Sistemas informáticos

Curso 2008-2009

JUEGOS ARCADE ONLINE SOBRE MAPAS REALES

Víctor Pinto San Macario
Javier Rodríguez Vidal
David Gómez Cordero

Dirigido por:
Prof. Natalia López Barquilla
Dpto. Sistemas informáticos y computación

Facultad de Informática
Universidad Complutense de Madrid

Resumen

Este proyecto implementa juegos arcade, que se desarrollan sobre mapas reales. Están hechos para poder jugar de forma online, a través de internet. La información geográfica necesaria es obtenida de la página de la fundación OpenStreetMap. Es una fundación para crear mapas libre y editables, que cuenta con la colaboración desinteresada de los usuarios de la red. Tenemos dos opciones para elegir el lugar donde se expondrá el juego. Las opciones son: bien jugar sobre unos mapas descargados previamente al realizar la aplicación por los desarroladores, o bien de forma online elegir el lugar sobre el cual queremos jugar. Para esto último nos apoyamos en aplicaciones ofertadas por esta Comunidad como puede ser el namefinder. Nuestra aplicación ha sido desarrollada bajo el lenguaje de programación Java de la empresa Sun Microsystems. Se ha usado como entorno de programación Netbeans, de la misma empresa citada con anterioridad. El proyecto se divide en dos, siguiendo la arquitectura cliente-servidor. El servidor se encarga de toda la computación y el cliente es una página web para poder jugar sobre ella. Para llevar a cabo el cliente se ha usado un applet de Java, que no es más que un programa incrustado en un documento HTML. La conexión entre las dos partes del proyecto citadas se ha realizado a través de sockets. Se ha seguido un protocolo creado por nosotros para que las comunicaciones sean correctas. Los juegos arcades disponibles en la aplicación son: el comecocos, el pilla pilla, la serpiente y el pincha globos.

Palabras clave: juegos, online, socket, applet, OpenStreetMap, cliente, servidor, pilla-pilla, serpiente, comecocos, pincha-globos.

Summary

This project implements arcade games, which are developed on real maps. They are made for playing online, over network. The geographic information required is got from OpenStreetMap foundation´s web page. This foundation creates free geographic data, with the unselfish collaboration of network users. We have two options for choosing the place where we will play. The options are either to play on maps downloaded by the developers previously or to choose the place when you are using this application. For the latter we use applications offered by OpenStreetMap, for example namefinder. Our application has been developed under the Java programming language of Sun Microsystems. The programming environment used has been Netbeans, of the same company previously mentioned. This project is divided into two parts, carrying on the client-server architecture. The server does all computation meanwhile and the client is a web page for playing on it. A Java applet has been used for implementing the client, which is a program incrusted in a HTML document. The connection between the two parts of the project is made using sockets. The application use a protocol defined by us for correct communications. The arcade games available in the application are: pacman, the running-running, the snake and click-balloons.

Key words: games, online, socket, applet, OpenStreetMap, client, server, running-runnig, snake, pacman, click-balloons.

Se **autoriza** a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos no comerciales el proyecto *Juegos arcade online sobre mapas reales* desarrollado por David Gómez Cordero, Javier Rodríguez Vidal y Víctor Pinto San Macario alumnos de la Facultad Informática y dirigido por la profesora Natalia López Barquilla durante el curso 2008/2009. Del mismo modo se ceden los derechos a dicha universidad de la memoria y el código utilizado en el desarrollo del mismo, así como de la documentación y el prototipo empleados.

Fdo.

David Gómez Cordero Javier Rodríguez Vidal Víctor Pinto San Macario

En Madrid, Julio del 2009.

Índice

1. Introducción

Cuando comenzamos a trabajar sobre este proyecto la idea inicial era poder trasladar míticos juegos arcade sencillos a un entorno real y acotado del mundo. En otras palabras ubicar al usuario en un entorno conocido y rutinario de su vida real. Con este hecho pretendíamos hacer volar su imaginación e intentar plasmar con la computación cosas familiares para el individuo. Unas situaciones tan familiares como, por ejemplo, conseguir con el juego pasear por las calles de tu barrio, de tu ciudad, o simplemente del sitio de tus sueños. Ese lugar donde te gustaría en un momento aparecer y perderte.

La idea nos resultó atractiva, ya que los ordenadores están hoy en día tan introducidos en nuestra rutina, que parece tan sencillo como inimaginable hacer un símil entre lo real y lo imaginario. Estos temas están de actualidad, como podemos ver en el cine plasmado en forma de películas, en videojuegos o en la música. Todo lo que se intenta es evadir a la persona de su vida real y trasladarla a otros puntos donde los problemas y preocupaciones no pasen de pasajeros, y con dar un botón todo finalizará. Eso sí, siempre manteniendo los pies sobre la Tierra.

Para conseguir las calles o barrios reales sobre los que presentar los juegos, iniciamos una amplia búsqueda. Esta búsqueda a través de la red para poder encontrar un servicio Web que nos proporcionase la información geográfica adecuada y adaptada a nuestras necesidades. Estas necesidades eran computacionales, queríamos algo con unas

características concretas para poder satisfacer los deseos de futuros jugadores de nuestra aplicación.

La primera tentación fue utilizar el servicio de mapas ofrecido por la reconocida empresa *Google*, conocida a nivel mundial y que tiene como producto estrella su famoso buscador además de otros tantos servicios dentro del mundo de Internet. El servicio de mapas ofertado por esta multinacional recibe el nombre de Google Maps. La pregunta es ¿por qué entró en nuestras mentes esta consideración? La respuesta más sencilla sería decir que es una de las empresas más conocidas de la red pero no sólo eso, sino que nos estaba ofreciendo algo que estábamos buscando, un servicio capaz de proporcionarnos unos mapas, utilizando un sencillo API[1], con bastantes tutoriales repartidos por toda la red, que nos facilitaría su uso. A parte de esto, destacar la calidad y precisión cartográfica de los mapas. Aun siendo todo lo comentado anteriormente ventajas, encontramos un gran inconveniente que nos impedía continuar considerando este servicio de cara a nuestro trabajo: el API de Google Maps no proporciona la información completa de la región requerida. Cuando se le solicita un mapa concreto a este servicio Web, la información que acompaña a este no es total ya que sólo nos muestra diversos puntos, por ejemplo de una serie de calles sólo proporciona los nodos de las confluencias de éstas. Por este motivo, se descartó definitiva e irrevocablemente esta opción y se pasó a considerar otra.

La segunda y definitiva alternativa fue la utilización de OpenStreetMap (OSM) [*Véase* 5]. La comunidad que forman los miembros del proyecto OpenStreetMap se encargan de crear mapas libres y con posibilidad de editarlos, muy enclavado dentro del surgimiento de la Web 2.0. Para crear estos mapas la información geográfica es capturada con dispositivos GPS móviles, así se asegura precisión en dicha información. El proyecto cuenta con un amplio grupo de colaboradores repartidos actualmente por todo el mundo, aunque sus orígenes fuesen en Londres, Reino Unido. Al ser uno de los impulsores del gran proyecto a nivel mundial de Software libre, poco a poco haciéndose un gran hueco dentro del mundo de la Informática, nos proporciona de forma gratuita la información necesaria y nos hace participes de forma activa de dicho proyecto, estas son dos de las grandes ventajas de esta iniciativa. Además la forma de obtener información es sencilla, disponen de una Wiki[2] [*Véase* 8,9] con tutoriales tanto para expertos, como para usuarios junior y encajan perfectamente con los requerimientos de nuestra aplicación. Los principales servicios de los que nos aprovechamos de este proyecto son la información detallada, punto a punto, de los mapas que nosotros escojamos, y la imagen de la región seleccionada siempre acorde con la información territorial escogida. En siguientes capítulos se dará una explicación más extensa y detallada de todos los servicios que ofrece OSM.

Siguiendo en la línea del Software libre se pensó en usar herramientas de este tipo para implementar la aplicación. Pero antes de ello había que tomar una decisión respecto al lenguaje que se usaría. Entre los posibles candidatos considerados estaban Java, C, C++,

[1] API: **A**pplication **P**rogramming **I**nterface. Es el conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

[2] Wiki: es un sitio web cuyas páginas web pueden ser editadas por múltiples voluntarios a través del navegador web.

Pascal, ¿por qué estos y no otros? Ante la falta de lenguajes de programación específicos para la tarea que íbamos a realizar, decidimos escoger entre los lenguajes de propósito general que nos eran más conocidos. De entre ellos nos decantamos por Java.

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principio de los años 90. Java frente a los otros tiene la ventaja de estar en plena actualidad, ya que la mayoría de las aplicaciones que se desarrollan hoy en día se hacen en dicho lenguaje, además está muy conectado al mundo de la telefonía móvil. Posee un cómodo recolector de basura que evita al programador tener que manejar la memoria de forma manual. Esta acción se lleva a cabo sin que el desarrollador se percate. En cuanto a protocolos y accesos a Internet hay aplicaciones ya hechas que se pueden usar e integrar perfectamente en tu proyecto. Aparte de esto, Java mantiene la independencia de poder ejecutarse igualmente en cualquier tipo de hardware, esto sale reflejado en un axioma de la empresa Sun Microsystems referido a Java: "escrito una vez, ejecutado en cualquier sitio". También, dispone de su propia maquina virtual. Así conseguimos mantener en nuestra aplicación una cierta autonomía. Para llevar esto a cabo compilamos el código fuente y se genera un código en bytecode, instrucciones maquinas de la plataforma Java. Después, la máquina virtual de Java (JVM) ejecuta este bytecode.

De cara al desarrollo se ha usado un entorno de trabajo llamado Netbeans, surgido de la misma empresa creadora de Java, que se puede descargar cómodamente de la página web de Sun Microsystems e instalar en tu máquina. Se dispone de diferentes plugins que se pueden añadir a este entorno, para hacerlo más competitivo y completo.

Continuando con el tema de implementación se discutió qué tipo de estructura tendría la aplicación a realizar, el abanico de posibilidades era abierto y bastante amplio. Nos centramos en las necesidades para construir una aplicación, alojada en algún sitio web, capaz de comunicarse a través de la red mediante una tecnología existente con un sitio remoto intercambiando diversa información. Dados estos parámetros llegamos a deducir que debíamos dividir el proyecto en dos partes bien diferenciadas para poder alcanzar la meta propuesta.

Esta partición debería contener una primera parte donde debería llevar toda la carga computacional y es la que estaría alojada en algún sitio web conocido. La otra parte se ejecutaría en el ordenador de nuestros futuros jugadores. Esta última no debería llevar ninguna carga de trabajo y no sería más que la pantalla, lo que podemos ver, de lo que se está ejecutando en un momento dado en la otra máquina remota. La primera parte mencionada será el servidor y la segunda parte el cliente, consiguiendo así la arquitectura cliente-servidor. La idea está clara pero ¿cómo llevarla a cabo? ¿Qué tecnologías usar? La respuesta a estas preguntas la podrá encontrar seguidamente.

La parte relacionada con el cliente se ejecuta en la máquina del usuario y no realiza ningún cálculo computacional, simplemente se nutre de información que obtienen del servidor cuando lo necesita. Para la implementación del cliente se ha utilizado applets de Java, componente de una aplicación que se ejecuta en el contexto de otro programa, en nuestro caso un navegador web. Para poder utilizar este componente de Java se han tenido en cuenta los problemas derivados de la seguridad de este tipo de aplicaciones. Por ejemplo, el applet no puede acceder a ninguna información de la máquina donde se

ejecuta, éste es uno de los motivos por los cuales todo el tratamiento computacional debe realizarse en la parte servidora.

El servidor, como ya hemos mencionado, es el encargado de nutrir al cliente de todo lo que necesita en cada instante, ya que el cliente tiene como única misión pedir los datos y mostrarlos. Esta partición debe estar clara y bien distinguida. Se encarga del acceso a Internet para obtener los datos necesarios referidos a la información de zonas geográficas elegidas por el usuario, para ello usa unos servicios web, API de OpenStreetMap. La empresa Sun enmarcado en su entorno Netbeans ha desarrollado un código que facilita la creación y uso de servicios web.

Los servicios web ("Web services" en inglés) son un conjunto de protocolos y estándares utilizados para intercambiar datos entre aplicaciones. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos a través de alguna red como Internet. Estos suelen seguir unos estándares preestablecidos, tanto OASIS como W3C son responsables de crearlos.

Una de las misiones principales de la parte servidora es ejecutar el algoritmo propio creado por nosotros y en éste es donde radica la idea principal de este proyecto y donde debemos hacer verdadero hincapié. El algoritmo toma como entrada un archivo escrito con un lenguaje de marcas, en este caso lenguaje XML, y haciendo un procesamiento adecuado es capaz de moverse a través del mapa. Esto ocurre gracias a que el API de OpenStreetMap proporciona toda la información de los puntos que aparecen y no solo los puntos de cruces entre calles como realizaba Google Maps, así el movimiento que se realiza es preciso.

Otro de entre los tantos cometidos que tiene el servidor es realizar la lógica de los juegos implementados, movimientos posibles e inteligencia artificial aplicada a ellos. Para poder realizar la comunicación entre la parte servidor y el cliente sufrimos grandes inconvenientes que finalmente solucionamos. Esta comunicación debería ser bidireccional y no unidireccional, este hecho fue el que nos hizo decantarnos por usar sockets para establecer dicha comunicación. Los sockets son una concepción teórica de la forma en que dos aplicaciones se comunican, necesitan una dirección IP, un protocolo y un número de puerto. Las propiedades de éstos dependen del protocolo que implementen bien sea TCP[3] o UDP[4], bien usando uno u otro ganamos o perdemos ciertas características.

En esta aplicación se han implementado un total de cuatro juegos arcade de toda la vida. El primero de ellos es el pilla-pilla, un juego en el que los niños juegan en la calle y que nosotros hemos querido trasladar al computador como indicábamos en la filosofía del proyecto expuesta al principio de este texto. El segundo, la serpiente, juego típico de teléfonos móviles y ordenadores donde la serpiente va aumentando de tamaño según va comiendo diferentes piezas de fruta distribuidas por el mapa. El tercero, el típico juego

[3]TCP: es uno de los protocolos fundamentales en Internet orientado a conexión.

[4] UDP: es un protocolo del nivel de transporte basado en el intercambio de datagramas no orientado a conexión.

del pincha globos en el que te van apareciendo globos que debes pinchar con el ratón sumando así puntos. Este último juego imita a las míticas maquinas con una pistola que había en los recreativos y que disponían de juegos de caza, en los que el objetivo era disparar o bien patos, platos, etc. Para finalizar el último juego es el come-cocos, ¿quién no ha jugado a este juego alguna vez? Consiste en un bichito que se mueve a través del plano que debe de coger todos los puntos que aparecen pero teniendo cuidado de que no le atrapen a él unos malvados fantasmas.

Esto es a grandes rangos una amplia visión de los temas que trataremos de aquí en adelante, más desarrollados y con mayor exactitud y tecnicidad. Con ésta hemos querido dar una idea de todos los asuntos que han servido para fraguar el producto final que es nuestra aplicación implementada.

Para empezar, en el siguiente capítulo, encuadraremos nuestro proyecto en la actualidad, daremos unas pequeñas pinceladas de las teconologías usadas. Nos centramos en Web 2.0, Java, Servidor Web Apache Tomcat, Sockets, programa Inkscape, Web services, RMI, y sockets.

El capítulo 3 lo hemos reservado para hablar sobre OpenStreetMap. Hemos relatado la creación y evolución de este proyecto, la forma de obtener información gracias a él, el editor de mapas JOSM y la Wiki con información acerca de esto.

Otro tema importante a destacar en nuestra aplicación, es la estructura que se ha seguido para desarrollarla. Por ello el capítulo 4 se encarga de la arquitectura Cliente Servidor y la comunicación entre ellos mediante sockets.

El capítulo siguiente, relatamos la forma de adquisición de información geográfica y cómo se hace el tratamiento para su uso.Obtener la información, conseguir la imagen y calibrar correctamente los mapas.

En el capítulo 6 desarrollaremos la parte central de nuestro proyecto, el algoritmo realizado.

Los dos capítulos posteriores son la implementación del servidor (capitulo 7) y la del cliente (capitulo 8). En el capítulo 8 dentro de la implementación del cliente se especifica temas más concretos del Applet, CardLayout, eventos, sonidos, pintado.

El capítulo 9 se tratan todos los juegos implementados: pilla pilla, mono platanero, snake, pincha globos, come cocos.

El capítulo 10 es el manual de usuario, en el que se proporcionan las pautas del uso de la aplicación.

Para terminar la última parte es la diferente bibliografía que se ha usado durante el desarrollo del mismo.

2. Estado del arte. Entornos de red y tecnologías usadas

Cuando se empezó a trabajar en este proyecto, más especialmente cuando se comenzó a pensar en las tecnologías a usar para llevarlo a cabo, nos hicimos el interrogante de qué tecnologías deberíamos usar. Este punto sería muy importante para la evolución de nuestro proyecto ya que marcaría tanto el desarrollo como la filosofía del mismo desde el inicio hasta el fin.

En pleno siglo XXI las tecnologías se desarrollan y avanzan con una velocidad cósmica gracias a la cantidad de gente y empresas que se ocupan de forma desinteresada o no a ellas. Este hecho es aún más constatable en el mundo de la informática, sobre todo gracias a internet que ha acercado a cada hogar o cada empresa, por pequeña que sea ésta, la posibilidad de ayudar a construir este universo tecnológico. Una de las ideas de nuestro proyecto, que queríamos remarcar, es ayudar un poco más a este fenómeno que se está desarrollando en nuestros días que parece no acabar, todo lo contrario, parece un túnel que no tiene fin.

Dentro de este maravilloso mundo de las tecnologías podemos concluir que existen dos tipos de éstas, las que se deben obtener mediante el pago de una licencia y las que no. Las primeras pertenecen a alguna persona, empresa o grupo que es propietario de las mismas. El uso de éstas está condicionado necesariamente con el pago de la licencia al o los propietarios. Sin embargo, las tecnologías libres se caracterizan por el uso gratuito

de la licencia sin estar condicionadas a pago alguno. En principio, dentro de este grupo, se englobaría el llamado *software libre* (en inglés *open source*) que se caracteriza por tener el código fuente del programa abierto a cualquiera que lo quiera ver. Esta corriente, que se ha convertido en filosofía de vida informática, está ganando peso día a día en la comunidad internauta sobre en los pequeños *desarrolladores domésticos*. Un ejemplo de esto, serían las distribuciones GNU-Linux.

Dentro de este panorama, teníamos bastante claro que las tecnologías que usaríamos debían ser de carácter gratuito ya que el proyecto no tenía asignado presupuesto alguno. En caso de que hubiéramos adquirido alguna licencia de uso de alguna tecnología no gratuita sabíamos que éstas suelen alcanzar un precio elevado en el mercado, así que desde el principio se pensó en contar solo con los medios y tecnologías a nuestro alcance, es decir, o bien gratuitas o bien con licencia adquirida por la Facultad.

Los primeros pasos de nuestro proyecto fueron asentar las ideas claves de qué íbamos a desarrollar. Estas primeras discusiones nos dieron diferentes pistas sobre qué queríamos que fuera nuestra aplicación, cómo iba a ser y dónde iba a ejecutarse.

Queríamos una aplicación abierta a la red, acorde con el siglo XXI, que se pudiera utilizar en cualquier ordenador en cualquier parte del mundo, a través del medio de todos, internet. Queríamos que fuera un proyecto capaz de provocar diversión de la gente a través de nuestros juegos basados en *OpenStreetMap*, un proyecto hecho por todos y para todos, pudiendo personalizar cada usuario donde localizar este entretenimiento: *el lugar favorito de vacaciones*, *en un maravillo parque*, *en su barrio*,… Antes de comenzar a tratar la parte más técnica nos propondremos ubicar la idea dentro de las aplicaciones ya creadas y disponibles en la Red.

