BreakArk: desarrollo de un clon de Breakout con una arquitectura cliente-servidor e integrador RK4

Autor: Alejandro Nápoles González
Ingeniería técnica de Telecomunicación. Especialidad de Telemática

Consultor: Manel Llopart Vidal

18/06/2014

FICHA DEL TRABAJO FINAL

Título del trabajo:	BreakArk: desarrollo de un clon de Breakout con una arquitectura cliente-servidor e integrador RK4
Nombre del autor:	Alejandro Nápoles González
Nombre del consultor:	Manel Llopart Vidal
Fecha de entrega (mm/aaaa):	06/2014
Área del Trabajo Final:	Aplicaciones multimedia de nueva generación
Titulación:	Ingeniería técnica de Telecomunicación. Especialidad de Telemática

Resumen del Trabajo (máximo 250 palabras):

En este Trabajo se ha desarrollado un clon del popular juego de la compañía Atari Breakout.

No solo tiene un modo de un jugador, sino que se ha pretendido ir más allá e implementar una arquitectura cliente-servidor. Los dos puntos claves de este clon son la arquitectura cliente-servidor y la implementación del método de Runge-Kutta de cuarto orden, o integrador RK4, frente al típico integrador de Euler que, aunque puede resultar excesivo dado las dimensiones de este juego, permite entender cómo se elaboran la simulación de cuerpos rígidos en programas más avanzados como pueden ser simulaciones físicas o juegos profesionales en las que se requiere una interacción realista con los objetos.

Especialmente interesante es la complejidad inherente que conlleva la realización de un producto de software como es el videojuego. Aunque un clon de Breakout puede ser considerado según los estándares normales de "sencillo", implementa un gran abanico de conceptos multidisciplinares, como pueden ser la programación gráfica, programación de red (sockets, conexiones virtuales sobre UDP), ramas como la

matemática (p.e algebra lineal), la física con un sistema rudimentario de simulación de cuerpos rígidos o colisiones en tiempo real y conocimientos asociados al desarrollo de software como máquinas de estado finito o programación orientada a objetos.

En definitiva, este Trabajo sirve, entre otras cosas, para darse cuenta de la ingente obra de ingeniería y creatividad que supone la creación de un videojuego, en donde se mezclan la ciencia y el arte.

Abstract (in English, 250 words or less):

In this work we have developed a clone of the popular game company Atari Breakout.

Not only has a single player mode, but has sought to go further and implement a client-server architecture. The two key points of this clone are client-server architecture and implementation of the Runge-Kutta fourth order or RK4 integrator versus typical Euler integrator, that although may be excessive given the size of this game, allows to understand how the simulation of rigid bodies in more advanced physical simulations, such as professional games, are done.

Especially interesting is the inherent complexity involved in the implementation of a software product such as a video game. Although a Breakout clone can be considered as a simple game, it implements a wide range of multidisciplinary concepts, such as graphics programming, network programming (sockets, virtual connections over UDP) as mathematical branches (eg linear algebra), physics with a rudimentary system of rigid bodies simulation or real-time collision and associated software development, as finite state machines or OOP knowledge.

Ultimately, this work serves, among other things, to realize the enormous feat of engineering and creativity that involves the creation of a video game, where science and art mix together.

Palabras clave (entre 4 y 8):

Juego, Breakout, integrador RK4, arquitectura cliente-servidor, FSM

Índice

Lista de figuras

1. Capítulo 1: Introducción

1.1 Contexto y justificación del Trabajo

El trabajo surge, en principio, como una necesidad personal del autor por embarcarse en una línea de negocio o laboral distinta al del sector de las telecomunicaciones y, a la vez, por poner en práctica conocimientos aprendidos a lo largo de la Ingeniería. Esta nueva línea de negocio o laboral es el desarrollo de software, concretamente el desarrollo de videojuegos, línea en la que el autor ha sentido siempre especial interés, ya que aúna un gran número de ramas de la ciencia y el arte para crear un producto cultural que, aunque originalmente siempre tuvo como fin el de entretener, tiene un potencial inmenso para remover conciencias y transportar a mundos diferentes al actual, dado su gran interactividad, concepto que no poseen otras formas de arte o cultura como la pintura, el cine o música. En el sector de los videojuegos se encuentran aquellos juegos únicamente pensados para entretener como los típicos Super Mario, basándose en conceptos como el reto, la competitividad o cooperación entre usuarios.

Ilustración 1 Mario Bros.

Otros se basan en un acto más contemplativo, en el que la unión de imagen, música e interacción intentan trasmitir sentimientos, ideas o crear estados de ánimo determinados,

como puede ser el juego desarrollado en Flash por Jordan Magnuson y música de Kevin MacLeod *Loneliness* (Soledad) [1], en el que lo único que se puede hacer es mover con las flechas del teclado un cuadrado en pantalla, y ver cómo los demás cuadrados se alejan inexorablemente del jugador cuando intenta acercarse a ellos.

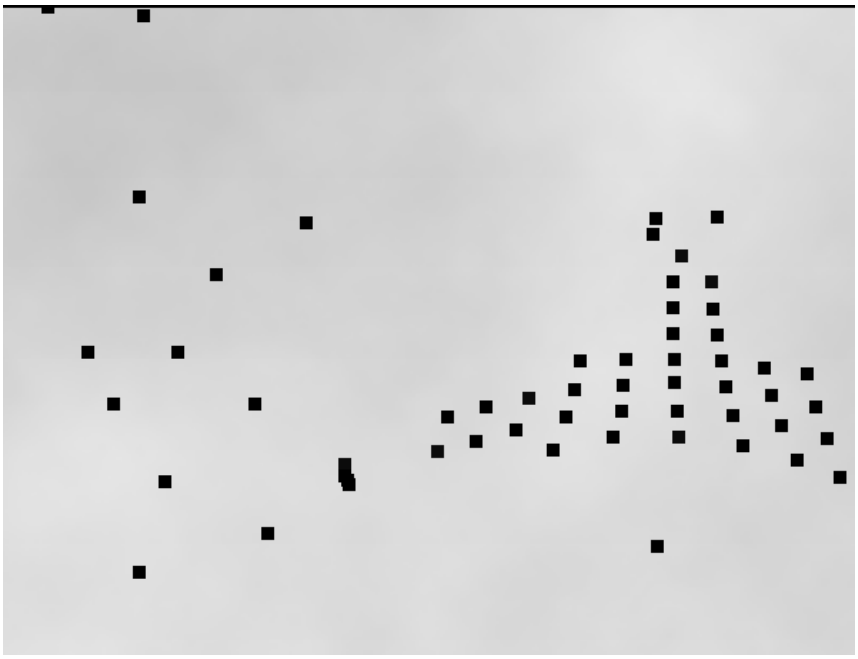

Ilustración 2 Loneniless

Otro ejemplo de esto es el juego Gone Home [2], en el que el único objetivo es explorar una casa con el fin de esclarecer por qué no hay nadie de la familia para recibir a la protagonista. El viaje por toda la casa descubriendo cartas, casetes, diarios, escuchando las cartas de la hermana de la protagonista... va desvelando la historia principal y hace al jugador partícipe y confidente.

Ilustración 3 Gone Home

Elaborar un videojuego, relativamente simple en apariencia, permite al autor conocer de primera mano en qué consiste exactamente la elaboración de un videojuego, qué conocimientos se deben tener. A su vez, añadiendo al juego una arquitectura cliente-servidor permite usar varios de los conocimientos aprendidos en la carrera, como los Sockets o los protocolos de transporte TCP y UDP.

Es un tema relevante, al margen de las inclinaciones del autor, por el hecho de que el sector de los videojuegos crece a pasos agigantados, aumenta el número de jugadores cada año y los beneficios derivados de la industria superan a la industria del cine o la música. Ya no es un sector, como podía serlo o parecerlo en el pasado, solo para apasionados de los videojuegos, frikis o personas en ese sector genérico y oscuro que es la informática, sino que, actualmente, se precisa de personal altamente cualificado y con amplios conocimientos en numerosas ramas, como puede ser el diseño, la programación gráfica o la inteligencia artificial, a medida que se incrementa el alcance del videojuego y avanza la tecnología.

Es un sector en constante innovación. Dada la importancia que tiene el realizar cálculos complejos en tiempo real siempre se están buscando nuevos algoritmos o mejores implementaciones de algoritmos ya existentes, ocurre lo mismo con las estructuras de datos. También hay que añadir que da un enorme impulso al sector gráfico, haciendo que compañías como AMD o Nvidia saquen productos nuevos constantemente en busca del salto gráfico, algo que tiene beneficios así como desventajas para el usuario final.

Ilustración 4 Logo Nvidia

Ilustración 5 Logo AMD

Con este Trabajo se pretende, además, elaborar o dejar constancia del proceso de elaboración de un videojuego, así como de las tecnologías usadas para ello.

1.2 Objetivos del Trabajo

- Definir máquina de estado finito que englobe los estados fundamentales por los que pasa la aplicación.

- Desarrollar una arquitectura cliente-servidor para un solo jugador (cliente y servidor local).

- (Opcional) Clientes en remoto podrán conectarse al servidor.

- Implementar integrador Runge-Kutta de cuarto orden.

- Implementar simulación de cuerpos rígidos para la pelota usando el integrador.

- Implementar carga de niveles a través de ficheros.

- (Opcional) Además de los diferentes niveles de dureza, es posible desarrollar diferentes efectos, como romper un número determinado de bloques dependiendo del radio, aumentar la velocidad de la pelota por cada bloque que se haya roto, conseguir bonificadores u objetos que caigan de los bloques y doten al jugador de efectos extra como múltiples pelotas.

- (Opcional) Implementar sonidos para los menús y las colisiones de la pelota con los demás elementos del juego.

- (Opcional) Incluir música de fondo mientras se juega.

1.3 Enfoque y método seguido

Hay diversas estrategias a seguir en la elaboración de cualquier trabajo de este calibre, sobre todo en el desarrollo de software, unas conllevarán un mejor o peor resultado.

Una estrategia típica y muy extendida en el mundo de la programación es el de ponerse a implementar directamente, sin tener en consideración cuestiones fundamentales del programa, como pueden ser:

- ¿Qué se pretende conseguir con este trabajo, es decir, que se obtendrá con la finalización del trabajo?

- ¿Cuáles son los objetivos o hitos en los que se debería dividir el trabajo y por los que se deberá pasar obligatoriamente si se quiere finalizar con éxito?

- Estimación de los conocimientos y los recursos que se requieren y del grado de complejidad que supondrá la elaboración del proyecto. ¿Se podrá afrontar satisfactoriamente y en un plazo razonable el proyecto con los recursos actuales (conocimientos, tiempo, dinero...)?

- Elección clara de las herramientas de trabajo, ¿se conseguirá terminar el proyecto antes si se escogen ciertas herramientas, aumenta o disminuye la complejidad del software?

Esta estrategia, la de ponerse “manos a la obra” de inmediato, lleva consigo más perjuicio que beneficio a largo plazo, ya que se tenderá a rehacer y repensar numerosas veces el alcance o las soluciones que se han aplicado para resolver el problema a mano, esto hace que probablemente el plazo final del proyecto se alargue más de lo esperado.

La estrategia más adecuada y la que se ha pretendido usar en este proyecto es la de pensar primero en el diseño, en la planificación inicial, sobre el papel, plasmar físicamente la línea de pensamiento, aspectos importantes a considerar son:

- Los estados por los que se cree que irá pasando la aplicación.

- Plasmar el flujo de la aplicación en líneas generales

- Determinar el paradigma más adecuado para el proyecto.

- Detectar las clases principales que se necesitarán (si se usa programación orientada a objetos), cuáles serán sus funciones, cómo se interrelacionarán entre ellas, qué métodos necesitarán...

- Si no se conocen las herramientas o bibliotecas habrá que aprenderlas antes de enfrascarse en la elaboración del Trabajo.

Conviene también seguir la estrategia de divide et impera (divide y vencerás), dividir el proyecto en una serie de hitos y subobjetivos en los que se vayan probando las herramientas y desarrollando las funcionalidades que formarán parte del producto final.

Con todo esto plasmado físicamente en papel se tendrá una hoja de ruta clara que facilitará muchísimo la elaboración del Trabajo. Esto no significa que el diseño inicial sea inamovible, puesto que en el desarrollo de software no es, en la mayoría de los casos, posible determinar con exactitud todo lo que se necesitará de antemano, y es posible que se tenga que reformular el diseño o modificarlo según nos enfrentamos al problema

1.4 Planificación del Trabajo

- PEC 1:
    - Elección de proyecto: durante este tiempo se determina cuál será el proyecto
    - Elección de herramientas: durante este periodo se deciden las herramientas que se utilizarán para la realización del Trabajo.
- PEC 2: en el estado del arte se realiza la recolección de información sobre el área de interés: tecnologías, lenguajes, proyectos similares o el estado del sector son puntos a tener en cuenta.

