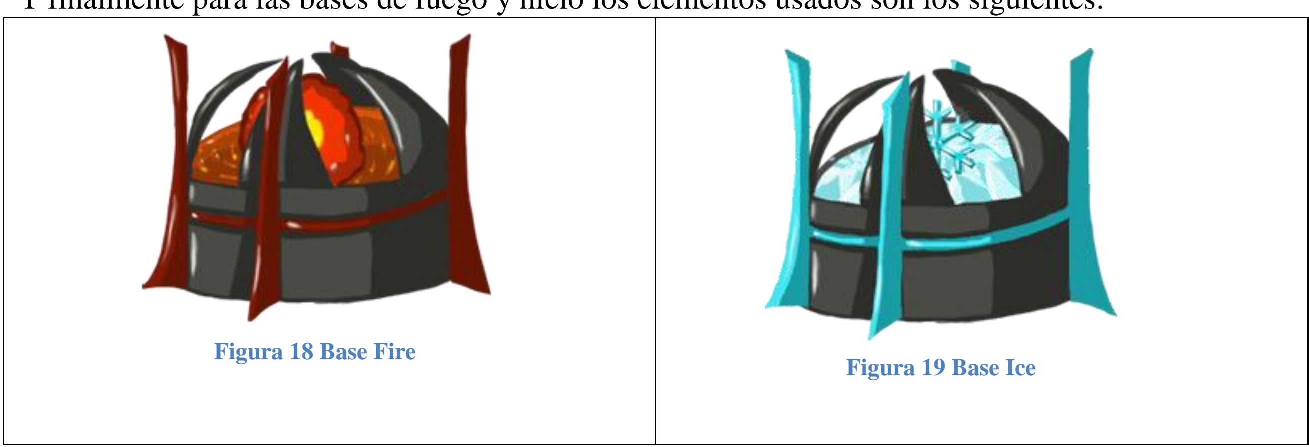

Figura 18 Base Fire

Figura 19 Base Ice

5.1.3 Mapeado

Para el mapeado se dividió en nueve sectores, el sector central será donde en principio jugarán los jugadores pero debido a que podremos alejar y acercar la cámara será necesario cubrir tanto bandas

superiores, inferiores como laterales, quedando de la siguiente manera:

Figura 20 Mapeado

La temática del juego como era el enfrentamiento entre fuego y hielo queda reflejado en cada mitad del mapa donde tenemos sus bases y torres correspondientes.

Hay más elementos en lo referente a grafismo como son las propias habilidades de los personajes, las explosiones o los disparos de las torres pero los elementos más importantes han sido reflejados en las imágenes anteriores.

5.1.4 Interfaz gráfica

Todo esto sería lo referente a diseño gráfico del propio juego en sí, pero tenemos más elementos que requerirán de un trabajo como grafista, nos referimos a la propia portada o a los de la interfaz.

Para el diseño de la portada y de las letras se eligió un diseño con dos elementos contrarios ya que se trata de un juego de confrontación y hay pocas cosas más antitéticas que el fuego y el hielo, por ello ya en la primera pantalla dividida en dos zonas vemos ese enfrentamiento.

Figura 21 Portada IceFire

Para la selección de jugadores tenemos la siguiente UI donde se ve cada personaje en una pose pasiva así como una descripción del personaje en sí y de sus habilidades:

Figura 22 Selección de Firewind

Figura 23 Selección de Iced

Por otra parte para el diseño de la UI se optó por elementos sencillos, sin recargar la pantalla de forma excesiva, se podrían haber puesto más menús o un mini mapa pero debido al reducido tamaño de éste se vio innecesario.

Para la UI del juego tenemos los retratos de cada personaje además de los dibujos de sus skills para reflejar si están activas o no:

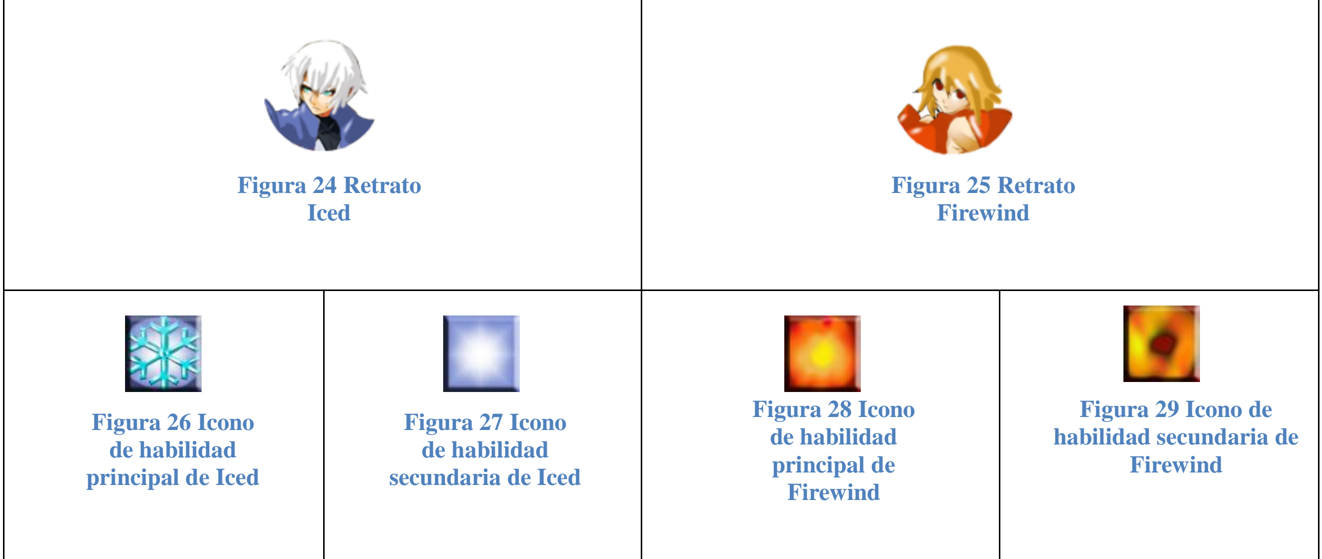

Figura 24 Retrato Iced

Figura 25 Retrato Firewind

Figura 26 Icono de habilidad principal de Iced

Figura 27 Icono de habilidad secundaria de Iced

Figura 28 Icono de habilidad principal de Firewind

Figura 29 Icono de habilidad secundaria de Firewind

5.1.5 Diseño gráfico del conjunto

El resultado final en el juego queda de la siguiente manera:

Figura 30 Pantalla de juego

Como se observa se intenta una integración tridimensional con algunos efectos entre el escenario y los personajes como el que las rocas del lago queden por delante lo que requirió poner algunos sprites a parte de los escenarios.
El grafismo por el que se ha optado es el propio de algunos de los **MOBAS** que triunfan actualmente.

6 IMPLEMENTACIÓN DEL PROYECTO

6.1 Introducción

En la realización del proyecto IceFire necesitaremos una serie de clases básicas que en prácticamente todos los juegos realizados en XNA tendremos y a partir de ahí algunas clases exclusivas de nuestro juego referente a la propia mecánica del juego.

Estas clases comunes se refieren a las que se ocupan de cambiar las screens o de la administración online que de una manera u otra siempre será parecido pero también necesitaremos otras que se ocupen de los héroes y sus habilidades que serán más exclusivas.

El análisis del diseño y codificación lo realizaremos de la siguiente forma, hablaremos primeramente de esta serie de clases más generales y reutilizables en otros proyectos de videojuegos para ir pasando a las clases más exclusivas y con solo validez en nuestro MOBA.

Otro punto importante antes de empezar las explicaciones de diseño y codificación es hablar de clases paralelas, como se explicó previamente el proyecto en sus primeras fases era offline y siempre estuvo en mente realizar un modo desconectado ya sea de práctica o de un jugador. En esas fases se diseñaron clases referentes a los personajes. El problema viene que para la parte online tenemos que ir accediendo a la información obtenida por el network sesión y prácticamente todas las clases en el modo online requieren de esa información, por tanto por lo que se optó es en un cierto paralelismo de las clases, tenemos clases que controlan los héroes online y otras que controlan los heros offline. En principio fue la mejor solución posible ya que de cara al diseño evolutivo se pudo dejar sin modificar la parte offline y trabajar seguidamente en la online con unos parámetros diferentes.

Por poner un ejemplo de la ventaja de esto, hay una cierta habilidad de un personaje que hace invisible a uno, en el modo offline, esta habilidad prácticamente no tiene dificultad ya que es un modo práctica y por tanto no tiene "efecto", sin embargo en el modo online se trata de que si nosotros hacemos la invisibilidad nos seguiremos viendo en nuestra pantalla mientras que el enemigo no podrá vernos en la suya lo que implica que enviemos en los paquetes de información a los jugadores si el personaje es visible o no para cada jugador. El tener que diseñar esta habilidad de forma genérica para modo online y offline seguramente sería más difícil.

Hay otras facilidades gracias a esto como es el tema de que para entrar en el modo online tenemos que crear sala y una serie de pasos previos que en el modo offline no, sea como sea el diseño en su etapa correspondiente se decidió hacer así con las ventajas y desventajas que conlleva.

Otros puntos que hay que aclarar antes de proseguir es que hay disponible mucha información mediante tutoriales en Internet, ya sea de parte de Microsoft o de parte de usuarios que intentan enseñar parte de sus conocimientos a la gente que se inicia en XNA. En la sección de Bibliografía ya se hace mención de esta información pero quería aquí citar dos tutoriales que me ayudaron mucho en los inicios y en el diseño inicial del juego.

El primero que quiero citar es el GameStateManagement de la propia Microsoft, en el que se puede ver un buen diseño para crear las clases necesarias para crear menús y el propio juego con su opción de pausa.

Tiene una versión ampliada que es la NetWorkState Management donde se puede ver un buen diseño para la realización de un videojuego con Lobby y con salas.

Los dos ejemplos no explican cómo hacer un juego y no entran en detalles de algunos puntos como por ejemplo en el caso del NetWorkStateManagement sobre el envió de paquetes y la actualización del juego. De todas formas ambos ejemplos son excelentes para empezar a trabajar en XNA ya que se puede ver un buen diseño y ver que clases pueden ser necesarias para trabajar a partir de ahí.

Otros ejemplos de mención y estudiados durante la realización del proyecto son los siguientes:

Creating A 2DSprite: Ejemplo muy simple de cómo poner un sprite 2D en pantalla, realmente es el primer ejemplo importante y el primer paso por el que empezar.

AnimatedSprite_Sample: En nuestro caso al realizar el juego con sprites en 2D necesitaremos animación en varios puntos, el saber cómo actualizar y activar estas animaciones es muy importante y con este ejemplo podemos aprender una parte.

PeerToPeerSample: En este tutorial se introduce en el tema de envío de paquetes a diferencia del NetWorkStateManagement. Es la base para poder entender cómo funciona la actualización en cada máquina cuando recibe un paquete.

PlatformerSample: Este ejemplo fue mirado en sus inicios por contener varios puntos de interés, se trata de un ejemplo de un juego de plataformas, tiene animaciones en sus personajes y colisiones entre sprites que son algunas de las claves en los juegos 2D además de funcionar a partir de una física basada en velocidades.

SpaceWar: Es un juego completo, un enfrentamiento entre naves en el que se pueden crear salas y unirse varias personas. Contiene también efectos gráficos interesantes, el único problema es que carecía de versión 3.1 y con la desaparición del soporte a la 3.0 su utilidad fue desapareciendo.

RolePlayingGame: Se trata de otro juego completo en este caso del género rol, uno de los puntos de interés en este juego era el tema de los stats y cómo manejarlos.

Durante el desarrollo del proyecto se visitaron muchos tutoriales y se estudiaron aun más ejemplos pero sin duda los citados anteriormente fueron los que tuvieron más impacto en el proyecto.

6.2 Clases generales

El tipo de clases más generales son las que controlan las pantallas de juego.

Tenemos por una parte la clase abstracta GameScreen, es el tipo de clase general de las que heredaran todas las demás pantallas.

Seguidamente como clases herederas tenemos varios tipos de Game Screen: **LobbyScreen, MenuScreen, NetworkBusyScreen, MessageBox, SelectionHeroScreen, BattleScreenOnline,** [...]

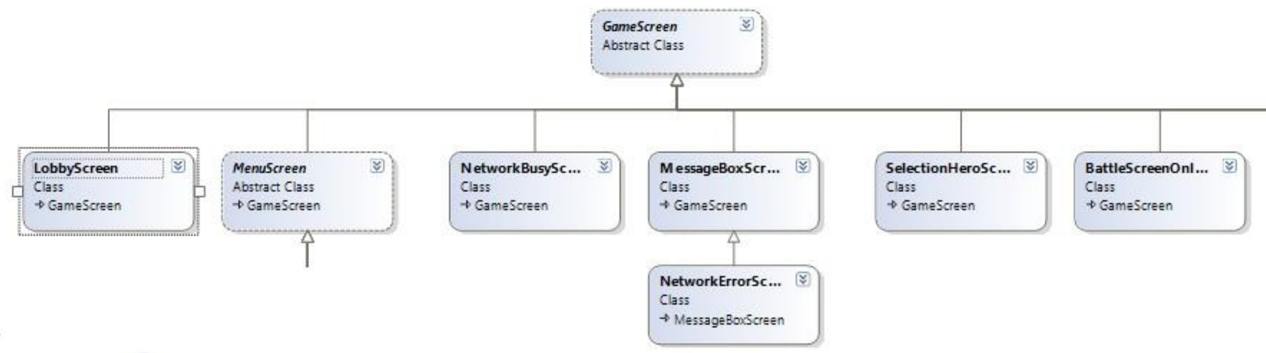

Figura 31 Diagrama 1 de clases de IceFire

[...], BackGroundScreen, GameplayScreen, SelectionHeroScreenOnline, ProfileSignScreen, LoadingScreen y WinMenuScreenOnline.

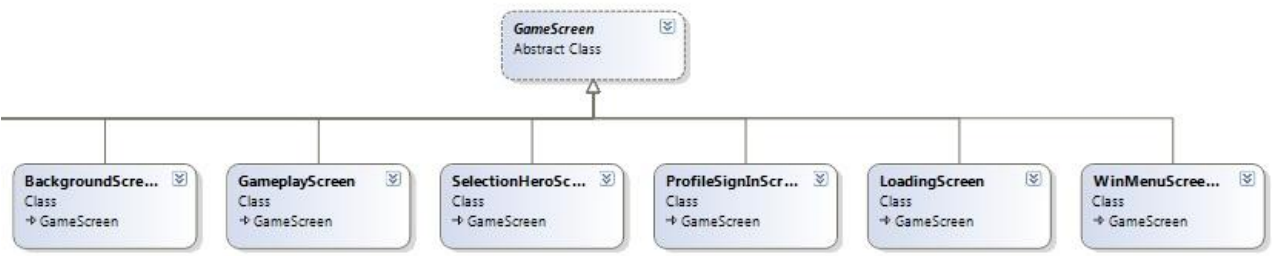

Figura 32 Diagrama 2 de clases de IceFire

De este tipo de clases de screens, podemos hablar de screens para la versión online y otras para la versión offline. En concreto sería:

- En ambas versiones: **MenuScreen, LoadingScreen**
- versión offline**: SelectionHeroScreen,GameplayScreen**
- versión online: **Lobby Screen, NetworkBusyScreen, ProfileSignScreen, MessageBox, SelectionHeroScreenOnline, BattleScreenOnline.**

De la clase Menu Screen se hablará con más detalle más adelante pero básicamente estamos hablando de una clase abstracta que representa a todas las pantallas en las que aparece un menú por ejemplo el menú inicial para elegir modo de juego opciones, menú de opciones, etc. Loading Screen por su parte es también una clase muy sencilla, es el tipo de GameScreen que aparecerá cuando hacemos algún tipo de carga, por ejemplo después de crear sala al crear los elementos de la pantalla de juego.