2.1 Estado del arte

Dentro del entorno en lo que se hace basar nuestro proyecto podemos afirmar que juegos online hay por doquier dentro de la red. Pero lo importante para nuestra aplicación no era que fuese un juego online sino que pudieses elegir la zona geográfica en la cual querías desenvolverte. Como en la red se puede encontrar casi de todo, iniciamos la búsqueda haber qué podíamos hallar referido a este tema. Encontramos variadas cosas sobre esto. De entre ellas destacaremos dos juegos de carreras que se desarrollaban usando Google Maps. Uno de ellos lo podemos encontrar en la siguiente dirección web:

http://geoquake.jp/en/webgame/DrivingSimulatorGM/

Este juego consiste en un vehículo que se va moviendo por el mapa. Se tienen las opciones de cambiar la forma de ver el mapa, se puede modificar el vehículo que se usa, elegir el lugar que se va a recorrer dando él unas sugerencias,

cambiar el zoom y tiene un volante que se mueve según gira el vehículo.

El otro juego se encuentra alojado en la siguiente página:

http://www.tomscott.com/realworldracer/

Es una carrera entre varios coches. El usuario elige un comienzo y un final de la carrera. A partir de ese momento empieza. El coche del usuario es diferente a los demás, es rojo y el resto blanco. La carrera continúa por el trayecto marcado por la línea azul.

El gran problema de ambos juegos es que no seguían por las calles y era posible ir por encima de los edificios. Esto ocurre porque el API de Google Maps no proporciona información completa, solo da los puntos de confluencia entre las calles. Verdaderamente solo se comprueba a la hora de contabilizar la carrera que has pasado por todos los puntos de cruce. Nosotros queríamos que esto no ocurriese. Fue la gran razón para descartar Google Maps y decantarnos por OpenStreetMap. Con ello conseguimos poder seguir el recorrido pero sin salir de los límites de las calles.

A parte de esto, hemos implementado más de un juego y no solo de coches sino de mas temáticas, aunque todos en la misma línea.

Uno de los objetivos para desarrollar está idea de proyecto era encontrar una tecnología que respetase a la misma y a la vez acorde a los días en que vivimos. Por ello nos apetecía adentrarnos en el mundo de internet de la mano de la tecnología Web 2.0.

2.2 Internet. Web 2.0

Hoy día Internet es el medio más efectivo para buscar información y mostrar al mundo nuestras propias creaciones pudiendo ser valoradas por otros usuarios de forma directa (ya sea por el éxito de acogida que tenga como por las críticas que reciba en foros especializados). Esto ha permitido a las empresas testear productos antes de sacarlos al mercado, en forma de programas beta o de evaluación y poder observar así la opinión del consumidor. Como consecuencia de este hecho, el movimiento de gran cantidad de recursos, tanto económicos como humanos a la hora de comprobar que los productos que se vayan a lanzar al mercado tengan éxito. Por todo ello queríamos que parte de nuestro proyecto se desarrolle con ciertos toques de esta filosofía de mundo

abierto, de compartir archivos e ideas a través de la red. Pero tanto nuestro proyecto como la historia de Internet no fueron así al principio.

Internet, en un principio, fue diseñada como una red de ámbito militar (usada por el ejército de EEUU) que poco a poco ha ido rompiendo fronteras llegando a ser de uso común para el resto de personas que vivimos en el planeta Tierra (aunque aún quedan reductos que se utilizan para diversos ámbitos más protegidos que necesitan una mayor privacidad). Cuando la red se fue abriendo ampliando al resto de la humanidad, Internet tuvo que implementar ciertos protocolos de funcionamiento básicos, en este momento dio comienzo la primera versión o la también llamada tecnología Web 1.0. Como hoy día se distribuían contenidos como páginas web, que los usuarios podían visitar, pero que no permitían la interactuación, tan buscada en nuestro proyecto, entre el usuario y los contenidos, por lo tanto los usuarios eran incapaces de poder comunicarse y expresar sus inquietudes, quejas y demás menesteres y compartirlos con otros usuarios.

Dada la gran expansión que se produjo, entre mediados y finales de los 90, y para dar cabida a la ingente demanda de peticiones de los usuarios fue necesario actualizar los protocolos de actuación y mejorar la versión inicial, como consecuencia nace la tecnología Web 2.0. El término de Web 2.0 fue introducido por Tim O'Reilly en el año 2004. Esta tecnología, es una evolución de las tecnologías existentes anteriormente de cara a que el usuario final pueda ser partícipe en la construcción de Internet. Dichos usuarios van a ser capaces de poder hacer un uso activo de Internet pudiendo hacer sus propias aplicaciones online, como pueda ser la creación de blogs o pudiéndose registrar en Webs sociales tales como Facebook o Tuenti.

Como hemos mencionado, anteriormente los internautas solamente podían acceder a la red para buscar algún tipo de información, no pudiendo comunicarse de manera fluida con Internet. La aparición de Web 2.0 supuso una revolución en aras de una participación constante y fluida por parte de los usuarios finales, provocando a su vez, un gran boom a la hora de la llegada de la red a todos los hogares del mundo. La ingente cantidad de información que las personas aportan a la red, hace que Internet sea la mayor biblioteca a nivel mundial sobre el conocimiento humano que jamás haya existido (mayor aún que la antigua biblioteca de Alejandría. Colocar como ejemplo la enciclopedia enmarcada en esta tecnología, la Wikipedia, conocida por todo el mundo y realizada con la aportación de cada miembro de la comunidad internauta. Todo este desarrollo ha concluido en que distintos sectores de la sociedad evolucionen muy rápidamente gracias al uso de esta tecnología y de sus derivados, siendo imprescindible para cualquier labor que queramos emprender en la actualidad.

Según la definición de Ribes la tecnología Web 2.0 engloba a "todas aquellas utilidades y servicios de Internet que se sustentan en una base de datos, la cual puede ser modificada por los usuarios del servicio, ya sea en su contenido (añadiendo, cambiando o borrando información o asociando datos a la información existente), pues bien en la forma de presentarlos, o en contenido y forma simultáneamente."

La infraestructura de esta tecnología se ha ido haciendo muy compleja a medida que se iba expandiendo basándose en redifusión, intercambio de mensajes entre diferentes servidores y máquinas, etc. El enfoque en el intercambio de mensajes se suele basar o bien en el uso de un servidor que englobe todas las funcionalidades o bien, en el uso de

plugins[5] de servidor Web con herramientas de publicación tradicionales mejoradas con interfaces API y otras herramientas. Independientemente del enfoque elegido, no se espera que el camino evolutivo hacia la Web 2.0 se vea alterado de forma importante por estas opciones. En lo tocante a la opción de redifusión, decir que usa protocolos estandarizados que permiten a los usuarios finales usar el contenido de la Web en otro contexto, ya sea en otra Web, en un conector de navegador o en una aplicación de escritorio. Entre los protocolos que permiten redifundir se encuentra el RSS[6], y Atom[7], todos ellos basados en el lenguaje XML. Otros protocolos específicos amplían la funcionalidad de los sitios y permiten a los usuarios interactuar sin contar con sitios Web centralizados.

La tecnología Web 2.0 ha sido fundamental a la hora de poder realizar nuestro proyecto tanto en los aspectos motivadores del mismo en sus inicios como en el desarrollo de la aplicación, por su gran utilidad a la hora de realizar consultas a la comunidad internauta y solucionar problemas aparecidos dentro del proyecto. Todo ello sin olvidarnos que la piedra angular del mismo se fundamenta totalmente en dicha tecnología, hablamos claramente de la comunidad OpenStreetMap.

Sabiendo que la anterior tecnología es claramente una de los pilares de nuestro proyecto, no debemos dejar sin mencionar el resto de las tecnologías usadas en el mismo, por ser también parte importante en nuestra aplicación y desarrollo de la misma.

2.3 Java. Tecnología y lenguaje de programación

Quizás el punto más importante de ésta discusión radicaba en la tecnología a utilizar para el desarrollo a la hora de programar nuestra aplicación. Gracias al avanzado estado tecnológico que vivimos en el mundo de la informática tuvimos difícil este punto ya que tuvimos que elegir entre gran cantidad de ellos. De parte de la facultad conocíamos los lenguajes más importantes, por cursarse en diferentes asignaturas, y habíamos desarrollado aplicaciones de mayor o menor dificultad con ellos. Algunos de éstos como C/C++ o Java más acordes con la actualidad y otros, como Haskell, Prolog, Pascal más característicos del estudio.

Al final, realizando una ardua investigación sobre las características de varios lenguajes de programación que se adaptaban a nuestras necesidades nos quedamos con el lenguaje orientado a objetos *Java*, desarrollado por la empresa *Sun Microsystems* en los años 90'.

En primer lugar, elegimos este lenguaje de programación por ser imperativo y orientado a objetos, dos características que creíamos indispensables y que darían forma a la filosofía que deseábamos en nuestro proyecto.

[5] Plug-in: Es una aplicación o componente que se relaciona con otra para aportarle una función nueva y generalmente muy especifica

[6] RSS: es una familia de formatos de fuentes web codificados en XML.

[7] Atom: es un fichero en formato XML usado para Redifusión web.

Además de estas dos características, más que importantes, vimos en él un lenguaje simple, bien diseñado y con diferentes herramientas que facilitan al programador su labor a la hora de hacer proyectos, motivos claves que nos harían decantarnos por él.

La razón quizás más relevante para la elección de Java como lenguaje es que es independiente del hardware donde se ejecute, es decir, se puede utilizar en cualquier plataforma (Windows, Linux,…), todo ello gracias a la *Máquina Virtual de Java (Java Virtual Machine,* JVM). Ésta es un programa capaz de ser ejecutado en la plataforma dada y tiene como función interpretar y ejecutar instrucciones expresadas en un código específico (el *Java bytecode*) generado por el compilador del lenguaje *Java*. Todo ello hace que el único requisito para poder hacer uso de aplicaciones Java es tener instalada la *Máquina Virtual de Java.* El hecho de que Java pudiera ser ejecutado en cualquier plataforma fue fundamental para la elección del lenguaje para nuestro proyecto.

Además de esto, el lenguaje para facilitar la programación, elimina de forma explícita los punteros aunque fuera de la vista del programador sigan existiendo. En relación con esto, Java añade un recolector de basura que es un mecanismo implícito de gestión de memoria que se ocupa de borrar la memoria creada y que ha dejado de ser útil para el programa.

Otras de las razones de la elección es que Java soporta multithread,varios hilos o procesos simultáneos, cosa fundamental para nuestro proyecto, ya que parte de él se basa en la sincronización.

En nuestro proyecto utilizamos la última versión sacada al mercado por la empresa, Java™ 6 *Standart Edition.* La actualización de versiones por parte de la multinacional se hace de forma regular a lo largo del año, estando estas de forma gratuitas disponibles en la Web de la empresa

2.4 Entorno de desarrollo de programación NetBeans

Para programar sobre Java hemos usado el entorno de desarrollo Netbeans concretamente la versión Netbeans IDE 6.5, recientemente sacada al mercado de forma gratuita por la empresa *Sun Microsystems,* dado que nos ofrecía bastantes comodidades para el desarrollo de aplicaciones sobre Java.

Una de las grandes características de esta plataforma de desarrollo es la capacidad de la misma en la integración de diversos y útiles plug-in para facilitar al usuario la tarea de programar, explicándose muchos de ellos en el extenso tutorial que ofrece el entorno. Estos plug-in, desarrollados por la compañía, se pueden descargar de la página Web de ésta de forma gratuita. Explicar que estos plug-in son de muy diversos ámbitos dependiendo del uso que se le quiera dar a la aplicación. Por ejemplo, a la hora del desarrollo de nuestra aplicación hemos hecho uso del plug-in de Apache Tomcat que dispone la plataforma. De esta forma se puede probar el proyecto conjuntamente con el servidor solamente sin movernos del entorno de programación, pudiendo comenzar o parar el curso del servidor cuando plazca. Otro de los plug-in utilizados en nuestro proyecto es el relativo al uso de Web-Services. Éste nos sirve para que el entorno pueda

asimilar la tecnología ASP[8] para poder utilizar la misma en nuestra aplicación. Usando este plug-in hemos podido realizar el web service implementado en nuestro servidor de forma fácil y solamente utilizando el lenguaje de programación Java, es decir, con el plug-in citado NetBeans es capaz de traducir nuestro código a la tecnología

Otra de las características que nos proporciona java y NetBeans es la posibilidad de una aplicación capaz de ejecutar el applet sin la necesidad de colgar el mismo en una Web, de forma rápida para poder hallar los fallos realizados en el mismo.

2.5 Servidor web Apache Tomcat

En relación con la tecnología a elegir para el servidor donde se ejecutaría parte de nuestra aplicación se pensó en una tecnología compatible y similar al lenguaje de programación usado, *Java*. Después de buscar a través de la red diferentes tecnologías y contrastar pros y contras de las mismas llegamos a la conclusión de usar Apache-Tomcat por ser el más fidedigno a Java.

Encontramos que Apache-Tomcat tiene una instalación sencilla, sobre todo en máquinas de pequeño tamaño y también que su coste es cero, por lo tanto está al alcance de cualquier usuario de la red y entraba dentro de los requerimientos de nuestro proyecto.

La fiabilidad avalada por los usuarios que lo han usado, encontrada en diversos foros especializados en el tema, además de su compatibilidad con las API de *Java* fueron otras razones de peso para afianzar nuestra decisión.

La versión utilizada para las pruebas de nuestro proyecto ha sido Tomcat 6.0, si bien se podrá instalar el mismo en otra versión siempre que sean compatibles. Hemos decidido utilizar dicha versión por ser la más moderna y estar instalada en los laboratorios de la facultad, por ser este, el lugar habitual de la realización del proyecto

2.6 Inkscape

Para la realización del tratamiento de las imágenes por parte de nuestra aplicación necesitábamos un programa que nos solucionara este problema.

Por ello, siguiendo el consejo de la página de desarrolladores de O*penStreetMap (OSM)*, contemplamos la posibilidad de usar el programa de tratamiento de imágenes Inkscape, un programa de pequeño tamaño y sencillo de usar. Después de leer parte del manual del programa, alojado en su página Web, vimos varios aspectos que nos serían de gran utilidad para nuestra aplicación.

Este programa es de carácter libre y puede ser usado en cualquier plataforma, del mismo modo que las tecnologías usadas anteriormente. Además, la aplicación puede tratar

[8] ASP: Active Server Pages. Es una tecnología del tipo "lado del servidor" para páginas web generadas dinámicamente

gráficos vectoriales lo que nos es útil para el manejo de los ficheros SVG (Scalable Vector Graphics.) que tratamos en nuestro proyecto para crear imágenes PNG (Portable Network Graphics).

Un dato fundamental que nos hizo decantarnos por la elección de este programa es que se puede usar el mismo a través de la consola de cualquier sistema operativo, lo que significa que se puede ejecutar desde los mecanismos de *Java* que se encargan de ejecutar programas externos a la aplicación programada.

2.7 Web services versus RMI y sockets

Nuestra aplicación se divide en dos bloques bien diferenciados, por una parte la aplicación servidor y por otra la aplicación cliente (este tema será explicado con detenimiento a lo largo de este texto). El cliente, implementado gracias a un *applet*, se encuentra dentro de una página web que se aloja dentro de un contenedor, un servidor web, para que se pueda acceder a ella a través de cualquier ordenador del mundo por medio de Internet. Por otro lado se encuentra la aplicación servidor, que se encuentra alojada dentro del mismo contenedor en nuestro caso, pero no es estrictamente necesario que sea así, es la que dará servicio al cliente.

La aplicación servidor cuenta con varios elementos cada uno con una función diferenciada. Uno de estos elementos descritos es un *Web services*, en nuestro caso basado en *Java*.

Un *Web services* es un conjunto de estándares y protocolos que se usan para intercambiar información entre distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma. La utilidad de ellos es que la aplicación que se conecte con el *web services* puede utilizar *los servicios* que preste el primero, intercambiando los datos y la información necesarios para darlos. *Los servicios* que presta esta aplicación pueden ser de muy diversa índole, siendo usados para cualquier campo. El reglamento y la arquitectura de los web services están tutelados por diferentes comités de las organizaciones *OASIS*[9] y *W3C*[10].

Cuando se pensó en la necesidad de tener una herramienta que actuara de forma similar, prestando servicios, se pasó a la búsqueda de tecnologías adecuadas para poder tratar el problema. Después de la búsqueda de las mismas, se llegó a la conclusión de que las tecnologías más apropiadas eran: los *sockets*, *web services* o *RMI* (*Remote Method Invocation*).

Aunque al principio se consideró la idea, al final se terminó descartando el uso de *sockets* para esta parte en concreto, ya que aún pudiéndose hacer con ellos, se partiría desde una programación muy a bajo nivel y el trabajo con ellos hubiera sido bastante

[9] OASIS: Es un consorcio que crea diversos estándares sobre internet principalmente.

[10] W3C: World Wide Web Consortium. es un consorcio internacional que produce recomendaciones para la World Wide Web

laborioso y complejo, a la vez de largo. Las tecnologías que ahora se explicarán utilizan la tecnología de sockets pero preparada para dar al usuario de la misma mayores facilidades.

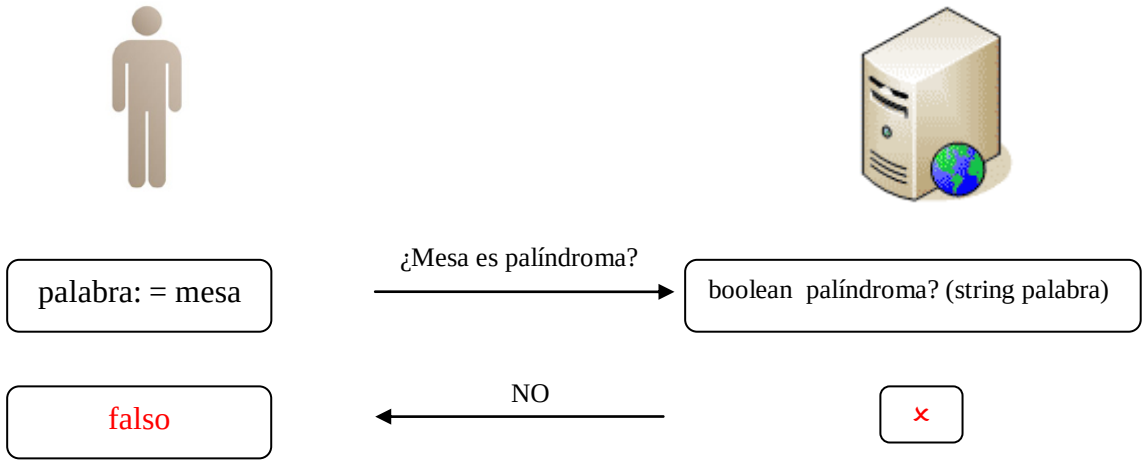
RMI es un mecanismo para poder invocar un método de forma remota. El uso de éste nos proporciona de un mecanismo simple para comunicar servidores de una aplicación distribuida que solamente necesite *Java*. Esta tecnología usa los sockets antes descritos de forma transparente al usuario, facilitando su uso por evitar diversos problemas que dan los sockets pero que con esta tecnología desaparecen. El uso de RMI hubiera sido una buena opción pero en la recta final del proceso de toma de decisión se acabo eligiendo la siguiente por tener un manejo sencillo, visual y dinámico como explicamos ahora.