- PEC 3:

    - Aprendizaje de herramientas: durante este tiempo se realizan pruebas con las herramientas y bibliotecas elegidas para analizar su funcionalidad y aprender a usarlas correctamente.

    - Diseño: una vez aprendidas las herramientas que se van a usar, se pasa a la fase de diseño, en donde se deciden a grandes rasgos los flujos de ejecución de la aplicación, los estados, las clases, funciones, las convenciones a la hora de programar, etc.

    - Implementación: terminado el diseño se pasa a la fase de implementación, en donde se programa lo que se ha decidido en la anterior fase.

- PEC 4: en esta fase se recolectan la mayoría de las PECs para elaborar la memoria final del Trabajo.

Nombre	Fecha de inicio	Fecha de fin
PEC 1	26/02/14	12/03/14
Elección de proyecto	26/02/14	4/03/14
Elección de herramientas	5/03/14	12/03/14
PEC 2	13/03/14	2/04/14
PEC 3	3/04/14	21/05/14
Aprendizaje de herramientas	3/04/14	16/04/14
Diseño	17/04/14	1/05/14
Implementación	1/05/14	21/05/14
Memoria	22/05/14	18/06/14

Ilustración 6 Planificación temporal

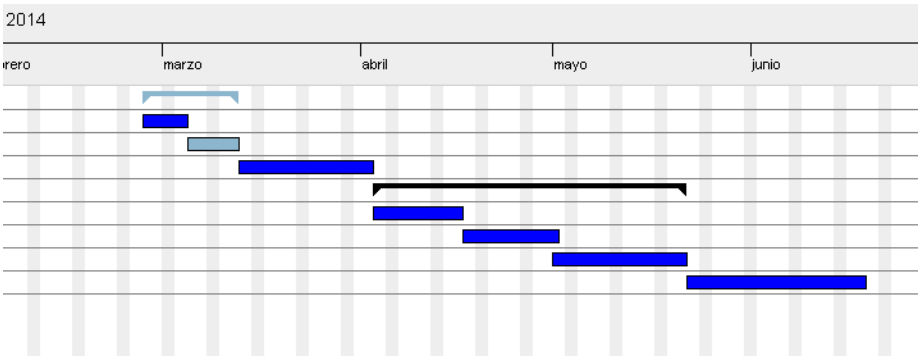

Ilustración 7 Diagrama de Gantt

1.5 Breve sumario de productos obtenidos

Al finalizar este Trabajo se obtendrán los siguientes productos:

- Código fuente del proyecto.
- Memoria del proyecto
- Diagrama UML.
- Programa funcional.

1.6 Breve descripción de los otros capítulos de la memoria

- Capítulo 2: Estado del arte. En este capítulo se pretende hacer un repaso de las diferentes técnicas y proyectos relacionados con el ámbito del Trabajo, esto permitirá:
    - Tener una visión global del estado actual del ámbito en cuestión.

    - Obtener una bibliografía que permita justificar las decisiones de diseño e implementación que se tomen a medida que se realice el proyecto.

- Capítulos 3: Diseño. En este capítulo se detallará el diseño elegido a la hora de realizar la aplicación y la implementación de estas decisiones de diseño en el código

- Capítulo 4: se detalla la implementación de las decisiones tomadas en el capítulo de diseño.

- Capítulo 5: se exponen las conclusiones del proyecto.

2. Capítulo 2: Estado del Arte

2.1 Lenguaje

El lenguaje de programación elegido, tanto para el front-end como para el back-end, para la realización del Trabajo de Fin de Carrera ha sido C++. Los motivos que llevan a su elección para este proyecto, frente a otros muchos más ágiles, rápidos, intuitivos, simples, fáciles de comprender o más populares o extendidos como pueden ser Python, Ruby, C# (que cuenta, además, con el framework .NET de Microsoft) o Java son los siguientes:

- Es el lenguaje *de facto* en la industria de los videojuegos. Su uso está ampliamente extendido en los videojuegos triple A (AAA), aquellos que aspiran a tener un alta calidad y gran recaudación.

- Es un lenguaje con una gran trayectoria (creado en 1983) y una enorme comunidad detrás. Esto implica que existen muchísimas librerías (entre ellas destacan las bibliotecas Boost, formando algunas de ellas parte del nuevo estandar del lenguaje C++11) de eficacia probada y una ingente cantidad de documentación, bien en libros o en páginas web.

- Rapidez. C++ siempre se ha caracterizado por ser muy rápido. Aunque en los últimos tiempos el rendimiento de lenguajes como C# o Java ha ido incrementandose, C++ no deja de ser una buena opción cuando se persigue este objetico.

- Gestión de memoria. En C++ no hay por defecto un recoletor de basura, el programador está más expuesto y tiene que lidiar con punteros. Normalmente se considera una desventaja (sobre todo para usuarios principiantes) a la hora de elegir usar este lenguaje frente a otros que esconden el uso de punteros, pero lidiar con ellos permite obtener un conocimiento más profundo del funcionamiento del

sistema, además de poder controlar con precisión el uso que se le da a la memoria, ideal para aplicaciones en hardware con memoria limitada por ejemplo, en consolas o sistemas empotrados (aunque para esto último posiblemente se prefiera C, un lenguaje de más bajo nivel)

2.1.1 C++

C++ es un lenguaje multiparadigma diseñado por Bjarne Stroustrup en la decada de los 80, el objectivo era añadir al lenguaje C mecanismos de manipulación de objetos, lo cual no quiere decir que no se pudiera aplicar el paradigma orientado a objetos en C, pero era más tedioso. Con C++ se permite la creación de clases y objectos (instancias de la clase) en vez de usar estructuras, así como, entre otras cosas, la visibilidad de los métodos de la clase con las palabras claves public, protected y private. Asimismo se añaden conceptos como el de herencia y polimorfismo.

2.1.2 Herencia

El concepto de herencia permite crear clases que deriven de otras, heredando los atributos y métodos (según la visibilidad) de las clases padre. Esto permite, a su vez, crear clases abstractas puras, tambien llamadas interfaces, en donde los métodos son implementados por las clases hijas.

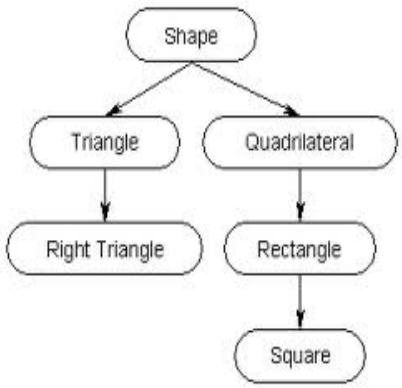

Ilustración 8 Herencia

A continuación un ejemplo del concepto de herencia, la clases Brick y Paddle heredan de la clase padre Square:

```
class Square : public Object
{
public:
Square(int posX, int posY);
~Square();

 const int getPosX() { return posX; }
 const int getPosY() { return posY; }

    //Método virtual. No se implementa aquí.
 virtual void move() = 0;

 void render(SDL_Renderer *renderer);

protected:
 int posX;
 int posY;

 SDL_Rect collider;

 void shiftCollider();

}

class Brick : public Square
{
public:

 enum HARDNESS { NONE, SOFT, MEDIUM, HARD };

 Brick(int posX, int posY, const HARDNESS hardness);

 bool isDestroyed() { return !hardness; }
```

```
//Método move() es implementado en la clase hija Brick.
void move();

void decreaseHardness();

private:
HARDNESS hardness;
};
```

A continuación se ve cómo la clase hija Brick llama al constructor de la clase padre Square y le pasa los atributos pertinentes:

```
Brick::Brick(int posX, int posY, HARDNESS hardness)
: Square(posX, posY)
{
this->hardness = hardness;
}
```

2.1.3 Polimorfismo

Esta característica permite que, a través de una misma interfaz, acceder a entidades de diferentes tipos. A continuación vemos un ejemplo simple de esto:

```
void render(SDL_Renderer *renderer);

void render(SDL_Renderer *renderer, std::string path);
```

2.2 Bibliotecas

2.2.1 Simple DirectMedia Layer

Para las funciones del juego relacionadas con el dibujo en pantalla, gestión de imagenes, música, eventos del teclado y ratón, y temporizadores se ha elegido la biblioteca Simple DirectMedia Layer (SDL), creada inicialmente por Sam Lantinga en

el lenguaje C, aunque existen numerosos empaquetadores (wrappers) en otros lenguajes de programación. La versión más actual de la biblioteca es la 2.0.1.

La elección de esta biblioteca ha sido condicionada por los siguiente motivos:

- Multiplataforma. SDL es compatible con Windows, Mac OS, GNU/Linux y QNX.

- Licencia GNU Lesser General Public (LGPL). Esta licencia, según Wikipedia:

"*Pretende garantizar la libertad de compartir y modificar el software cubierto por ella, asegurando que el software es libre para todos sus usuarios*"

- Extensa documentación en forma de artículos, tutoriales o la propia wiki oficial

- Biblioteca usada con éxito en videojuegos comerciales, siendo algunos ejemplos los siguientes:

    - World of Goo.

    - Starbound.

    - Don't Starve.

- Posibilidad de usar OpenGL.

- Facilidad de uso.

Otras bibliotecas que se contemplaron fueron:

- Allegro

- SFML: similar a SDL, programada en C++ teniendo en cuenta el paradigma orientado a objetos.

2.2.2 SDL_net

Para las funciones relacionadas con la red se ha utilizado la biblioteca SDL_net. Esta biblioteca se usa con SDL. Simplifica la programación de red: creación de sockets TCP/UDP, envío y recepción de paquetes y resolución de nombres. Sin un gran nivel de abstracción (comparado con BSD Sockets), provee la suficiente como para trabajar de una manera más eficaz y rápida. El gran inconveniente es la falta de documentación, siendo una de las pocas fuentes de información la documentación oficial, aunque con conocimientos básicos en programación y los sockets en GNU/Linux se puede entender con facilidad esta biblioteca.

2.3 Estado del sector

El sector del videojuego crece a pasos agigantados, se buscan constantemente nuevos talentos, gente altamente cualificada, nuevas ideas, nuevo hardware.

Según el libro blanco emitido por la ADESE en 2014 [7] en España hay una 330 empresas en activo y el número crece cada año.

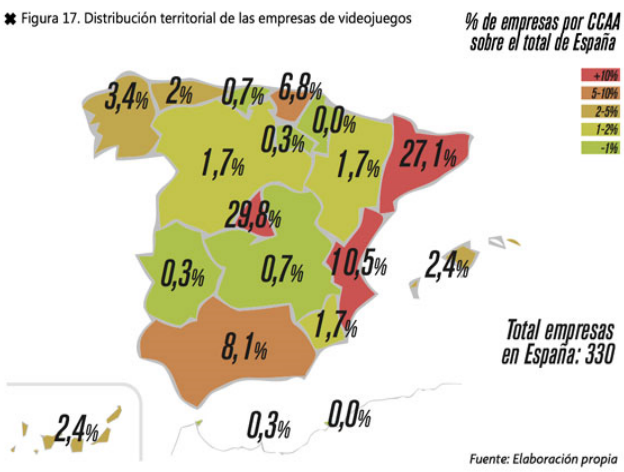

Ilustración 9 Empresas en españa

Además, aunque las ventas del formato físico estén bajando, suben las del formato digital, facturando un 78% del total y el gobierno español pretende dar 3.5 millones de euros en subvenciones y 35 millones en préstamos durante el 2014 y el 2015.

En definitiva, es un sector con una gran expansión y en donde se aúna la ciencia, la ingeniería con el arte y se encuentra gente altamente cualificada que busca la innovación constante en cuanto a tecnologías.

Capítulo 3: Diseño

3.1 Diseño de los pilares básicos del juego

Cualquier clon de Breakout/Arkanoid se compone de varios elementos básicos perfectamente reconocibles:

- Raqueta.
- Pelota.
- Bloques
- Contadores de vida, tiempo y nivel.
- Distintos niveles con diferente disposición de bloques y dificultad.