Una vez comentadas las clases compartidas en ambas versiones hablaremos de las clases de la versión offline. En concreto tenemos dos referentes a pantallas del propio juego.

SelectionHeroScreen no es un tipo de pantalla que podemos encontrar en todo juego pero si en prácticamente en todos los MOBA o cualquier juego donde haya selección de personajes, se trata como no de la pantalla que nos aparecería cuando tenemos que seleccionar las héroes disponibles. En nuestro caso se trata de un puntero que podemos mover por la pantalla y al pulsar el botón de confirmar encima de un héroe lo seleccionamos.
Podemos ver un ejemplo en la siguiente imagen:

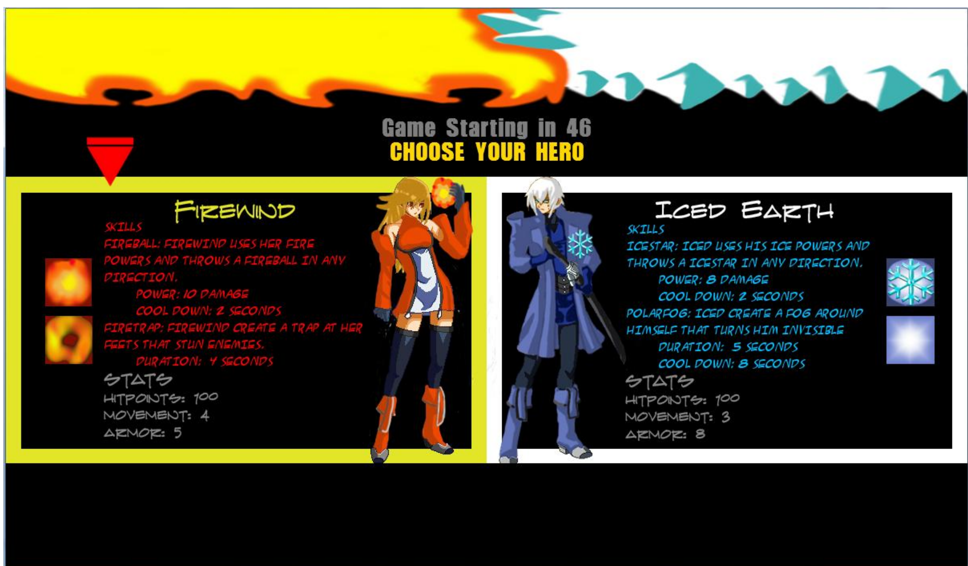

Figura 33 Pantalla de Selección de Héroes

Como último dato citar que solo selecciona un jugador, se trata del modo offline, en el caso del personaje enemigo que aparecerá en el modo práctica se selecciona aleatoriamente.

La otra clase del modo offline es GamePlayScreen, como ya dice el nombre es la pantalla del propio juego, la pantalla que aparece cuando jugamos. Es un tipo de clase general que en la que dentro podríamos definir desde un tetris a un juego de coches. Dentro de ella hace la llamada a las clases y métodos creados para poder definir nuestro videojuego.

En el caso del modo online tenemos las siguientes clases:

La clase Lobby Screen es la encargada de definir los parámetros de la pantalla que gestiona las salas de conexión a internet. Cuando un usuario quiera crear o unirse a una sala lo hace a través de la Lobby Screen.

Un ejemplo de esta pantalla sería:

Figura 34 Pantalla de Lobby

Esta clase es la que aparecerá cuando el usuario necesite autenticarse en los servidores de Microsoft, usando la clase ProfileSignScreen veremos si puede usar el servicio LIVE.

MessageBox será el modelo para los mensajes que puedan aparecer entre los menús, por ejemplo accediendo a la red o mensajes por el estilo.

SelectionHeroScreenOnline es básicamente el mismo tipo de pantalla que tenemos en el modo offline pero añadiendo el envió de paquetes de cada cursor a cada jugador mediante la Network Session.

BattleScreenOnline es el equivalente al GamePlayScreen del modo offline, es la pantalla que controla el juego en sí. Tiene métodos más complejos además de usar las clases de los modos online ya que tiene que controlar el envió y recepción de paquetes y actualizar de forma adecuada dependiendo de la información de éstos.

Algunas de las clases citadas aquí en el proyecto final no se usan, siguen en el proyecto porque se podrían utilizar de cara a futuras líneas de expansión pero en la versión final presentada no se usan.

Las clases no utilizadas son BackGroundScreen y WinMenuScreenOnline.

La clase BackGroundScreen es una clase que puede crear un efecto de scroll, el scroll es cuando el fondo de pantalla avanza, es una especie de animación del fondo. Cuando se estuvo diseñando el tipo de movimiento de la cámara del jugador fue una de las maneras que hubo de proporcionar efectos de movimiento en los jugadores, finalmente con el diseño de la clase Camara2D no hizo falta aunque se podría añadir por ejemplo para hacer scroll de alguna imagen o de algún tipo de título o frase que pasase de una dirección a la contraria.

Por otra parte la clase WinMenuScreenOnline era una pantalla que aparecería cuando se acababa un juego online. En las fases finales se decidió poner una frase dentro del propio juego diciendo quien ha ganado al estilo de otros MOBAS como Heroes of NewErth, pero en el caso de que se prefiriese por ejemplo poner una imagen dependiendo del equipo que ganase con esta clase se podría hacer.

A continuación pasaremos a hablar con más detalle de las MenuScreen. Las pantallas de menú tienen una gran importancia, hacen falta varias pantallas para ir seleccionando los modos de juego que quiera el usuario por tanto su creación es necesaria.

En el caso del proyecto tenemos las siguientes pantallas de menú:

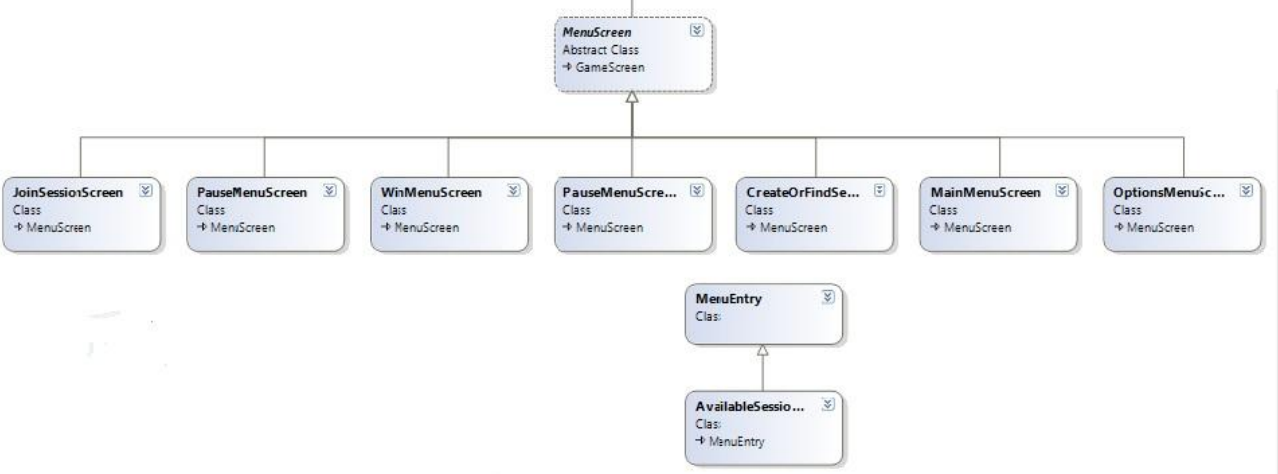

Figura 35 Diagrama 3 de clases IceFire

Referente a las clases relacionadas con el modo online tenemos:

CreateOrFindSession: Esta pantalla aparecerá cuando queramos crear o buscar a una sesión online, llamará a los métodos correspondientes para poder crear la sala o buscar una. Un ejemplo de esta MenuScreen la tenemos en la siguiente imagen:

Figura 36 Pantalla de sala creada

JoinSessionScreen: Esta tipo de pantalla es usada cuando queremos unirnos a una sesión online, es la encargada de llamar a los métodos necesarios para realizar la conexión correspondiente entre las dos máquinas. Un ejemplo de esta sala la tenemos en las siguiente imágenes:

Figura 37 Seleccionamos Sala

Figura 38 Entramos en Sala

Al elegir Join Seasson nos aparecerá la primera pantalla con las salas disponibles, veremos el nombre del Host, el número de jugadores teniendo en cuenta que el máximo son dos por sala, y la latencia que tendremos.

Si seleccionamos esa sala aparecerá la siguiente pantalla donde veremos el nombre de los jugadores y entre paréntesis quien es el host. En este momento ya está realizada la conexión y se pueden dar servicios de LIVE como es el de hablar vía micrófono. Cuando ambos jugadores den al botón de confirmar empezará la partida.

Mención especial en este sentido a la clase abstracta MenuEntry que será la clase de la cual AvailableSessionEntry heredará los métodos para añadir su entrada a un menú. Es decir cuando

hacemos una búsqueda de sesiones online y se nos tienen que ir añadiendo al menú esas entradas lo hacemos a partir de esta clase.

Sobre las demás clases de menú tenemos:

MainMenuScreen es la clase base de los menús, el primer menú que nos aparecerá al iniciar el juego, el menú principal donde podremos elegir modo: versión offline, Live o acceder a las opciones disponibles.

Figura 39 Pantalla de Selección Inicial

OptionsMenuScreen es la clase que genera el menú de opciones, en ella podremos elegir modo de dificultad y subir o bajar el volumen de base del juego.

Figura 40 Pantalla de Selección de Opciones

La importancia de PauseMenuScreen es muy alta, es la encargada de pausar el juego y que al reiniciarlo todo siga igual. Por lo tanto hay que guardar la información para que una vez des pausado el juego no haya problemas.

Figura 41 Pantalla de Pausa

WinMenuScreen esta clase controla el menú que aparece cuando en el modo offline se gana una partida, pudiendo empezarla otra vez o ir al menú principal.

Figura 42 Pantalla de Victoria

Con esto acabamos las MenuScreen, prosiguiendo con las clases que nos quedaron en el tintero, el otro gran bloque que veremos a continuación se refiere a algunas de las más importantes, la propia clase que genera el programa, el controlador de las Screens y el controlador de las conexiones online.

Junto a ellas comentaremos clases no utilizadas o de difícil ubicación en alguno de los bloques anteriores.

En la imagen de a continuación se puede ver el diagrama de clases de ellas:

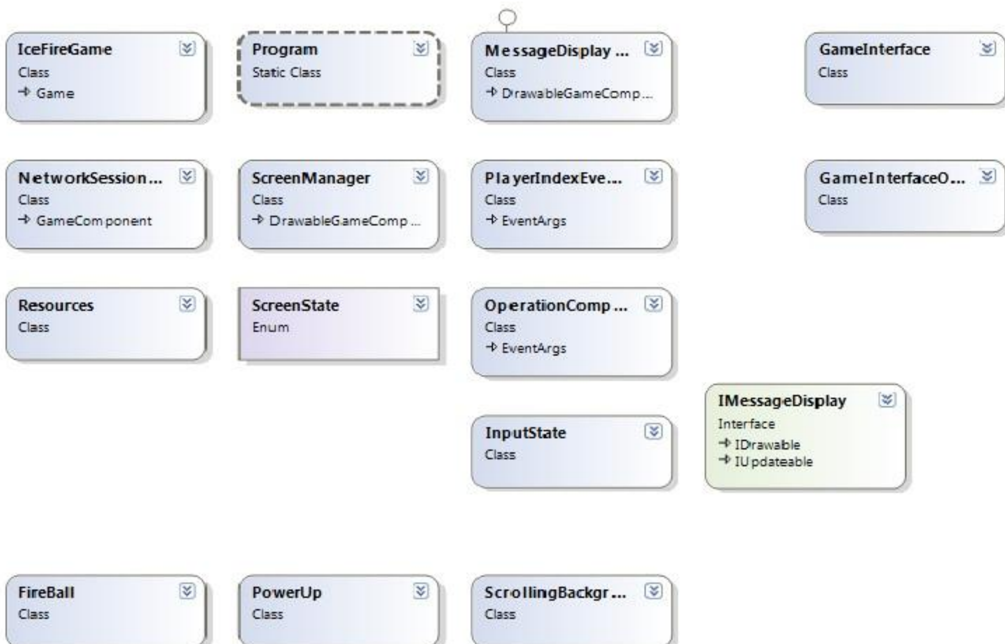

Figura 43 Diagrama 4 de clases IceFire

La clase Screen Manager es como las otras relacionadas en las screens basadas en el tutorial del GameStateManagement para poder manejar fácilmente la creación de nuevas imágenes de juego.

El concepto de screen es lo que veremos en pantalla, por ejemplo una pantalla de menú (MenuScreen) o la propia pantalla de juego (GameplayScreen), el screen manager es el encargado de gestionar estas screens y tiene una serie de métodos para poderlo realizar con facilidad.

Este gestor es del tipo DrawableGameComponent, es decir un componente del juego que puede ser dibujado y consiste en grandes rasgos en un gestor de listas de gamescreens y listas de gamescreens

para actualizar, tiene otro tipo de listas pero estas dos son las más importantes en lo referente a los juegos.

Se encarga de controlar que pantalla es la prioritaria y tiene métodos como el AddScreen que es para añadir pantallas a estas listas o el RemoveScreen para eliminarlas.

Una función extra es para las pantallas en transición el FadeBackBufferToBlack que hace que la pantalla pase a negro de forma progresiva y va perfecta para por ejemplo pantallas de cargas entre diferentes Screens.

El ScreenState se trata de un enum con las siguientes posiciones dentro de la clase GameScreen: TransitionOn, Active, TransitionOff, y Hidden. La primera posición y la tercera son en el caso de transiciones entra pantalla, la segunda para saber si esta activa y la última el caso de que estuviese oculta.

NetworkSessionComponent es en parte la clase que permite realizar las conexiones a internet, pero se hablará con detalle de ella y de cómo realizar estas conexiones junto con el funcionamiento online del juego en el apartado posterior de Conexión Online.

Las clases GameInterface y GameInterfaceOnline son las encargadas de realizar las pantallas de interfaz tanto en el menú de selección de personajes como en la propia partida en sí.

Los componentes de este interfaz pueden ser desde simples textos a imágenes de los retratos de los jugadores, de sus habilidades o las barras de vida de los personajes, torres y bases.

Para tratar todo este diferente de objetos esta clase tiene diferentes métodos para ello.