Se terminó concluyendo que los *web services* de *Java* eran la mejor opción por las facilidades que aportaba al trabajo del programador gracias a que contaba que el entorno de desarrollo de la aplicación que usábamos era *Netbeans*. Este entorno hace que el uso de esta tecnología sea sencillo ya que contiene una interfaz que facilita la comprensión y ayuda bastante a la hora de la creación, modificación y mantenimiento de los mismos. Todo gracias a que Netbeans cuenta con un paquete específico para el uso de *web services* de *Java* en el entorno.

La seguridad de los *web services* es una de las razones por las que se ha elegido esta tecnología, dado que al apoyarse en el protocolo HTTP pueden seguir beneficiándose del sistema de seguridad firewall sin necesidad de hacer cambios importantes en la configuración, además esta tecnología permite que desde diferentes puntos geográficos se puedan combinar servicios y de esto modo proveer al usuario de unos servicios integrados de calidad.

Para que se pueda entender mejor el concepto de *servicio web*, se ilustrará con un ejemplo. Imaginemos que tenemos un *web services* que ofrece como servicio decir si una palabra es palíndroma.

El cliente se conectaría al *web services* por medio de la red y usará el servicio del *web services*. El *web service* contestará afirmativa o negativamente después de procesar la palabra que quiera enviar el cliente.



En nuestro caso, tenemos dos web services: uno que se ocupa del proceso de generación de la imagen a través de un XML y el otro que se encarga de arrancar la aplicación servidor que se encarga de hacer de motor de los diversos juegos que tiene nuestra aplicación.

El web service que contiene nuestro proyecto y que se encarga de la comunicación con los API de OpenStreetMap para la obtención del archivo XML, que procesa éste y lo transforma a un archivo PNG, una imagen, contiene dos métodos que prestan servicio al usuario que lo desee.

El primer método se utiliza para que dado una cadena de caracteres que representen un lugar el servicio devuelve un conjunto de lugares posibles para esa cadena. Por ejemplo, si la cadena de entrada fuera "oporto", el programa devolvería diferentes posibilidades como "Oporto, Portugal", "Metro de Oporto, Madrid","Calle Oporto, Madrid". Lo que realiza el método es una comunicación con el servicio *namefinder* perteneciente a *OpenStreetMap* que le devuelve un archivo XML con los posibles lugares referidos a la cadena entrante junto con las coordenadas de los mismo. El método procesa este XML y lo transforma a un conjunto de lugares que posteriormente se le muestran al usuario para que elija el lugar deseado.

El segundo método recibe un entero que hace referencia a uno de los distintos lugares que previamente se habían procesados con el método primero. Este entero, significa que el usuario ha seleccionado un lugar exacto como opción. A partir de aquí el método se comunica por medio de la API de OpenStreetMap con el servicio del mismo grupo obteniendo el archivo XML propio del lugar elegido. Cuando tenemos este archivo el propio método procesa el archivo XML, a través de una hoja de estilo, y lo transforma a un archivo SVG desde donde se parte para obtener un archivo PNG que es la propia imagen del mapa que necesitamos.

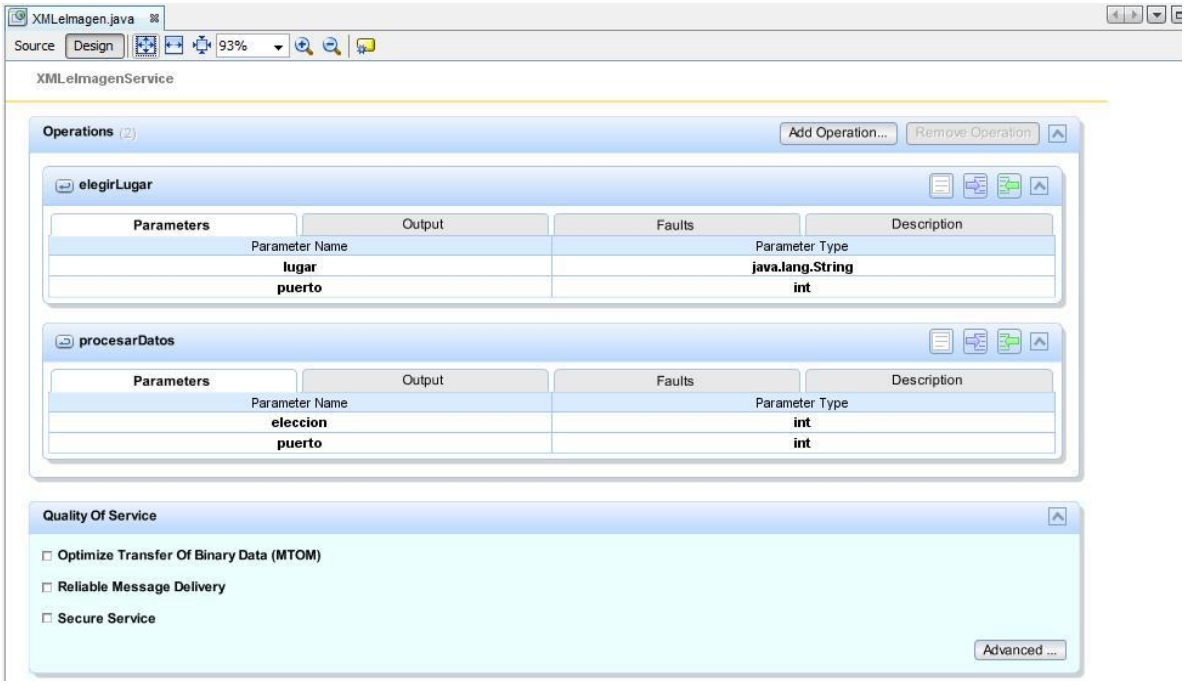


Imagen de nuestro web services de la aplicación con NetBeans

2.8 Introducción a la comunicación mediante sockets

Los sockets han sido usados para poder realizar la comunicación entre el cliente y el servidor. Un socket es definido mediante una dirección del Protocolo de Red, un número de puerto y un protocolo para el intercambio de información. Para poder realizar la comunicación es imprescindible que sean capaces de localizarse lo primero. Necesario también disponer de los tres recursos con anterioridad nombrados. Los sockets son usados para implementar una arquitectura cliente-servidor. Se inicia la comunicación por parte de uno de ellos, programa cliente. El otro programa está a la espera que el anterior la inicie, por ello recibe el nombre de programa servidor. El tema de sockets para la comunicación, tanto como la estructura será explicada con detenimiento en el capítulo correspondiente, aunque nos parecía conveniente nombrarlo en este capítulo.

3. Servicio de Mapas: OpenStreetMap

Después de en capítulos anteriores haber introducido conceptos generales como Web 2.0, nos vamos a centrar en los servicios específicos para obtener información de tipo geográfica. Entre los servicios existentes en la Web y de cara a la realización del proyecto, comparamos dos alternativas a la hora de poder descargarnos los mapas que vamos a necesitar. Estos portales son Google con su aplicación de mapas (Google Maps) y el portal de software libre OpenStreetMap [*Véase* 5]. Posteriormente detallaremos una pequeña introducción al servicio de Google Maps y nos extenderemos en el servicio ofrecido por OpenStreetMap, que fue por el que finalmente nos decantamos. Expondremos algunas ventajas y desventajas.

Nuestra primera intuición para hacer efectiva la descarga de los mapas, fue el uso de Google Maps, ya que los productos lanzados por esta empresa son de los más usados y tienen una amplia popularidad y acogida por parte de los usuarios de la Red. Este servicio, nos permite ver mapas reales a través de Internet. Actualmente se están desarrollando nuevas herramientas que permitan ver la Luna (Google Moon) y Marte (Google Mars) con la ayuda de las imágenes obtenidas de la NASA. Ahora, pasaremos a detallar el uso de esta herramienta.

Como ya hemos comentado, Google permite descargarse mapas a través de su página usando un API especial que nos permite interaccionar con él. Dicha API está realizada

en JavaScript[11]. El problema que surge, es que la información que otorga dicho API únicamente proporciona los puntos de confluencias entre los caminos. Es decir, solo tememos referencias entre los cruces de las vías. Para realizar nuestra práctica, nosotros necesitamos los datos completos de todos los puntos que forman la vía o los necesarios para poder trazar la trayectoria de la vía. La solución a este problemas pudiese ser rellenar manualmente puntos ausentes, pero resultaría imposible porque no disponemos de la localización de éstos.

Además del archivo de los puntos, nos podemos descargar la imagen de las calles en formato PNG (formato de imagen). Para poder usar el API de Google debemos registrarnos primero para poder obtener una clave para dicha aplicación, una vez obtenida podremos programar nuestras propias aplicaciones siguiendo las instrucciones que se nos adjuntarán junto con la clave proporcionada. De cada mapa podremos obtener tanto su latitud como su longitud con los métodos específicos para su obtención, así como obtener un nivel de acercamiento en los mapas (zoom). Google define 20 niveles de acercamiento que podremos ir modificando a voluntad. Una vez que hayamos definido todo lo que deseemos, obtendremos un objeto de tipo GMap2 (el cual deberá estar inicializado previamente con la posición inicial en la cual debe presentarse en la aplicación que tengamos). El aspecto que tendrá dicho objeto es similar al que tiene el mapa que se aloja en el sitio Web de Google Maps incorporando muchos comportamientos que están integrados en dicha Web. Aunque es atractivo visualmente, el problema anterior es un gran impedimento para nuestro fin.

La segunda alternativa que contemplamos a la hora de poder realizar nuestro trabajo, fue el uso del portal llamado OpenStreetMap. Convenciéndonos enteramente de su uso ya que se adapta perfectamente a los requerimientos que buscamos para poder realizar con éxito esta misión. Realizaremos una pequeña introducción de esta comunidad para ir tomando contacto con ella y más adelante comentaremos las diferencias con la primera alternativa y desarrollaremos en más detalle su uso.

3.1 Creación y evolución de OpenStreetMap

OpenStreetMap (OSM) [*Véase* 5] es una fundación sin ánimo de lucro que nació en Londres (Reino Unido) en el año 2004, de la mano de Steve Coast (su fundador), para llevar a las personas datos cartográficos de lugares del mundo de manera gratuita. Hasta que no vio la luz esta Web, los usuarios tenían que pagar, en sus respectivos países, una tasa para utilizarlos. Más tarde nacieron sitios en Internet que permitían a los usuarios trazar sus propios mapas y mandarlos a estos lugares, pero el beneficio se lo repartían entre estas empresas. Con la venida al mundo de OSM, los navegantes eran capaces de editar sus propios mapas sin tener que pagar/beneficiar a otras empresas.

Transcurrido algún tiempo desde su fundación, OSM fue recibiendo importantes donaciones tanto de dinero de particulares como de datos orográficos por parte de los Estados que así lo consideraron oportuno (p. Ej.: Canadá). Los últimos datos estimados

[11] JavaScript: es un lenguaje de programación interpretado, es decir, que no requiere compilación, utilizado principalmente en páginas web, con una sintaxis semejante a la del lenguaje Java.

arrojan que cerca de 400000 personas se hallan registradas en este sitio colaborando en su expansión, realizando mapas de diversas zonas, así como editando otros con nuevos puntos de interés, nuevas carreteras, etc. Además de las zonas de mapas (que describiremos posteriormente), consta de una Wiki propia [*Véase* 8] donde los desarrolladores podrán obtener información acerca de la edición de mapas, como la subida de éstos como el significado de los XML que conforman las diversas áreas mundiales. Lo que nos interesa en este caso es sacar todo el poder posible de la utilidad para poder utilizarla lo mejor posible de acuerdo a los requerimientos impuestos por los requisitos dados para la realización de nuestro proyecto, por eso nos hicimos la siguiente pregunta: ¿Qué se puede hacer con OpenStreetMap? A continuación explicaremos los diferentes usos de OpenStreetMap.

3.2 Obtención de información desde OpenStreetMap

OSM es un servidor de mapas mundiales a través de Internet. Con él podemos descargarnos cualquier mapa cartográfico mundial así como descargarnos el callejero asociado de una manera totalmente gratuita. Para ello, debemos introducir en el cuadro de diálogo correspondiente la dirección que queremos buscar (por ejemplo: Madrid, Ciudad Universitaria), a continuación se nos mostrará el callejero de la zona seleccionada así como sus coordenadas terrestres el zoom que tiene puesto, el cual podremos modificar a voluntad. Una vez que hayamos seleccionado todos los requisitos pedidos, podemos exportar este trozo, para ello, nos saldrán varias opciones entre las que tenemos, exportar el mapa como XML (una vez obtenido podemos ver el mapa en este formato, pudiendo ver todos los puntos que han sido editados hasta el momento, tanto carreteras, como metro, edificios de interés, etc.). Para poder exportarlo, haremos uso del API que OSM pone en nuestras manos. Mediante el API de OSM vamos a poder ver un nodo dado su identificador, veamos un ejemplo:

```
http://api.openstreetmap.org/api/0.5/node/30894545
```

Esto nos proporcionaría información sobre ese nodo. Se puede ver que el API añade ciertos atributos: el último usuario en modificar la entidad, la visibilidad (los objetos históricos no son visibles a priori), y la fecha de la última modificación. Otra operación importante sobre el API es la petición de todos los datos dentro de un *bounding box* (especificado como oeste, sur, este, norte) ya que nos permitirá exportar los mapas de la zona que nosotros deseemos (sin tener que retocar). Para editar datos a el API 0.5 (existe una nueva versión de este API que todavía está en fase de pruebas versión 0.6, la visión de los nodos y la descarga de mapas es exactamente igual que la 0.5 sólo que hay que sustituir en la ruta especificada el 0.5 por el 0.6 correspondiente a la versión actual), se llama a la misma URL que para recibir la información de un objeto, enviando a el API el XML correspondiente a la nueva versión de un objeto. Para crear un nuevo objeto, se realiza la misma operación, pero sustituyendo el ID del objeto por "create" el API nos devolverá el identificador asignado para ese nuevo objeto. Hay otras operaciones para pedir datos al API, por ejemplo, solicitar el histórico de un objeto, solicitar los datos de todos los miembros de una relación, o borrar un objeto. Todo ello está documentado en las páginas SM_Protocol_Version_0.5 de la Wiki.

Otra forma de poder exportar el mapa, es como una imagen para ello tendremos que seleccionar la opción de bajarnos el mapa como imagen (se guardará en formato PNG, el cual se abre con cualquier editor de imágenes que tengamos instalado en el ordenador).

A medida que el proyecto OpenStreetMap ha ido creciendo así como su base de datos, ésta ha sido mejorada tanto en calidad como en cobertura. Además han ido surgiendo a su alrededor diversas herramientas informáticas y servicios, convirtiéndose en una fuente de datos factible para proyectos complejos. Así, por ejemplo, han surgido numeroso software que facilitan la captura, edición, tratamiento y presentación de la cartografía. Un claro ejemplo de este software es JOSM.

3.3 Herramienta de edición de mapas JOSM

JOSM (Java OSM Editor, totalmente gratuito) [*Véase* 10], es una de esas herramientas, con ella, vamos a poder modificar datos de zonas que ya han sido cartografiadas o modificadas por otras personas. Con ella vamos a poder modificar un XML, previamente descargado de OSM. Para ello contamos con un repertorio de widgets[12] útiles como son la de añadir nuevos puntos al mapa, seleccionarlos para ver sus coordenadas, el nombre de la calle a la cual pertenecen, etc. Usando distintos plugins que instalaremos, podemos editar mediante un GPS así como poder observar el mapa mediante la vista del satélite (sólo la parte orográfica, se puede conectar a un servidor de mapas reales si queremos, además, ver las casas, etc.). Una vez que hayamos modificado todo aquello que creamos oportuno, podemos subir los cambios realizados o por el contrario cerrar la herramienta y descartar todo aquel trabajo que hemos realizado.

El uso de JOSM se hace fundamental a la hora de hacer el proyecto, ya que así hemos podido verificar mejor nuestro algoritmo de movimiento a través del mapa, ya que con sólo acceder a cada uno de los puntos, hemos podido ir observando si los resultados que nos daba eran los correctos. Otra forma de editar los datos es modificando el XML, añadiendo nuevas etiquetas y todos aquellos datos que sean menester, esta forma de edición es más compleja por lo que se recomienda el uso de la herramienta anteriormente descrita para las posibles modificaciones que se quieran crear de los mapas. Existen otras herramientas de edición de mapas como pueden ser Potlacht que es un programa flash y Merkaator que es un programa desktop de QT.

Como podemos observar, OSM es un mundo en expansión que poco a poco irá comiendo el terreno a otros editores de mapas que hay en el mercado gracias a la inestimable ayuda y colaboración de todo aquel que tenga algo de tiempo para ayudar y colaborar con esta fundación.

Por último comentar que para poder hacer uso de todas las herramientas disponibles y saber cómo utilizarlas, OSM posee una potente ayuda para desarrolladores en forma de

[12] Widgets: Es una aplicación entre cuyos objetivos están los de dar fácil acceso a funciones frecuentemente usadas y proveer de información visual.

Wiki, la cual se hace imprescindible para poderse orientar en el intrincado mundo de OSM.

3.4 Wiki de OpenStreetMap

Uno de los puntos fuertes de esta Web es su ayuda para los desarrolladores, ésta se encuentra en formato Wiki pudiendo ser editada por los usuarios de dicho espacio. Una vez que entramos en ella, nos encontramos con diferentes opciones que nos permitirán obtener ayuda (por supuesto, toda la información que encontremos aquí, se encuentra en inglés). La primera, que nos encontramos es la guía para principiantes ("Beginners" Guide), una vez que accedemos aquí, podemos ver los cinco pasos que debemos hacer si queremos editar/crear un mapa, esta información la vamos a explicar de manera un poco más detallada en el siguiente párrafo.

Otra de las posibles entradas que podemos seleccionar es la de crear/editar un mapa (Map Making). Seleccionando esta opción aparecerá una nueva ventana en la cual se nos explicará el modo y método para conseguir lo que nos estamos proponiendo (editar/crear nuestro propio mapa). Para ello, debemos seguir los siguientes pasos. Primero de todo, debemos obtener los datos (con los dispositivos necesarios para tal fin, esto lo podemos hacer andando, montando en bicicleta...), asimismo debemos estar registrados en la página y unirse a las listas de correo (mailing lists) para tener comunicación con otras personas. Una vez hecho esto, pasaremos a la fase de mapeado (maping), podemos ver que es lo que ya se ha hecho sobre esa zona que deseamos modificar (para no repetir trabajo que otros ya han realizado). Lo siguiente que debemos hacer, es saber las técnicas de mapeado que debemos seguir (maping techniques), la cual es una guía muy completa sobre los modos de editar un mapa. Sobre todo el modo más sencillo para crear un mapa es usando el GPS, y volcarlos a nuestro ordenador para la edición usando JOSM. Otra cosa que podemos hacer, es tomar nuestras propias tomas aéreas o de satélite (usando para ello los mapas de Yahoo, Yahoo Maps, ya que existe un acuerdo de colaboración entre estos dos portales).Otras de las cosas a realizar, son la toma de los datos del GPS, apuntando en cada una de estas coordenadas qué es lo que vamos a encontrar (ya sea un parque, un colegio, etc.) Si así lo deseamos, tomaremos fotografías digitales que luego podremos incluir en el proyecto como también añadir voces, videos… Debemos de todos modos hacernos un esquema de cómo es la zona que vamos a diseñar para luego poder plasmarlo en lo que queremos hacer. Debemos añadir colores a cada una de las regiones que componen nuestro mapa ya sea para distinguir los parques, las carreteras, lagos, ríos… Por último añadiremos los edificios que deseemos, las líneas de transporte público, las longitudes de los diversos caminos y todo aquello que consideremos oportuno. Para acabar, decir que sólo cartografiaremos aquellas zonas que sean espacios públicos y que nunca debemos poner aquello que aparezca como privado (ya que estaremos violando el derecho de privacidad de la gente). Otra de las opciones que podemos ver, son los diversos tags que podemos incluir en nuestro mapa y que luego se pondrán como palabras reservadas en nuestro XML, así como el significado que tienen (como por ejemplo, highway residential que nos indicará que la carretera en la que estamos pertenece a un barrio residencial, hay muchas señalizaciones que podemos leer accediendo a esta página).Y por último, decir que también se nos explica cómo subir datos GPS. Podemos, unirnos a un grupos para editar

mapas, etc. También obtendremos ayuda para editar nuestros mapas ya sea para añadir marcas o tags, ver lo que otros usuarios han hecho en nuestra zona de interés, etc. El penúltimo paso, es la subida de todo el trabajo que hayamos realizado y por último, ver el resultado final en la Web.