Además de esto, se necesitará obtener la entrada del teclado de alguna manera, un reloj para medir el tiempo de juego con posibilidad de detenerlo, pausarlo o reanudarlo. Por otra parte, la lógica del juego debería quedar encapsulada y, en la medida de lo posible, independiente de la implementación que puedan tener los elementos que lo componen. Se necesitará también realizar un sistema de colisiones que pueda detectarlas y resolverlas usando los objetos involucrados en ellas (pelota, raqueta y bloques).

Una vez aclarado los requerimientos se hace evidente que, siguiendo un paradigma orientado a objetos, se podrá encapsular la mayoría de los sistemas en clases. A continuación se muestran las clases requeridas.

3.1.1 Clases

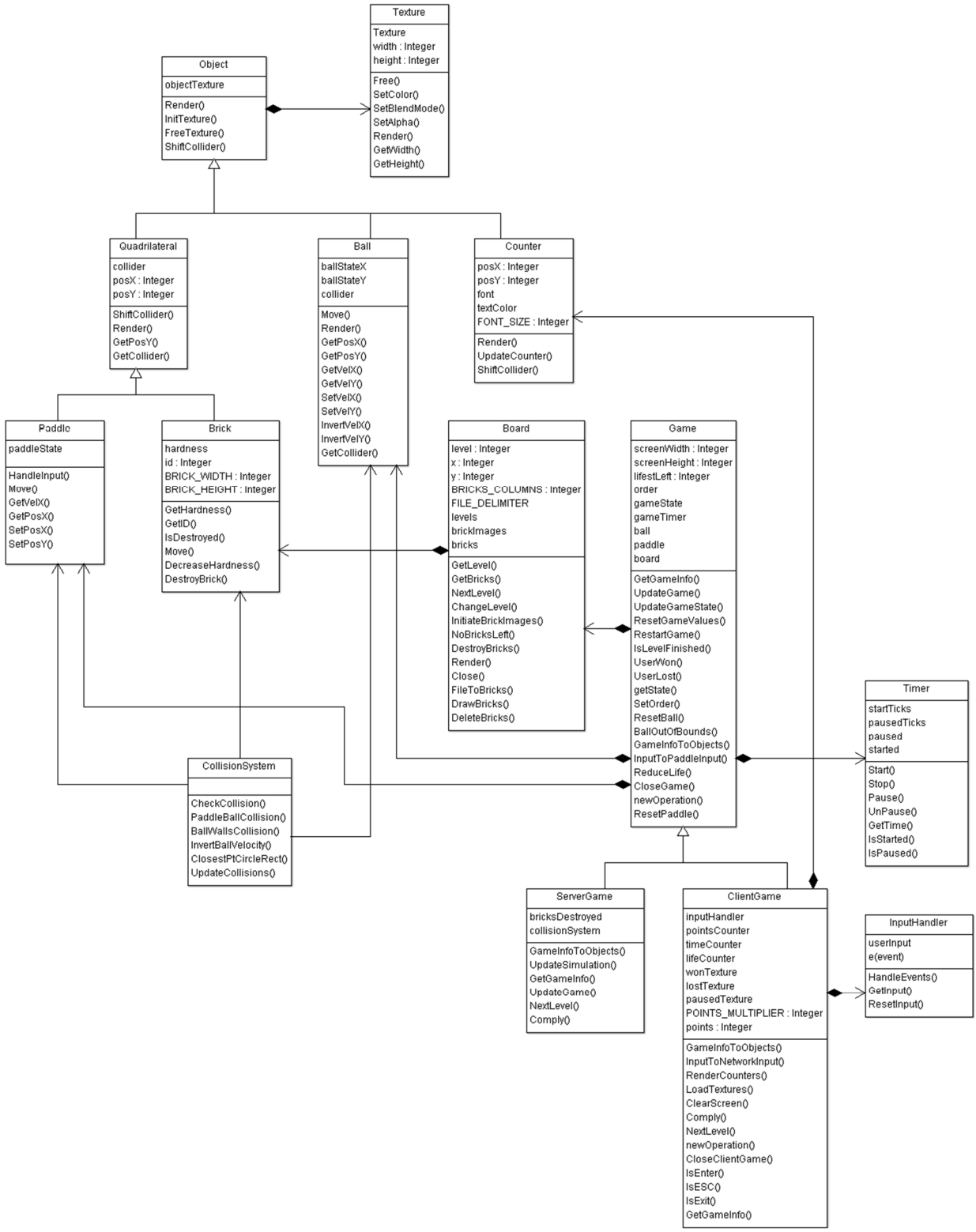

Ilustración 10 Clases BreakArk sin incluir red

3.1.2 Sistema de colisiones

El sistema de colisiones juega un papel vital en este videojuego, puesto que la interacción entre los elementos básicos es gran parte de este, la pelota golpea los bloques destruyéndolos y rebotando, rebota a su vez en la raqueta y en las paredes con un cierto ángulo.

El objetivo del juego es destruir los bloques en la pantalla, una vez que esto pase se gana, pudiendo pasar de nivel en el caso de que haya más. Claramente se necesitará un sistema de físicas en el que se detecte si la pelota ha colisionado con uno o varios bloques o con la raqueta o con los muros.

Para detectar si la pelota ha colisionado con los bloques o con la raqueta se usará el método de intersección de líneas, es decir, se detectará si alguno de los lados de la pelota colisiona con algún lado de los bloques o la raqueta (la raqueta y los bloques se consideran iguales, son cuadrados o rectángulos).

En el caso de que exista colisión se invierte el componente de la velocidad que corresponda.

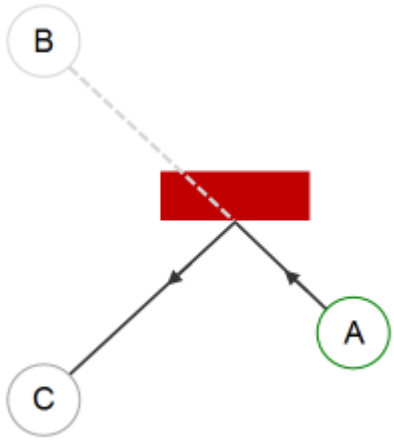

Ilustración 11 Pelota rebotando en un bloque

Hay que tener en cuenta que puede haber múltiples colisiones, en el caso en el que la pelota rebote en un bloque y toque otro, en este caso se ha optado por realizar recursivamente la comprobación de colisiones, terminando solo cuando no haya más. Por lo tanto el procedimiento básico es el siguiente:

- Comprobar colisiones entre pelota-bloques, pelota-raqueta, pelota-paredes.

- Si existen colisiones invertir componente de velocidad correspondiente, si no entonces salir.

- Mover la pelota a la posición siguiente y repetir primer paso.

Una entidad llamada CollisionSystem se encargará de realizar este procedimiento.

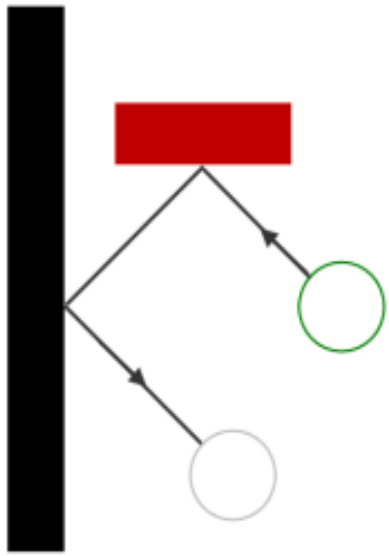

Ilustración 12 Pelota rebotando varias veces

3.1.2.1 Detección de colisiones en tiempo real

La detección de colisiones en tiempo real es un ámbito de estudio enorme y muy complejo. Su objetivo es poder detectar las colisiones que se dan entre dos objetos de la manera más eficiente y en el menor tiempo posible, dado las imposiciones que supone detectarlas en tiempo real. Los cálculos deben realizarse en menos de un fotograma por segundo, para permitir a otros sistemas, como puede ser el renderizado 3D o la inteligencia artificial, realizar sus cálculos. Para ello se utilizan métodos y algoritmos de diversa índole, desde intersección de volúmenes AABB-AABB (Axis-aligned Bounding Boxes) o OBB-OBB (Oriented Bounding Boxes), hasta particionamiento espacial con árboles k-d o jerarquías usando árboles BSP (Binary space partitioning). En el proyecto se pretende mostrar la detección de colisiones usando lo primero, intersección de volúmenes AABB y un círculo que conforman el 100% de todas las colisiones que se encontrarán en el juego.

Por lo tanto el objeto pelota, como raqueta y bloque tendrán un volumen que se usará para calcular la colisión. En el caso de la raqueta y el bloque será un rectángulo (AABB). En la pelota, aunque la imagen de esta es también un rectángulo y se podría calcular la intersección AABB-AABB, se optará por crear un círculo.

3.1.2.1.1 AABB-círculo

Un volumen AABB consiste en rodear el objeto del que se quiere detectar la colisión con un rectángulo y efectuando la detección a partir de este. De esta manera detectar la colisión AABB-AABB simplemente consiste en analizar si dos rectángulos se sobreponen.

Ilustración 13 : AABB-AABB

En el caso de este proyecto se necesita detectar colisiones AABB-Círculo (o esfera), con lo que consistirá en detectar si la círculo ha atravesado el rectángulo. Para ello se deberá hallar el punto más cercano entre el círculo y el rectángulo. Esto se reduce a obtener el punto más cercano entre un rectángulo y un punto (el centro del círculo).

Se irá recorriendo cada coordenada y, si está fuera del rectángulo se tomará la coordenada del rectángulo, sino se mantendrá la coordenada del punto.

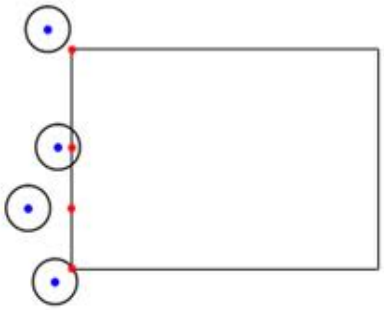

Ilustración 14 Punto más próximo a una caja

Finalmente, un AABB y un círculo se superponen si la distancia desde el centro de este último al punto más cercano calculado anteriormente es menor que el radio del círculo [10]

3.1.2.1.2 Simulación del movimiento de la pelota y raqueta

En un juego simple es posible no tener en cuenta demasiados factores, pero en este proyecto se ha pretendido realizar técnicas más cercanas a la realidad del desarrollo de videojuegos comercial. El objetivo en el movimiento de la pelota era conseguir realizar una simulación del movimiento elástico y, además, realizar la simulación en tiempos muy cortos llamados delta. Para ello se necesita aplicar un integrador no euclideo (ya que este tiene un gran error al cambiar las derivadas en cada paso). Para ello se ha decidido usar el integrador RK4, aunque existen numerosos integradores como la integración velvet o euler implícito, se ha decidido usar el método de Runge-Kutta por ser el integrador de uso genérico más preciso.

Con este integrador se podrá a través de un estado de la pelota conseguir una simulación "realista" de la posición, la velocidad y la aceleración de esta en un tiempo muy pequeño delta. Que sea en un tiempo lo más pequeño posible es lo más recomendable.

A continuación se muestra la ecuación matemática del método RK4, aunque la implementación será más sencilla de lo que parece.

$$y' = f(x, y), \quad y(x_0) = y_0$$

Ilustración 15 Ecuación 1

$$y_{i+1} = y_i + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

Ilustración 16 Ecuación 2

$$\begin{cases} k_1 &= f\left(x_i, y_i\right) \\ k_2 &= f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h\right) \\ k_3 &= f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2 h\right) \\ k_4 &= f\left(x_i + h, y_i + k_3 h\right) \end{cases}$$

Ilustración 17 Kn

$$\text{pendiente} = \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}.$$

Ilustración 18 Pendiente

Las ecuaciones del movimiento son las siguientes:

a = f/m

dv/dt = a = F/m

dx/dt = v

Esto significa que:

- Aceleración es igual a fuerza divida por masa.
- Cambio en velocidad es igual a aceleración por un tiempo delta.
- Cambio en posición es igual a velocidad por un tiempo delta.

3.2 Autómatas de estado finito

Los autómatas de estado finito (AF), autómata finito, o máquinas de estado finito (Finite State Machine o FSM) son ampliamente usados en la industria del software para multitud de usos. Permiten afrontar un gran número de situaciones relacionadas con el concepto de "comportamiento".