El método GenerateBar tiene el siguiente código:

```
public void GenerateBar(int health, Vector2 posicion, Texture2D textura, SpriteBatch batch, int equipo)
    {
        Color colorBarra = Color.White;
        if (equipo == 0)
        {
            colorBarra = Color.LightBlue;
        }
        else
        {
            colorBarra = Color.Red;
        }
        batch.Draw(textura, posicion, new Rectangle(TexturaHealth.Width / 2, textura.Height, textura.Width, 25),
Color.Gray, 0, new Vector2(0, 0), 1, SpriteEffects.None, 1);
        batch.Draw(textura, posicion, new Rectangle(TexturaHealth.Width / 2, textura.Height, (int)(textura.Width *
((double)health / 100)), 25), colorBarra, 0, new Vector2(0, 0), 1, SpriteEffects.None, 1);
    }
```

Básicamente consiste en crear un rectángulo de color azul o rojo encima del elemento (personaje, torre o base) y debajo de este otro rectángulo de color gris, al ir perdiendo vida el rectángulo rojo o azul ira haciéndose más pequeño quedando a la vista el gris produciendo el efecto de pérdida de vida.

El método Draw hace distintas cosas, desde imprimir texto con código del estilo:

```
batch.DrawString(gameFont, "RED TEAM WINS!!", posVictory, Color.Red);
```

Las fuentes es un archivo que tenemos que integrar en el programa del tipo gamefont,

posteriormente ya sólo restará poner lo siguiente:

```
private SpriteFont gameFont;
```

Otro punto de interés en el interfaz es por ejemplo hacer que diga el nombre del Nick del jugador conectado a los servidores LIVE, lo haremos con:

```
foreach (NetworkGamer gamer in networkSession.AllGamers)
        {
          if (gamer.IsHost == true)
          {
             nombre1 = gamer.Gamertag;
          }
          else
          {
             nombre2 = gamer.Gamertag;
          }
        }
```

de esta forma obteniendo el gamertag podemos decir el nombre del jugador.

La clase Power Up es una clase implementada que finalmente no se usa, en las primeras versiones existían estos objetos, sprites que si algún personaje pasaba encima de ellos recibía mejoras de sus características, más adelante cuando el diseño progreso a un sistema más igualitario y menos aleatorio se eliminaron. Los métodos de los que disponía la clase eran los típicos de los elementos de juego, Load para cargar las texturas de los diferentes Power Up, una función Update que controla si algún personaje pasa encima realizando las mejoras correspondientes en los stats del héroe y una función Draw que se encarga de dibujar todos estos efectos. Como características curiosas de la clase decir que los Power Up iban apareciendo aleatoriamente por el mapeado para que los jugadores nunca supieran exactamente donde podían salir.

La clase Scrolling Background es tratada en la sección de gráficos posteriormente.

La clase FireBall que aparece aquí es una clase espejo de la clase Fire, utilizada para hacer tests o cambios sobretodo en la etapa de la mejora de la sincronía entre los disparos de los clientes conectados.

Para finalizar, la clase más importante de todas ya que es donde se inicia el programa, IceFireGame, ésta es la clase que hace la llamada a la creación del juego y donde se irán creando las clases necesarias.

```
public IceFireGame()
      {
        // GeneralData generaldata;
         Content.RootDirectory = "Content";
         graphics = new GraphicsDeviceManager(this);
         graphics.PreferredBackBufferWidth = LimiteX;
         graphics.PreferredBackBufferHeight = LimiteY;
         // Creamos el componente screen manager.
         screenManager = new ScreenManager(this);
         Components.Add(screenManager);
         Components.Add(new MessageDisplayComponent(this));
         Components.Add(new GamerServicesComponent(this));
         // Activamos las primeras screens.
         screenManager.AddScreen(new BackgroundScreen(), null);
         screenManager.AddScreen(new MainMenuScreen(), null);
         //Escuchamos para la invitación de notificación de eventos.
         NetworkSession.InviteAccepted += (sender, e)
```

```
            => NetworkSessionComponent.InviteAccepted(screenManager, e);
        }
```

Por tanto al código de esta clase crea el screenManger y los elementos que necesitaremos para la ejecución del programa.

La otra clase que podríamos llamar raíz es la clase Program, que como se observará a continuación es una clase estática que crea un objeto game del tipo de nuestra clase IceFireGame y lo arranca, por tanto está clase es la que inicia el programa.

```
static class Program
    {
        static void Main()
        {
            using (IceFireGame game = new IceFireGame())
            {
                game.Run();
            }
        }
    }
```

6.3 Clases del juego

A continuación vamos a hablar de las clases más relacionadas con la parte jugable en sí, que son las clases de los personajes, habilidades de cada uno o movimiento de la cámara.

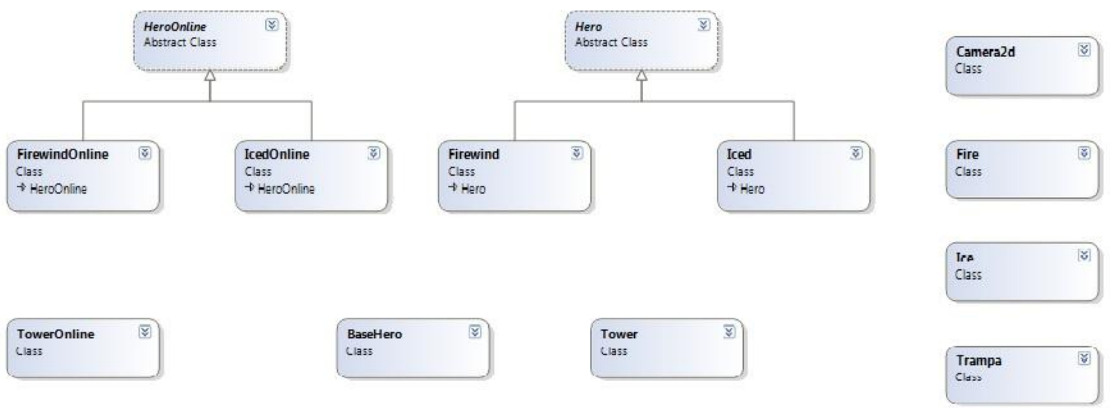

Figura 44 Diagrama 5 de clases de IceFire

Para empezar tenemos que hablar de las necesidades para el juego, hay que tener en cuenta que tendremos varios personajes, estos tendrán algunas características comunes como por ejemplo el que cuando les impacte alguna habilidad de otro personaje pase alguna cosa (bajar su vida, que se queden quietos, que les empujen hacia atrás) y por otra parte tendrán habilidades propias. Teniendo en cuenta esto tenemos dos opciones, hacer una clase por cada personaje y que no tengan ninguna relación entre sí, o lo que parece más obvio es que ya que hay ciertas características comunes necesitaremos de una clase padre de las que las herederas recibirán esas características comunes.

Esta clase tendría que ser una clase abstracta virtual, necesitamos definir un objeto que nos sirva para todos los personajes, pero además necesitábamos usar este objeto para los casos citados anteriormente, si impacta en algún objeto de esta clase que nos sirva para las demás clases

herederas.

A efectos prácticos, queremos esto para si se impacta en Firewind no tengamos que poner if impacto(firewind) == true, ya que implicaría por cada personaje hacer su caso lo cual nos haría agrandar el código, por ejemplo impacto(iced) == true, impacto(personaje_nuevo) == true, es mucho mejor usar esta clase impacto(hero)==true y a partir de ahí da igual si tenemos dos, tres o mil personajes herederos de esa clase.

Las clases Hero y HeroOnline son básicamente la misma clase para responder a la necesidad de que en la versión online el tratamiento es distinto debido a las networksession y por el tema de envío de paquetes.

Luego tenemos los dos personajes elegibles: Firewind e Iced.

Cada uno tiene características distintas, tanto de stats como de habilidades por lo tanto necesitamos de unas variables y métodos concretos para conseguir la mecánica de juego que necesitamos.

En principio en el caso de las clases de personajes como las de torre y base comentaremos con detalle solo las versiones online, las versiones offline básicamente son una versión simplificada que no utiliza la network sesión y con algunas diferencias en la implementación de habilidades.

Comencemos con detalle con la clase FirewindOnline. Este héroe tiene 2 habilidades distintas y sus propios sprites de personaje, interfaz, habilidades, retrato... Obviamente para cargar todo esto necesitaremos la función Load que estará dentro de la clase, de esta forma al crear un objeto de este tipo de héroe ya le cargaremos los sprites correspondientes. También dispone de un método Draw y de esta forma cada clase se ocupa de dibujar sus elementos.

Estas dos métodos ya los habíamos visto anteriormente en el primer ejemplo de un programa de XNA, los métodos propios son controles, donde esta función controla lo que realizan los jugadores tanto con un teclado o con el mando de X 360 y responde a sus necesidades.

Un detalle importante es en lo referente a las habilidades en sí, por ejemplo en el caso del disparo creamos un elemento de la clase Fire, en el caso de Iced se llamaría al método Ice. Al hacer esto, cómo cada habilidad tiene su propia clase, sería posible crear nuevos personajes con habilidades de otros fácilmente.

El método Update es el más importante, aquí cada héroe actualiza sus acciones, si impacta con un héroe, torre o base, si pone una trampa que el otro jugador quede bloqueado y básicamente todo lo que necesite algún tipo de comportamiento por parte de los héroes estará en esta parte de código, probablemente la más importante de la clase.

Finalmente la función de respawn del héroe que le permitirá revivir cuando éste sea eliminado.

En el caso de la clase IcedOnline básicamente es una clase muy parecida a Firewind, la principal diferencia a parte de la carga de diferentes texturas es la segunda habilidad, que en el clase de Firewind es una trampa que atrapa al enemigo y en el caso de Iced es volverse invisible.

Para hacer esta habilidad lo que hay que realizar es comprobar si el objeto de la clase Iced que esta invisible es el personaje que controla el jugador o no, en el caso de que si lo sea pues verá al personaje, y en el caso de que no lo sea desaparecerá. Para ello con un booleano que nos diga si esta visible o no y comprobando si es el jugador local será suficiente.

El código extraído de BattleScreenOnline en la parte de draw:

```
if (player2.Is_Visible == true)
         {
            //dibujamos heroe2
            player2.Draw(spriteBatch);
            //dibujamos  barras de energia
            //heroe2
            posicionBarra.X = player2.Posicion.X;
            posicionBarra.Y = player2.Posicion.Y - 50;
            spriteBatch.Draw(gameinterfaz.TexturaHealth, posicionBarra, new Rectangle(player2.Textura.Width / 2, player2.Textura.Height, player2.Textura.Width, 25), Color.Gray, 0, new Vector2(0, 0), 1, SpriteEffects.None, 1);
            spriteBatch.Draw(gameinterfaz.TexturaHealth, posicionBarra, new Rectangle(player2.Textura.Width / 2, player2.Textura.Height, (int)(player2.Textura.Width * ((double)player2.Health / 100)), 25), Color.LightBlue, 0, new Vector2(0, 0), 1, SpriteEffects.None, 1);
         }
         else
         {
            if (localHero == player2)
            {
               player2.Draw(spriteBatch);
               //dibujamos  barras de energia
               //heroe2
               posicionBarra.X = player2.Posicion.X;
               posicionBarra.Y = player2.Posicion.Y - 50;
               spriteBatch.Draw(gameinterfaz.TexturaHealth, posicionBarra, new Rectangle(player2.Textura.Width / 2, player2.Textura.Height, player2.Textura.Width, 25), Color.Gray, 0, new Vector2(0, 0), 1, SpriteEffects.None, 1);
               spriteBatch.Draw(gameinterfaz.TexturaHealth, posicionBarra, new Rectangle(player2.Textura.Width / 2, player2.Textura.Height, (int)(player2.Textura.Width * ((double)player2.Health / 100)), 25), Color.LightBlue, 0, new Vector2(0, 0), 1, SpriteEffects.None, 1);
            }
         }
```

La siguiente clase que vamos a tratar son las citadas anteriormente Fire e Ice. Ambas clases controlan los dos tipos de disparos de cada personaje. En muchos aspectos funcionan igual y tan solo cambian características como la velocidad y daño base de cada uno de los disparos que es distinto.

La parte importante de código de ambas clases es el método update donde tenemos:

```
public void Update()
      {
         if ((pos.X > limiteX1) || (pos.X <= limiteX2) || (pos.Y > limiteY1) || (pos.Y <= limiteY2))
            vivo = false;
         if (vivo)
         {
            pos.X += Direccion.X*10;
            pos.Y -= Direccion.Y*10;
         }
      }
```

El booleano vivo se actualiza si llega a los límites del escenario o si impacta el disparo con algún personaje.

Mientras esté vivo a cada instante el disparo ira progresando en las dirección que haya sido enviado.

Prosiguiendo con el repaso del diseño del programa, tenemos la clase Trampa, esta clase es otra habilidad de Firewind, una trampa que al ser pisada por algún personaje hace que éste se quede inmovilizado, el código básicamente es el mismo que el de un disparo de clase Fire o Ice

simplemente en esta ocasión no hay método update y la trampa es estática.

La parte más importante de código relacionada con esta clase está en la zona de la clase de FirewindOnline en el método Update donde tenemos:

```
if (Tramp != null && hero.State == true)
            {
                if (Tramp.Rect.Intersects(hero.Rectangulo))
                {
                    CuentaStun = 0;
                    hero.IsStun = true;
                    Tramp = null;//contacta la trampa
                }
            }
            CuentaStun += elapsed;
            if (CuentaStun >= TiempoStun)
            {
                hero.IsStun = false;
            }
```

Con éste código si algún héroe pasa por una trampa pasa al estado IsStun, se activa la cuenta de stun y una vez el contador iguale el tiempo establecido de la habilidad el jugador volverá a estar libre.

Prosiguiendo con las clases del programa tenemos la de Tower y TowerOnline, esta clase define las torres de nuestro juego, la labor de estas torres es defensiva, hasta que no se han destruido no se puede atacar a la base y además van disparando intentando de dar al jugador enemigo apuntando a su posición.

Los métodos de esta clase son Load donde cargaremos las textura y todo lo correspondiente a esta clase, el método Draw que nos dibujará las torres y método Update donde tenemos también el sistema de disparo. El sistema es el siguiente:

```
if (Equipo == 1)
            {
                if ((fire == null)&&(State == true))
                {
                    if (random.Next(1000) < probabilidadDisparo)
                    {
                        float m;
                        if (Posicion.X != hero.Posicion.X)
                        {
                            m = ((Posicion.Y - hero.Posicion.Y) / (Posicion.X - hero.Posicion.X));
                            if (Posicion.X > hero.Posicion.X)//torre por delante
                            {
                                DireccionDisparo.X = -1;
                                DireccionDisparo.Y = -m * DireccionDisparo.X;
                            }
                            if (Posicion.X < hero.Posicion.X)//torre por detras
                            {
                                DireccionDisparo.X = 1;
                                DireccionDisparo.Y = -m * DireccionDisparo.X;
                            }
                            if (DireccionDisparo.Y < -3)
                            {
                                DireccionDisparo.X = DireccionDisparo.X / 20;
                                DireccionDisparo.Y = DireccionDisparo.Y / 20;
                            }
                        }
                        else
```

```
                    {
                       DireccionDisparo.X = 0;
                       DireccionDisparo.Y = -1;
                    }
                    DireccionDisparo.X = DireccionDisparo.X / 3;
                    DireccionDisparo.Y = DireccionDisparo.Y / 3;
                   soundEffect3.Play();
                     fire = new Fire(new Vector2(pos.X + textura.Width, pos.Y + textura.Height/2), texturaFire,
DireccionDisparo);
                 }
              }
           }
```

El ejemplo es un disparo de una torre del equipo 1, es decir de fuego que lanzará disparos de la clase Fire, en este caso se trata de una ecuación de la recta para buscar al jugador enemigo y saber qué dirección tiene que tener el disparo para darle. Posteriormente se le rebaja la velocidad al disparo ya que los tiros de las torres son más potentes pero más lentos.