La opción que más hemos utilizado es la de desarrolladores (Development), en ella nos vamos a encontrar con un esquema que representa cómo están relacionados los diversos componentes de OSM, con las diversas herramientas que lo editan, el servidor de mapas que utiliza. En la parte derecha de la ventana, aparecen distintas secciones que vamos a ir comentando. En General Development, tenemos distintas subsecciones. En Componente Overview, nos explicarán cada una de las distintas componentes de manera más detallada, como pueda ser la API (comentada en la sección el API de OSM). La parte de OSM Front End (compuesta por Slippy Map, lo que aparece en la página principal de OSM, mostrándonos los mapas según las diversas coordenadas que le pongamos en la URL, y por Potlatch que es un editor online de mapas que está escrito en lenguaje ActionScript). En la siguiente parte, se nos explicará la forma que tienen de renderizar los mapas (ya sea utilizando Mapnik, que es la aplicación usada cuando se usa Slippy Map por defecto, también se explica el modo de funcionamiento o bien, usando el sistema distribuido de renderizado como puede ser Osmarender el cual irá corriendo a través de varios clientes que cogiendo los datos de la API, los pasarán a formato SVG utilizando Inkscape y una vez que las hayan renderizado las subirán al servidor).Luego, también se mencionan sistemas renderizadores y editores, ambos ya han sido explicados anteriormente. Otra de las subsecciones que aparecen aquí, es la de cosas que hacer (Things to Do), en ella, se anima a la gente a participar de nuevos proyectos que se necesitan para completar OSM, ya sea haciendo guías, ampliando documentación, etc. En la subsección top ten tasks, se muestra una lista con las diez tareas que se pueden realizar, aparecerán tachadas aquellos proyectos que ya han sido asignados. La siguiente subsección es la comunidad de desarrolladores (Developer community), en el que se nos anima a participar de esta experiencia. En la última subsección se nos pondrá al corriente de la historia de los desarrolladores y nos mostrarán el estatus que han alcanzado a lo largo del tiempo. Todo esto que hemos comentado se encuentra englobado en lo que se denomina, introducción al desarrollo de OSM.

Otra sección es la de instalación de OSM en nuestra máquina para poder desarrollar (Installing OSM on your machine so you can develop). En la que nos encontraremos información de cómo instalar puertos rails (aquí se nos explicará la forma de hacerlo de manera muy detallada ya sea para instalarlo en Ubuntu, etc.) Luego también podemos echar un rápido vistazo a cómo está distribuido y las partes de las que consta el código. Otra parte, es la de obtener el código (Getting the Source), como su nombre indica, se nos explicará la forma de obtener el código fuente de las aplicaciones ya programadas para que podamos empezar a desarrollar las nuestras. Y por último se nos muestra la opción de obtener datos (Getting Data), en la cual se nos explicará la forma de editar/crear mapas. Lo primero de todo sería escoger nuestra región. En segundo lugar, sería la construcción de una URL para nuestro mapa como por ejemplo:

```
http://api.openstreetmap.org/api/0.5/map?bbox=11.54,48.14,11.543,48.145
```

A través de la cual vamos a ser capaces de acceder a la región que estemos creando o editando. En tercer lugar, la forma de bajarnos la información, se explica cómo hacerlo a través de wget. También se nos proporciona el protocolo a usar para poder hacer todo el proceso.

Otra de las cosas que podemos hacer si queremos entrar a desarrollar, es crearnos una cuenta, ya que es un requisito imprescindible a la hora de poder realizar los cambios que deseemos (siempre y cuando sean admitidos), en ella se nos explicará la forma de acceder a la lista de correos electrónicos de otros desarrolladores, etc. También se nos adjuntará otra información de interés como puede ser la de probar nuevas APIs que se han hecho o aprender a usar el servidor de pruebas.

Otra de las secciones que podemos encontrar aquí es la forma de exportar mapas a nuestros propios sitios Web ya sea construyéndonos nuestro propio Slippy Map o la posibilidad de exportar los mapas con otros formatos diferentes. Y por último, mencionar algunas otras secciones como son la miscelánea (que nos explicarán diversos proyectos, iconos de los mapas y otras discusiones) y la importación de datos (Data imports).

Otra de las opciones que nos encontraremos en la ventana principal es la de FAQS[13] en ella se nos mostrará las preguntas que con más frecuencia suelen hacerse y las respuestas que se dan desde una óptica de los creadores del sitio Web del que estamos hablando. Y por último, Press que es la opción para reporteros que quieran escribir sobre este portal, en ella se mostrará información sobre el proyecto OSM. En la parte izquierda, debajo del esquema que nos aparece, podemos ver que se nos muestra los distintos proyectos con dos opciones, Code svn (pinchando sobre esta opción aparecerá una nueva ventana con carpetas que contienen los códigos fuentes de la aplicación a la cual estemos accediendo) y en la parte Bugs trac tickets, se nos mostrará los errores que se han encontrado en esa aplicación y que por supuesto, han de ser revisados.

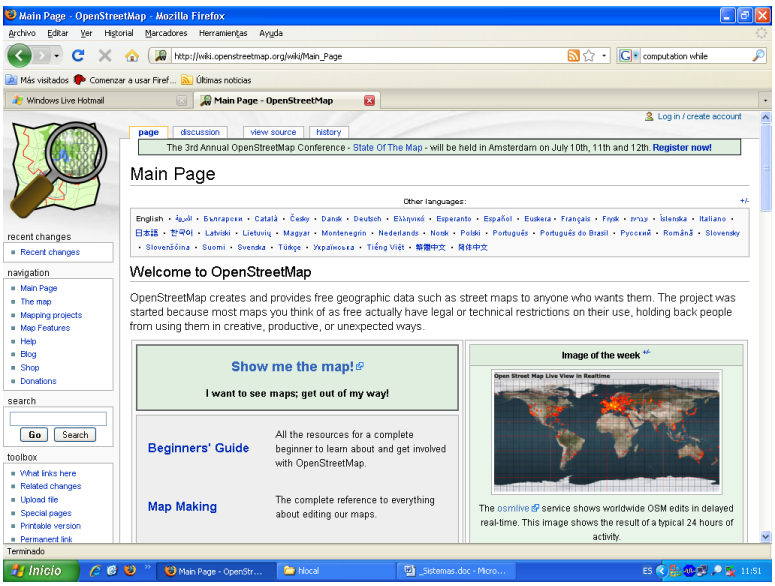


Imagen de la Wiki de OSM.

[13] FAQS: Conjunto de preguntas frecuentes.

Si nos fijamos en la parte superior de la página encontraremos varias pestañas en las cuales podemos navegar. La primera de ella es page, que es donde nos encontramos actualmente. La segunda es discusión en la que accederemos a diversos que temas que se están o han sido tratados. View source que nos permitirá ver y modificar (si tenemos los permisos adecuados) el código fuente de la Wiki. Y por último, la pestaña history en la que veremos los avances que se han logrado, de manera cronológica.

También existen otras cosas de interés, como pueden ser noticias que se van publicando periódicamente, en las cuales se nos irá informando sobre nuevos mapas que próximamente podremos utilizar. Un calendario con los próximos eventos que van a suceder (como charlas) y los lugares en dónde se van dar. Una serie de portales en donde se obtendrán las diversas herramientas que usaremos para modificar los mapas tales como JOSM, Merkaator, etc. (mencionadas todas ellas en apartados anteriores a este). Imagen de la semana, en la cual se va a mostrar el mejor mapa del que se tienen los datos cartográficos. Y por último mencionar la sección de la meta información que podemos obtener, tales como la suscripción a la página a través de RSS (en la cual se nos informará de manera puntual sobre las últimas novedades concernientes a esta materia. Novedades como pueda ser la inclusión de una nueva API, su estado actual de desarrollo y los posibles problemas que puede tener. Estadísticas y reportajes aparecidos en diarios. Y proyectos hermanos basados en la tecnología OSM como pueda ser OpenStreetPhoto y cosas parecidas.

Anteriormente hemos mencionado, tanto en el uso de Google Maps como en el de OSM, varias veces las siglas API pero la gente profana a estos términos querrá saber qué es un API. Intentaremos dar una definición exacta de qué es y para qué sirve. Un API sirve para poderse comunicar entre componentes software. Se trata del conjunto de llamadas a ciertas bibliotecas que ofrecen acceso a ciertos servicios desde los procesos y representa un método para conseguir abstracción en la programación, generalmente (aunque no necesariamente) entre los niveles o capas inferiores y los superiores del software. Uno de los principales propósitos de una API consiste en proporcionar un conjunto de funciones de uso general, por ejemplo, para dibujar ventanas o iconos en la pantalla. De esta forma, los programadores se benefician de las ventajas de la API haciendo uso de su funcionalidad, evitándose el trabajo de programar todo desde el principio. Las APIs asimismo son abstractas: el software que proporciona una cierta API generalmente es llamado la implementación de esa API.

Otras de las cosas que hemos mencionado en esta parte, es el término software libre, a continuación expondremos qué es y por qué es mejor el uso de este tipo de programas a la hora de trabajar con ellos.

El software libre es un tipo software mediante el cual los usuarios tienen la posibilidad de modificar, copiar y redistribuir a voluntad de manera libre. Estas aplicaciones suelen ser gratuitos o en su defecto tener un precio, que es el de distribución. Por lo que se deduce de la anterior frase, hay que tener en cuenta que el mero hecho de ser libre no quiere decir que es gratis, ni tampoco que sea de dominio público, esto último se produce cuando el autor de la obra la dona de manera desinteresada a la humanidad o por el contrario han caducado los derechos de autor, de acuerdo con las leyes vigentes que reglan cada uno de los países (como ocurre con las películas, las cuáles sus derechos caducan pasados 70 años).

El software libre garantiza las siguientes libertades:

- ✓ *Libertad 0*: la libertad de usar el programa, con cualquier propósito
- ✓ *Libertad 1*: la libertad de estudiar cómo funciona el programa, y adaptarlo a tus necesidades.
- ✓ *Libertad 2*: la libertad de distribuir copias, con lo que puedes ayudar a tu vecino.
- ✓ *Libertad 3*: la libertad de mejorar el programa y hacer públicas las mejoras a los demás, de modo que toda la comunidad se beneficie.

Las libertades 1 y 3 requieren que esté disponible el código fuente porque estudiar y modificar software sin su código fuente es muy poco viable. A su vez existen multitud de licencias en este mercado pero que no expondremos aquí (para más información consultar en Internet sobre éstas). Como hemos podido comprobar el término de software libre no significa que sea totalmente gratuito o que no posea licencias y derechos de autor, todo al contrario, se puede llegar a generar todo un negocio a través de él. Pero entonces, ¿qué diferencia el software libre del que no lo es? En primer lugar, el software que no es libre no puede ser modificado por otros usuarios y lo que es peor, no se puede redistribuir ni hacer copias del mismo si no se poseen permisos especiales por parte de las empresas creadoras. Otro de los motivos para usar este tipo de aplicaciones, es que al ser compartido por millones de usuarios en el mundo y mejorado constantemente, se evitan ataques tanto de virus, creados por gente poco ética, como de hackers que hay en la red, convirtiendo a este tipo de herramientas en las más seguras que pueden existir en la actualidad. Existen proyectos en ayuntamientos españoles para sustituir el software de código cerrado, software libre con el fin de optimizar recursos y ahorrar dinero en licencias de programas. Es un hecho contrastado el avance de estas tecnologías sobre la población y el alto impacto que tendrán en un futuro, pudiendo llegar a que todo tipo de software sea libre.

Para terminar con este bloque, daremos una explicación del por qué nos decantamos por la elección de OSM en detrimento de Google Maps exponiendo los puntos fuertes y además los beneficios que hemos obtenido al hacer uso de esta herramienta a la hora de poder implementar los algoritmos de procesado y movimiento (que explicaremos detalladamente en secciones posteriores) que utilizamos.

El uso principal de OSM por nuestra parte, es debido sobre todo a que es código abierto y de herramientas de procesamiento de mapas, lo cual ha facilitado el trámite de descargarnos los mapas (ya que así no hemos tenido que pedir ningún tipo de contraseña para el API, cosa que sí nos ocurría con Google Maps), la comprobación de errores, etc. Otro de los puntos a su favor ha sido el hecho de que exista una completa ayuda y explicación del uso de OSM así como los códigos fuentes que nos han permitido asimilar mejor el modo de uso, lo cual no quiere decir que no hayamos encontrado algunas dificultades a la hora de usarlo. Como ejemplo ilustrativo de estas dificultades es el caso de conseguir el mapa. Para ello, nos descargamos un XML con la ruta del mapa que queremos (con sus coordenadas al final, longitudes y latitudes mínimas y máximas) que previamente habremos obtenido gracias a un buscador de nombres (nameFinder) que nos proporciona OSM y que explicaremos más adelante (le damos un nombre de una ciudad y él nos devuelve sus coordenadas). Una vez obtenido

el XML, mediante el uso de un conversor y una hoja de estilos, lo pasamos a un SVG y de ahí a PNG. Este proceso ha sido muy lento, por la multitud de puntos a procesar, y hemos tenido que depurar y optimizar todo el código de manera que no tardase tanto. Como hemos comentado anteriormente, también nos ha sido de utilidad el uso de JOSM para la comprobación del correcto funcionamiento del algoritmo de movimiento a través de los mapas, ya que hemos podido observar que la salida de éste (el siguiente punto al que nos movemos) coincidía o no con el siguiente punto real del mapa, ahorrándonos un trabajo y esfuerzo considerable al no tener que ir buscando por el XML los puntos.

4. Arquitectura Cliente-Servidor. Comunicación mediante sockets

Este proyecto está estructurado con una arquitectura de cliente-servidor (C/S). Esta arquitectura ofrece diferentes ventajas frente a otras. Entre estas se encuentran:

- Fácil escalabilidad: facilidad a la hora de aumentar el número de clientes y se servidores.

- Fácil mantenimiento: Al estar repartido las funciones, se podrá reparar los problemas surgidos en un servidor sin afectar a sus clientes.

- Control centralizado: un programa cliente irregular no podrá dañar el sistema, ya que los recursos son controlados exclusivamente por el servidor.

- Amplia oferta de tecnologías para desarrollar este modelo.

- Pero como en todo, no van a ser todo ventajas también se encuentran desventajas dentro de este modelo. Entre ellas destacamos:

- Gran tráfico en la transferencia de datos.

- Cuando el servidor se encuentra fuera de servicio las peticiones provenientes de los clientes no pueden ser atendidas.

- No todos los hardware son capaces de alojar al servidor y dar cabida a las peticiones de los clientes.

- Los clientes no disponen de todos los recursos que tiene el servidor, por ejemplo en una Web (como es nuestro caso) en el cliente no se dispone de disco duro.

Después de haber resumido en pocas palabras una idea general del modelo C/S, entremos en el tema de nuestra aplicación.

4.1 Comunicación entre los elementos de la arquitectura

El servidor, en nuestro caso es el responsable de realizar todos los cargos computacionales, y el cliente es una mera presentación de los cálculos. La división entre estas dos partes debe ser clara y estar bien distinguida. Ahora el problema viene ante la necesidad de comunicar ambas partes tanto al cliente con el servidor como al servidor con el cliente. Nosotros necesitábamos que el cliente en cualquier momento pidiese datos al servidor, como el servidor le pasase datos al cliente para actualizaciones o demás actuaciones. Esto en nuestro proyecto se refleja de la siguiente forma: la parte cliente necesitara actualizar los datos de los juegos, posición de jugadores, puntos de estos, etc. Para ello en esta parte saltara continuadamente un temporizador que pedirá datos actualizados al servidor. También el servidor necesita de información del cliente por ejemplo las teclas que son pulsadas, posición donde se ha pulsado con el ratón sobre el panel, etc. Como bien se ha explicado la comunicación que se debe establecer debe ser bidireccional ya que ambos necesitan notificarse información de forma reciproca. Estos motivos expuestos con anterioridad hacen la elección de la forma de comunicación se hiciese mediante sockets y no mediante otras tecnologías. Entre las otras tecnologías que se barajaron estaban los web services y RMI (invocación remota de métodos). Fueron descartadas por lo anteriormente comentado, con estas últimas únicamente conseguíamos una comunicación direccional, en un único sentido, y nosotros lo que necesitábamos era una comunicación en ambos sentidos, una comunicación full-dúplex. Vamos a intentar explicar el concepto de socket, su uso, forma de usarlos, etc. Los sockets nos permiten una conexión entre dos programas en red para que pueda haber un intercambio de información entre ellos. Java pone a disposición del programador estas clases ya implementadas y lo único que se debe hacer es un protocolo entre ellos, para asegurarse de que toda la información que se van a intercambiar es correcta y en un orden preestablecido. Dicho protocolo esta creado por nosotros y le definiremos a continuación.

La forma de utilizar sockets gracias a Java es sencilla [*Véase* 15], para la parte del servidor ya disponemos de una clase ServerSocket que podemos instanciar facilitando un número de puerto. El numero de puerto debe ser un entero entre 1 y 65535.los números del 1 al 1023 son usados para servicios como web, telnet, ftp, etc. El resto del 1023 en adelante los podremos usar a nuestro antojo. Ejemplo:

```
ServerSocket socket = new ServerSocket (45555);
```

Para aceptar clientes disponemos de la siguiente instrucción:

```
Socket cliente = socket.accept ();
```

Existe la posibilidad de poder aceptar más de un cliente. Para ello hay que realizar una programación multitarea mediante hilos, es decir cada vez que solicite un cliente al servidor se debe crear un hilo que atienda esta tarea. Como nosotros necesitamos esto hemos implementado lo comentado previamente.

Para la parte del cliente se crea de la siguiente forma, poniendo la dirección IP del servidor y el número de puerto por el que queremos que se comunique.

```
Socket socket = new Socket ("dir IP servidor", 45555);
```

Para finalizar la conexión se debe cerrar correctamente estos canales de una forma segura y así la comunicación queda suprimida. Se usa la instrucción "close".

```
Cierre de la conexión con el cliente      cliente.close();

Cierre del socket servidor                socket.close();
```

La información que se envía a través de estos canales de sockets la podemos diferenciar en dos: envío de datos de tipos básicos (integer, float, string, boolean, etc.), y envío de objetos de clases creadas por nosotros y más complejas.

Para los tipos básicos no se plantean problemas únicamente usar bien los siguientes métodos. Siempre el método se llama o bien si lo que quieres es escribir write o si quieres leer read, seguido del tipo. Por ejemplo para integer: writeInt(),readInt().Para strings o cadenas de caracteres usaremos writeUTF() y readUTF().

El problema viene cuando los tipos no son básicos sino objetos para ello usaremos los métodos readObject() y writeObject() de ObjectInputStream y ObjectOutputStream. Una cosa para que todo funcione a la perfección debe ser que las clases que viajen por el socket deben implementar la interfaz Serializable[14]. Si no es así no podemos realizarlo. Si la clase tiene atributos de clases primitivos, integer, float, String, no hay problema ya que estos implementan dicha interfaz. Pero si tenemos atributos de otras clases, estas deben también que implementar dicha interfaz. Para que lo implemente simplemente debemos poner que implementar dicha interfaz:

```
class Datos implements Serializable
{
  int a;
  boolean b;
  ClaseSerializable c;
}
```

[14] Serializable: El objeto en cuestión puede ser transformado a una cadena de bits.