3.2.1 Definición

Un autómata de estado finito, es:

"Un modelo computacional que realiza cómputos en forma automática sobre una entrada para producir una salida". - Wikipedia ES

Este modelo computacional consta de un alfabeto, estados y transiciones. Según su definición formal es una quíntupla formada por los siguientes objetos:

- Q: estados.
- Σ: alfabeto.
- $q_{0:}$ estado inicial (siempre único).
- δ: función de transición.
- F: estados finales. (puede ser más de uno)

Es posible ver este tipo de modelo como una cinta lectora que avanza hacia delante de una celda a otra (de uno en uno) dependiendo de la función de transición.

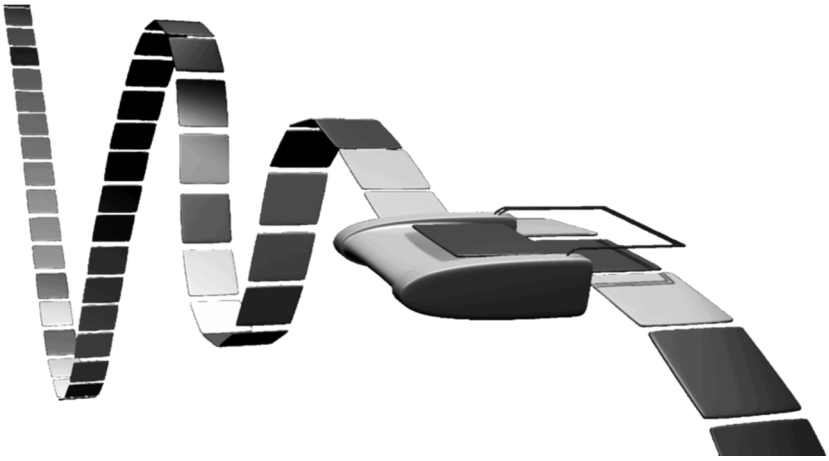

Ilustración 19 Cinta lectora

Este tipo de modelo computacional se considera débil, es decir, está limitado a un cierto número de tareas. Hay principalmente dos tipos:

- **Autómatas de estado finito determinista**s: este tipo de autómatas es un sistema determinista, es decir, los futuros estados del sistema no dependen del azar y solo hay una transición para cada estado, esto significa que no pueden darse transiciones del tipo:

  1: $\delta(q,a) = q_1 \;\; \delta(q,a) = q_2 \;\; q_1 \neq q_2$
  2: $\delta(q,\varepsilon)$

En el segundo caso ε se refiere a la cadena vacía, esto significa que un autómata determinista no puede tener transiciones vacías, para cambiar de estado debe siempre procesar datos de entrada.

- **Autómatas de estado finito no deterministas**: en este tipo de autómatas puede existe más de una transición posible para un estado y, además, puede haber transiciones vacías, lo que permite cambiar de estado sin procesar una entrada.

3.2.2 Representación

Este tipo de autómatas pueden ser representados de las siguientes maneras:

- **Diagrama de estados** (grafos). Este tipo de diagramas permite ver de un vistazo rápido y claramente los diferentes estados y las transiciones entre ellos.

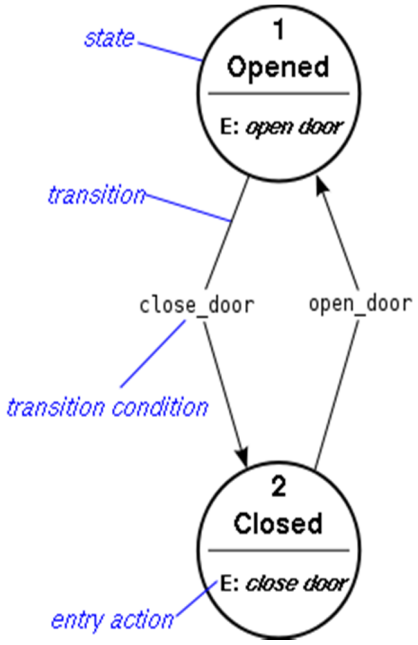

Ilustración 20: Diagrama de estados

- **Tabla de transiciones**: este tipo de tablas son tablas de verdad/matrices de estado.

State transition table

Current state → Input ↓	State A	State B	State C
Input X	...	...	...
Input Y	...	State C	...
Input Z	...	...	...

Ilustración 21 Tabla de transiciones

Para el proyecto se ha elaborado el siguiente autómata:

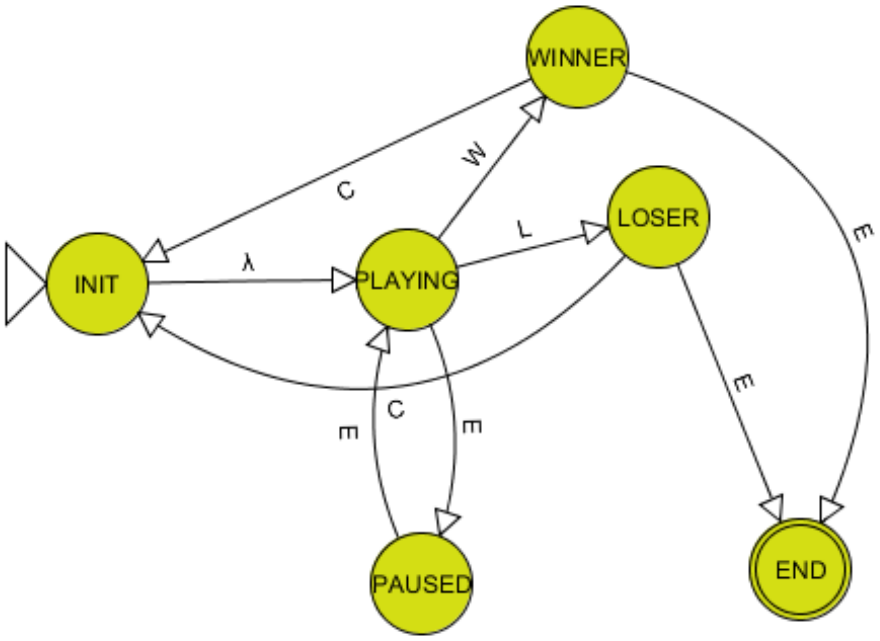

Ilustración 22 Autómata BreakArk

state	>	successor
PLAYING	W	WINNER
PLAYING	L	LOSER
PLAYING	E	PAUSED
PAUSED	E	PLAYING
LOSER	E	END
WINNER	E	END
INIT	λ	PLAYING
WINNER	C	INIT
LOSER	C	INIT

Ilustración 23: Tabla de transiciones BreakArk

Las transiciones son las siguientes:

- WIN (W)
- LOSE (LOSE)
- ESC (E: ESCAPE)
- CONTINUE (C)

El grueso de la aplicación radica en el estado PLAYING, en el que se realizan la mayoría de las funciones del servidor y el cliente, como puede ser actualizar la simulación o renderizar los objetos en pantalla.

Para comunicar la transición por la red se usan una serie de órdenes, algunas exclusivas del servidor y otras exclusivas del cliente. Esto se ha pensado así para independizar ciertos aspectos del cliente del servidor, como la gestión de las teclas en el cliente relacionadas con el pausado, reiniciado o cierre del juego o el paso de nivel en el servidor que implica un reseteo de la escena en el cliente. Las órdenes son las siguientes:

```
NONE, NEXT, PAUSE, CONTINUE, EXIT, WIN, LOSE, LIFE
```

- NONE: indica la ausencia de orden.
- NEXT: orden usada por el servidor para señalizar al cliente que se pasa al siguiente nivel.
- PAUSE: orden exclusiva del cliente para indicar al servidor que el jugador ha pausado el juego (pulsando la tecla Esc).
- CONTINUE: usada por el cliente para señalar al servidor la reanudación del juego, es decir, el jugador ha pulsado Esc mientras el juego estaba pausado.

- EXIT: el cliente lanza esta orden cuando el usuario cierra la ventana del juego para indicar al servidor del cierre.

- WIN: exclusiva del servidor, esta orden permite a este señalar al cliente que el jugador ha ganado la partida.

- LOSE: homóloga a la anterior orden

3.2.3 Usos

Los autómatas de estado finito tienen diversos usos en la industria del software, entre ellos está:

- Reconocimiento de lenguajes formales.

- "Refactorización" del código. Esto se conoce de manera casual como limpiar el código, consiste en reestructurar el existente sin cambiar su funcionalidad, de manera que mejore la claridad (más organización, más facilidad de lectura) y más escalable y facil de mantener. Si la aplicación tiene un comportamiento que varía dependiendo del estado en el que esté, implementar autómatas hace que el código sea mucho más limpio y manejable.

- Programación de Inteligencia Artificial.

- Modelado de sistemas dinámicos, es decir, el modelado de sistemas que tengan un comportamiento que cambie con el tiempo.

3.3 Arquitectura cliente-servidor

La arquitectura cliente-servidor es un modelo de aplicación distribuido en el que se dividen las funciones del programa en dos entidades diferentes: cliente y servidor. Estas dos entidades están claramente diferenciadas. El cliente demanda servicios, realizando peticiones al servidor, que es el que los provee. Es posible ver al servidor como un proveedor de servicios y a los clientes como demandantes de servicios.

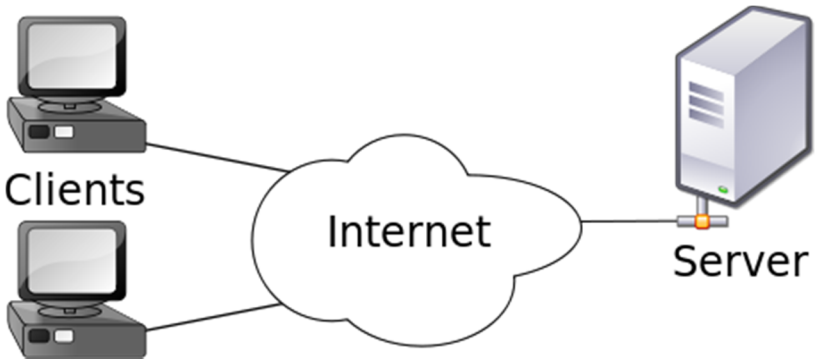

Ilustración 24 Cliente-Servidor

Esta arquitectura contrasta con el Peer to Peer (P2P), un modelo de aplicación descentralizada en el que no hay clientes ni servidores, sino que todos los nodos de la red tienen el mismo tratamiento. Normalmente este tipo de redes se usa para intercambio de ficheros, Voice over IP (VoIP), sistemas de ficheros distribuidos...

Ilustración 25: Peer-to-Peer

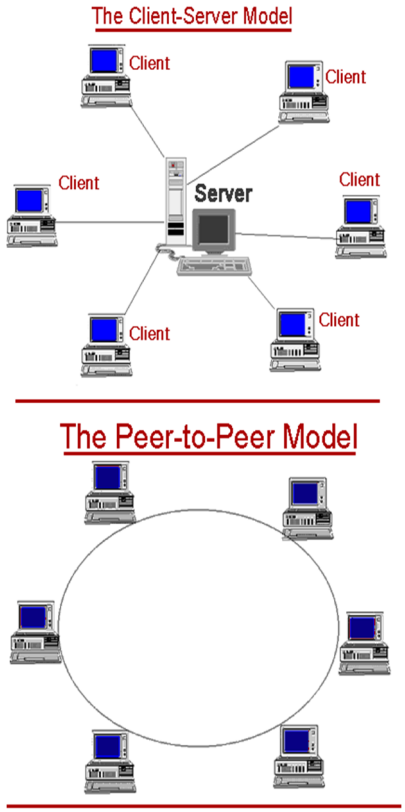

Ilustración 26 Cliente-Servidor frente Peer-to-Peer

A continuación se muestran las ventajas y desventajas de la arquitectura cliente-servidor.

3.3.1 Ventajas y desventajas de la arquitectura cliente-servidor

Ventajas

- Centralización: este tipo de arquitectura permite centralizar toda la administración en un solo punto, el servidor, que permite controlar tanto los permisos de acceso como la asignación de los recursos. Asimismo, mejora la capacidad de mantener los datos y la arquitectura así como la actualización de esta última.

- Copia de seguridad: al estar los datos centralizados se facilita sustancialmente la copia de seguridad y la recuperación de estos.

- Seguridad: al implementar listas de acceso y controlar los permisos de acceso podemos aumentar la seguridad.

- Escalabilidad: tener este tipo de arquitectura permite realizar cambios, si es necesario actualizar el software o aumentar las capacidades del servidor o añadir otro, solo cambiando la parte del servidor. El cliente no necesita ser modificado.