Si el disparo da en un héroe enemigo tenemos el mismo tipo de impacto que en las clases Firewind o Iced, se resuelve el impacto de la siguiente forma:

```
//comprobamos si algún disparo fuego da en heroe
           if (Fuego != null && hero.State == true)
           {
              if (Fuego.Rect.Intersects(hero.Rectangulo))
              {
                 soundEffect7.Play();
                 Fuego = null;
                   hero.Health -= ((Attack)*7 – (hero.Defence));
                 hero.Health = (int)MathHelper.Clamp(hero.Health, 0, 100);
                 if (hero.Health <= 0)
                 {
                    if (hero.Nombre == 0)
                    {
                       soundEffect2.Play();
                    }
                    else
                    {
                       soundEffect8.Play();
                    }
                    hero.State = false;
                    hero.Ndeaths++;
                 }
              }
           }
```

Comprobando si el rectángulo del disparo intersecciona en el rectángulo de la textura del héroe enemigo, hacemos las operaciones correspondientes d reducción de vida y los efectos sonoros que necesitan oírse.

La última función de interés de la clase Tower es DrawExplosion, que dibujará en caso de que la torre tenga la vida a cero la animación de una explosión.

La clase Base es la última en lo referente a clases de elementos del juego propiamente, esta clase controla el elemento que los jugadores tienen que proteger, un sprite que solo puede ser destruido cuando la torre defensiva de su equipo es eliminada. Por tanto se trata de un elemento pasivo, en el que solo tenemos que implementar que le baje la vida cuando es golpeado y destruido si no le

quedan más puntos de vida.

Teniendo en cuenta esto los métodos que tenemos son un Load para cargar las correspondientes texturas, un Update que nos mira si la vida de la base es cero para realizar la explosión o no, y una función Draw que nos dibujará la base que sea.

Con esto damos por acabado la visión general de la mayoría de clases de nuestro programa, a continuación veremos más concretamente algunos apartados como es la conexión online, la creación de gráficos, sonido y la colisión de sprites.

6.4 Conexión Online

6.4.1 .Implementación

Realizar una conexión entre dos máquinas utilizando los servidores LIVE tiene una ciertas peculiaridades y reglas, en este apartado vamos a hablar con detalle como realizamos la conexión en IceFire utilizando el código del ejemplo de PeerToPeer de Microsoft y hablando de la clase NetworkSession Component.

Antes de seguir habrá que citar dos peculiaridades de estas conexiones, se permite conectar a varios PC y a varios Xbox 360 utilizando este tipo de servidores pero sin embargo no se permite entre PC y Xbox 360. En el fondo como se ha explicado en la sección de compilación aunque compartan código se compilan de forma diferente y por tanto son dos versiones del mismo juego.

La otra peculiaridad es en lo referente a lo necesario para realizar estas conexiones. Usar XNA es gratuito, sin embargo, si intentamos realizar una conexión a internet entre dos PC utilizando el protocolo de XNA nos saltará un error en la compilación o al intentar conectar las dos máquinas diciéndonos que no se puede usar el component NetworkSession y dará un error de Content. Content es el apartado de nuestro programa donde se guardará todos los archivos que utilizaremos, como por ejemplo sprites, pero también es donde tenemos las XNA Frameworks.

Para poder utilizar el Content de conexión a internet necesitamos tener una cuenta Premium de XNA, entonces podremos realizar la conexión entre dos PC.

Si probásemos en este momento de realizar la conexión compilando el programa para nuestra Xbox 360 también nos daría un error, ya que en el caso de la X 360 además de que nuestra cuenta tiene que ser Premium, necesitamos tener el servicio Live Gold de pago de Microsoft

Una vez cumplido el tener cuenta Premium de XNA y servicios Live Gold podremos conectar dos X 360 y jugar con nuestro programa sino nos dará error de ejecución en nuestro PC vinculado o se quedará colgado el juego.

Citadas estas peculiaridades que afecta en lo referente a los tests del juego, vamos a proseguir explicando la conexión a internet que permite XNA.

Para poder conectarnos, necesitaríamos en nuestro código añadir primeramente lo siguiente:

```
NetworkSession networkSession;
PacketWriter packetWriter = new PacketWriter();
PacketReader packetReader = new PacketReader();
```

De esta forma crearemos una networkSession y un paquete tanto de escritura como de lectura. Los

paquetes que enviamos no se definen más, es decir no hace falta especificar su tamaño o el tipo de información que tendrá previamente, esto vendrá definido por el primer paquete que enviemos, es decir que el primero es el "default" y se considera que los siguientes paquetes serán iguales a este.

Por esto mismo el primer paquete tiene bastante importancia, además el tamaño de este paquete es recomendable que no sobrepase los 8Kb/s, típicamente el ancho de banda nos permitiría desde 12 a 250 kilobytes por segundo pero debido a que los anchos de banda proporcionados por las compañías no siempre son los reales y a estudios realizados con Halo3 donde el 99% de los usuarios tenían ese ancho de banda recomiendan que los paquetes no sobrepasen ese tamaño.

Prosiguiendo con el código, una vez creada la networksession tendremos que ir pasándola vía parámetro en las clases y funciones que la necesiten.

En el apartado de Update de nuestro programa tendremos que añadir lo siguiente:

```
if (networkSession == null)
        {
          // Si no tenemos una  network session, actualizar la pantalla de menu para crear o unirnos a una.
          UpdateMenuScreen();
        }
        else
        {
          //Si tenemos una nerwork session la actualizamos.
          UpdateNetworkSession();
        }
```

la función de MenuScreen podría ser de la siguiente forma:

```
void UpdateMenuScreen()
      {
        if (IsActive)
        {
          if (Gamer.SignedInGamers.Count == 0)
          {
            // Si no hay perfiles logeados, no se puede seguir. Enseñamos la pantalla de sign in
            Guide.ShowSignIn(maxLocalGamers, false);
          }
          else if (IsPressed(Keys.A, Buttons.A))
          {
            // Creamos una nueva sesión
            CreateSession();
          }
          else if (IsPressed(Keys.B, Buttons.B))
          {
            // Nos unímos a una sesión ya creada
            JoinSession();
          }
        }
      }
```

A destacar el tag “Gamer” proporcionado por las network Session, cada Gamer representaría un jugador, y tiene varias funciones propias de las que hablaremos con más profundidad más adelante.

Por otra parte la función para crear una sesión por internet:

```
void CreateSession()
      {
        DrawMessage("Creating session...");
        try
```

```
        {
            networkSession = NetworkSession.Create(NetworkSessionType.PlayerMatch,
                                      maxLocalGamers, maxGamers);
            HookSessionEvents();
        }
        catch (Exception e)
        {
            errorMessage = e.Message;
        }
    }
```

Con éste código crearíamos la sesión, a destacar un punto, en "NetworkSession.Create(NetworkSessionType [...] ", tenemos varios tipos de posibilidades de conexión, en nuestro caso hemos elegido PlayerMatch porque es la que permite conectar dos consolas o PCs vía internet con los servidores LIVE pero tenemos otras posibilidades que a continuación son explicadas:

- **Local:** Este tipo de network session no tiene ningún tipo de conexión a internet, pero nos posibilita realizar split screens fácilmente y poder jugador dos personas en el misma pantalla de forma paralela.

- **SystemLink:** Esta conexión es lo que en PC se conocería como LAN, permite conectar vía cable a dos consolas XBOX 360.

- **Ranked:** Este tipo de conexión está disponible para juegos que han sido presentados para ponerse a la venta y Microsoft da la aprobación, es decir para juegos comerciales, y posibilita hacer rankings de las partidas.

Visto las posibilidades queda claro que PlayerMatch es el tipo de sesión que necesitamos.

La función de unirse a una partida:

```
void JoinSession()
    {
        DrawMessage("Joining session...");
        try
        {
            // Buscamos sesiones.
            using (AvailableNetworkSessionCollection availableSessions =
                        NetworkSession.Find(NetworkSessionType.SystemLink,
                                    maxLocalGamers, null))
            {
                if (availableSessions.Count == 0)
                {
                    errorMessage = "No network sessions found.";
                    return;
                }
                // Nos unimos a la primera sesión que encontremos.
                networkSession = NetworkSession.Join(availableSessions[0]);
                HookSessionEvents();
            }
        }
        catch (Exception e)
        {
            errorMessage = e.Message;
        }
    }
```

En este caso buscaríamos sesiones y nos uniría a la primera sesión que encontrase. En nuestro juego esto es un poco más complejo ya que podemos elegir la sesión que queremos pero para ilustrar de forma práctica el funcionamiento de la networkSession se está utilizando estas funciones un pelín más simplificadas.

Lo siguiente que hay que explicar es que necesitaremos una función para ir actualizando la networkSession, está función será de la siguiente forma y será usada en el método Update general del juego actualizándose por tanto a cada instante:

```
void UpdateNetworkSession()
        {
           // Actualizamos los gamer Locales ( podemos tener 2 mandos), y enviamos la información.
           foreach (LocalNetworkGamer gamer in networkSession.LocalGamers)
           {
              UpdateLocalGamer(gamer);
           }
           networkSession.Update();
           // Miramos que la sesión no se haya acabado.
           if (networkSession == null)
              return;
           // Leemos cualquier paquete que nos de información.
           foreach (LocalNetworkGamer gamer in networkSession.Gamers)
           {
              ReadIncomingPackets(gamer);
           }
        }
```

El primer foreach es debido a que podemos tener varios jugadores locales por tanto primero actualizamos a cada Gamer y luego leemos los paquetes con la información de los otros jugadores de la red.

Por otra parte la función de ReadIncomingPackets es de la siguiente manera:

```
void ReadIncomingPackets(LocalNetworkGamer gamer)
        {
           // Seguimos leyendo mientras lleguen paquetes.
           while (gamer.IsDataAvailable)
           {
              NetworkGamer sender;
              // Leemos cada paquete de la network.
              gamer.ReceiveData(packetReader, out sender);
              // Descartamos los paquetes enviados por los jugadores locales
              if (sender.IsLocal)
                 continue;
              // Miramos el Heroe asociado con el que es enviado el paquete.
              Hero remotehero = sender.Tag as Hero;
              // leemos el estado de ese Hero del paquete de la red.
              remoteHero.info1 = packetReader.ReadVector2();
              remoteHero.info2 = packetReader.ReadSingle();
           }
        }
```

A destacar que en el caso del leer paquetes, hay que especificar el tipo de estructura o variable que vamos a leer, si es un Vector2, o Vector 3, un booleano o un int32.

De esta forma nos actualizamos a nosotros mismos y tenemos la información que nos envían los otros jugadores, pero claro nosotros también tenemos que enviar nuestra información, por ello será necesario:

```
void UpdateLocalGamer(LocalNetworkGamer gamer)
        {
            // Miramos que héroe está asociado para el jugador local.
            Hero localHero = gamer.Tag as Hero;
            // Actualizamos el heroe.
            ReadHeroInputs(localHero, gamer.SignedInGamer.PlayerIndex);
            localHero.Update();
            // Escribimos en el paquete dos informaciones.
            packetWriter.Write(info1);
            packetWriter.Write(info2);
            //Enviamos la información a cada uno de la sesión.
            gamer.SendData(packetWriter, SendDataOptions.InOrder);
        }
```

Como se puede ver en este código uno de los factores clave es la asignación de lo que leemos. Nuestra máquina lee paquetes recibidos, y tenemos una serie de elementos, por ejemplo tres personajes, ¿a quién asignamos el paquete recibido? Una posible forma de solucionarlo es usar el tag Gamer, todo jugador conectado a la red tiene una identidad gamer y es única, por tanto si al entrar en el juego hemos asignado un héroe con un tag gamer será fácil ponerle la información correspondiente.

En el caso del proyecto que nos ocupa, hubiese sido posible utilizar el tag Gamer, pero debido a la dificultad para testear con dos jugadores, tener 3 o más disponibles era inviable. Por tanto al limitar la partida a dos jugadores se optó por otro método de asignación, ver si el jugador es Host o no.

Con todo lo explicado anteriormente ya se podría hacer un juego con conexión online ya que básicamente los puntos importantes son crear una networkSession, enviar y recibir paquetes, y asignar correctamente estos paquetes a cada elemento de juego, estas son las claves de un juego online.