Pero si alguna de ellas no es serializable debemos implementar los siguientes métodos, tanto el de lectura como el de escritura. En el de escritura da igual el orden, en el de lectura no ocurre así.

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException {
    out.writeInt(a);
   out.writeBoolean(b);
}

private void readObject(java.io.ObjectInputStream in) throws IOException,
ClassNotFoundException

{
   a = in.readInt();
   b = in.readBoolean();
}
```

En cualquier caso para ir leyendo o escribiendo se va o volcando en el flujo o recogiendo del flujo (Stream).

Un gran problema de los sockets es que puede haber momentos en que la información se pierda, por ejemplo porque el servidor cuando envíe la información al cliente cierre el socket.

Para ello hay que esperar un tiempo para que el cliente recoja los datos, si no es así al cerrarlo si el cliente no ha tenido tiempo de obtenerlos se pierden. Para esto precisamente es para lo que sirve la instrucción setSoLinger que espera un tiempo en segundos que le pasas por parámetro, también le debes poner a true para evitar que el servidor cierre inmediatamente a continuación de haber puesto los datos. Ejemplo en el que no cerramos inmediatamente se espera 10 segundos.

```
cliente.setSoLinger (true, 10);
```

Empecemos a hablar del protocolo creado por nosotros. Antes de ello contextualicemos nuestra estructura para así entender mejor todo.

El usuario elegirá un juego del aparte cliente el que el prefiera de entre los disponibles. Tendremos en la parte servidor una clase que hemos llamado SuperServidor que recibe esta petición. Estas peticiones siempre entran por el mismo número de puerto. El SuperServidor que siempre está a la espera crea un hilo delegando la tarea ya asignando un número nuevo de puerto para todas las siguientes comunicaciones entre ese cliente y servidor. El SuperServidor volvería a la espera de nuevas peticiones y la otra tarea ya habrá quedado delegada a una clase denominada SocketServidor con un juego. El código que implementa la espera y delegación de tareas del SuperServidor es el siguiente:

```
    puertoPrin = 45700;

    sigPuerto = puertoPrin + 1;
```

```
        while (true) {
            try {
                socket = new ServerSocket(puertoPrin);
                System.out.println("**** SUPERSERVIDOR Esperando cliente");
                cliente = socket.accept();
                //Recibir nombre del juego
                this.obtieneFlujoDatosIn();
                String nombreJuego = this.bufferIn.readUTF();
                System.out.println("Recibido el nombre del juego");
                //Envio Puerto
                this.obtieneFlujoDatosOut();
                this.bufferOut.writeInt(sigPuerto);
                System.out.println("**** SUPERSERVIDOR Enviado puerto");
                System.out.println("**** SUPERSERVIDOR Conectado con cliente de "
+                       cliente.getInetAddress());
                this.cerrarConexion();
                if (nombreJuego.compareTo("PILLA-PILLA") == 0) {
                    Runnable nuevoCliente = new PillaPillaSolo(sigPuerto);
                    Thread hilo = new Thread(nuevoCliente);
                    hilo.start();
                    sigPuerto++;
                }
                else if (nombreJuego.compareTo("MONO_PLATANERO") == 0) {
                    Runnable nuevoCliente = new MonoPlatanero(sigPuerto);
                    Thread hilo = new Thread(nuevoCliente);
                    hilo.start();
                    sigPuerto++;
```

```
            }
            else if (nombreJuego.compareTo("COME-COCOS") == 0) {
                Runnable nuevoCliente = new ComeCocos(sigPuerto);
                Thread hilo = new Thread(nuevoCliente);
                hilo.start();
                sigPuerto++;
            }
            else if (nombreJuego.compareTo("SNAKE") == 0) {
                Runnable nuevoCliente = new Snake(sigPuerto);
                Thread hilo = new Thread(nuevoCliente);
                hilo.start();
                sigPuerto++;
            }
            else if (nombreJuego.compareTo("PINCHA") == 0) {
                Runnable nuevoCliente = new PinchaGlobos(sigPuerto);
                Thread hilo = new Thread(nuevoCliente);
                hilo.start();
                sigPuerto++;
            }
          else {
                System.out.println("ERROR: EL JUEGO NO EXISTE");
            }
} catch (IOException ex) {
            Logger.getLogger(SuperServidor.class.getName()).log(Level.SEVERE,
null, ex);
          }
  }
```

```
}
```

El protocolo es el SuperServidor está escuchando por el numero de puerto identificado en puertoprin, cuando un cliente necesita de sus servicios el cliente le envía el nombre del juego. El SuperServidor a continuación envía un número de puerto que será el de todas las siguientes comunicaciones. Así conseguimos tener más de un cliente conectado a la vez ya que en cada una se crea un hilo que es la encargada de futuras comunicaciones. El servidor volvería a la espera.

Cuando el cliente decide que empieza a jugar el cliente pedirá una nueva conexión por el numero de puerto que le ha indicado el SuperServidor, la tarea ya ha sido delegada a un socket servidor que como en el caso del SuperServidor estará a la espera. Cuando este pida la conexión, el SocketServidor que está escuchando la aceptará y se empezara la transmisión de datos en un cierto orden. Lo primero que envía el cliente es la dificultad del juego y el nombre del mapa, ambas cosas elegidas por el usuario. El SocketServidor arrancara el juego correspondiente, información disponible en el SuperServidor, y enviara al cliente: imagen del mapa sobre el que se va a desarrollar el juego, una lista con los jugadores sus posiciones en el mapa e información de estos, y las máximas y mínimas latitudes y longitudes de la región solicitada. El socket servidor seguirá trabajando en el juego calculando puntos y demás. Cuando el cliente solicite una nueva conexión puede ser por dos cosas o bien porque ha saltado el temporizador del cliente y quiere actualización de los datos para que se muestren correctamente o bien porque el usuario ha tocado alguna tecla y desea moverse, esto sería análogo al juego del pincha-globos pero con el click del ratón. Para diferenciar ambos casos se envía un testigo en forma de entero que le dice al servidor que es lo que ha pasado. Si ha ocurrido la primera opción, ha saltado el temporizador pero no se ha tocado tecla o click del ratón, el SocketServidor recibe el testigo únicamente y responde enviando si se ha finalizado el juego por alguna causa y a continuación la información sobre los jugadores, lo mismo que antes sus posiciones, vidas, etc. Si por el contrario lo que ha pasado es que se ha tocado tecla o se ha clickeado con el ratón, el cliente envía el testigo y aparte la tecla o posición, el SocketServidor lo jugadores de nuevo actualizados con el procesamiento de la información recibida ya procesada. Cuando todo esto termina se cierra la conexión y todo quedaría igual. La parte de la aplicación que realiza esto es la siguiente:

```
while(true){
  abrirConexion();
  obtieneFlujoDatosIn();
  etapa = this.bufferIn.readInt();
  System.out.println("Recibida etapa");
  // Se envía un entero y una cadena de caracteres.
  if (etapa == 1) {
     limpiezaVar();
```

```
                obtieneFlujoDatosIn();
                this.dificultad = this.bufferIn.readInt();
                System.out.println("Recibido dificultad");
                obtieneFlujoDatosIn();
                fuenteMap=this.bufferIn.readUTF();
                System.out.println("Recibido nomMApa");
                comenzar();
                ponerTiempo(); //el poner tiempo ya hace el start
                tiempo.start();
                ImageIcon fondo;
                if(fuenteMap.compareTo("PERSONALIZADO")==0){
                fondo = new
ImageIcon("c:\\hlocal\\GOPIRO\\Imagen\\map"+this.puerto+".png");
                }
                else{
                   fondo=new
ImageIcon("c:\\hlocal\\GOPIRO\\MapasFijos\\"+this.fuenteMap+"\\map"+this.fuent
eMap+".png");

                }
                obtieneFlujoObjetosOut();
                this.bufferObjetosOut.writeObject(fondo);
                System.out.println("Enviado fondo de pantalla");
                obtieneFlujoObjetosOut();
                this.bufferObjetosOut.writeObject(jugadores);
                System.out.println("Enviado jugadores");
               //después de comenzar enviamos las máximas latitudes
                obtieneFlujoDatosOut();
```

```
            this.bufferOut.writeDouble(this.uXML.getMaximaLat());
            System.out.println("Enviado maxima latitud");
            this.bufferOut.writeDouble(this.uXML.getMinimaLat());
            System.out.println("Enviado minima latitud");
            this.bufferOut.writeDouble(this.uXML.getMaximaLon());
            System.out.println("Enviado maxima longitud");
            this.bufferOut.writeDouble(this.uXML.getMinimaLon());
            System.out.println("Enviado minima longitud");
        } else if (etapa == 2) {
            //tecla
            synchronized(jugadores){
            obtieneFlujoDatosIn();
            String opcion = this.bufferIn.readUTF();
            System.out.println("Recibido String opción");
            if (opcion.compareTo("teclayjugadores") == 0) {
                //ES LA OPCION CUANDO SE HA PULSADO TECLA
                this.obtieneFlujoDatosIn();
                this.mover(this.bufferIn.readInt());
                obtieneFlujoObjetosOut();
                synchronized(bufferObjetosOut){
                this.bufferObjetosOut.writeObject(this.jugadores);
                }
                System.out.println("Enviado jugadores");
            }
            //cuando es el juego del ratón
            else if (opcion.compareTo("ratonyjugadores") == 0){
```

```
                this.obtieneFlujoObjetosIn();
                this.clickeado((Pixel) this.bufferObjetosIn.readObject());
                obtieneFlujoObjetosOut();
                synchronized(bufferObjetosOut){
                this.bufferObjetosOut.writeObject(this.jugadores);
                }
                System.out.println("Enviado jugadores");
              }
            //cuando salta el timer y no tecla
            else{
            obtieneFlujoDatosOut();
            this.bufferOut.writeBoolean(finJuego);
            System.out.println("Enviado fin de juego????");
            obtieneFlujoObjetosOut();
            synchronized(bufferObjetosOut){
            this.bufferObjetosOut.writeObject(this.jugadores);
             }
            System.out.println("Enviado jugadores");
             }
        }
            //  }
            //OPCION CUANDO HA SALTADO ELTIMER DEL CLIENTE
        } else {
            this.vivoApplet = false;
        }
        cerrarConexion();
```

```
        }
```

Volver a repetir que la comunicación es en ambos sentidos pero notar que el cliente es el que decide cuando hay comunicación y cuando no, aunque ambos envían y reciben datos. Si no fuese de esta forma se perdería el espíritu de la arquitectura cliente-servidor y ambos serían servidores.

Importante detallar los métodos usados para: abrir las conexiones, obtener los buffers tanto de objetos como de tipos básicos, cerrar las conexiones. Estos han servido para transmitir los datos, sin ellos no sería posible.

Función que abre la conexión.

```
public void abrirConexion() throws IOException {
        try {
        socket = new ServerSocket(puerto);
       // Se acepta una conexion con un cliente. Esta llamada se queda
        // bloqueada hasta que se arranque el cliente.
        System.out.println("Esperando cliente");
        cliente = socket.accept();
        System.out.println("Conectado con cliente de " + cliente.getInetAddress());
        cliente.setSoLinger(true, 20);
        }
        catch(IOException e){
            System.out.println("Error");
        }
    }
     /*Función que se encarga de cerrar una conexión desconectando el socket*/
    public void cerrarConexion() throws IOException{
        socket.close();
        System.out.println("cerrada conexión");
    }
```

```
    /*Función que devuelve el flujo de datos de entrada para los
datos(enteros,booleanos,etc)*/
    public void obtieneFlujoDatosIn() throws IOException{
          if(cliente!=null){
               bufferIn = new DataInputStream(cliente.getInputStream());
           }
    }

 /*Función que devuelve el flujo de datos de salida para los
datos(enteros,booleanos,etc)*/
    public void obtieneFlujoDatosOut() throws IOException{
           if(cliente!=null){
               bufferOut = new DataOutputStream(cliente.getOutputStream());
           }
    }

    /*Función que devuelve el flujo de datos de entrada para el resto de objetos*/
    public void obtieneFlujoObjetosIn() throws IOException{
        if(cliente!=null){
          bufferObjetosIn = new ObjectInputStream(cliente.getInputStream());
        }

    }

   /*Función que devuelve el flujo de datos de salida para el resto de objetos*/
    public void obtieneFlujoObjetosOut() throws IOException{
      if(cliente!=null){
```

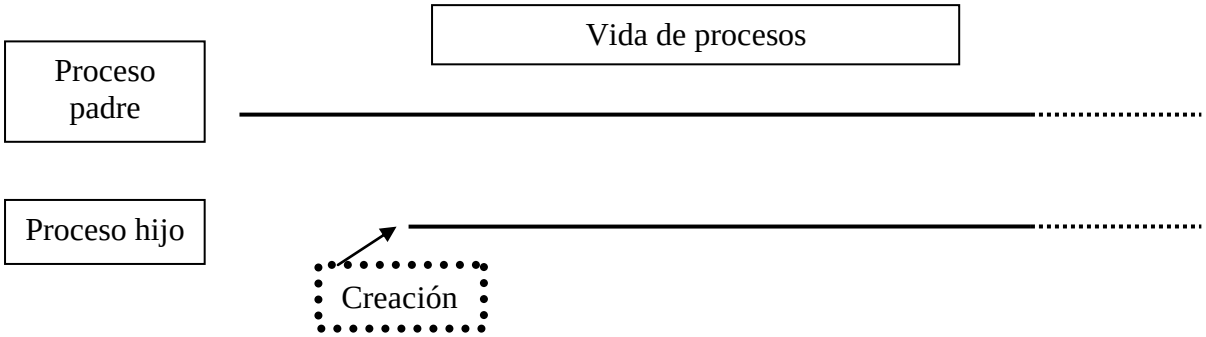
```
        if(bufferObjetosOut==null){
        bufferObjetosOut = new ObjectOutputStream(cliente.getOutputStream());
        }
        else{
           synchronized(bufferObjetosOut){
        bufferObjetosOut = new ObjectOutputStream(cliente.getOutputStream());
           }
           }
       }
}
```

4.2 Programación multitarea

Como se ha comentado previamente para conseguir que el servidor pueda atender varios juegos es imprescindible que se implemente la multitarea para hacer esto posible.

La multitarea consiste en crear procesos y delegar la tarea en estos. Se usan para programas que tengan que realizar varias tareas de manera simultánea, en los que la ejecución de una parte requiera tiempo y no deba detener el resto del programa.

Para nosotros, cuando el usuario ha elegido el juego y el SuperServidor recibe dicha petición, éste crea un nuevo hilo o proceso que se va a encargar de ejecutarlo. De esta forma se consigue que el SuperServidor siga a la espera de futuras peticiones de clientes y no se detenga atendiendo al actual juego.



Para hacerlo todo posible hay que definir clases un poco especiales si las tareas deben ejecutarse en un proceso diferente. Necesario que incluyan el método run() en la propia clase para poder arrancarlos desde fuera de ellos mediante start().

El siguiente ejemplo ilustra la creación del hilo y el arranque de éste.

```
Runnable nuevoCliente = new ComeCocos(sigPuerto);

Thread hilo = new Thread(nuevoCliente);

hilo.start();
```

La clase comecocos como todos los juegos heredan de una clase SocketServidor que implementa la interfaz "runnable" y en la que se dispone del método run. Como muestra pondremos las cabeceras de las clases y la del método.

```
public class ComeCocos extends SocketServidor{…..}



public abstract class SocketServidor implements Runnable{

…

…

   public void run()  {…}

…….

}
```

Un gran problema, que plantea muchos quebraderos de cabeza dentro de la programación multitarea es la concurrencia. Esta sucede cuando dos procesos que están siendo ejecutados intentan acceder a un recurso que tienen en común. Es inadmisible que ocurra, ya que el resultado que proporcionará con toda seguridad será erróneo. Un ejemplo de problema de concurrencia sería: un hilo escribe en un fichero "patata" y el otro escribe "cebolla", al final quedarán todas las letras entremezcladas. Hay que conseguir que uno de ellos escriba primero su palabra y el otro después. Para evitar estas desagradables situaciones, debemos de arbitrar entre las hebras. Arbitrar sobre los hilos será lo que llamaremos la sincronización de procesos o de hilos.

La sincronización de hilos no es más que establecer un orden entre los procesos para acceder al recurso común. Cuando un hilo este haciendo uso del recurso debe marcar de alguna forma que lo está usando. De esta forma el otro hilo verá que el recurso está ocupado y debe esperar a su desocupación. Para esto Java pone a nuestra disposición la palabra reservada "synchronized". Dicha palabra se utiliza para indicar que ciertas partes del código están sincronizadas, es decir, que solamente un subproceso puede acceder a dicho método a la vez. Si se pone sobre el nombre de una variable indicamos que mientras uno use esa variable nadie más puede acceder. Esto funciona parecido a un cerrojo, si alguien está dentro cierra el cerrojo y el que llegue y quiera entrar debe esperar. Se espera hasta que el otro salga y quite el cerrojo.

En nuestro proyecto hemos usado "synchronized" sobre jugadores y sobre el buffer sobre el que escribe el socket. El motivo de hacerlo sobre jugadores es para que ningún otro hilo modifique esta variable, mientras ya hay otro modificándola. Así se evita que haya incoherencias en la información. El de hacerlo sobre el buffer es para que todos los datos aparezcan en su orden correcto y no haya problemas del tipo que uno escribe antes que otro o viceversa. Muestra de lo explicado es:

```
synchronized(jugadores){

                    this.bufferObjetosOut.writeObject(this.jugadores);

                }
```

Tener en mente, que lo que se haga dentro de estos fragmentos debe ser algo ligero. Con ligero nos referimos a algo que no tarde mucho tiempo en ejecutarse porque sino detendría mucho a nuestro programa y podrían surgir otros tipos de problemas.

5. Adquisición de información geográfica y tratamiento para su uso

En este capítulo nos centraremos en definir el fichero del que recibimos toda la información necesaria (estructura y demás), en explicar el paso de dicho fichero a una imagen, y en explicar cómo usar la imagen en nuestra aplicación.

5.1 Fichero con la información geográfica

El fichero para manejar los datos que proporciona el API de OpenStreetMap es un fichero con extensión ".OSM". Son ficheros escritos utilizando un lenguaje de marcas. Un lenguaje de marcas o de marcado es un modo de encriptar un texto. Las etiquetas aportan o bien un significado especial de lo que encierran entre ellas o bien definen una forma de presentación. Estos lenguajes son muy conocidos actualmente gracias al auge del lenguaje HTML, muy presente en el mundo de la Web. Existen diferentes tipos de lenguaje de marcados, entre ellos destacar: de presentación, que define una forma de presentar los datos; de procedimientos, orientado al mismo fin que el anterior pero los diferencian en que en este caso la persona que manipula el texto puede tenerlo accesible; descriptivos, utiliza las etiquetas para englobar texto pero sin ocuparse de la presentación. En este último tipo es donde se encuadra el lenguaje XML. Los ficheros obtenidos de la comunidad OpenStreetMaps siguen estas directrices, destacar su

estructura en forma jerárquica. Tampoco debemos olvidar el significado que tiene cada una de sus etiquetas. Ahora intentaremos profundizar un poco más en cada uno de estos aspectos, tanto estructura como significado de las etiquetas, siempre orientándolo hacia las cosas de las que hemos hecho mayor uso.

Estructura del fichero y significado de etiquetas

El fichero dispone de una cabecera:

```
<?xml version="1.0" encoding="UTF-8"?>

<osm version="0.5" generator="OpenStreetMap server">
```

A continuación tendría un cuerpo que es donde está la información que usará el algoritmo.

Las partes más importantes que nosotros manejamos son tres: definición de máximas y mínimas coordenadas, nodos que componen el mapa y caminos que hay en el mapa.

Lo primero que aparecerá será una etiqueta “bounds” con información de latitudes y longitudes máximas y mínimas que tiene el mapa, es decir el lugar que abarca expresado en coordenadas.