Desventajas

- **Congestión**: la congestión es un problema recurrente en este tipo de sistemas, ya que a mayor número de clientes, mayor número de peticiones y un incremento considerable de la congestión.

- **Robustez**: el hecho de que este modelo de aplicación distribuido sea centralizado hace que se tenga un solo punto de fallo, el servidor, por lo que no es tan robusta como puede ser, por ejemplo, una red Peer to Peer. Esto se mejora incrementando la redundancia, es decir, incrementando el número de servidores que sirven a los clientes.

- **Costes**: mantener una arquitectura de este tipo no es barato, hay que disponer de un hardware potente capacitado para poder procesar todas las peticiones de los clientes, así como un software lo más eficiente posible, como, por ejemplo, las bases de datos (BBDD). Asimismo la arquitectura debe ser mantenida por personal de comunicaciones cualificado, tanto en la parte de gestión de los servidores como en la de la red, es necesario tener una red correctamente dimensionada y en buenas condiciones.

3.3.2 Flujo de ejecución del servidor y el cliente

Uno de los objetivos fundamentales es obtener una arquitectura cliente servidor, en el que un cliente se conecte al servidor y toda la "inteligencia" del juego (detección de colisiones, cálculo de estados, etc) se realiza en este último, siendo la función del primero la de recoger la entrada y renderizar los objetos en pantalla. A continuación se ven las funciones de cada uno de manera simplista, con el objetivo de visualizar claramente estas, lo que permitirá realizar la implementación con unos objetivos claros:

Servidor

- Esperar clientes.
- Si llega un paquete:
    - Si es nuevo cliente: crear conexión.
    - Ya existe el cliente: procesar paquete.
- Procesar datos del cliente. Dependiendo del evento:
    - Pausar/reanudar juego.
    - Salir del juego.
    - Actualizar estado de la simulación:
        - Actualización del estado de los objetos.
        - Detectar y resolver colisiones.
        - Comprobar condiciones de victoria/derrota.

        - Comprobar si es posible cambiar de nivel y en caso afirmativo hacerlo.

        - Enviar el estado de los objetos y órdenes al cliente.

Cliente

- Iniciar cliente, objetos, constantes, etc.

- Conectarse con servidor:

    - Local (localhost)

    - Remoto (IP)

- Recoger entrada: guardar las teclas pulsadas por el usuario y eventos:

    - Pausar/reanudar juego.

    - Salir.

    - Enviar entrada y órdenes al servidor.

    - Recibir estados actualizados de los objetos y órdenes del servidor:

        - Ejecutar órdenes del servidor y cambiar de estado si es necesario.

        - Actualizar objetos locales.

    - Renderizar los objetos basándose en el estado del juego.

Todas estas funciones, menos el envío y recepción de datos, no son propias del cliente y el servidor, sino del juego, con lo que se encapsularán en dos entidades llamadas

ServerGame y ServerClient. De esta manera se consigue dividir el concepto de red y el de juego, independizándolos.

A su vez, se crearán dos entidades llamadas Server y Client, que se encargarán fundamentalmente de enviar y recibir los datos y pasárselos al juego.

3.3.3 Protocolos de transporte

Ya que uno de los puntos claves del proyecto es conseguir una arquitectura cliente-servidor, hay que considerar el protocolo de transporte que se usará (por motivos obvios se considera que el protocolo de red usado es IP). Existen dos opciones:

Transmission Control Protocol (TCP)

- Protocolo enfocado a conexión:
- 3-way handshake para conexión.
- 4-way handshake para desconexión.
- Fiable:
- Control del flujo de datos.
- Mecanismos ante la pérdida de paquetes (ack)
- Segmentos como unidad de datos.
- No apropiado para aplicaciones en tiempo real o con necesidad de baja latencia debido a la sobrecarga que lleva la fiabilidad y la creación de conexiones.

User Datagram Protocol (UDP)

- Protocolo no orientado a conexión.
- Datagramas como unidad de datos.
- No hay fiabilidad:
- Sin control de flujo.
- Sin mecanismos ante la pérdida de paquetes.
- Apropiado para aplicaciones en tiempo real o con necesidades de baja latencia.

3.3.4 Elección del protocolo de transporte

Los videojuegos son aplicaciones que trabajan con grandes restricciones, debido al gran número de computaciones necesarias en un tiempo muy corto para poder mostrar correctamente la información), con lo que se pueden considerar que son aplicaciones en tiempo real. Si además de esto se añade que se debe transmitir tráfico importante (como las físicas) a través de la red se llega a la conclusión de que TCP no es la opción a seguir, sino UDP.

Para la mayoría de los juegos comerciales no solo basta con UDP, es necesario tener conexiones entre cliente y servidor y, además, tener algún sistema de control de flujos y fiabilidad. Es posible que se cuestione entonces el uso de UDP si es necesario usar características que brinda TCP.

3.3.5 Diseño de arquitectura de red

Se requiere tener ciertos mecanismos para crear una conexión virtual entre los clientes y los servidores y un sistema de control de flujos, pero se requiere tener estos mecanismos sin la gran sobrecarga añadida que presenta TCP. Para esto se creará una entidad llamada Connection, que permitirá obtener las ventajas de una conexión virtual sobre UDP. También se necesitará crear una entidad socket llamada UDPSocket que encapsule todas las funciones del socket UDP, de manera que la clase Connection contenga un socket por el que poder comunicarse con el otro extremo.

Para conseguir esta conexión virtual se añade un id de protocolo único en los datos del paquete UDP, de tal manera que el servidor solo acepte paquetes que contengan este id y, además, si no existe una conexión previa con el emisor del paquete, se crea.

Las entidades Server y Client harán uso de esta clase Connection para realizar una conexión virtual entre ellos. Para permitir que el servidor pueda tener varias conexiones, se puede tener un vector de conexiones en el que se vayan creando estas según sean necesarias.

Por tanto, se tiene el servidor, que se considera autoritativo, es decir, la simulación que se ejecuta en este siempre la verdadera, que recibe las teclas pulsada por el usuario más las ordenes y actualiza la simulación del mundo, mandando a continuación los datos actualizados al cliente. Hay que notar que el servidor no depende de los datos del cliente para simular el juego (esto haría que el jugador tuviera que estar constantemente pulsando las teclas para ver como la pelota se mueve y que cuando no pulsara nada el juego estuviera congelado), sino que, una vez empiece este, la simulación siempre se ejecuta con los datos que posee el servidor en ese momento. Cuando llegue el paquete del cliente con la nueva posición de la raqueta, esta se actualiza y a continuación se simula el mundo.

3.3.5.1 Paquetes

Es especialmente importante optimizar el tamaño de los paquetes, solo enviando lo que realmente se ha actualizado y es estrictamente necesario. Para el cliente los campos que conformarán los datos del paquete serán los siguientes

- Protocol Id.
- Input (solo flechas seleccionadas por el servidor).
- Orden.

Los campos de datos que contendrán los paquetes del servidor son los siguientes:

- Protocol Id.
- Posición x de la pelota.
- Posición y de la pelota.
- Posición x de la raqueta.
- Nº de bloques destruidos.
- Array conteniendo el id de los bloques destruidos.
- Orden.

3.3.5.2 Clases

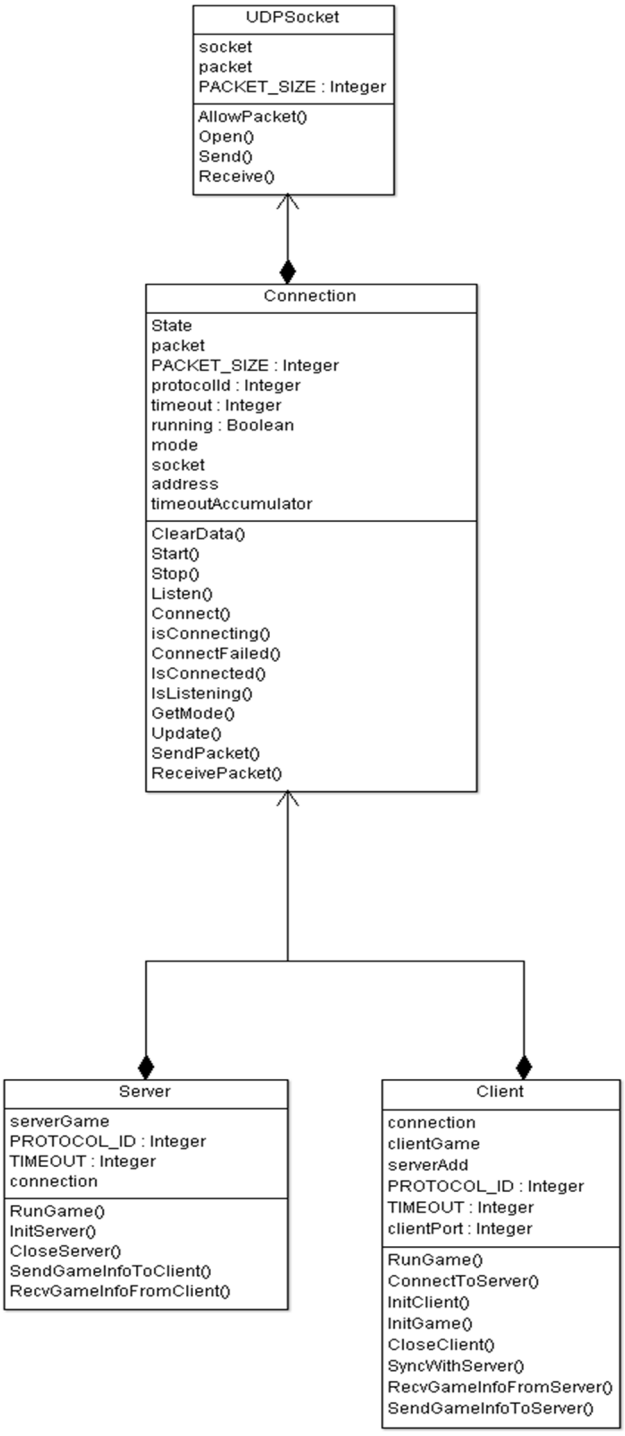

Ilustración 27: Clases arquitectura cliente-servidor

3.4 Conclusión y consideraciones

Una vez realizado la arquitectura cliente servidor, nada impide tener un juego con un solo jugador en una misma máquina ejecutando el cliente y el servidor encima. Es indiferente dónde se encuentra el cliente o el servidor. También es importante notar que no hace falta que el servidor y el cliente se ejecuten a una misma velocidad. El cliente puede ir a 60 fotogramas por segundo mientras que el servidor se actualiza a 30.

Se puede ir un paso más allá creando mecanismos de control de flujo fiabilidad, pero cada nueva funcionalidad incrementa la complejidad y el tiempo de desarrollo, con lo que se ha optado por no implementar estas funciones.

Otro aspecto del diseño de juegos en red es el de la predicción en el lado del cliente [4]. En un principio los juegos en red consideraban (y consideran) al servidor autoritativo (tiene la última palabra) y al cliente como un terminal tonto que simplemente manda la entrada al servidor y dibuja en pantalla lo que le dice este último. Esto funciona bien en entornos donde hay un gran ancho de banda y poca latencia como puede ser el juego a través de la LAN, pero a través de Internet (teniendo en cuenta que en un principio se usaban modems con velocidades bajísimas comparado con las actuales como, por ejemplo, 56k) este método es inviable, es ahí cuando se decidió crear el concepto de predicción en el lado del cliente. Esto consiste en que, el cliente, además de enviar la entrada al servidor, calcula localmente el estado de los objetos, como, por ejemplo, la posición del jugador, de esta manera este no nota ningún retraso entre la acción de pulsar una tecla y el movimiento del personaje en pantalla. Mientras el jugador no interaccione con otros jugadores directa o indirectamente, la predicción del cliente será exactamente igual o muy aproximada a la del servidor, pero en el momento en que alguna interacción ocurra (como puede ser una explosión provocada por un misil de otro jugador que desplaza al jugador original) se verá que la predicción del cliente es incorrecta y, por tanto, será sobrescrita por el servidor. Para evitar que este salto abrupto a la nueva posición no sea notado por el jugador (es importante notar que esta nueva posición esta en el pasado, con lo que invalida toda la predicción del cliente hasta el momento actual), el cliente mantiene un buffer circular en el que guarda todas las entradas del jugador y estados anteriores y,

descartando todos los estados más viejos que el tiempo de la nueva posición, vuelve a simular el juego desde la posición indicada por el servidor con todas las entradas emitidas por el jugador hasta ahora, como si rebobinara hasta el momento indicado por el servidor y simulara el juego de nuevo.