A continuación y tras las explicaciones pondremos una parte del código de esta parte para que se vean algunos de los cambios citados.

```
foreach (LocalNetworkGamer gamer in networkSession.LocalGamers)
        {
            if (gamer.IsHost == true)
            {
                localHero = player1;
            }
            else
            {
                localHero = player2;
            }
        }

void UpdateLocalGamer(LocalNetworkGamer gamer)
      {
            if (gamer.IsHost == true)
            {
                packetWriter.Write(player1.Posicion);
                packetWriter.Write(player1.IsStun);
                packetWriter.Write(player1.IsRight);
                packetWriter.Write(player1.DireccionDisparo);
                packetWriter.Write(player1.State);
                packetWriter.Write(player1.Health);
                packetWriter.Write(player1.Is_Firing2);
                packetWriter.Write(player1.Main_Skill);
                packetWriter.Write(player1.Second_Skill);
                packetWriter.Write(player1.Is_Firing);
                packetWriter.Write(player1.Is_Tramping);
                packetWriter.Write(tower1.Is_Firing);
                packetWriter.Write(tower1.Health);
                packetWriter.Write(base1.Health);
                packetWriter.Write(player1.Ndeaths);
```

```
                packetWriter.Write(player1.Nkills);
                packetWriter.Write(tower1.State);
                packetWriter.Write(base1.State);
                packetWriter.Write(player1.Is_Visible);
                packetWriter.Write(tower1.DireccionDisparo);
                packetWriter.Write(tower1.PosicionDisparo);
                packetWriter.Write(tower1.DisparoVivo);
                packetWriter.Write(tower2.Is_Firing);
                packetWriter.Write(tower2.DireccionDisparo);
                packetWriter.Write(tower2.PosicionDisparo);
                packetWriter.Write(tower2.DisparoVivo);
                packetWriter.Write(player2.Health);
                packetWriter.Write((int)player1.estadoAnime);
            }
            else
            {
                packetWriter.Write(player2.Posicion);
                packetWriter.Write(player2.IsStun);
                packetWriter.Write(player2.IsRight);
                packetWriter.Write(player2.DireccionDisparo);
                packetWriter.Write(player2.State);
                packetWriter.Write(player2.Health);
                packetWriter.Write(player2.Is_Firing2);
                packetWriter.Write(player2.Main_Skill);
                packetWriter.Write(player2.Second_Skill);
                packetWriter.Write(player2.Is_Firing);
                packetWriter.Write(player2.Is_Tramping);
                packetWriter.Write(tower2.Is_Firing);
                packetWriter.Write(tower2.Health);
                packetWriter.Write(base2.Health);
                packetWriter.Write(player2.Ndeaths);
                packetWriter.Write(player2.Nkills);
                packetWriter.Write(tower2.State);
                packetWriter.Write(base2.State);
                packetWriter.Write(player2.Is_Visible);
                packetWriter.Write(tower2.DireccionDisparo);
                packetWriter.Write(tower2.PosicionDisparo);
                packetWriter.Write(tower2.DisparoVivo);
                packetWriter.Write(tower1.Is_Firing);
                packetWriter.Write(tower1.DireccionDisparo);
                packetWriter.Write(tower1.PosicionDisparo);
                packetWriter.Write(tower1.DisparoVivo);
                packetWriter.Write(player1.Health);
               packetWriter.Write((int)player2.estadoAnime);

            }
void ReadIncomingPackets(LocalNetworkGamer gamer)
        {
            while (gamer.IsDataAvailable)
            {
                NetworkGamer sender;
                gamer.ReceiveData(packetReader, out sender);
                if (sender.IsLocal)
                    continue;
                    if (gamer.IsHost == true)
                    {
                        bool placeholder1 = false;
                        Vector2 placeholder2 = Vector2.Zero;
                        int placeholder3 = 0;
                        player2.Posicion = packetReader.ReadVector2();
                        player2.IsStun = packetReader.ReadBoolean();
                        player2.IsRight = packetReader.ReadBoolean();
                        player2.DireccionDisparo = packetReader.ReadVector2();
                        player2.State = packetReader.ReadBoolean();
                        player2.Health = packetReader.ReadInt32();
                        player2.Is_Firing2 = packetReader.ReadBoolean();
                        player2.Main_Skill = packetReader.ReadBoolean();
                        player2.Second_Skill = packetReader.ReadBoolean();
                        player2.Is_Firing = packetReader.ReadBoolean();
                        player2.Is_Tramping = packetReader.ReadBoolean();
                        tower2.Is_Firing = packetReader.ReadBoolean();
                        tower2.Health = packetReader.ReadInt32();
                        base2.Health = packetReader.ReadInt32();
                        player2.Ndeaths = packetReader.ReadInt32();
                        player2.Nkills = packetReader.ReadInt32();
                        tower2.State = packetReader.ReadBoolean();
```

```
        base2.State = packetReader.ReadBoolean();
        player2.Is_Visible = packetReader.ReadBoolean();
        tower2.DireccionDisparo = packetReader.ReadVector2();
        tower2.PosicionDisparo = packetReader.ReadVector2();
        tower2.DisparoVivo = packetReader.ReadBoolean();
        placeholder1 = packetReader.ReadBoolean();
        placeholder2 = packetReader.ReadVector2();
        placeholder2 = packetReader.ReadVector2();
        placeholder1 = packetReader.ReadBoolean();
        placeholder3 = packetReader.ReadInt32();
        int x = packetReader.ReadInt32();
        if (x == 0)
        {
            player2.estadoAnime = HeroOnline.EstadoAnime.quieto;
        }
        if (x == 1)
        {
            player2.estadoAnime = HeroOnline.EstadoAnime.caminando;
        }
        if (x == 2)
        {
            player2.estadoAnime = HeroOnline.EstadoAnime.disparando;
        }
        if (x == 3)
        {
            player2.estadoAnime = HeroOnline.EstadoAnime.herido;
        }
        if (x == 4)
        {
            player2.estadoAnime = HeroOnline.EstadoAnime.muerto;
        }
        if (x == 5)
        {
            player2.estadoAnime = HeroOnline.EstadoAnime.trampa;
        }

    }
    else
    {
        player1.Posicion = packetReader.ReadVector2();
        player1.IsStun = packetReader.ReadBoolean();
        player1.IsRight = packetReader.ReadBoolean();
        player1.DireccionDisparo = packetReader.ReadVector2();
        player1.State = packetReader.ReadBoolean();
        player1.Health = packetReader.ReadInt32();
        player1.Is_Firing2 = packetReader.ReadBoolean();
        player1.Main_Skill = packetReader.ReadBoolean();
        player1.Second_Skill = packetReader.ReadBoolean();
        player1.Is_Firing = packetReader.ReadBoolean();
        player1.Is_Tramping = packetReader.ReadBoolean();
        tower1.Is_Firing = packetReader.ReadBoolean();
        tower1.Health = packetReader.ReadInt32();
        base1.Health = packetReader.ReadInt32();
        player1.Ndeaths = packetReader.ReadInt32();
        player1.Nkills =packetReader.ReadInt32();
        tower1.State = packetReader.ReadBoolean();
        base1.State = packetReader.ReadBoolean();
        player1.Is_Visible = packetReader.ReadBoolean();
        tower1.DireccionDisparo = packetReader.ReadVector2();
        tower1.PosicionDisparo = packetReader.ReadVector2();
        tower1.DisparoVivo = packetReader.ReadBoolean();
        tower2.Is_Firing = packetReader.ReadBoolean();
        tower2.DireccionDisparo = packetReader.ReadVector2();
        tower2.PosicionDisparo = packetReader.ReadVector2();
        tower2.DisparoVivo = packetReader.ReadBoolean();
        player2.Health = packetReader.ReadInt32();
        int x = packetReader.ReadInt32();
        if (x == 0)
        {
            player1.estadoAnime = HeroOnline.EstadoAnime.quieto;
        }
        if (x == 1)
        {
            player1.estadoAnime = HeroOnline.EstadoAnime.caminando;
        }
        if (x == 2)
```

```
                {
                    player1.estadoAnime = HeroOnline.EstadoAnime.disparando;
                }
                if (x == 3)
                {
                    player1.estadoAnime = HeroOnline.EstadoAnime.herido;
                }
                if (x == 4)
                {
                    player1.estadoAnime = HeroOnline.EstadoAnime.muerto;
                }
                if (x == 5)
                {
                    player1.estadoAnime = HeroOnline.EstadoAnime.trampa;
                }
            }
        }
    }
```

Puntos a remarcar, en la primera función se ve la asignación de los héroes y como se había dicho se hace a través de ser el Host o no.

Por otra parte se observará que hay variables placeholders, esto es debido a que una vez empezados a enviar paquetes no se puede cambiar el tamaño de estos ya que se coge el diseño del primero como defecto tanto en tamaño como en posición del tipo de variables en él.

Tal como se plantea en esta conexión el Host tiene preferencia, y envía al otro jugador los datos de los disparos de las torres, vida de las bases etc., lo único que lee del huésped es hacía donde se mueve el movimiento del otro jugador y qué tipo de skill realiza, todo lo demás lo controla el anfitrión.

Con esto se consiguió eliminar ciertas asincronías que habían en los primeros prototipos.

6.4.2 Algoritmos de predicción

Este pequeño apartado se ha incluido por que se estudió en los últimos meses antes de la finalización, en ese tiempo aparecieron nuevos tutoriales y guías en la web oficial de XNA explicando un poco como realizar algoritmos de predicción, básicamente se trata de que el programa prediga hacía dónde irá un elemento y así el juego da la impresión de ir más fluido.

Por ejemplo, jugando a un juego tenemos información de los jugadores aunque sea información pasada y ella se puede combinar pare predecir hacia donde irá.

Esta predicción puede ser cierta o no, pero en el caso de que no sea cierta provocaría un salto en nuestro avatar y ese es el principal problema.

No se ha aplicado en la versión final, pero seguramente su aplicación junto con algunos cambios en cómo se realizan los disparos de los jugadores podría ser la clave para la completa sincronía entre los competidores de la partida.

6.5 Implementación de Gráficos

6.5.1 Introducción

Anteriormente hemos visto la parte artística de los gráficos, en parte la realización de esa parte va íntimamente ligada a el diseño de la programación de la parte gráfica, ya que como pensemos que haremos los gráficos influirá en la parte de cómo moverlos.

Para entender mejor esto lo más claro es la animación, en nuestro caso lo diseñamos mediante Sprite Sheets, pero podría hacerse perfectamente mediante frames en cada imagen, el hacerlo de nuestra forma lo que mejora es el numero de cargas que el programa realizará. Pero dependiendo del método elegido cambiará el algoritmo y el tratamiento de las imágenes por esa razón la parte artística con la programación de los gráficos van en simbiosis y se influencian.

6.5.2 TexturaAnimada

La clase TexturaAnimada se encarga de realizar las animaciones a través de un **Sprite Sheet**, el formato de creación de la clase es del tipo:

```
public TexturaAnimada(Vector2 pos, int frameCount, int framesPerSec)
```

Siendo pos, un Vector2 (dos dimensiones) donde habrá la X y la Y del recuadro que queremos mostrar, framecount es el número de frames en el que se divide la Sprite Sheet y framespersec será el parámetro que nos mostrará como de rápida queremos que sea la animación.

Lo que hace esta clase es dividir la Sprite Sheet en el número de frames que le hayamos dicho y pasarlos a la velocidad pedida. El método que realiza esto es:

```
public void UpdateFrame(float elapsed)
    {
        totalElapsed += elapsed;
        if (totalElapsed > timePerFrame)
        {
            frame++;
            //Los frames son circulares, al llegar al último volvemos al primero.
            frame = frame % framecount;
            totalElapsed -= timePerFrame;
        }
    }
```

Pasamos por parámetro el elapsed, que es el gametime del juego a cada instante.

Otro método interesante es el UpdateDeath, en la animación de la muerte necesitamos que haga la animación de caída y posteriormente se quede estática, con lo cual simplemente poniendo que una vez recorrido la animación no progrese el contador de frames quedándonos en el último frame del personaje en el suelo.

Otros métodos de la clase son el DrawFrame que consiste en dibujar el frame correspondiente de una animación:

```
if (IsRight == true)
        {
            batch.Draw(myTexture, position, sourcerect, Color.White,
                0, new Vector2(0, 0), 1, SpriteEffects.FlipHorizontally, 1);
        }
        else
        {
            batch.Draw(myTexture, position, sourcerect, Color.White,
                0, new Vector2(0, 0), 1, SpriteEffects.None, 1);
        }
```

IsRight es un booleano que nos dice hacia donde esta encarado el personaje si izquierda o derecha, aprovechando la potencia del método Draw haremos un giro de la imagen de ciento ochenta grados girándola gracias al SpriteEffects.FlipHorizontally.

6.5.3 Camara2D

Otra clase importante para la parte gráfica es la Camara2D que solucionó uno de los grandes problemas de la parte online. Cuando el juego estaba en la fase previa simplemente teniendo dos jugadores en la misma pantalla era suficiente, ya que se suponía que ambos jugadores tenían el mismo monitor o televisor, sin embargo, al posibilitar la conexión online y que los jugadores jueguen desde sitios distintos permite que cada uno vea cosas distintas, ampliar el campo de juego o crear habilidades relativas a la visibilidad.

Se trataba de lograr que la cámara siguiese el jugador local. Pero gracias al tutorial realizado por David Amador(el enlace a la página web está reseñado en el apartado de links) se pudo realizar esta cámara.

Realizarla no es tarea fácil básicamente se trata de crear una matriz de la imagen, y recolocar los pixeles cada vez que el personaje que es el centro se mueva, además de seguir al personaje permite también realizar zoom in y zoom out lo cual es perfecto para este tipo de juegos. El zoom out se limito al visto de serie ya que generalmente los jugadores juegan con el mayor campo de vista con lo que si diésemos más alcance que el normal los jugadores siempre al comenzar partida irían al máximo.

Tendremos varias variables para poder usar las transformaciones en la pantalla que necesitamos se trata de zoom, rotation y pos, en los dos primeros casos se trata de floats y en el último de un vector con la posición de la cámara y multiplicadores que usaremos a la hora de hacer los cambios.

Este es el método que obtiene la matriz transformada dependiendo de los valores de rotación, zoom y posición del centro de la cámara(en este caso nuestro personaje). Para ello usaremos los métodos proporcionados por la propia framework del Visual Studio de Matrix.

```
public Matrix get_transformation(GraphicsDevice graphicsDevice)
        {
            _transform =
              Matrix.CreateTranslation(new Vector3(-_pos.X, -_pos.Y, 0)) *
                                         Matrix.CreateRotationZ(Rotation) *
                                         Matrix.CreateScale(new Vector3(Zoom, Zoom, 1)) *
                                         Matrix.CreateTranslation(new Vector3(ViewportWidth * 0.5f, ViewportHeight * 0.5f, 0));
            return _transform;
        }
```

Una vez con esa matriz podemos utilizarla para crear la imagen con el spritebatch:

```
spriteBatch.Begin(SpriteBlendMode.AlphaBlend,
                  SpriteSortMode.Immediate,
                  SaveStateMode.SaveState,
                  cam.get_transformation(device));
```

6.5.4 Scrolling Background

Finalmente la última clase implementada en el apartado gráfico aunque no usada es ScrollingBackground, esta clase se creó para poder hacer scroll de la pantalla, era una de las posibilidades previas al uso de la Camara2D, la idea sería ir moviendo en una dirección u otra según se moviese el personaje, en parte es lo mismo que realiza la Camara2D a un nivel menos sofisticado y sin posibilidad de hacer zoom in y zoom out. Finalmente esta clase podría ser usada para por

ejemplo poner un mensaje que pase de punta a punta de la pantalla.

Los métodos de la clase son los siguientes:

Con el método Update vamos actualizando la posición de la variable screenpos que representa la posición de la imagen en la pantalla, se inicializa inicialmente a la mitad de la pantalla con:

```
screenpos = new Vector2(screenwidth / 2, screenheight / 2);
y pasamos por parámetro la posición del personaje con Pos
 public void Update(Vector2 Pos)
        {

            screenpos.X = Pos.X;
            screenpos.X = screenpos.X % mytexture.Height;
        }
```

Con lo cual cuando vayamos caminando irá avanzando o retrocediendo la pantalla según nuestra posición.

Del método Draw la parte importante es la siguiente:

```
            // Dibujamos la textura si aun está en la pantalla
            if (screenpos.X < screenheight)
            {
                batch.Draw(mytexture, screenpos, null,
                     Color.White, 0, origin, 1, SpriteEffects.None, 0f);
            }
            // Dibujamos la textura una segunda vez, detrás de la primera para crear el aspecto de ilusión
            batch.Draw(mytexture, screenpos - texturesize, null,
                 Color.White, 0, origin, 1, SpriteEffects.None, 0f);
```

6.6 Implementación del Sonido

6.6.1 Introducción

El sonido, ya sea la música o los efectos sonoros es un aspecto muy importante en los videojuegos actuales. El dar un buen ritmo el juego, el énfasis en algunos aspectos como explosiones, impactos o la ambientación en algunas zonas hacen de la música un aspecto muy importante.

Para el tratamiento de la música y de los efectos sonoros existen tres formas distintas, las tres han sido estudiadas y usados en el proyecto, aunque en el cliente final solo dos de ellas se utilizan.

6.6.2 MediaPlayer

Xbox 360 no deja de ser un dispositivo multimedia, no solo sirve para jugar sino que podemos ver películas, reproducir música u otras opciones. Básicamente este sistema se trata de utilizar las librerías que contralan el reproductor de música de la propia consola.