```
<bounds minlat="40.43075" minlon="-3.71594" maxlat="40.43552" maxlon="-
3.70997"/>
```

Luego aparecerán todos los nodos del mapa “node id” ellos contendrán un identificador de nodo, único, sus coordenadas (latitud y longitud), usuario que lo crea, si es visible, y más información adicional.

```
  <node id="310655027" lat="40.4309843" lon="-3.7135327" user="Remiguel"
visible="true" timestamp="2008-11-08T20:48:30+00:00">

    <tag k="highway" v="bus_stop"/>

    <tag k="Numero" v="Nr: 192"/>

    <tag k="Source" v="EMT; RLM"/>

    <tag k="note" v="Linea 2"/>

    <tag k="name" v="Guzman El Bueno Nº 5"/>

  </node>
```

Después de la definición de todos los nodos podremos encontrar los caminos que los unen entre ellos, si pensamos que esto es un grafo lo anterior serían sus nodos y los caminos las aristas que los unen.

Los caminos vienen precedidos por las etiquetas “way id”, con un identificador único de camino, para poder diferenciarlos (“nd ref= *numero nodo*”) y dentro de ellos hay un

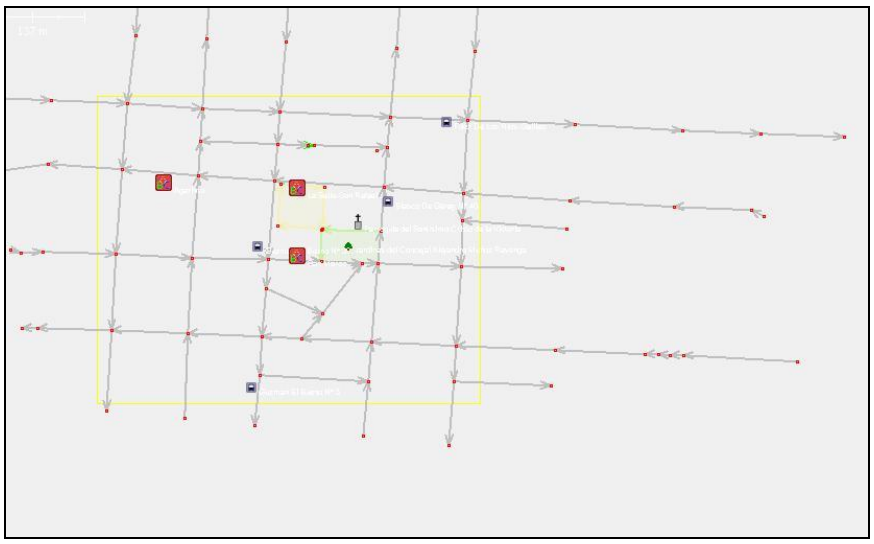
listado con los nombres de los nodos que componen este camino. Trasladándonos al mundo de los grafos, estos seria como si varios nodos compartieran la misma arista. Para identificarlo más claro son diferentes puntos de una misma calle, carretera, etc. Tener en cuenta que un mismo nodo puede pertenecer a varios caminos, por ejemplo el cruce entre una calle A y una calle B, ese mismo punto tanto pertenece a la calle A como a la calle B. Esto nos es de gran utilidad para el desarrollo de la aplicación.

```
<way id="4314415" visible="true" timestamp="2007-02-25T14:17:22+00:00"
user="Quico">

   <nd ref="26153007"/>

   <nd ref="26153008"/>

   <tag k="highway" v="steps"/>

   <tag k="name" v="Calle de Abdón Terradas"/>

   <tag k="created_by" v="JOSM"/>

 </way>
```

Las etiquetas que hemos mencionado y sobre las que hemos hecho mayor hincapié son las que afectan y se usan en la aplicación desarrollada, además de estas existe múltiple información sobre los lugares que se puede trabajar, como líneas de metro, nombre de calle. De todas maneras, si se está interesado en otro tipo de proyectos, el significado de todas las etiquetas lo podemos encontrar en la siguiente Web: “ http://wiki.openstreetmap.org/wiki/Map_Features”.

Una cosa importante a comentar sobre la información que nos proporciona el XML es que pueden aparecer nodos que estén fuera del rango de máximas, mínimas latitudes y longitudes. Esto parece contradictorio con el hecho de que ya tenemos un cuadro limitado que es lo que nos indica el cuadro que limita las máximas y mínimas latitudes y longitudes. Pero lo único que pasa con estos nodos que están fuera del rango es que son de algún camino que aparece en nuestro mapa y el propio camino continua fuera de éste. Para nuestro caso, estos nodos no tienen ninguna relevancia, por eso para poder trabajar correctamente hay que ignorarlos durante la ejecución del algoritmo.

La siguiente imagen sirve de ejemplo a lo anteriormente explicado, las máximas latitudes y longitudes son las que definen el cuadro amarillo, todo lo que está delimitado por este cuadro es lo que se va a usar, sin embargo se puede ver que fuera de éste hay también puntos rojos, que son nodos, que nosotros debemos despreciar, y como se aprecia son nodos de caminos que están dentro del cuadro y continúan también fuera. Es decir en el archivo aparecerán todos los nodos del camino que se encuentre del cuadro, y nosotros nos encargamos de seleccionar los que nos interesan y los que no.



Sabiendo que estructura tiene el fichero y las partes mas importantes para nosotros explicaremos el paso de dicho fichero a una imagen, y como hacer para que aparezcan correctamente los puntos sobre la imagen en nuestros juegos.

5.2 Transformación de la información de texto a imagen

Cuando hablamos de nuestro proyecto no podemos pasar por alto una de las partes más importantes de nuestra aplicación, la cual tiene que ver con la obtención del archivo XML (*Extensible Markup Language*, lenguaje de marcas en castellano), para poder ser tratado por el algoritmo de nuestra la aplicación, y el archivo PNG (Portable Network Graphics, gráficos de red portables en castellano), que correspondería con la imagen del mapa referenciada por el archivo XML anteriormente citado.

En el principio del estudio de las medidas a tomar para poder llegar a conseguir dichos objetivos teníamos bastante claro que la obtención del archivo XML debía ser a través del completo API que nos ofrecía OpenStreetMap ya que, como hemos comentado anteriormente, tienen puesto a disposición de los internautas un sencillo tutorial en su página web que facilita la comprensión del mismo.

Para ello sólo nos hacía falta conectarnos con la página web que hace referencia el API de OpenStreetMap introduciendo los valores de las coordenadas superiores e inferiores que delimitan al XML del terreno del cual se quiere obtener la información.

```
www.openstreetmap.com/api/0.5/map?bbox=left,bottom,right,top
```

Sabiendo que los valores que debemos introducir son *left* (es la longitud más al oeste del mapa), *bottom* (es la latitud más al sur del mapa), *right* (es la longitud de más al este del mapa) y *top* (es la latitud más al norte del mapa).

Tenemos que dejar claro que la petición al API debe ser razonable sino queremos que su respuesta sea errónea, es decir, los datos de la petición deben ser coherentes y siempre dentro de unas normas, por ejemplo, no podemos seleccionar un archivo XML

correspondiente a un mapa cuya latitud máxima y mínima sea la misma porque de ese modo no se podría mostrar el mapa, sería una petición un tanto absurda.

Con ello obtendremos directamente el contenido del archivo XML que utilizaremos como posteriormente comentaremos en la memoria.

Por otro lado, la obtención de la imagen del mapa (PNG) correspondiente al archivo XML quedaba menos clara y se abrieron desde el principio dos claras vías de trabajo para conseguirlo.

La primera alternativa de cómo obtener el archivo imagen del mapa era descargarlo directamente a través de la red por medio de algún servidor de mapas. Esta idea surgió al ver la forma de trabajar del programa JOSM, ofrecido por la iniciativa OpenStreetMap, que es el cauce para la creación y modificación de los mapas almacenados en la base de datos de la iniciativa por parte de los colaboradores anónimos de la red. Para ello se pasó a la fase de adquisición de información sobre los servidores de mapas disponibles en la red, pensando siempre en la compatibilidad de los mismos con la idea de nuestro proyecto. La mayoría de las propuestas encontradas a través de la red fueron finalmente rechazadas por sus grandes diferencias con lo que verdaderamente nosotros queríamos para nuestro proyecto. Por un lado, había candidatos que nos ofrecían mapas con gran cantidad de información que nuestra aplicación no necesitaba y cargaba enormemente el tráfico de nuestra aplicación con Internet o por el contrario escaseaban de esta. En muchas ocasiones nuestro problema fue que alguno de los servidores de mapas hallada por la red no dispensaba de forma gratuita estos, estos se rechazaron directamente.

La segunda alternativa, la que finalmente se ha implementado, nació gracias a la búsqueda en las páginas de consulta para desarrolladores de la iniciativa OpenStreetMap. En esta página aconsejaban este método y explicaban el mismo, con alguna otra alternativa según el sistema operativo o los programas a utilizar, paso a paso para poder llevarlo a buen fin.

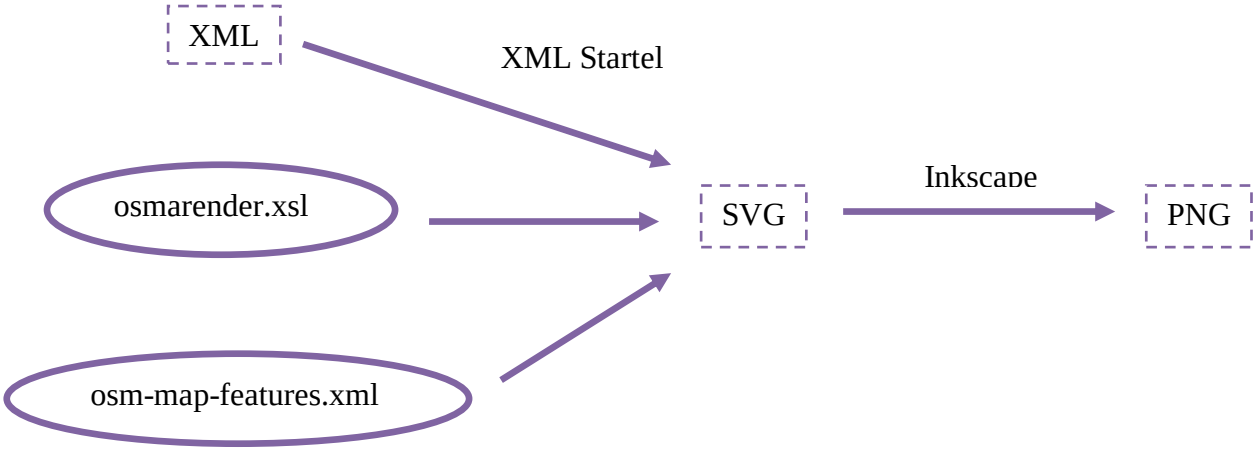
En nuestra aplicación se desarrolla una de estas versiones explicada en la página de desarrolladores de OpenStreetMap que consiste de forma global en la obtención del archivo imagen, PNG, a partir del archivo XML correspondiente al mismo utilizando para ello hojas de estilo y un programa ofrecido por la iniciativa.

Una vez obtenido el archivo XML a través del API, como se ha comentado a lo largo de esta explicación, se procede a crear un archivo de tipo SVG (Scalable Vector Graphics) que sirva de nexo entre el archivo XML y el archivo PNG, para ello se procede a utilizar el programa *XML Startel* proporcionado por la página de desarrolladores. Este programa crea un archivo SVG a partir del ya obtenido archivo XML con la ayuda de una hoja de estilo (*osmarender.xsl*) y un archivo que contiene unas reglas propias de OpenStreetMap (osm-map-features-z17.xml). Estas reglas observan la forma del archivo XML y según esta dan un formato u otro al archivo resultante. Se quiere obtener este tipo de fichero ya que en su esencia es un vector de gráficos, una imagen.

Después de haber utilizado el programa citado para conseguir el archivo SVG, como paso final, transformamos el archivo SVG a PNG. El motivo de este paso es poder pasar el fichero a un tipo de archivo compatible con nuestra aplicación y de este modo poder

hacer uso de él. Este proceso se realiza mediante la aplicación aconsejada por OpenStreetMap, *Inkscape*, estando especializada en dichas actuaciones. Dado que el applet de nuestro proyecto tiene unas dimensiones características a la vez de la transformación anterior se procede a modificar su tamaño, dejando este en 580x530 pixeles. Este dato es muy importante de cara al calibrado de las coordenadas UTM dentro del marco de los pixeles.

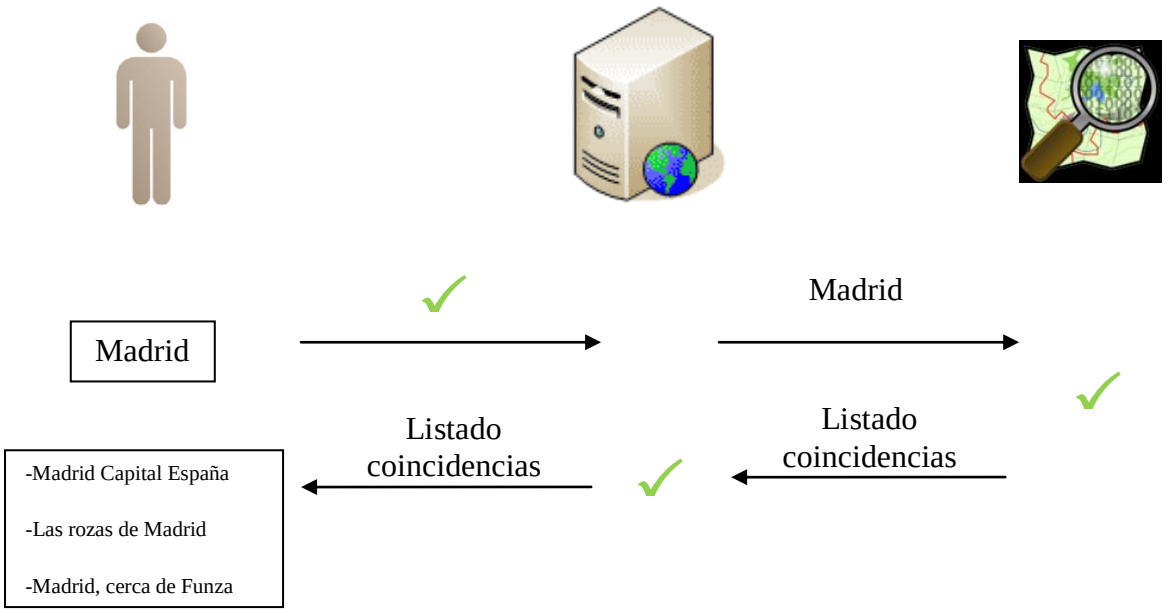
A continuación un pequeño esquema de este proceso.



Todo este proceso se encuentra implementado dentro de nuestro proyecto en forma de web service dentro de nuestro gran módulo de la aplicación servidor, que como se ha comentado anteriormente, es una aplicación colgada en una web que resuelve peticiones y ofrece diversos servicios a un cliente dado.

El servicio web que se encuentra diseñado en nuestro proyecto oferta dos tipos de servicios: uno que obtiene un archivo XML desde la web y otro que busca un nuevo archivo XML en la web y convierte este en una imagen apta para utilizarla.

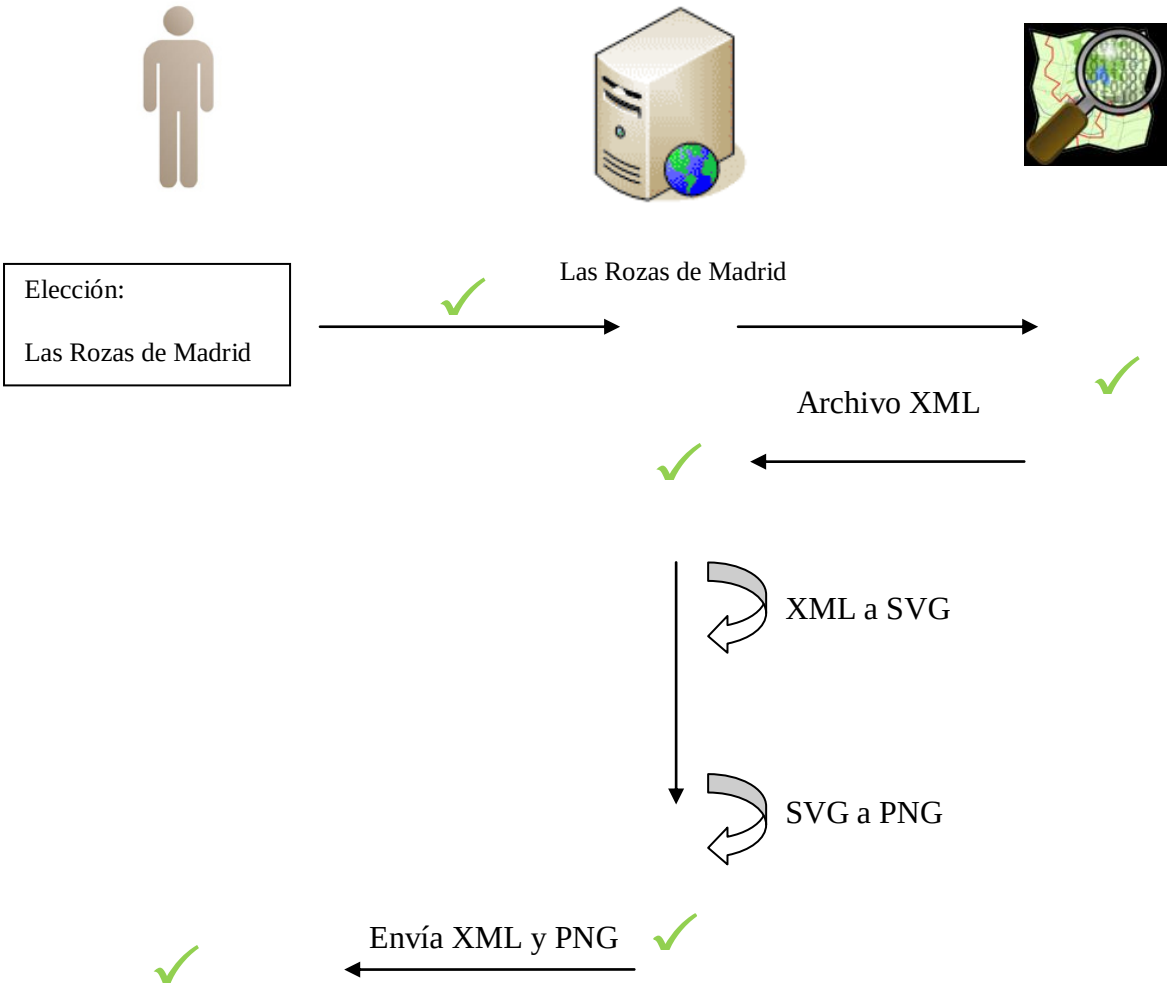
El primero recibe una petición del cliente junto con una cadena de caracteres. Esta cadena de caracteres representa un lugar arbitrario elegido por la conciencia del usuario. Al recibir esta petición, nuestro servicio web contacta con una página web propiedad de OpenStreetMap consiguiendo en un archivo XML un listado de los posibles lugares, con las coordenadas UTM de su localización, que coinciden con la cadena de caracteres introducida. Una vez realizado este proceso, el servicio web devuelve el listado de los posibles coincidencias con la cadena de caracteres introducida. Expresaremos lo explicado con un ejemplo.



El segundo servicio que oferta nuestro web service es el que recibe la elección de una de las coincidencias devuelta por el servicio web y ejecuta el proceso que anteriormente se ha explicado, es decir, se comunica con OpenStreetMap para conseguir el archivo XML del lugar exacto requerido por el usuario, transforma este a un archivo SVG y finalmente convierte este último en un PNG.

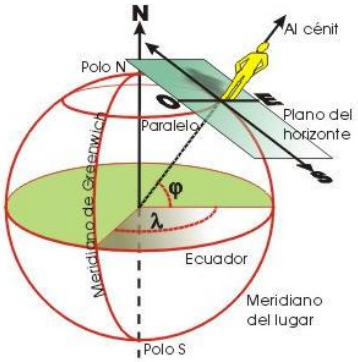
Todo este proceso se realiza dentro de nuestra aplicación Java utilizando la clase Process, la cual es un hilo que ejecuta programas externos a Java. Es necesario el uso de hilos en este proceso al ser necesaria la sincronización dado que el orden de ejecución de los mismos está directamente relacionado con la terminación exitosa del proceso.

Para que resulte más visual, se ilustrará mediante un dibujo esquemático el proceso aquí descrito.



5.3 Calibrado de mapas

Los mapas de *OpenStreetMap* que utiliza nuestra aplicación están caracterizados por un archivo XML lleno de información en forma de nodos. Cada nodo viene descrito, además de por otras características, por su posición en forma de coordenada geográfica o *UTM* relativa al globo terráqueo. Éstas determinan la posición exacta en la superficie terrestre utilizando las dos coordenadas angulares de un sistema de coordenadas esféricas que está alineado con el eje de rotación de la Tierra:



Latitud

Mide el ángulo entre cualquier punto y el ecuador. Las líneas de

latitud se llaman paralelos y son círculos paralelos al ecuador en la superficie de la Tierra.

Longitud

Mide el ángulo a lo largo del ecuador desde cualquier punto de la Tierra. Se acepta que Greenwich en Londres es la longitud 0 en la mayoría de las sociedades modernas. Las líneas de longitud son círculos máximos que pasan por los polos y se llaman meridianos.

En nuestro proyecto utilizamos esta información para podernos desplazar por el mapa según desee el usuario, haciendo coincidir la coordenada geográfica con la posición del plano que corresponde a ésta. El problema radica en que tenemos que transformar las coordenadas geográficas que corresponden con la localización en la tierra en coordenadas planas que localizan el punto en el mapa.

Este cálculo se consigue realizando rigurosos cálculos matemáticos que tienen que ver con el ángulo que se forma con el punto en la superficie de la Tierra (φ) y la distancia al centro de la tierra (λ), como se puede ver en la figura.

Para poder hacer los cálculos necesitamos primero coordenadas de las esquinas del plano. Lo tenemos en forma de atributos de tipo *coordenada*, teniendo coordenada la latitud y longitud.