Esta funcionalidad no ha sido tampoco implementada por cuestiones de tiempo y complejidad, aunque es una propuesta para futuros proyectos que quieran profundizar en este ámbito.

3.5 Clases arquitectura final

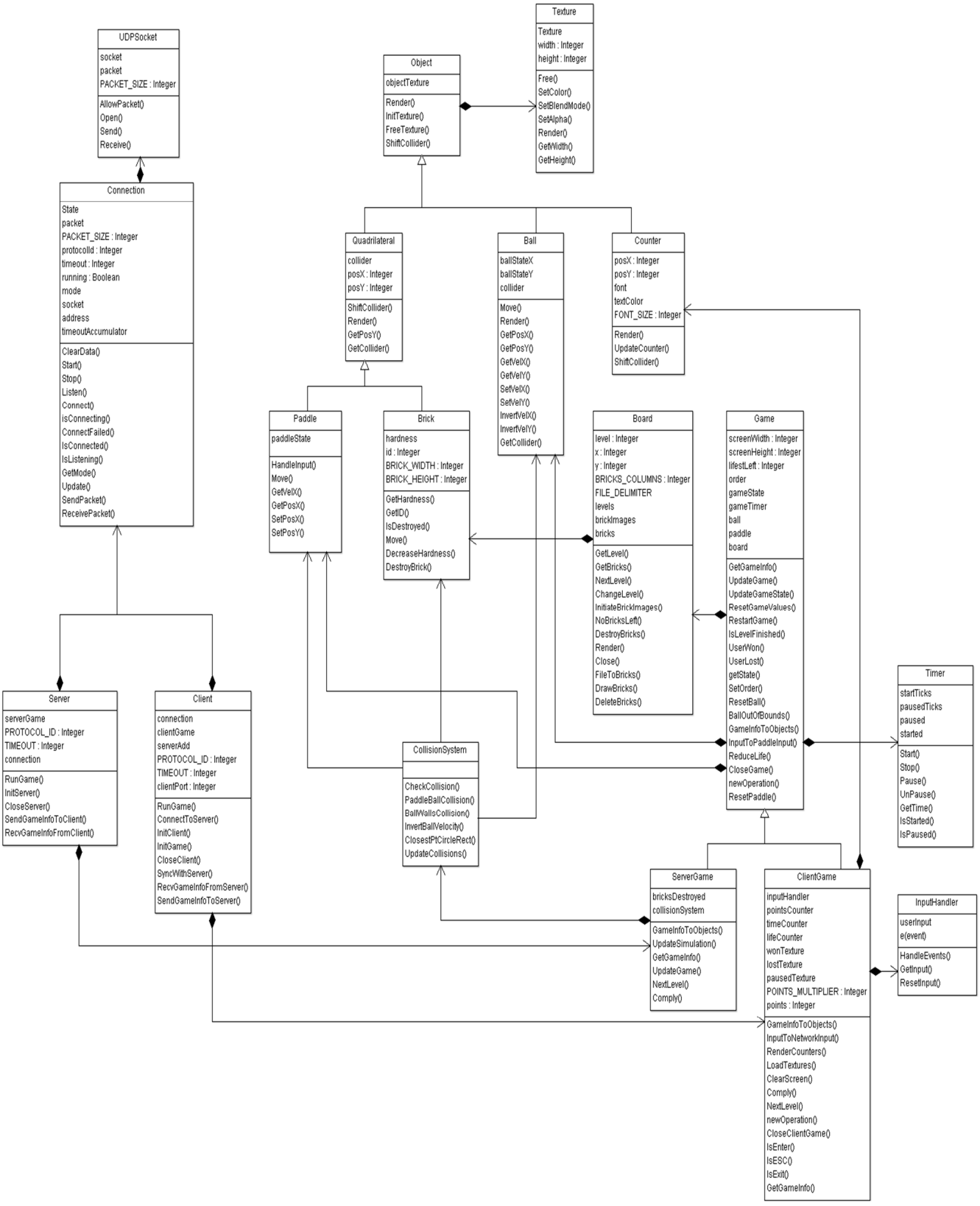

Ilustración 28 Clases arquitectura final

4. Capítulo 4: Implementación

4.1 Código de red

Como se vio en el capítulo anterior, se han diseñado varias clases con el objetivo de dotar al programa de una arquitectura cliente-servidor. Para ello se ha creado una clase que encapsula un socket UDP y otra que permite crear una conexión virtual encima de UDP usando un id de protocolo.

4.1.1 UDPSocket

A continuación se muestran los métodos más importantes de la clase UDPSocket, que permiten abrir un socket y enviar y recibir datos por la red usando el protocolo de transporte UDP.

```
bool UDPSocket::Open(int port)
{
	if (!(socket = SDLNet_UDP_Open(port)) || !AllocPacket(&packet))
	{
		return false;
	}

	return true;
}
```

```
bool UDPSocket::Send(const IPaddress &destination, const void * data, size_t size)
{
	packet->data[0] = 0;

	//Fill packet
	memcpy((char*)packet->data, data, size);
	packet->address.host = destination.host;
	packet->address.port = destination.port;
	packet->len = size+1;

	//Send packet. Return false if the packet cannot be sent;
	if (!SDLNet_UDP_Send(socket, -1, packet))
	{
		return false;
	}

	return true;
}



size_t UDPSocket::Receive(IPaddress& sender, void * data)
{
	if (socket == nullptr)
	{
		return 0;
	}

	packet->data[0] = 0;

	if (SDLNet_UDP_Recv(socket, packet))
	{
		sender.host = packet->address.host;
		sender.port = packet->address.port;

		memcpy(data, (char*)packet->data, packet->len);

		return packet->len;
	}

	return 0;
```

Es importante notar que los datos que se introducen en el paquete han sido convertidos previamente a Big Endian por el servidor y el cliente.

4.1.2 Connection

A continuación se muestran las funciones más importantes de esta clase, que permiten crear un socket usando la clase anterior, poner un servidor en escucha o conectarse a uno, así como recibir y enviar datos.

```
bool Connection::Start(int port)
{
	if (running)
	{
		std::cerr << "connection is already running" << std::endl;
		return false;
	}
	else
	{
		if (!socket.Open(port))
		{
			return false;
		}

		running = true;
		return true;
	}
}
```

```
void Connection::Listen()
{
	ClearData();
	mode = Server;
	state = Listening;
}
```

```
void Connection::Connect(const IPaddress &address)
{
	ClearData();
	mode = Client;
	state = Connecting;
	this->address = address;
}
```

```
bool Connection::SendPacket(const unsigned char data[], size_t size)
{

	if (!running)
	{
		std::cerr << "connection is not running" << std::endl;
		return false;
	}

	if (address.host == 0)
	{
		std::cerr << "address host is empty";
		return false;
	}

	//Insert the protocolId + data into the packet data
	unsigned char packet[PACKET_SIZE];
	Uint32 protocolIdAux = SDL_SwapBE32(protocolId);

	memcpy(packet, &protocolIdAux, sizeof(Uint32));
	memcpy(packet + sizeof(Uint32), data, size);

	return socket.Send(address, packet, sizeof(Uint32)+size);
}



int Connection::ReceivePacket(unsigned char data[])
{
	if (!running)
	{
		std::cerr << "connection is not running" << std::endl;
		return 0;
	}

	IPaddress sender;
	unsigned char packet[PACKET_SIZE];

	size_t bytes = socket.Receive(sender, packet);

	//If packet has no data or only the protocolId return 0
	if (bytes <= sizeof(Uint32))
	{
		return 0;
	}

	Uint32 protocolIdAux = 0;
	memcpy(&protocolIdAux, packet, sizeof(Uint32));

	if (protocolId != SDL_SwapBE32(protocolIdAux))
	{
		return 0;
	}

	//If packet comes from a client the server does not know anything about then
server connects with the client (just one client at the moment)
	if (mode == Server && !IsConnected())
	{
		state = Connected;
		address = sender;
	}
```

```
	if (sender.host == address.host && sender.port == address.port)
	{
		if (mode == Client && state == Connecting)
		{
			std::cout << "client completes connection with server" <<
std::endl;
			state = Connected;
		}
		timeoutAccumulator = 0.0f;
		memcpy(data, packet + sizeof(Uint32), bytes - sizeof(Uint32));

		return bytes - sizeof(Uint32);
	}

	return 0;
}
```

Es importante notar que el protocolo id se convierte a Big Endian usando la funcion SDL_SwapBE32 y luego es convertido a Little Endian usando la misma función al recibir un paquete. También es importante pasar a las funciones que se encargan de enviar los datos a través de la red el tamaño exacto de lo que se envía, así se evita hacer suposiciones sobre el tamaño, lo que puede llevar a errores como acceder a un lugar de memoria no utilizado.

4.1.3 Server

La clase servidor se encarga de la comunicación con el cliente, a la vez que llama a ServerGame, que contiene toda la información de la lógica del juego.

La función más importante en esta clase es RunGame, que es la que ejecuta el juego, recibe y envía los datos y actualiza la simulación del mundo. A continuación se muestra parte de esta función, en la que se ve la mayor parte del funcionamiento del juego:

```
	switch (serverGame->GetState())
	{
	caseGame::GameStates::INIT:caseGame::GameStates::PLAYING:

		if (serverGame->UserWon())
		{
			serverGame->SetOrder(Game::GameOrders::WIN);
			serverGame->UpdateGameState(Game::GameStates::WINNER);
		}
		else if (serverGame->UserLost())
		{
```

```
                        serverGame->SetOrder(Game::GameOrders::LOSE);
                        serverGame->UpdateGameState(Game::GameStates::LOSER);
                }
                else if (serverGame->IsLevelFinished())
                {
                        serverGame->NextLevel();
                        serverGame->SetOrder(Game::GameOrders::NEXT);
                }


                if (connection->IsConnected())
                {
                        serverGame->UpdateSimulation(t, dt);
                        SendGameInfoToClient();
                }

                break;

        case Game::GameStates::PAUSED:
                break;

        case Game::GameStates::WINNER: case  Game::GameStates::LOSER:

                //Waiting for a CONTINUE order from the client to restart the game
                if (serverGame->GetOrder() == Game::GameOrders::CONTINUE)
                {
                        serverGame->RestartGame();
                }
                break;

        case Game::GameStates::END:
                return true;}
```

4.1.4 Client

Al igual que en la anterior clase, el cliente se encarga del envío y recepción de los paquetes provenientes del cliente y hacia el cliente. Ejecuta la función RunGame, que lleva el grueso de la aplicación en la parte del cliente. A su vez esta clase obtiene la entrada del jugador, renderiza todos los objetos en pantalla y se encarga de gestionar las teclas y las pantallas relacionadas con la victoria, pérdida o pausa del juego delegando en ClientGame

```
switch (clientGame->GetState())

{
   case Game::GameStates::INIT:

                clientGame->RenderGame(renderer);
                clientGame->UpdateGameState(Game::GameStates::PLAYING);
```

```
		break;

	case Game::GameStates::PLAYING:

		if (!renderScreen)
		{
			//Clear renderScreen
			renderScreen = true;
		}

		data[0] = 0;
		if (RecvGameInfoFromServer(data))
		{
			clientGame->UpdateGame(data);
			clientGame->Comply(renderer);
			clientGame->RenderGame(renderer);
		}

		break;

	case Game::GameStates::PAUSED:

	// Render the paused screen just once in a row
		if (renderScreen)
		{
			clientGame->RenderGame(renderer);
			renderScreen = false;
		}
		break;

	case Game::GameStates::WINNER: case  Game::GameStates::LOSER:

		// Render the winner or loser screen just once in a row
		if (renderScreen)
		{
			clientGame->RenderGame(renderer);
			renderScreen = false;
		}
		break;

	case Game::GameStates::END:
		return true;
	}
```

4.2 Código del juego

4.2.1 ServerGame y ClientGame

ServerGame se encarga de la lógica del juego, entre otras cosas de actualizar la simulación, pasar las colisiones al sistema de colisiones para que las detecte y responda en consecuencia o dar los datos del estado del juego a Server para que proceda a enviarlos al cliente.