En la zona de LOAD pondríamos:

```
CombatMusic = content.Load<Song>("CombatMusic");
MediaPlayer.Play(CombatMusic);
```

Definiendo previamente Combatmusic como tipo music, simplemente cargamos el archivo de música mp3 que queremos y lo reproducimos. Tenemos luego opciones como que vaya repitiéndose la música continuamente con lo cual esta opción es ideal ahora la de crear la música general del juego

6.6.3 XTC

En las versiones previas de XNA 3.0 solo existía esta opción para crear la música, el XTC es el sistema de archivos sonoros internos de la consola, es su forma de crear sonido.

Para utilizarlo necesitamos utilizar archivos Wav que con un conversor a XTC llamado XNA Game studio Center pasan a ser XTC.

El XNA Game Studio Center nos da varias opciones para esos archivos, como hacerles mix o subirles el volumen de base, con lo cual para tratar a posteriori los efectos sonoros es una opción muy adecuada. Los problemas como se puede entender es que de base hay que tratar con Wav lo cual limita además de que el conversor tiene sus dificultades para usarse y reproducirse.

6.6.4 Reproducción de sonido

A parte del Mediaplayer, ya en XNA 3.1 y posteriores versiones se puede reproducir sonidos de forma relativamente sencilla. Esta es la opción que finalmente se eligió para los efectos sonoros debido a su gran facilidad y versatilidad en la utilización.

En el LOAD tendríamos definido soundEffect como efecto sonoro y soundname como la ruta al archivo de sonido:

```
SoundEffect soundEffect1;
string soundName1 = "Sounds/bigboom";

LOAD
soundEffect1 = content.Load<SoundEffect>(soundName1);
```

y luego cuando ocurra la acción deseada para que suene, por ejemplo un impacto de un disparo con un héroe haríamos:

```
UPDATE
soundEffect1.Play();
```

De esta forma tan sencilla se puede aplicar efectos sonoros en XNA tanto para PC como para consola. Las ventajas respecto al XTC es que puede reproducir más tipos de sonido no solo .Wav, no hay que hacer conversión y es bastante más sencillo de definir y utilizar.

6.6.5 Elección de música y efectos sonoros

Normalmente en la producción de un videojuego comercial se contratara a músicos profesionales para la realización de la música y efectos sonoros.

En el caso de IceFire se ha optado por buscar efectos sonoros en internet que estuviesen libres de licencia. Con lo cual todos los efectos sonoros son extraídas de páginas de ese estilo.

En lo referente a la música se ha optado por dos temas de 2 bandas comerciales disponibles en sus webs como demos de su música.

6.7 Colisiones entre Sprites

6.7.1 Introducción

Para el desarrollo del juego hay una parte crucial que es el tratamiento de las colisiones entre los

Sprites. Para la mecánica del juego tenemos unos personajes con habilidades que en ocasiones son disparos que hay que saber cuándo impactan y cuando no. Para el desarrollo del juego se trataron diferentes tipos de colisiones para diferentes mecánicas. El juego dispone de dos tipos de colisiones:

6.7.2 Intersects

Las propias librerías de XNA nos proporcionan una herramienta muy potente y son las posibilidades de tratamiento de polígonos, Este tipo de colisiones se trata de la siguiente manera, creamos un rectángulo del mismo tamaño que la textura-sprite que queremos tratar, y decimos que cuando ese rectángulo se interseccione con otro haga x cosa. En el programa quedaría de la siguiente manera:

```
if (RectDisparo.Intersects(tower.Rectangulo))
            {
                soundEffect7.Play();
                DisparoVivo = false;
                tower.Health -= ((Attack) - (tower.Defence));
                //forzamos a la vida a estar entre 0 y 100
                tower.Health = (int)MathHelper.Clamp(tower.Health, 0, 100);
           [...]
```

Es decir que cuando el rectángulo que rodea a un disparo intersecciona con el rectángulo que rodea a una torre pues hacemos las operaciones indicadas, en este caso una resta.

Este tipo de colisiones a partir de intersecciones de rectángulos que rodean a texturas es el más usado en IceFire. Lo utilizamos para las habilidades del tipo disparo de los héroes y para los disparos de las torres. Es una herramienta muy potente ya que el polígono no tiene por qué ser solo un rectángulo aunque en general y para el tratamiento de los Sprites que nos ocupa es lo más adecuado.

6.7.3 Colisiones mediante Path

Este tipo de colisiones es distinto al anterior y es útil para tratar el problema de los trazados de los sprites. En el juego que nos ocupa con una perspectiva áerea y en el que básicamente vemos el mapeado, el hacer que un héroe vaya por el camino que queremos puede ser difícil. Una opción básica es delimitar mediante la posición, por ejemplo que no puede caminar a partir de cierta posición de la X y de la Y, pero ese tipo de trazados es bastante rectangular, es difícil hacerlo con curvas enrevesadas además de la dificultad del tratamiento.

La mejor opción es tratarlo con un path, una array de puntos blancos o negros y que nos dice si esa posición está permitida o no.

Debajo de nuestro mapa principal y que vemos tendremos una capa que no vemos pero que lee el programa:

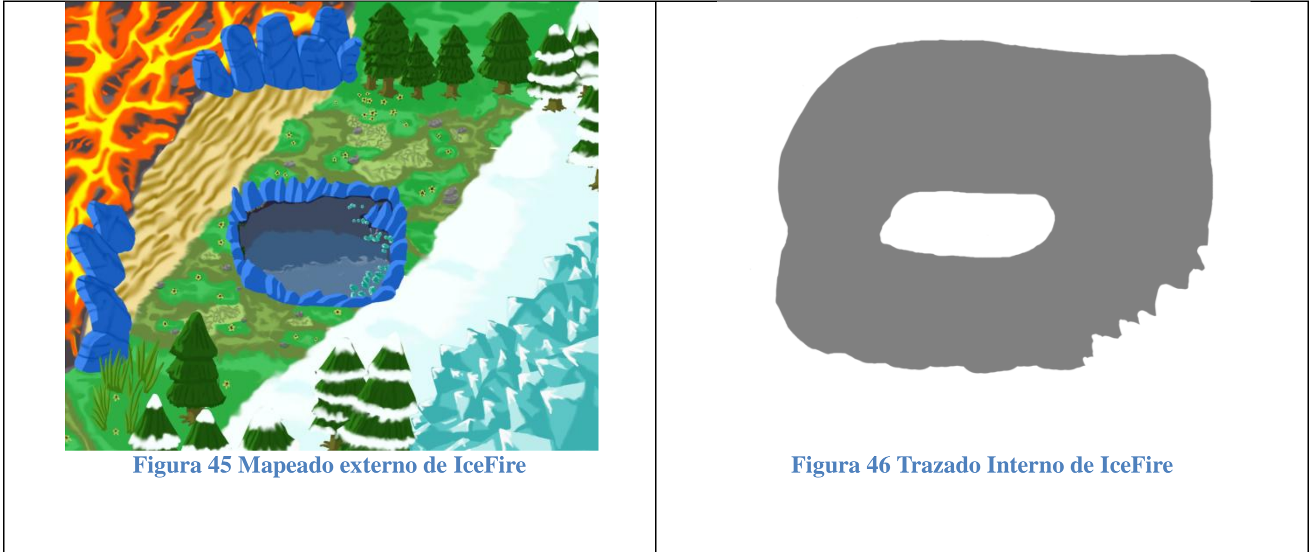

Figura 45 Mapeado externo de IceFire

Figura 46 Trazado Interno de IceFire

Esta última capa nos dice por donde está permitido movernos. Para ello se desarrollo la siguiente función:

```
public override bool IsOnPath()
    {
        int aPixels = Textura.Height * Textura.Width;
        Color[] myColors = new Color[aPixels];
        Recorrido.GetData<Color>(0, new Rectangle((int)Posicion.X,
            (int)Posicion.Y, Textura.Width, Textura.Height), myColors, 0, aPixels);
        /*TextureDataRoad*/
        foreach (Color aColor in myColors)
        {
            //Si uno de los pixeles no es gris, entonces el sprite esta moviendose en esa zona permitida
            if (aColor != Color.Gray)
            {
                return false;
            }
        }
        return true;
    }
```

Esta función lo que hace es crearnos una array del tipo Color y donde metemos si color es blanco o gris, entonces al recorrerla nos devuelve si está en una zona permitida o no.

En el desarrollo de esta función un problema fue las diferencias entre desarrollar para PC o para consola.

Inicialmente en esta array se introducían los datos de todo el mapeado, en la versión para PC no había problema ya que podía recorrer esa array de forma fluida y se podía jugar perfectamente, pero en la versión para 360, el cálculo de toda esa array ralentizaba el juego.

Por eso se cambio de política y aunque la idea era la misma se redujo la información introducida, esta vez metíamos en la array la parte donde estaba el jugador, o sea era calcular en una array del tamaño de la textura del jugador si había partes grises o blancas, con lo cual se agilizaron mucho los cálculos y la consola los realizaba sin mayores problemas.

6.7.4 Opciones alternativas

Hay más opciones a la hora de tratar colisiones, éstas son las usadas en IceFire, pero otras opciones

por ejemplo es a partir de los query de oclusión, la consola envía querys de oclusión cuando hay un sprite delante de otro y para decidir cual se muestra, a partir de estos querys se puede saber si algo colisiona o no. Este es un método estudiado para el proyecto pero que finalmente no fue utilizado.

6.8 Timers

El transcurso del tiempo es algo muy importante en cualquier videojuego, a cada instante se tiene que ir actualizando la pantalla con los cambios adecuados para formar la ilusión de movimiento o de efectos gráficos que queremos realizar. Pero el tiempo tiene también otras funciones, en nuestro caso las habilidades de los personajes queremos que no sean spameables o sea que no se puedan pulsar indiscriminadamente sin pensar antes.

En los juegos con habilidades aparece lo que se denomina en inglés cool down time o lo que podríamos traducir como tiempo de refresco, el tiempo en el que vuelve a estar disponible. Éste fue otro de los problemas de IceFire sufrió en las fases preliminares.

La primera opción que se utilizó fue desarrollar unos timers a partir de los métodos proporcionados por Visual Studio, éstos funcionaban y las habilidades tenían un tiempo de refresco en la versión de PC sin ningún problema y funcionando perfectamente.

Sin embargo en la versión de consola daban errores de compilación, el problema fue muy simple y es que esos métodos no se permiten en XNA, no están soportados.

El método natural de timer para el juego en XNA es el GameTime, un método que nos proporciona esta API y que se trata de un puro contador de cada instante de juego, por tanto a partir de este GameTime se construyeron los métodos para proporcionar habilidades con tiempo de refresco, el código es como sigue mirando la clase BattleScreenOnline, en el método Update:

```
public override void Update(GameTime gameTime, bool otherScreenHasFocus,
                                    bool coveredByOtherScreen)
        float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
        /activamos heroe1
        player1.Update(player2, tower2, base2, elapsed, field2);
```

Y ahora dentro de por ejemplo Firewind tendríamos en su método Update:

```
public override void Update(HeroOnline hero, TowerOnline tower, BaseHero baseHero, float elapsed, Texture2D
field2)
    {
        if (Main_Skill == false)
        {
            if (Is_Firing == true)|
            {
                soundEffect3.Play();
                 fire = new Fire(new Vector2(Posicion.X + Textura.Width / 2, Posicion.Y), TexturaFire,
                DireccionDisparo);
                 Main_Skill = false;
                 Is_Firing = false;
            }
            Cuenta_Main_Skill += elapsed;
              if (Cuenta_Main_Skill >= tiempo_Main_Skill)
             {
                Cuenta_Main_Skill = 0;
                Main_Skill = true;
            }
        }
```

Nada más disparar tendríamos el booleano de Main_skill en false, por tanto cuando este en cool down la habilidad entrará en el código y a cada instante comprobará si el tiempo que vamos sumando es el del tiempo de refresco, una vez sobrepasado se pondrá el booleano de la habilidad en true y ya la tendremos otra vez activa.

7 PRUEBAS BETA TESTING

7.1 Introducción

Para la realización de cualquier videojuego hay una parte muy importante que es la del testeo. La mayoría de juegos tienen bugs en su desarrollo o incluso en su lanzamiento que pueden influir desde en el correcto funcionamiento del juego o bugs menos visibles que afectan a la jugabilidad del producto.

7.2 Etapas

En el caso del proyecto que nos ocupa, la importancia era doble ya que no solo había que testear el propio juego sino testear la funcionalidad online. La fase de testeo la podemos dividir en varias etapas:

7.2.1 Funcionalidades Offline

En esta primera etapa el proyecto ocupaba poco espacio y las funcionalidades que había que testear eran offline. Los primeros tests los realizó el desarrollador del programa y básicamente consistían en ver si el juego realizaba lo que se buscaba. Si se movían correctamente los personajes, los disparos o movimiento de la cámara.

En esta etapa inicial el cliente era repartido entre los Betatesters mediante el correo online y adjuntando el cliente.

7.2.2 Funcionalidades Online

Una vez estaban implementadas las funcionalidades básicas offline, entre Noviembre y Diciembre se decidió empezar con las funcionalidades online, el cliente había ido aumentando considerablemente de tamaño con lo que el método de repartir el cliente mediante el correo era obsoleto.

Se buscaron maneras de repartir el cliente de la forma más rápida posible, google Docs por ejemplo hubiese sido una alternativa pero tenía los mismos problemas que enviar un correo. La mejor opción era algún programa que para la sincronización entre diferentes PC de archivos, uno de los más populares ahora mismo es el DropBox y éste fue el programa elegido para repartir el cliente entre los diferentes Beta Testers.

Una vez solucionada la disponibilidad rápida entre los Beta Testers del cliente se empezaron los tests exhaustivos del programa. Otro de los problemas eran los requisitos para testar el programa, se necesitaba: Visual Studio 2008, cuenta Live Gold para compilar en X 360, Silver para compilar en PC, librerías XNA y cuenta Premium del XNA creator's club. Con lo cual en parte los Beta Testers se reducían a compañeros de la propia Universidad que dispusiesen de la cuenta XNA y a ser posible la consola.

Pese a estas dificultades para encontrar los Beta Testers adecuados el testeo fue progresando y en ésta área empezó el desarrollo de los paquetes, el diseño del programa respecto a Host y cliente.

Pero como se puede entender, el tener que depender siempre de otro compañero para realizar los testeos y progresar en el proyecto fueron un gran problema y retrasaron en esta fase el desarrollo del proyecto.

7.2.3 Testeo exhaustivo Online

Para progresar a buen ritmo las funcionalidades online se necesito una nueva metodología de trabajo. Se necesitaba realizar los tests rápidamente para no tener que depender en todo momento de personas externas. Para ello se preparó la siguiente estación de trabajo.

Un ordenador de mesa con todos los requisitos y un portátil que los cumplía también. De esta manera mediante un portátil y un PC de sobremesa trabajando en paralelo se podía ver los problemas del programa en la visión de Host y cliente.

En esta etapa fue cuando se hicieron mayores avances en esa funcionalidad gracias a la libertad de poder trabajar en cualquier momento en el programa sin necesidad de otro Beta Tester.

El único problema que tenía esta metodología es que al ser dos ordenadores en la misma sala tenían una conexión entre sí demasiado buena con latencía entre ellos muy bajas, menos del 10 ms con lo cual los dos clientes iban muy sincronizados y futuros problemas del programa no se veían.

7.2.4 Testeo con latencias reales

La última etapa del testeo online fue volver a testear con los Beta Testers de la segunda etapa. Al testear con ellos y con latencias más altas que variaban de 60-80 ms a 120 ms se veían problemas que en la anterior etapa no aparecieron, problemas entre la sincronización de los disparos.

Es aquí cuando se decidió cambiar la arquitectura de los clientes, que hasta aquel momento era relativamente igualitaria y que consistía básicamente en enviar la información de lo que hacía un jugador al otro, a una arquitectura que daba preferencia al Host.

Un ejemplo concreto de esto por ejemplo serían los disparos de las torretas, inicialmente cada cliente controlaba sus torretas y avisaba al otro de cuando estas se disparaban, no se enviaba información de la posición de los tiros y se presuponía que al avisar casi simultáneamente la posición de las dos balas en los dos clientes sería aproximadamente la misma.

En latencias bajas esto es así pero cuando es más alta esto cambia. Es por eso que se optó por el otro modelo. Actualmente el Host controla las dos torretas, y envía información de la posición de los disparos de éstas, lo que ve el Host es lo "real" y envía al cliente que se conecta donde ve y cuando disparan las torres.

De esta manera se paliaron en gran parte los problemas de sincronización que en ocasiones aparecían.

8 CONCLUSIONES

8.1 Consecución de objetivos

Los objetivos iniciales que teníamos en el estudio de viabilidad se han cumplido en su gran mayoría aunque por supuesto hay algunos que no se han cumplido o no en toda su totalidad pero en líneas generales podríamos decir que la consecución de objetivos ha sido satisfactoria.

Mirando uno por uno los objetivos cumplidos tendríamos:

- Realizar juego offline de un jugador y dos jugadores que funcione correctamente
- Realizar la conexión online y gestionarla correctamente
- Realizar interfaz gráfica externa al juego
- Creación de más de un personaje distinto
- Añadir efectos sonoros al videojuego
- Creación de un mapeado adecuado para 1 vs 1
- Realización de la interfaz gráfica del juego.

Por otra parte algunos objetivos secundarios que han quedado sin resolver por algunas decisiones en la mecánica de juego posteriores a la realización del estudio, es el caso de:

- Añadir mecánicas extras al proyecto como NPCs
- Creación de contenido extra, más de un mapa disponible, diferentes modos 1 vs 1, 2vs2 y 3vs3.

Finalmente en lo referente a la consecución de objetivos tenemos el último de ellos:

- Eliminar todos los posibles bugs del programa y que funcione correctamente.

En este caso podemos decir que la resolución ha sido casi total, el videojuego actualmente permite a dos jugadores conectarse y realizar una partida de IceFire con dos consolas Xbox 360, el único pero en ese sentido es que no se ha podido resolver totalmente el problema de algunos disparos que se desincronizan cuando aparecen latencias medias-altas, se pudo solucionar en el caso de las torres, pero en los jugadores pese a cambios de código constantes y muchos tests en ocasiones da algún problema.

Tras finalizar el proyecto y visto en perspectiva, la realización del videojuego ha sido muy difícil por varias razones.

Ya desde su inicios había un gran problema, el diseño del juego en sí, de cómo iba a funcionar y que elementos tendría. Pasos previos a la realización de código pero de vital importancia, habían pocas cosas definidas, algunos de los preámbulos es que tenía que ser online por tanto competitivo, algún

tipo de modo versus en el que los dos jugadores tendrían que demostrar su habilidad divirtiéndose con el juego. Porque en el fondo los videojuegos son para eso, jugar y pasar un buen rato y tener el concepto, la idea inicial no es nada fácil.

La primera dificultad ya definido el juego fue comenzar con C#, un programa con el que no se estaba familiarizado pero que ha resultado ser muy intuitivo y una herramienta potentísima de la que se ha intentado sacar el máximo provecho para realizar el juego. Los principios como era de esperar fueron lentos pero una vez superada esta primera etapa los progresos fueron rápidos en la siguiente fase de la creación offline.

En lo referente a la parte de código no ha sido fácil, realizar las clases y métodos necesarios para el GameState, para las Screens, animaciones de los personajes, para la selección diferente de personajes, habilidades de cada uno y su funcionamiento, implementar los otros elementos como son las torres , bases, y en definitiva realizar el videojuego ha traído muchos quebraderos de cabeza.

Pero el gran problema sin duda ha sido la programación con conexiones online. Ya al principio se presuponía que seguramente sería el gran contratiempo del programa, incluso en algunas etapas había dudas si finalmente el juego podría ser online o no, pero finalmente las dificultades se han ido superando hasta llegar al resultado final.

8.2 Desviaciones sobre la planificación

Cuando se realizó la planificación inicial en el estudio de viabilidad algunas de las mecánicas del juego no estaban claras ni tampoco se sabía con exactitud que dificultades y problemas se iba a tener en el desarrollo del producto.

Es por ello que la planificación real es un poco diferente de la presentada en el estudio de viabilidad.

La comparativa entre las dos planificaciones la tenemos en el siguiente cuadro:

Prototipo	Duración prevista	Duración real	Diferencia	Fechas previstas		Fechas reales	
Pre Alpha	30 h	30	0	11-10-10	24-10-10	11-10-10	24-10-10
Alpha 1	30 h	30	0	25-10-10	7-11-10	25-10-10	7-11-10
Alpha 2	30 h	75	+45 h	8-11-10	21-11-10	8-11-10	12-12-10
Alpha 3	45 h	60 h	+15 h	22-11-10	12-12-10	13-12-10	9-1-11
Alpha 4	60 h	-	-60 h	13-12-10	9-1-11	-	-
Beta 1	60 h	60 h	0	10-1-11	6-2-11	10-1-11	6-2-11
Beta 2	60 h	60 h	0	7-2-11	6-3-11	7-2-11	6-3-11
Beta 3	120 h	60 h	-60 h	7-3-11	1-5-11	7-3-11	3-4-11
Beta 4a	-	60 h	+60 h	-	-	4-4-11	1-5-11
Beta 4b	-	60 h	+60 h	-	-	2-5-11	5-6-11
Total	435 h	495 h	+60 h	11-10-10	1-5-11	11-10-10	5-6-11

Las razones principales para los cambios de planificación en su mayoría es por nuevos objetivos en los prototipos provocando la creación de nuevas versiones o la eliminación de otras. En el inicio de la planificación no estaban del todo perfiladas las mecánicas del juego que fueron variando en su desarrollo.

A continuación se presenta los diferentes prototipos reales, sus notas de parche con las mejoras y cambios producidos en cada versión y una explicación de los motivos de la desviación de la planificación. Éste es el desarrollo real del proyecto acompañado también con las fechas reales y previstas de cada versión:

Prototipo Offline IceFire Pre Alpha
Notas:
-Implementación de menús iníciales.
-Creado escenario en una "pantalla" donde dos personajes se mueven con sus controles correspondientes (mando o teclado).
Fechas previstas: 11 Octubre 2010-24 Octubre 2010
Fechas reales: 11 Octubre 2010-24 Octubre 2010
Duración total en horas: 30 horas

Prototipo Offline IceFire Alpha 1
Notas:
-Clases Torre y base creadas. Objetos que pueden ser destruidos por los jugadores.
-Los héroes tienen barras de vida que van bajando progresivamente.
-Cuando la barras llegan a 0, tienen un tiempo inhabilitados hasta revivir.
-Se implementan Power Ups que aparecen aleatoriamente y al cogerlos mejoran características de héroes.
-Implementación de algunos textos de interfaz con datos de Kills y Deaths.
Fechas previstas: 25 Octubre 2010-7 Noviembre 2010
Fechas reales: 25 Octubre 2010-7 Noviembre 2010
Duración total en horas: 30 horas

Prototipo Offline IceFire Alpha 2
Notas:
-Creación de clase padre Hero y clases herederas Firewind e Iced, por el momento comparten las mismas características y solo cambia el sprite que los representa.
-Implementadas clases para animar con sprite sheets.
-Implementado el código para crear y unirse a una sala.
Fechas previstas: 8 Noviembre 2010- 21 Noviembre 2010
Fechas reales: 8 Noviembre 2010-12 Diciembre 2010
Duración total en horas: 75 horas
Motivo de Desviación: Aquí tenemos la primera gran variación, la primera impresión al ver las fechas puede ser el creer que ha habido retrasos en este cliente, pero lo que realmente ocurre es que se fusiona el Alpha 2 y 3 al progresar más rápido de lo previsto.

Prototipo Online IceFire Alpha 3
Notas:
-Enviamos paquetes entre los jugadores de la sala con la posición de cada uno. Los jugadores ya ven al otro moviéndose.
-Las torres disparan aleatoriamente en todas las direcciones, el disparo que realizan es mayor.
Fechas previstas: 22 Noviembre 2010- 12 Diciembre 2010
Fechas reales: 13 Diciembre 2010-9 Enero 2011
Duración total en horas: 60 horas
Motivo de Desviación: El cliente previsto Alpha 4 pasa a ser el Alpha 3 y se empieza a trabajar en la programación online.

Prototipo Online Icefire Beta 1
Notas:
-Se agranda escenario (ocupa más de una pantalla), y cámara sigue a cada jugador en su cliente correspondiente.
-Primeros Beta Tests online con otros jugadores.
-Primera implementación de habilidades de disparo.
Fechas previstas: 10 Enero 2011- 6 Febrero 2011
Fechas reales: 10 Enero 2011-6 Febrero 2011
Duración total en horas: 60 horas
Motivo de Desviación: En este prototipo ya el desarrollador está familiarizado con las herramientas y se intenta completar las partes incompletas relacionadas con la parte online y habilidades.

Prototipo Online Icefire Beta 2
Notas:
-Se cambia mecánica de las torres, ahora disparan a los jugadores enemigos.
-Eliminación de los Power Ups.
-Mejoras de implementación de disparos (aún dan problemas).
-Implementación de gráficos y sonidos propios.
-Corrección de bugs referentes a la creación de salas y salida de éstas (el juego crasheaba en ocasiones).
Fechas previstas: 7 febrero 2011- 6 Marzo 2011
Fechas reales: 7 Febrero 2011-6 Marzo 2011
Duración total en horas: 60 horas
Motivo de Desviación: En este prototipo, aunque en fechas se cumpla lo previsto hay grandes diferencias de objetivos, se define la mecánica de juego y se intenta solucionar algunos bugs aún existentes en la creación y abandono de salas.

Prototipo Online Icefire Beta 3
Notas:
-Implementación de nuevas habilidades: la trampa y la invisibilidad en modos online.
-Inicio de implementación de nuevos sistemas de disparos.
-Cambio en el sistema de envíos de paquetes, ahora hay un cliente que es el host principal y es el que controla todo y el que se conecta recibe los paquetes de información de éste
Fechas previstas: 7 Marzo 2011- 1 Mayo 2011
Fechas reales: 7 Marzo 2011-3 Abril 2011
Duración total en horas: 60 horas
Motivo de Desviación: El cambio de fechas es debido a que se acaban de programar las habilidades de los personajes y se intenta mejorar la sincronía de clientes.

Prototipo Online Icefire Beta 4a
Notas del prototipo:
-Se cambia el sistema de disparos del modo online en las torres por la asincronía que produce el antiguo sistema, ahora el Host es el que envía los datos de los disparos de las torres.
-Corrección de bugs generales.
-Nuevas implementaciones gráficas.
Fechas previstas: -
Fechas reales: 4 Abril 2011-1 Mayo 2011
Duración total en horas: 60 horas
Motivo de Desviación: Se detectan aún ciertas problemas con los disparos con lo que se cambia parte del código para intentar solucionarlos.

Prototipo Alternativo IceFire 4b
Notas del prototipo:
-Se intenta mejorar la respuesta de los disparos al problema de latencia y de lag que puede ocurrir en ocasiones cambiando el funcionamiento del sistema de disparos de los personajes al igual que el de las torres.
Fechas previstas: -
Fechas reales: 2 Mayo 2011-5 Junio 2011
Duración total en horas: 60 horas
Motivo de Desviación: Este cliente no se utiliza en la versión final, se intenta cambiar el sistema de disparos de los jugadores de forma insatisfactoria con lo que se decide utilizar el anterior cliente como versión final.

Información Global
Fecha Inicio: 11 Octubre
Fecha Finalización: 5 de Junio
Total de horas: 495 horas.

A parte de los cambios en los objetivos de los prototipos y la creación de más versiones que en la planificación original hay también un incremento de trabajo llegando la fecha de finalización hasta el 5 de Junio incrementándose en 60 horas más.

8.3 Líneas de ampliación

Una vez finalizado siempre hay cosas mejorables o ampliables. El caso de IceFire no es una excepción y hay varios puntos que se podrían mejorar en futuras ampliaciones del producto.

El punto más fácilmente ampliable y más obvio es el de añadir más personajes, tal como está el código actualmente añadir nuevos héroes sería sencillo, sería añadir nuevas clases que corresponderían a estos personajes con sus correspondientes habilidades.

Quizás lo más complicado en este sentido es como diseñar estas nuevas avatares de forma que no sean desbalanceados y sigan igualados antiguos héroes y nuevos. Uno de los errores de los MOBA actualmente es el ir añadiendo un sin fin de nuevos personajes consiguiendo desigualdades con los personajes iníciales ya que los últimos personajes son generalmente mejores para mantener el interés en el juego.

Otro punto fácilmente ampliable y que sería más un trabajo de grafistas y diseñadores de juego que del programador en sí sería el añadir nuevos mapas a los jugadores, actualmente solo hay un mapa para 1 vs 1, se podrían añadir nuevos con el mismo diseño base o con diferentes.

Siguiendo está línea de ampliación se podrían crear mapas con tamaños más grandes aunque también es cierto que para mapas de mayor tamaño sería recomendable más jugadores en la misma partida.

Otra línea de ampliación fácilmente realizable y que en parte está como clase no usada en la versión final es el añadir Power Ups o objetos que den mejoras a los jugadores. Requeriría un análisis de diseño de juego para ver qué tipos de mejoras no desbalancean los personajes, ajustar los valores de estos cambios testeando con los Beta Testers. La implementación de esta ampliación en el código sería relativamente sencilla y en parte está hecha

En este sentido y teniendo en cuenta la ampliación de mapas, se podría hacer nuevos tipos de modos, 2 vs 2, 3 vs 3 o el típico 5 vs 5. Realizar está ampliación ya sería más complicada y requeriría algunos cambios importantes en el código actual, pero sería posible sin duda. Para hacerlo para empezar habría que cambiar el sistema de asignar a los jugadores, ahora mismo simplemente miramos si es host o no, sería cambiar el identificador por el gamer tag, o lo que es lo mismo el nombre de usuario en los servidores LIVE. Una vez realizado este cambio el añadir más jugadores es sencillo, de base las conexiones de salas mediante LIVE permiten hasta 16 jugadores por tanto no habría ningún problema técnico sino más bien de si es bueno para el diseño del juego o no.

Líneas de ampliación diferentes que alejarían algunos puntos iníciales de diseño del juego sería cambiar la política de encararlo a e-Sport y llevarlo al terreno de los juegos MOBA generales con NPC que se pueden matar y permiten subir de nivel a los personajes.