En UpdateSimulation se ve cómo ServerGame mueve los objectos y detecta las colisiones.

```
void ServerGame::UpdateSimulation(double t, double dt)
{
	// Move objects, detect and react to collisions

	ball->Move(t, dt);
	paddle->Move(t, dt);

	bricksDestroyed.clear();
	collisionSystem.UpdateCollisions(*ball,    *paddle,    board->GetBricks(),
screenWidth, screenHeight, t, dt, bricksDestroyed);

	if (BallOutOfBounds())
	{
		ResetBall();
		ReduceLife();

		order = LIFE;
	}
}
```

Además, en GetGameInfo se puede observar como se pasa la información del juego a Server (se muestra parte del código):

```
size_t ServerGame::GetGameInfo(unsigned char* data)
{
	//Fill NetworkGameState structure in Big Endian (network byte order)
	NetworkGameState state;

	state.ballX = SDL_SwapBE32((Uint32)ball->GetPosX());
	state.ballY = SDL_SwapBE32((Uint32)ball->GetPosY());
	state.paddleX = SDL_SwapBE32((Uint32)paddle->GetPosX());
	state.idQuantity = SDL_SwapBE32((Uint32)bricksDestroyed.size());
	state.order = SDL_SwapBE32((Uint32)order);

. . .

}
```

ClientGame cumple una función similar en la parte del cliente, a continuación el código que renderiza la escena en pantalla.

```
void ClientGame::RenderGame(SDL_Renderer* renderer)
 {
  ClearScreen(renderer);

  switch (gameState)
```

```
{
case INIT: case PLAYING:
        RenderCounters(renderer);
        ball->Render(renderer);
        paddle->Render(renderer);
        board->Render(renderer);

        break;

case WINNER:
        wonTexture->Render(0, 0, renderer);
        break;

case LOSER:
        lostTexture->Render(0, 0, renderer);
        break;

case PAUSED:
        pausedTexture->Render(0, 0, renderer);

        break;
}

SDL_RenderPresent(renderer);
}
```

4.2.2 InputHandler

InputHandler permite obtener la entrada del jugador y depositarla en una estructura, que es leída por el cliente.

Es importante notar cómo HandleInput para cuando se ha obtenido uno de los eventos que se buscaba, devolviendo true, si esto no se hace así el bucle no para hasta que no hay más eventos.

```
while (SDL_PollEvent(&e) != 0)
{
        ResetInput();

        //If a key was pressed
        if (e.type == SDL_KEYDOWN && e.key.repeat == 0)
        {
                switch (e.key.keysym.sym)
                {
                case SDLK_LEFT:   userInput.leftDown = true; break;
                case SDLK_RIGHT:  userInput.rightDown = true; break;
                case SDLK_ESCAPE: userInput.esc = true; break;
                case SDLK_RETURN: userInput.enter = true; break;
                }
```

```
		event = true;
	}

	//If a key was released
	else if (e.type == SDL_KEYUP && e.key.repeat == 0)
	{
		switch (e.key.keysym.sym)
		{
		case SDLK_LEFT:   userInput.leftUp = true; break;
		case SDLK_RIGHT:  userInput.rightUp = true;  break;
		case SDLK_ESCAPE: userInput.esc = false; break;
		case SDLK_RETURN: userInput.enter = false; break;
		}
		event = true;
	}

	else if (e.type == SDL_QUIT)
	{
		userInput.quit = true;
		event = true;
	}
```

4.2.3 CollisionSystem

En esta clase se puede observar la detección de las colisiones. A continuación se muestra lo comentado en el diseño acerca de cómo detectar colisiones entre AABB-Círculo:

Cálculo del punto más cercano:

```
void   CollisionSystem::ClosestPtCircleRect(const Circle& a, const SDL_Rect& b,
int& pX, int& pY)
{
	// If the center is outside the rectangle, clamp it to the rectangle
	// Else the center is the closest point

	// Closest point to the collision box
	double x, y;

	x = a.x;
	y = a.y;

	// Find closest x offset
	if (a.x < b.x)
	{
		x = b.x;
	}
	if (a.x > b.x + b.w)
	{
		x = b.x + b.w;
	}

	//Find closest y offset
	if (a.y < b.y)
	{
		y = b.y;
```

```
        }
    if (a.y > b.y + b.h)
        {
                y = b.y + b.h;
        }

        pX = x;
        pY = y;
 }
```

Cálculo de la superposición entre AABB y círculo:

```
bool CollisionSystem::CheckCollision(const Circle& a, const SDL_Rect& b)
{
 //Closest point on collision box
 int pX, pY;
 ClosestPtCircleRect(a, b, pX, pY);

 //If the Closest point is inside the circle
 return DistanceSquared(a.x, a.y, pX, pY) < a.r * a.r;
}
```

4.2.4 Integration

En el archivo de cabecera Integration se observa el integrador RK4:

```
Derivative Evaluate(const State &Initial, double t)
{
        Derivative output;
        output.dx = Initial.v;
        output.dv = Acceleration(Initial, t);
        return output;
}

Derivative Evaluate(const State &Initial, double t, double dt, const Derivative &d)
{
        State state;
        state.x = Initial.x + d.dx*dt;
        state.v = Initial.v + d.dv*dt;
        Derivative output;
        output.dx = state.v;
        output.dv = Acceleration(state, t + dt);
        return output;
}

void Integrate(State &state, double t, double dt)
{
        Derivative a = Evaluate(state, t);
        Derivative b = Evaluate(state, t, dt*0.5f, a);
        Derivative c = Evaluate(state, t, dt*0.5f, b);
        Derivative d = Evaluate(state, t, dt, c);

        const double dxdt = 1.0f / 6.0f * (a.dx + 2.0f*(b.dx + c.dx) + d.dx);
        const double dvdt = 1.0f / 6.0f * (a.dv + 2.0f*(b.dv + c.dv) + d.dv);

        state.x = state.x + dxdt*dt;
```

```
        state.v = state.v + dvdt*dt;
}
```

4.2.5 Cliente y servidor actualizados a diferente velocidades

En este pequeño trozo de código se ve como se actualiza el cliente a 60 fotogramas por segundo mientras que el servidor lo hace a 30.

```
dtReal = 1.0 / 60.0;

if (local)
{
        dtServer = 0.0;

        timer.Start();
        tBegin = timer.GetTime();

        while (!quit)
        {
                //Server runs at 33,33ms intervals (30 FPS)
                dtServer += dtReal;

                if (dtServer >= 0.033)
                {
                        localServer.RunGame(timer.GetTime(), 0.033);
                        dtServer -= 0.033;
                }

                //Run client at maximum frame rate (60 FPS)
                quit = client.RunGame(renderer, dtReal);

                double tEnd = timer.GetTime();
                dtReal = tEnd - tBegin;
                tBegin = tEnd;
        }
```

4.3 Demostración

4.3.1 Capturas del juego

Nivel 1:

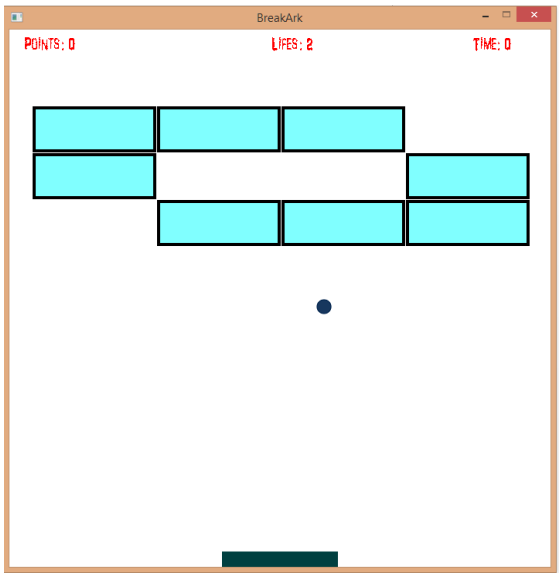

Ilustración 29 Nivel 1

Nivel 2:

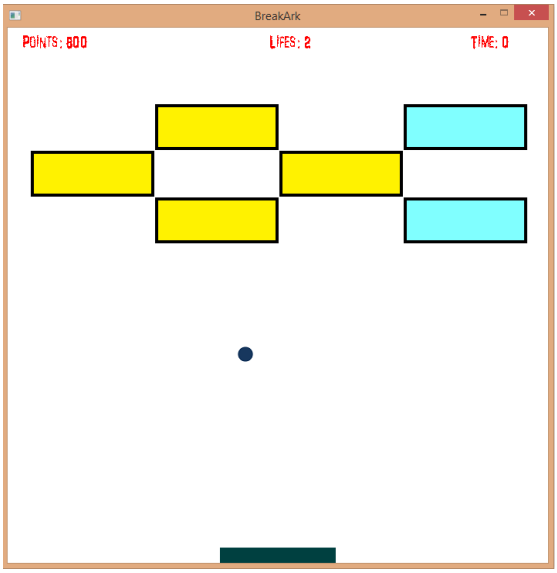

Ilustración 30 Nivel 2

Nivel 3:

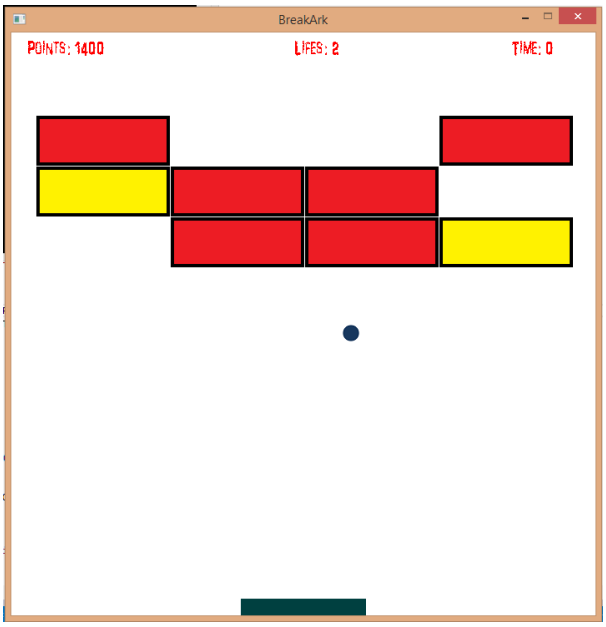

Ilustración 31 Nivel 3

Juego pausado:

Ilustración 32 Juego pausado

Jugador pierde:

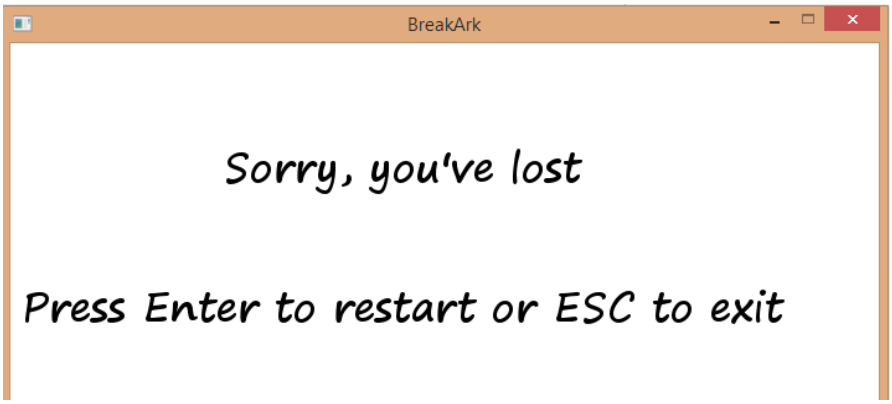

Ilustración 33 Jugador pierde

Jugador gana:

Ilustración 34 Jugador gana

4.3.2 Capturas del entorno de desarrollo

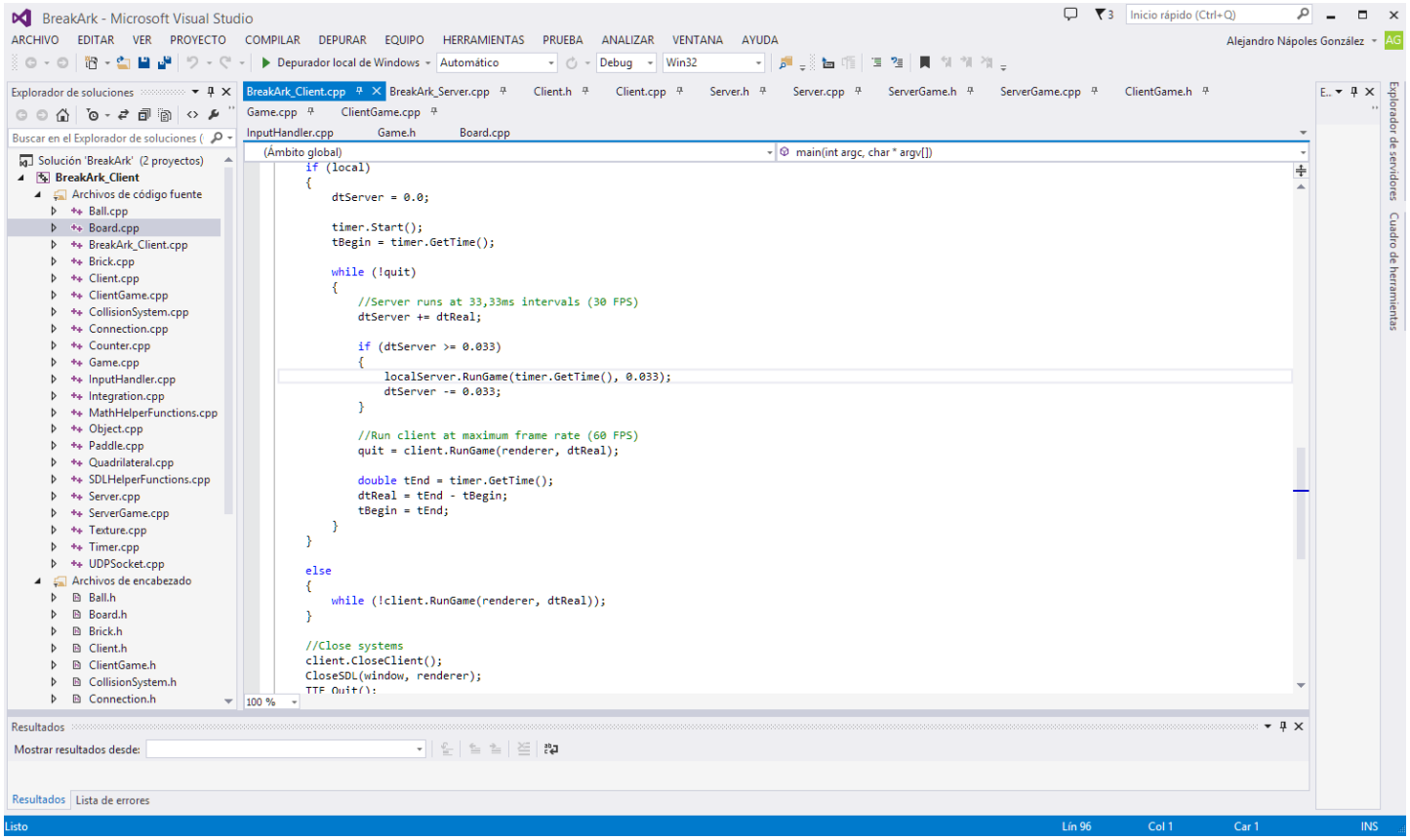

Ilustración 35 Visual Studio 2013

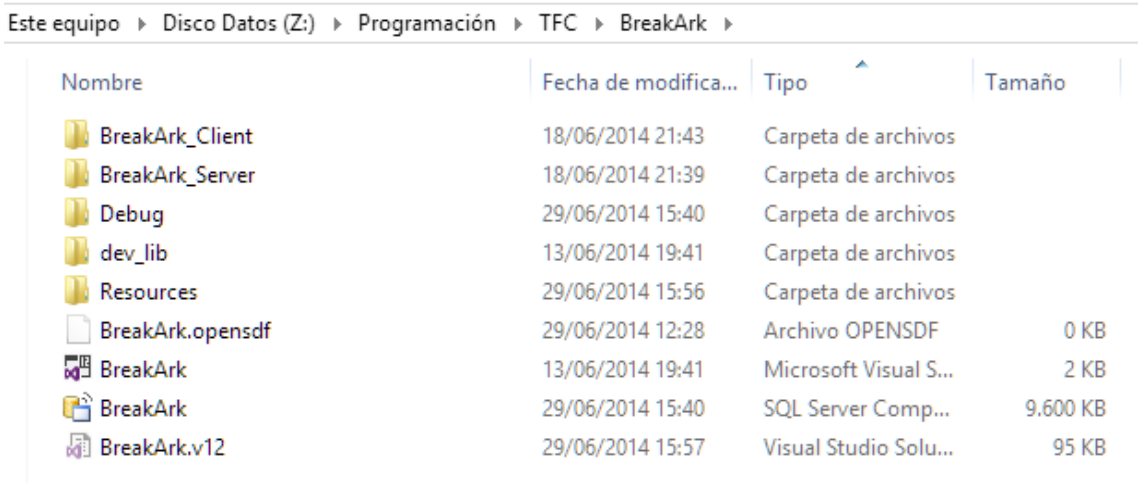

Ilustración 36 Directorio del proyecto

5. Capítulo 5: Conclusiones

5.1 Lecciones aprendidas

- La estimación del tiempo de desarrollo de software es complicado de estimar, surgen problemas inesperados, cuestiones que aparentemente están cubiertas pueden fallar. A la hora de estimar el tiempo que tomará aplicar una funcionalidad al programa, es mejor estimar añadiendo un cierto margen de seguridad y cuestionarse el coste y tiempo añadido que se suma al proyecto y si merece o no la pena.

- Uso de máquinas de estado finito para establecer claramente los cambios de estado dentro del código, que lo simplifica y lo aclara.

- Programación en C para el apartado de red. Sockets. Arquitectura cliente-servidor. Conexión virtual a través de UDP. Manipulación a nivel de bits. Cambiar de little endian a big endian al enviar los datos por la red. Usar formatos números fijos al pasar datos por la red como Uint32 en vez de int, que dependen de la plataforma.

- Mecánica del sólido rígido. Integrador RK4 frente a Euler. Algebra lineal. Detección de colisión en tiempo real. AABB-AABB, AABB-Círculo.

- Programación gráfica básica. Uso de la biblioteca SDL.

- Diseño e implementación de una aplicación funcional con tecnologías y métodos usados habitualmente en el sector. Conocer cómo es el desarrollo del software y especialmente del videojuego, así como su complejidad y enfrentarse a los diferentes retos que aparecen en el día a día del desarrollo.

- Fortalecimiento de los conocimientos básicos de programación. El diseño e implementación del proyecto ha fortalecido las bases de programación.

5.2 Reflexión crítica

No se han logrado todos los objetivos que se marcaron en un principio. Lo más destacados son la inclusión de varios menús, efectos de sonido al golpear la pelota con otros objetos, tabla de puntuaciones o diferentes reflexiones al golpear la pelota con la raqueta (en vez de rebotar con el mismo ángulo de entrada). Conseguir estos objetivos no ha sido posible por la errónea estimación del tiempo y el coste que suponía elaborar el proyecto. Al centrarse en las bases de este se hizo evidente que por una estimación incorrecta había que eliminar objetivos de la lista y centrarse en las funcionalidades principales del juego.

5.3 Análisis crítico del seguimiento de la planificación

La metodología prevista ha sido la adecuada aunque el seguimiento no ha sido el óptimo. Como se expone en los anteriores puntos, la estimación errónea del coste y tiempo que suponía añadir las funcionalidades básicas del video juego hubo que centrarse en ellas, desestimando las otras.

El gran problema para el autor ha sido la gran incertidumbre al inicio del proyecto, proveniente de la libre elección a la hora de realizar la aplicación, lo que hizo que, tras escoger un tema del que tenía nula experiencia y conocimientos y del que no parecía haber extensa documentación, se tuvo que cambiar radicalmente casi al acabar el plazo de la elección del proyecto.

5.3 Líneas futuras

Hay muchas líneas que se pueden seguir para mejorar el proyecto, a continuación se enumeran las más interesantes:

Seguir extendiendo y mejorando la arquitectura de red:

- Incluir mecanismos de control de flujo y fiabilidad sobre UDP.
- Implementar la predicción en el lado del cliente.
- Añadir debugging síncrono al proyecto.
- Crear una clase abstracta de la que hereden Server y Client.
- Añadir soporte para varios clientes en el mismo servidor. Actualmente solo se permite un cliente.

Lógica de juego:

- Cambiar la manera en la que colisiona la pelota con la raqueta para hacer el movimiento más imprevisible, esto haría el juego mucho más divertido. Posibles ideas son:
- Añadir giro (spin) y parte del momento (momentum) de la raqueta a la pelota.
- Cambiar el ángulo de reflexión según el lado de la raqueta en el que impacte la pelota.
- Aumentar la velocidad de la pelota a medida que vaya pasando el tiempo o se vayan destruyendo bloques.

- Añadir bonificadores que caen de los bloques y otorgan beneficios o desventajas al jugador (es posible que incluya el uso de patrones de diseño).

Programación:

- Revisitar el diseño de las clases e intentar independizar al máximo la parte gráfica y de red de la lógica del juego.

6. Glosario

- **Socket:** Socket designa un concepto abstracto por el cual dos programas (posiblemente situados en computadoras distintas) pueden intercambiar cualquier flujo de datos, generalmente de manera fiable y ordenada. El término socket es también usado como el nombre de una interfaz de programación de aplicaciones (API) para la familia de protocolos de Internet TCP/IP, provista usualmente por el sistema operativo.

- **TCP:** TCP es un protocolo de comunicación orientado a conexión fiable del nivel de transporte, actualmente documentado por IETF en el RFC 793. Es un protocolo de capa 4 según el modelo OSI.

- **UDP:** es un protocolo del nivel de transporte basado en el intercambio de datagramas (Encapsulado de capa 4 Modelo OSI).

- **Integrador:** métodos genéricos iterativos, explícitos e implícitos de integración numérica para resolver ecuaciones diferenciales ordinarias.
- **OOP:** es un paradigma de programación que usa los objetos en sus interacciones, para diseñar aplicaciones y programas informáticos

- **FSM/AF:** autómata finito (AF) o máquina de estado finito es un modelo computacional que realiza cómputos en forma automática sobre una entrada para producir una salida.

7. Bibliografía

[1] Loneliness (05/2014)

http://www.necessarygames.com/my-games/loneliness/flash

[2] Gone Home (06/2014):

http://www.gonehomegame.com/

[3] Programming Principles and Practice Using C++ Bjarne Stroustrup, Addison-Wesley (Pearson Education, Inc.) 2009

[4] Predicción en el lado del cliente (Client Side Prediction) (05/2014):

http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/

[5] Real-Time Collision Detection Christer Ericson, CRC Press 2014.

[6] Game Engine Architecture Jason Gregory, A K Peters, Ltd 2009

[7] Libro blanco ADESE (06/2014)

http://www.micromania.es/blogs/de-ratones-y-juegos/libro-blanco-del desarrollo-la-industria-en-espana

[8] Game Coding Complete Cuarta Edición, Mike McShaffry & David Graham Course Technology PTR 2013

[9] State Machine in Games (05/2014)

http://www.gamedev.net/page/resources/_/technical/game-%20%20programming/state-%20%20machines-in-games-r2982

[10] Documentación SDL_net (04/2014):

http://www.libsdl.org/projects/SDL_net/docs/SDL_net_frame.html

[11] Beej's Guide to Network Programming (04/2014):

http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html

[12] Documentación SDL 2.0 (04/2014):

http://wiki.libsdl.org/Tutorials

[13] Autómata finito. (05/2014):

http://es.wikipedia.org/wiki/Aut%C3%B3mata_finito

http://en.wikipedia.org/wiki/Finite-state_machine

[14] Autómatas de estado finito en videojuegos (04/2014)

http://www.gamedev.net/page/resources/_/technical/game-programming/state-machines-in-games-r2982

[15] Autómatas de estado finito en el desarrollo de videojuegos (04/2014)

http://jessewarden.com/2012/07/finite-state-machines-in-game-development.html

[16] Expresiones regulares y autómatas de estado finito. (04/2014)

http://www.gamedev.net/page/resources/_/technical/general-programming/finite-state-machines-and-regular-expressions-r3176

[17] ¿Por qué nunca usan los desarrolladores autómatas de estado finito? (04/2014)

http://www.skorks.com/2011/09/why-developers-never-use-state-machines/

[18]Arquitectura cliente-servidor. (04/2014)

http://es.wikipedia.org/wiki/Cliente-servidor

http://en.wikipedia.org/wiki/Client%E2%80%93server_model

[19] Depurando videojuegos multijugador. (04/2014)

http://gafferongames.com/networking-for-game-rogrammers/debugging-multiplayer-games/

[20] Redes Peer to Peer. (04/2014)

http://es.wikipedia.org/wiki/P2P

[21] Ventajas y desventajas de la arquitectura cliente-servidor. (04/2014)

http://www.ianswer4u.com/2011/05/client-server-network-advantages-and.html#axzz2xkxnUHMM