Realizar la anterior ampliación requeriría varios cambios, el más sencillo sería añadir en la clase base abstracta de Hero la característica de nivel que sería una simple variable, luego otra que sería experiencia y teniendo esas dos variables hacer que se incremente el nivel cuando la experiencia llega a los límites correspondientes. Añadir esto requeriría una vez más Beta Tests para tener esos niveles de experiencia de manera que no fuese tedioso subir de nivel y con el ritmo adecuado al juego, o sea una subida relativamente rápida.

Por otra parte el añadir NPCs enemigos de cada bando sería más complicado, requeriría de una IA, nuevos grafismos para esos personajes, ciertas habilidades y características que los definiesen. Por tanto requeriría bastante esfuerzo tanto en código nuevo, en lo referente a programación, y como trabajo de diseño del juego poniéndoles los stats adecuados.

Hasta ahora hemos hablado de líneas de ampliación realizables con el código de juego actual, pero podemos ir más allá.

Con más inversión en el proyecto se podría cambiar a la modalidad servidor y cliente, a partir de ahí se pueda crear ampliaciones con "registros" de los jugadores en nuestros servidores para poder jugar, y con esas cuentas poder realizar transacciones monetarias, por ejemplo realizando compras de nuevos aspectos para sus personajes o personajes que puedan ir comprando progresivamente.

También se puede cambiar la modalidad de jugar y hacer que los niveles subidos en las partidas se mantuviesen y se quedasen esos datos en los servidores con lo cual se añadiría el factor de ir progresando lentamente y enganchando al usuario en el juego.

Finalizando con las líneas de ampliación de aspectos jugables del juego hay dos vías más, la línea gráfica y sonora. Cambiar los gráficos y la música. Este tipo de cambios pueden ser muy profundos si por ejemplo se opta por pasar a gráficos 3D, con el consiguiente retraso en el tiempo y posible inversión. Realizar la música requeriría también de más inversión al necesitar contratar a un músico

profesional.

Líneas de ampliación respecto a aspectos exteriores al juego sería realizar una página web para publicitar el proyecto o juego, enlazar esa página con los servidores hablados anteriormente siendo las mismas cuentas las de esa página web con las del juego, sirviendo para poder postear en foros, realizar compras online de aspecto o lo que se quisiese vender.

La realización de la página web requeriría mucho más trabajo extra con el tema de diseño web, diseño gráfico, código php, y además para realizar las transacciones monetarias también se requeriría encriptación y otro tipo de medidas de seguridad.

Respecto al tema de la seguridad la última línea de ampliación que vamos a comentar será respecto a eso, todo juego tiene el peligro de ser hackeado, y es uno de los mayores males para la popularidad de éste, si se utilizan los servidores LIVE es difícil hackearlos pero no imposible. Si se trabajase además con servidor propio implicaría incrementar las medidas para la conexión a este servidor y vigilar mucho los paquetes que se envían entre los jugadores y el servidor.

Queda claro por tanto que el proyecto aunque está acabado tiene bastantes líneas de ampliación, algunas de ellas son relativamente sencillas realizarlas, otras es bastante más complicado y requeriría de más inversión y trabajo por parte de programadores, diseñadores y grafistas.

8.4 Comercialización del videojuego

Una vez tenemos finalizado el título una de las posibilidades es el Marketplace de Microsoft.

Este Marketplace para vender juegos online básicamente consiste en una página donde se puede descargar juegos indies, es decir, juegos realizados por personas o compañías pequeñas muy distintas de las grandes producciones de las multinacionales de los videojuegos.

Otro Marketplace de juegos comerciales e indies en plena expansión en la actualidad es Steam, que empezó como una forma de cerciorarse que la gente compraba el juego Counter Strike de la compañía Valve y ha acabado siendo otra gran plataforma para juegos donde apareció recientemente uno de gran éxito realizado con XNA, Magicka.

Los juegos indie aunque generalmente mucho más sencillos en gráficos y en efectos sonoros que los juegos comerciales suelen tener originalidad, buenas ideas, o simplemente son divertidos por lo tanto es un mercado que ahora mismo está en expansión, es presentar a mucha gente juegos de personas que de otra manera no podría hacer llegar sus ideas.

Dejando de lado estos aspectos, vamos a explicar en este apartado cómo funciona el Marketplace de Microsoft para los juegos de XNA.

En el caso de XNA para poder publicar en el Marketplace primero tendremos que pasar una primera aprobación por parte de Microsoft, si nuestro juego pasa esta criba inicial se pondrá en formato de review para que lo puedan probar los otros usuarios con XNA Premium.

Los usuarios podrán probar el juego y si encuentran contenido prohibido del tipo de desnudos, contenido sexual, pederasta o racista podrán bannear ese producto e impedir que se publique en el Marketplace.

Si no incumple el contenido prohibido restará valorar el juego y poner que categoría creemos que es: si es muy violento, si hay heridas, crueldad o desnudos parciales.

Tras estas valoraciones si nuestro producto sale airoso tendrá la oportunidad de aparecer en el Marketplace y ponerse a la venta. Parte de los beneficios serán nuestros pero una proporción irá a Microsoft por cada venta, con lo cual se trata un negocio en los que los dos salen beneficiados.

En el caso de IceFire sería posible utilizar el Marketplace ya que hasta cierto punto es un juego acabado, pero teniendo en cuenta el estándar de la potente industria del videojuego y la gran competencia que tendría entre todos los juegos distribuidos en XNA requeriría algunos meses más de desarrollo para acabar de perfilar algunas de las líneas de ampliación comentadas anteriormente para mejorar el producto y poder tener una mayor repercusión en ventas.

BIBLIOGRAFÍA Y LINKS:

Para la realización del proyecto de XNA se ha tenido suerte que tanto Microsoft como muchos otros usuarios de XNA ponen a la disposición de los iniciados al tema muchos tutoriales, ejemplos prácticos y código a disposición de los que quieren aprender de sus posibilidades.

Durante el transcurso de la programación encontramos en este índice de enlaces los más destacables, además hay que tener en cuenta que algunas de las páginas solo se pone el link general, si se navega en su interior se puede encontrar mucha información importantísima, por ejemplo en la citada página de Microsoft de XNA o en los blogs de Shawn Har tenemos muchas partes interesantes entre sus miles de entradas.

Otra parte de links es más general o de otro tipo de contenidos, como por ejemplo el Visual Studio C# o de cómo funcionan algunas de sus características.

Sin duda sin toda esta información disponible en la web el proyecto no hubiese sido posible.

- **Página principal de XNA.**
  - **Microsoft (2011, -). APP Hub develop for windows [Online]. Disponible en:** http://create.msdn.com/en-US

- **Tutoriales C#.**
  - **Mayo, Joe. (2000, Octubre). Introduction to classes. C# Station [Online]. Disponible en:** www.csharp-station.com/Tutorials/Lesson07.aspx
  - **Allen, Sam. (2011, -) . C# Enum exemples. dotnetperls.com [Online]. Disponible en:** http://www.dotnetperls.com/enum

- **Tutorial de prerrequisitos de juegos XNA**
  - ***Zman. (2007, Mayo). What do I need to make XNA Framework games run on other computers?.*** **APP Hub develop for windows** ***[Online]. Disponible en:*** http://forums.create.msdn.com/forums/t/1988.aspx
  - **Microsoft. (2011, -). Prerequisites for Developing Xbox LIVE Multiplayer Games on Xbox 360. http://msdn.microsoft.com/[Online]. Disponible en:** http://msdn.microsoft.com/en-us/library/bb975642.aspx
  - **Microsoft. (2011, -). Distributing Your Finished Windows Game. http://msdn.microsoft.com [Online]. Disponible en:** http://msdn.microsoft.com/library/bb464156(XNAGameStudio.31).aspx

- **Tutoriales generales XNA**
  - **XNA Mexico. (2011, -). XNA Mexico[Online]. Disponible en:** http://www.xnamexico.com/
  - **Microsoft. (2008, -). XNA Game-Themed Assignment (XGA). http://depts.washington.edu [Online] . Disponible en:** http://depts.washington.edu/cmmr/Research/XNA_Games/2008.5.R.1/Assignments/500_ClassHierarchy/XNA_Specifics/XNA_ImplementationGuide.html
  - **George. (-, -).Tutorials. http://www.xnadevelopment.com [Online]. Disponible en:** http://www.xnadevelopment.com/tutorials.shtml

- **Tutorial sobre Menús en XNA**
  - **Edman196. (2010, Agosto). XNA Menus. Code Project [Online] . Disponible en:**

http://www.codeproject.com/KB/graphics/XNA_Menus.aspx

- **Tutoriales sobre animacion2D**
    - **XNA tutorial. (2009, Diciembre). Moviendo un sprite por la pantalla. xna-tutorial.com [Online]. Disponible en:** http://www.xna-tutorial.com/moviendo-un-sprite-por-la-pantalla/
    - **Mariscal. David (2009, Marzo). Animar Sprites usando una hoja de sprites(Sprite Sheet). aprendiendoxna.wordpress.com [Online]. Disponible en:** http://aprendiendoxna.wordpress.com/tutoriales/tutoriales-2d/animar-sprites-usando-una-hoja-de-spritessprite-sheet/

- **Tutorial sobre Camara2D**
    - **Amador, David. (1998, April). XNA camara 2D with zoom and rotation. David-amador.com [Online]. Disponible en:** http://www.david-amador.com/2009/10/xna-camera-2d-with-zoom-and-rotation/

- **Tutorial sobre fondo con scroll(Scrolling Background)**
    - **Piroshi. (2010, Enero). Fondos con scroll. kalabaza.com [Online] . Disponible en:** http://kalabaza.com.mx/?cat=1
    - **OSD. (2008, Julio). XNA - Scrolling background + stop with final frame.Gamecareerguidecom [Online]. Disponible en:** http://www.gamecareerguide.com/forums/showthread.php?p=5744
    - **Microsoft. (2011, -). Making a Scrolling Background. msdn.microsoft.com [Online]. Disponible en:** http://msdn.microsoft.com/en-us/library/bb203868.aspx

- **Tutoriales sobre colisiones entre sprites**
    - **Ioannis (2010, Enero). XNA 2D Basic Collision Detection. www.progware.org [Online]. Disponible** en: http://www.progware.org/Blog/post/XNA-2D-Basic-Collision-Detection.aspx
    - **Ayucar, Iñaki (2007, Marzo). Xna collision detection. graphicdna.blogspot.com [Online] . Disponible en:** http://graphicdna.blogspot.com/2007/03/xna-collision-detection-part-i.html
    - **Riemer Grootjans (-, -). Collision detection. www.riemers.net [Online]. Disponible en:** http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series2/Collision_detection.php
    - **Scilla (2009, Diciembre). Efficient Platform Collisions XNA. forums.create.msdn.com/ [Online]. Disponible en:** http://forums.create.msdn.com/forums/t/43949.aspx
    - **George (-, -). The road not taken. www.xnadevelopment.com [Online]. Disponible en:** http://www.xnadevelopment.com/tutorials/theroadnottaken/theroadnottaken.shtml
    - **Shawn Har (2008,Diciembre). pixel perfect collision detection using oclusion queries. http://blogs.msdn.com [Online]. Disponible en:** http://blogs.msdn.com/b/shawnhar/archive/2008/12/31/pixel-perfect-collision-detection-using-gpu-occlusion-queries.aspx

- **Tutoriales sobre timers**
    - **Smith, Chad. (2007, Junio). Countdown in xna. www.gamedev.net [Online]. Disponible en:** http://www.gamedev.net/topic/452387-countdown-in-xna/
    - **Microsoft (2011, -). system.timers.timer. http://msdn.microsoft.com [Online].**

Disponible en: http://msdn.microsoft.com/es-es/library/system.timers.timer(VS.80).aspx
- **Microsoft (2011, -). Updating with Variable or Fixed Timing. http://msdn.microsoft.com [Online]. Disponible en:** http://msdn.microsoft.com/en-us/library/bb203878.aspx

- **Tutorial sobre funcionamiento de Mando de 360**
  - **SixofEleven (2010, Enero). xna screen manager part. www.dreamincode.net [Online]. Disponible en:** http://www.dreamincode.net/forums/topic/149373-xna-screen-manager-part-3/

- **Tutoriales sobre Networking**
  - **Microsoft (2011, -).Network p2p. create.msdn.com [Online]. Disponible en:** create.msdn.com/en-US/education/catalog/sample/network_p2p
  - **Brady Dial, Charles McGarvey (2010, Marzo). xna network specification. www.eng.utah.edu [Online]. Disponible en:** http://www.eng.utah.edu/~cs3505/Assignments/project06/xna_network_specification_v2.pdf
  - **Shawn har (2007, Diciembre). http://blogs.msdn.com [Online]. Disponible en:** http://blogs.msdn.com/b/shawnhar/archive/2007/12.aspx
  - **CS420 (-,-). Networking. www.cs.usfca.edu [Online]. Disponible en:** http://www.cs.usfca.edu/~galles/cs420/lecture/Networking/Networking.html
  - **XNAwiki (2008, Noviembre). Network Packet Serializer. xnawiki [Online]. Disponible en:** http://www.xnawiki.com/index.php?title=Network_Packet_Serializer

- **Tutoriales de strings en XNA**
  - **Microsoft (2011, -). XNA Game Studio 3.0. msdn.microsoft.com [Online]. Disponible en:** http://msdn.microsoft.com/en-us/library/bb447673(v=XNAGameStudio.30).aspx
  - **Microsoft (2011, -).Drawing Text with a Sprite. msdn.microsoft.com [Online]. Disponible en**: http://msdn.microsoft.com/en-us/library/bb447673.aspx

- **Tutoriales XNA Sonido**
  - **Sabin (2009, Marzo). Reproducir música y sonido. sabinx.wordpress.com [Online]. Disponible en:** http://sabinx.wordpress.com/xna-tutorial-2-reproducir-musica-y-sonido/
  - **SoldierJaw (2009, Febrero). Adding Background Music to Beginner's Guide to 2D Games. forums.create.msdn.com/ [Online]. Disponible en:** http://forums.create.msdn.com/forums/p/25134/136549.aspx
  - **Microsoft (2011, -).Tutorial 3: Making Sounds with XNA Game Studio. msdn.microsoft.com [Online] . Disponible en:** http://msdn.microsoft.com/en-us/library/bb203895(v=xnagamestudio.31).aspx
  - **Riemer Grootjans (-,-).Sound in XNA. www.riemers.net [Online]. Disponible en:** http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series2D/Sound_in_XNA.php
  - **Cuno Klopsfenstein Lorenzo. (2009, Marzo). Playing audio in xna using xact. Klopfenstein.net [Online]. Disponible en:** http://www.klopfenstein.net/lorenz.aspx/playing-audio-in-xna-using-xact

ANEXO 1: GUÍA RÁPIDA DEL JUEGO

IceFire es bastante intuitivo en los controles y debido a que solo hay dos habilidades por personaje tampoco es excesivo el número de botones que tenemos para pulsar.
Como hay dos versiones, la versión compilada para PC y la versión para consola, tenemos dos tipos de controles, en las siguientes imágenes vienen explicados los controles al seleccionar los personajes:

Cuando entramos en la partida propiamente dicha tenemos que controlar nuestros avatares con lo que para ello tendremos las siguientes posibilidades:

Josep Rius López, Setembre 2011