```
private Coordenada supizdaUTM;

private Coordenada supdchaUTM;

private Coordenada infizdaUTM;

private Coordenada infdchaUTM;
```

Además necesitamos el tamaño de la imagen donde tenemos alojado el plano. Lo tendremos como atributos que marquen el tamaño horizontal y vertical de la imagen.

```
private Pixel tamanyoImagen;
```

Para nosotros, las coordenadas planas son los pixeles de la imagen donde está alojado el plano. Se debe tener en cuenta que la posición (0,0) se encuentra en la esquina superior izquierda de la imagen, siendo la esquina inferior derecha de la imagen la tupla (pixelHorizontal,pixelVertical).

Ahora para poder calcular partiendo de unas coordenadas geográficas las coordenadas planas debemos averiguar un factor que relacione ambas. Esto se consigue realizando un cálculo que haga de proporción entre ambas.

Para poder transformar coordenadas geográficas a coordenadas planas realizamos los siguientes cálculos:

```
/*Función que pasa unas coordenadas UTM a pixel*/
```

```
public Pixel UTMaPixel(double lon, double lat) {

    int px, py;

    px = (int) ((Math.abs(supizdaUTM.getLongitud() - lon)) / pixelHorizontal);

    py = (int) ((Math.abs(supizdaUTM.getLatitud() - lat)) / pixelVertical);

    return (new Pixel(px, py));

}
```

La operación inversa, la que transforma coordenadas planas a coordenadas geográficas se escribe:

```
public Coordenada PixelaUTM(int puntox, int puntoy){

  int lat, lon;

  lon = (int)((double)supizdaUTM.getLongitud() + pixelHorizontal *
(double)puntox);

  lat = (int)((double)supizdaUTM.getLatitud() - pixelVertical * (double)puntoy);

  return (new Coordenada(lat,lon));

}
```

6. Algoritmo

El algoritmo es la columna vertebral de nuestro proyecto, en lo que se basado la mayoría de nuestros esfuerzos. Claro está que ha sido lo más complejo de realizar y lo que se ha llevado la mayor parte del tiempo invertido en la aplicación.

Intentaremos explicar en este párrafo de forma rápida y clara una idea general del algoritmo. El algoritmo está preparado y funciona para recibir un fichero obtenido de OpenStreetMaps, que el mismo procesa, y mediante un tratamiento computacional hace que puedas realizar desplazamientos de un punto a otro cumpliendo ciertas restricciones. Entre estas restricciones destacar algunas importantes en este momento. Una de ellas y que parece obvia, es que el desplazamiento suceda a otro punto dentro de tu mismo camino, y que no vaya saltando de un lado a otro del mapa sin ningún sentido. Añadir que el punto objetivo del movimiento sea un punto dentro de la visión actual y no fuera de ésta (la visión quedará definida detalladamente posteriormente). En resumidas cuentas, poder desplazarse por un mapa de forma correcta, siguiendo ciertos criterios.

A continuación nos enfrentamos a relatar de forma más precisa y técnica los avatares del algoritmo.

Nuestro algoritmo como medios de almacenamiento tendrá dos hashmap y un arraylist para almacenar nodos y las calles. Los hashmap van a contener en cada tupla un par clave, valor. Uno de los hashmap, llamado tabla de nodos, contendrá como clave la

coordenada de un nodo y como valor de esa coordenada el identificador del nodo. La otra tabla, llamada tabla de calles, contendrá como clave el identificador del camino y como valor una lista (arraylist) con los identificadores de nodo que componen ese camino. Como se ha explicado en el apartado de XML solo se introducen nodos que estén dentro de rango del rectángulo delimitado por las máximas y mínimas coordenadas, los otros se ignoran.

```
this.tablaNodos = new HashMap();

this.tablaCalles = new HashMap();
```

El fichero se lee utilizando un objeto de la clase de java "Document", pasándole la ruta en la cual está el fichero, la forma de obtenerlo es externa a este apartado, en este punto solo nos preocuparemos de facilitar una ruta correcta de un archivo ".OSM", bien declarado.

El primer paso a realizar es hacer un escaneo del fichero para obtener de este toda la información que necesitamos, el fichero se lee una única vez y con ello rellenamos todo lo necesario para ejecutar nuestro algoritmo. Para el recorrido del XML utilizaremos objetos de clases que nos proporcionan las bibliotecas de java: Element, NodeList, NamedNodeMap.

Se crea un objeto Element con el objeto Document anterior, y en este tenemos el archivo completo, nodo raíz. Luego cada etiqueta se recoge en un NodeList y de cada NodeList (etiqueta) extraemos los campos que nos conviene diferenciándolos por su identificador. Para entenderlo mas fácilmente saber que esto tiene estructura jerárquica en forma de árbol.

```
Element elemento = (Element) this.doc.getDocumentElement();
```

La primera información que vamos a obtener va a ser de las coordenadas que definen el rectángulo de nuestro mapa. Se nos proporciona la máxima latitud, máxima longitud, mínima latitud y mínima longitud, que nosotros almacenaremos en cuatro respectivas variables. Extraemos la etiqueta "bounds" y analizamos sus campos:

```
NodeList listaCoord = elemento.getElementsByTagName("bounds");

 NamedNodeMap listaAtributosCoord = listaCoord.item(0).getAttributes();

 minlat = listaAtributosCoord.getNamedItem("minlat").getNodeValue();

minlon = listaAtributosCoord.getNamedItem("minlon").getNodeValue();

maxlat = listaAtributosCoord.getNamedItem("maxlat").getNodeValue();

maxlon = listaAtributosCoord.getNamedItem("maxlon").getNodeValue();
```

A continuación analizaríamos todos los nodos que aparecen en el mapa, identificados por las etiquetas "node", extrayendo uno por uno.

```
NodeList listaNodos = elemento.getElementsByTagName("node");
```

De los nodos extraeremos lo único que nos interesa, su coordenada y su número de identificador. Haremos una comprobación si dicho nodo está dentro del rango que nos interesa y lo introduciremos en la tabla de nodos. La comprobación de que esté dentro del rango no es más que comprobar dicha coordenada con las máximas y mínimas coordenadas.

```
NamedNodeMap listaAtributos;

        Coordenada coord;

        String lat, lon;

        for (int i = 0; i < listaNodos.getLength(); i++) {

            listaAtributos = listaNodos.item(i).getAttributes();

            lat = listaAtributos.getNamedItem("lat").getNodeValue();

            lon = listaAtributos.getNamedItem("lon").getNodeValue();

            coord = new Coordenada(Double.valueOf(lat).doubleValue(),
Double.valueOf(lon).doubleValue());

            if (!fueraRango(coord)) {

                this.tablaNodos.put(coord,
listaAtributos.getNamedItem("id").getNodeValue());

                this.Ptos.add(coord);

            }

        }
```

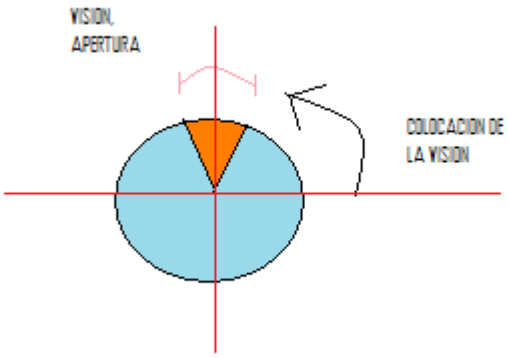
Luego introducimos las calles con igual procedimiento que el anterior, pero eso si cambiando el nombre de las etiquetas.

```
        Integer f;

        NamedNodeMap nm;

        NodeList nl;

        Element e;

        String idCalle;
```

```
        ArrayList aux;
        for (int i = 0; i < listaNodos.getLength(); i++) {
            e = (Element) listaNodos.item(i).getChildNodes();
            idCalle =
listaNodos.item(i).getAttributes().getNamedItem("id").getNodeValue();
            nl = e.getElementsByTagName("nd");//

            for (int j = 0; j < nl.getLength(); j++) {
                //PondrÃƒÂ¡ todos los valore a true porque todos estarÃƒÂ¡n.
                nm = nl.item(j).getAttributes();
                //f es el identificador del nodo no de la calle
                f = Integer.valueOf(nm.getNamedItem("ref").getNodeValue());
                if (tablaNodos.containsValue(f.toString())) { //sacar le id del nodo

                    if (tablaCalles.containsKey(f)) {
                        aux = (ArrayList) tablaCalles.get(f);
                        aux = this.copiar((ArrayList) tablaCalles.get(f));
                        aux.add(new Integer(Integer.valueOf(idCalle)));
                        this.tablaCalles.put(f.intValue(), aux);
                    } else {
                        ArrayList aux2 = new ArrayList();
                        aux2.add(new Integer(Integer.valueOf(idCalle)));
                        this.tablaCalles.put(f.intValue(), aux2);
                    }
                }//if
            }//for
        }
```

En este primero paso ya tenemos lo imprescindible para poder usar y movernos a través del mapa.

A partir de aquí dispondremos de un nodo A, que será del cual partamos. Obtenido bien de forma aleatoria o elegido explícitamente de entre los disponibles. Hay que tener en cuenta que nuestro jugador que se moverá por el mapa va a disponer de la coordenada en la que se encuentra, su posición, y una visión que va a venir dada por una medida en grados. El personaje podrá ver únicamente lo que entre dentro de ese rango de visión prolongándolo a través de sus líneas correspondientes. La visión puede estar colocada en 0 grados como que puede estar en otros grados ubicada. Por ejemplo a 10 grados, 20. La siguiente imagen sirve a modo de explicación y para que se entienda de mejor modo:

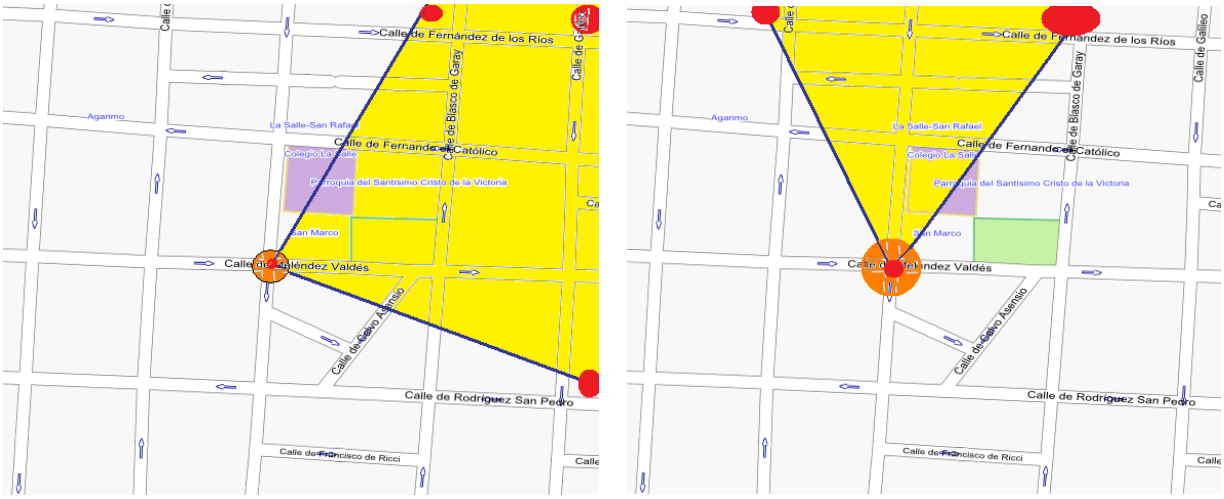


Entonces el jugador desde su posición, coordenada, podrá pedir si hay un punto al cual se pueda desplazar y que esté dentro de su visión, con inclinación de la visión de la que disponga en ese momento. El jugador pediría el punto, se comprobaría qué puntos hay dentro de esa visión, qué puntos son candidatos a ser su próxima posición. De estos se descartarían los que no están en su propio camino o calle, y de los restantes, los que están en su calle, nos quedaríamos con el que se encuentra más cerca de su posición. Con una visión pequeña tendremos mayor precisión en nuestros movimientos. Esto ha sido una forma de explicarlo para entender el concepto general, ahora profundicemos más en este tema.

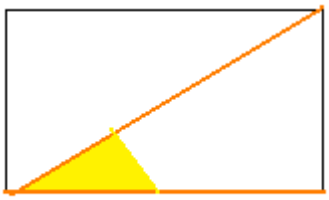
A partir del jugador A que tenemos, pediríamos su siguiente posición o coordenada, teniendo en cuenta la visión de ese nodo, la inclinación en la que está colocada esa visión y la posición, coordenada, del nodo actual. Comprobaremos si hay algún punto que pueda ser alcanzado desde nuestra posición y con nuestras características anteriormente descritas. Se miraría cada nodo si se cumple o no.

Para ello cogemos el nodo actual, obtenemos los puntos de un polígono sobre el mapa, ese polígono es la prolongación de las líneas de visión hacia las paredes del rectángulo que define el mapa. Con esto hemos definido todo lo que puede ver nuestro jugador desde una posición, todos los puntos que ve. En el dibujo se puede apreciar que con una visión grande, existe un dibujo para poder apreciarlo mejor, todo lo que está en la zona pintada de amarillo son los puntos candidatos a ser accedidos, de éstos aún habría que descartar otros nodos. Los puntos rojos que aparecen en la figura son los puntos que nosotros calculamos para construir nuestro polígono. Un detalle al calcular los puntos es

que no es tener en cuenta que la prolongación de las líneas visión pueden tocar una única pared o bien tocar en dos, ya que limitamos que la visión sea mayor de 45 grados, en el caso de que toque en una única pared habrá tres puntos (el propio, y ambos de la misma pared), y en el caso en que toque en dos paredes, tendremos los puntos anteriores mas el punto de la esquina entre ambas paredes. En estas dos imágenes podemos apreciar lo explicado anteriormente.



La forma de obtener los puntos, marcados en el dibujo como rojos, es bastante complicada y requiere de cálculos geométrico. Se debe diferenciar entre diferentes casos para llevar a cabo el cálculo de dichos puntos. Antes de diferenciar estos casos ,calcularemos el ángulo que se forma entre la recta inferior, pared inferior del rectángulo que es lo que nosotros tomamos como referencia para todo, como eje X de coordenadas cartesianas , y la recta que va desde una esquina inferior a la esquina superior contraria:



A este ángulo le llamaremos “anguloaux”. La forma de calcular este ángulo es calcular la inclinación de una recta respecto al eje X de coordenadas que como hemos dicho anteriormente es la pared inferior del rectángulo. Únicamente nos hace el segmento del que queremos conocer su inclinación. Facilitamos los puntos extremos del segmento y se realiza lo siguiente: calculamos la distancia entre ambos puntos, eso sería la hipotenusa de un triangulo, las longitudes de los otros dos segmentos por medio de restas conociendo la longitud y latitud máximas y mínimas las tenemos. Por tanto no es más que realizar el cálculo geométrico correspondiente:

```
arcoCos(a)=cateto opuesto/hipotenusa
```

Una vez obtenido esto pasamos a obtener los puntos. Todos los casos van a tener en común que el punto en el que se encuentran es un punto del futuro polígono por tanto este punto le añadimos. Ahora para empezaremos a distinguir los casos para añadir unos puntos u otros.

Para empezar a meternos en materia con la introducción de puntos para el polígono "ficticio" definimos ciertas variables, entre ellas importantes son: beta, almacena la inclinación de la visión en grados; *bv*, es la suma del valor de la variable anterior más el valor de la propia visión en sí; *auxgrados*, es el ángulo aux pasado a grados; *angContRect*, es la diferencia entre 2π radianes y el ángulo aux. También hay otras variables beta grados y bvgrados que son beta y bv pasadas a grados.

En el método "puntos de comprobación" añadimos los puntos de los que se va a componer nuestro polígono, añadiendo claro está el punto actual, es decir que "puntos comprobación" ya tiene la posición del actual cuando entra en la siguiente diferencia de casos.

Iniciemos la distinción de casos. Haremos un dibujo explicativo por cada caso considerando explicando que puntos hay que introducir. Fijarse que en todos los casos el plano queda dividido por dos ejes que son similares a los ejes de coordenadas cartesianas de dos dimensiones.

El esquema de la distinción de casos se basa en las líneas de prolongación, es el siguiente:

- Ambas líneas tocan un único lado
  - Ambas líneas tocan la parte derecha del mapa
  - Ambas líneas tocan la parte superior del mapa
  - Ambas líneas tocan la parte izquierda del mapa
  - Ambas líneas tocan la parte inferior del mapa

- Ambas líneas tocan dos lados diferentes
  - Una de las líneas choca con la pared derecha del mapa y la otra con la pared superior
  - Una de las líneas choca con la pared superior del mapa y la otra con la pared izquierda
  - Una de las líneas choca con la pared izquierda del mapa y la otra con la pared inferior
  - Una de las líneas choca con la pared inferior del mapa y la otra con la pared derecha

Antes de la explicación detallada de la distinción de casos, destacar que todos añaden al polígono la coordenada actual.

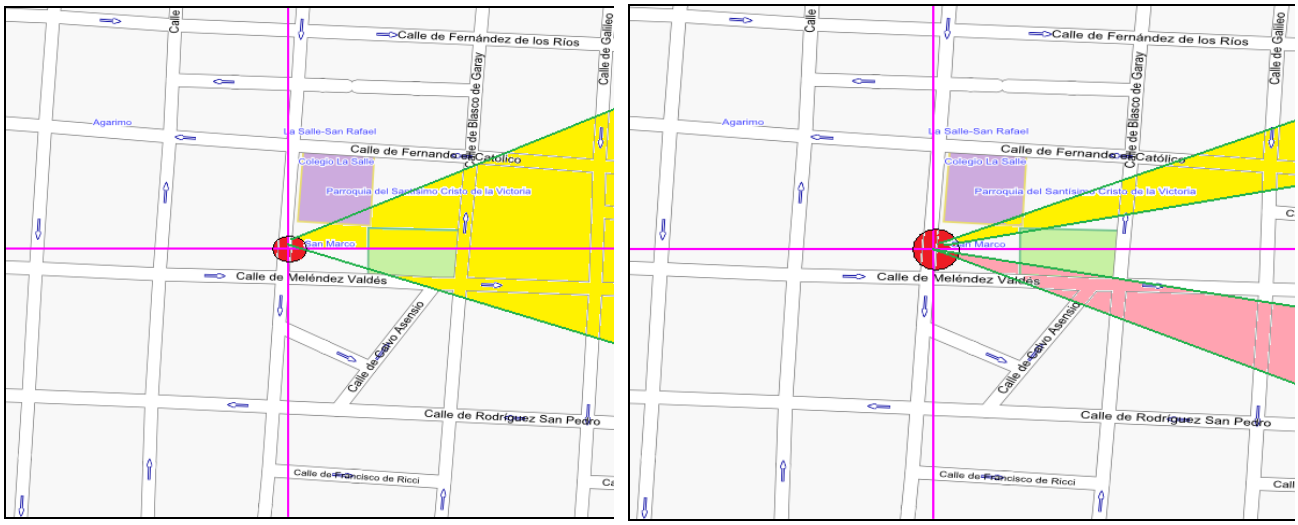
```
puntosComprobacion.add(coordActual);
```

- TOCAN UN UNICO LADO

  - PARED DERECHA DEL MAPA

En este caso podemos tener más casos que tienen en común que tocan en el lado derecho de la pared del plano. Dentro de aquí se distinguen 3 casos. Un caso que las 2 líneas prolongadas toquen por encima del eje de cartesianas horizontal, es decir que

ambas líneas estén por encima de los cero grados siempre tocando ambas en la pared derecha. El caso contrario sería que ambas líneas estén por debajo de los cero grados, y el último caso es que una de ellas este por encima de los ceros grados y la otra por debajo.

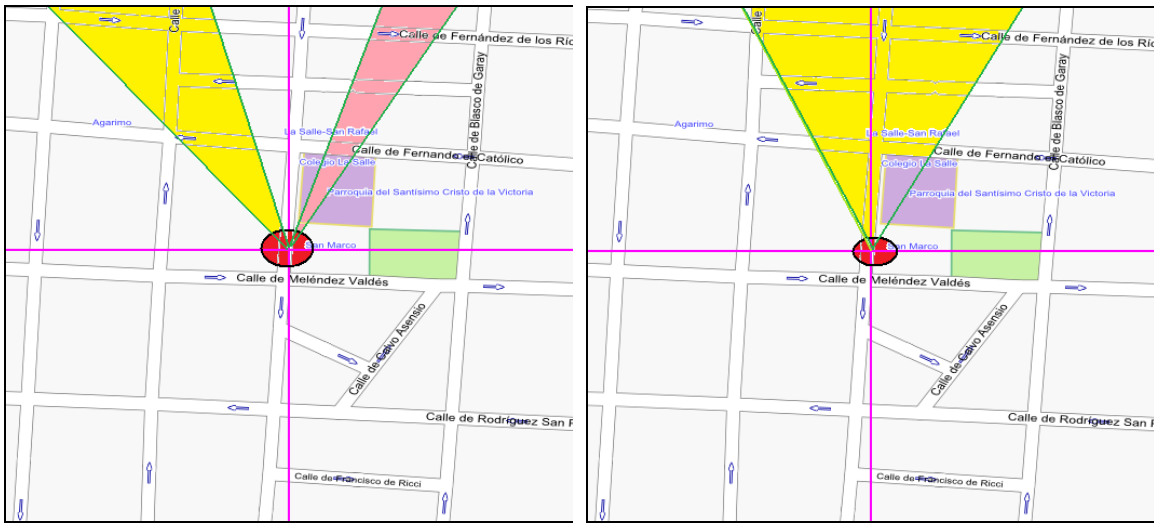



```
//Corresponde con la  "<"

        if ((((beta >= (2 * Math.PI - anguloAux)) && (beta <= (2 * Math.PI))) ||
((beta >= 0) && (beta <= anguloAux))) && (((bv >= (2 * Math.PI - anguloAux))
&& (bv <= (2 * Math.PI))) || ((bv >= 0) && (bv <= anguloAux)))) {

            //toca los doscon lo de la dcha

            //para beta

            if ((beta >= (2 * Math.PI - anguloAux)) && (beta <= (2 * Math.PI))) {

                //Calculo el triangulo que contiene BETA

                cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.maximaLon));

                cateOpu = Math.tan((2 * Math.PI) - beta) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordA = new Coordenada(coordActual.getLatitud() - cateOpu,
this.maximaLon);

                this.puntosComprobacion.add(coordA);

            } else {
```

```
                //Calculo el triangulo que contiene BETA

                cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.maximaLon));

                cateOpu = Math.tan(beta) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordA = new Coordenada(coordActual.getLatitud() + cateOpu,
this.maximaLon);

                puntosComprobacion.add(coordA);

            }

            if ((bv >= (2 * Math.PI - anguloAux)) && (bv <= (2 * Math.PI))) {

                cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.maximaLon));

                cateOpu = Math.tan((2 * Math.PI) - bv) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(coordActual.getLatitud() - cateOpu,
this.maximaLon);

                this.puntosComprobacion.add(coordB);

            } else {

                //Calculo el triangulo que contiene BETA

                cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.maximaLon));

                cateOpu = Math.tan(bv) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(coordActual.getLatitud() + cateOpu,
this.maximaLon);

                this.puntosComprobacion.add(coordB);

            }

        }
```

- PARED SUPERIOR DEL MAPA

Como el caso anterior también se van a dividir en tres casos que comparten que ambas líneas tocan en la pared superior. En este caso ambas líneas están entre 45 y 90 grados, en el primer cuadrante. El segundo caso sería que ambas estén entre 90 grados y 135. Y el tercer caso que una línea este entre 45 y 90 grados y la otra este entre 90 y 135 grados.

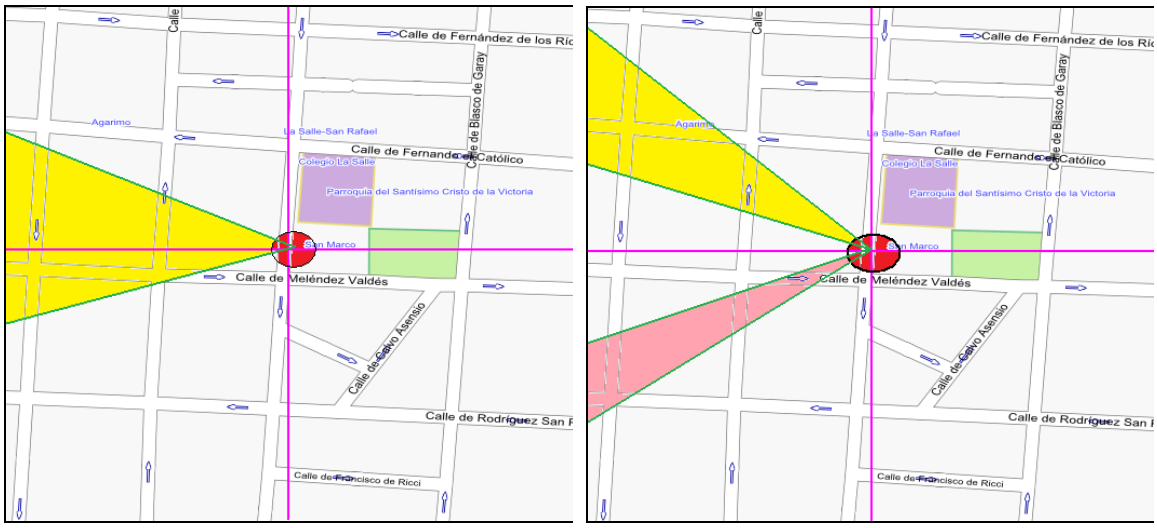



```
else if (((beta >= anguloAux) && (beta <= (Math.PI - anguloAux))) && (((beta + this.vision) >= anguloAux) && ((beta + this.vision) <= (Math.PI - anguloAux)))) {//toca los doscon lo de la arriba

            //para beta



            if (beta <= Math.PI / 2) {

                cateCont = u.distancia(coordActual, new Coordenada(this.maximaLat, coordActual.getLongitud()));

                cateOpu = Math.tan((Math.PI / 2) - beta) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordA = new Coordenada(this.maximaLat, coordActual.getLongitud() + cateOpu);

                this.puntosComprobacion.add(coordA);

            } else {

                cateCont = u.distancia(coordActual, new Coordenada(this.maximaLat, coordActual.getLongitud()));

                cateOpu = Math.tan(beta - (Math.PI / 2)) * cateCont;
```

```
                //Hemos hallado el punto donde la recta corta con una frontera.

                coordA = new Coordenada(this.maximaLat, coordActual.getLongitud()
- cateOpu);

                this.puntosComprobacion.add(coordA);

            }

            //para beta +vision

            if (bv <= Math.PI / 2) {

                cateCont = u.distancia(coordActual, new Coordenada(this.maximaLat,
coordActual.getLongitud()));

                cateOpu = Math.tan((Math.PI / 2) - bv) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(this.maximaLat, coordActual.getLongitud()
+ cateOpu);

                this.puntosComprobacion.add(coordB);

            } else {

                cateCont = u.distancia(coordActual, new Coordenada(this.maximaLat,
coordActual.getLongitud()));

                cateOpu = Math.tan(bv - (Math.PI / 2)) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(this.maximaLat, coordActual.getLongitud()
- cateOpu);

                this.puntosComprobacion.add(coordB);

            }

        }
```

- PARED IZQUIERDA DEL MAPA

Caso en el que ambas líneas choquen con la pared izquierda. Tres casos igualmente uno en que las dos líneas estén entre 135 y 180 grados. Otro en que las dos líneas estén entre 180 y 225 grados y otro caso en que una línea sea mayor que 135 grados y menor que 180 grados y la otra este entre 180 y 225 grados.

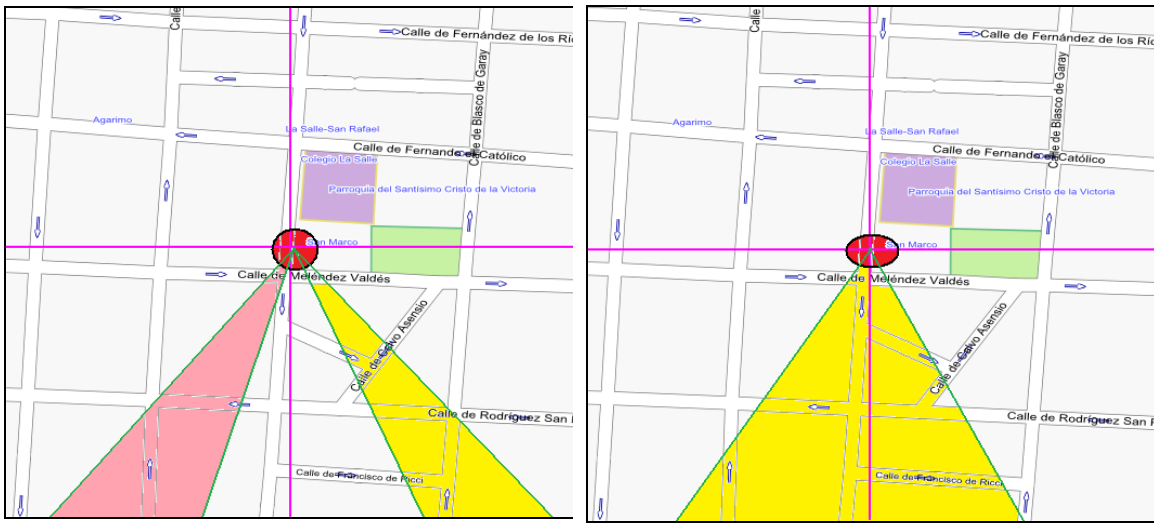



```
    else if (((beta >= Math.PI - anguloAux) && (beta <= Math.PI + anguloAux)) &&
(((beta + this.vision) >= Math.PI - anguloAux) && ((beta + this.vision) <= Math.PI
+ anguloAux))) {

            //toca los doscon lo de la izq

            if (beta <= Math.PI) {

                cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.minimaLon));

                cateOpu = Math.tan((Math.PI) - beta) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordA = new Coordenada(coordActual.getLatitud() + cateOpu,
this.minimaLon);

                this.puntosComprobacion.add(coordA);

            } else {

                cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.minimaLon));

                cateOpu = Math.tan(beta - (Math.PI)) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordA = new Coordenada(coordActual.getLatitud() - cateOpu,
this.minimaLon);

                this.puntosComprobacion.add(coordA);

            }
```

```
            //para beta +vision

            if (bv <= Math.PI) {

                cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.minimaLon));

                cateOpu = Math.tan((Math.PI) - bv) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(coordActual.getLatitud() + cateOpu,
this.minimaLon);

                this.puntosComprobacion.add(coordB);

            } else {

                cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.minimaLon));

                cateOpu = Math.tan(bv - (Math.PI)) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(coordActual.getLatitud() - cateOpu,
this.minimaLon);

                this.puntosComprobacion.add(coordB);

            }

        }
```

- PARED INFERIOR DEL MAPA

Ambas líneas tocan con la pared inferior. Es posible que ambas líneas se encuentren entre 225 grados y 270, o entre 270 y 315 o bien que una línea se encuentre entre 225 grados y la otra entre 270 y 315 grados.



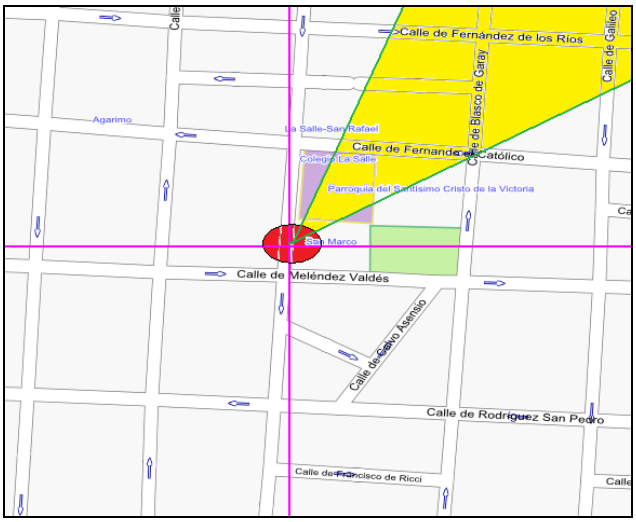

```
//Corresponde con "/\"

        else if (((beta >= Math.PI + anguloAux) && (beta <= (3 * (Math.PI / 2)) + angContRect)) && (((beta + this.vision) >= Math.PI + anguloAux) && ((beta + this.vision) <= (3 * (Math.PI / 2)) + angContRect))) {

            //toca los doscon lo de la abajo

            if (beta <= (3 * Math.PI) / 2) {

                cateCont = u.distancia(coordActual, new Coordenada(this.minimaLat, coordActual.getLongitud()));

                cateOpu = Math.tan((3 * Math.PI / 2) - beta) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordA = new Coordenada(this.minimaLat, coordActual.getLongitud() - cateOpu);

                this.puntosComprobacion.add(coordA);

            } else {

                cateCont = u.distancia(coordActual, new Coordenada(this.minimaLat, coordActual.getLongitud()));
```

```
                cateOpu = Math.tan(beta - (3 * Math.PI / 2)) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordA = new Coordenada(this.minimaLat, coordActual.getLongitud()
+ cateOpu);

                this.puntosComprobacion.add(coordA);

            }

            if (bv <= (3 * Math.PI) / 2) {

                cateCont = u.distancia(coordActual, new Coordenada(this.minimaLat,
coordActual.getLongitud()));

                cateOpu = Math.tan((3 * Math.PI / 2) - bv) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(this.minimaLat, coordActual.getLongitud() -
cateOpu);

                this.puntosComprobacion.add(coordB);

            } else {

                cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.minimaLon));

                cateOpu = Math.tan(bv - (3 * Math.PI / 2)) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(this.minimaLat, coordActual.getLongitud()
+ cateOpu);

                this.puntosComprobacion.add(coordB);

            }

        }
```

- TOCAN EN DOS LADOS

  - UNA DE LAS LÍNEAS CHOCA CON LA PARED DERECHA DEL MAPA Y LA OTRA CON LA PARED SUPERIOR.

La primera línea estará entre 0 y 45 grados y la segunda línea entre 45 y 90 grados

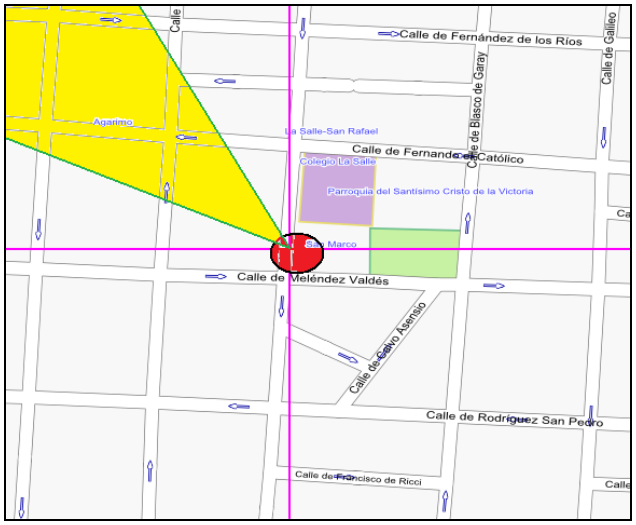



```
//Corresponde con "-/"

        //beta derecha y (beta+ vision)arriba

        else if ((((beta >= (2 * Math.PI - anguloAux)) && (beta <= (2 * Math.PI))) ||
((beta >= 0) && (beta <= anguloAux))) && (((beta + this.vision) >= anguloAux)
&& ((beta + this.vision) <= (Math.PI - anguloAux)))) {

            this.puntosComprobacion.add(new Coordenada(this.maximaLat,
this.maximaLon));

            //para beta

            if ((beta >= (2 * Math.PI - anguloAux)) && (beta <= (2 * Math.PI))) {

               //Calculo el triangulo que contiene BETA

               cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.maximaLon));

               cateOpu = Math.tan((2 * Math.PI) - beta) * cateCont;

               //Hemos hallado el punto donde la recta corta con una frontera.

               coordA = new Coordenada(coordActual.getLatitud() - cateOpu,
this.maximaLon);

               this.puntosComprobacion.add(coordA);
```

```
            } else {

                //Calculo el triangulo que contiene BETA

                cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.maximaLon));

                cateOpu = Math.tan(beta) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordA = new Coordenada(coordActual.getLatitud() + cateOpu,
this.maximaLon);

                this.puntosComprobacion.add(coordA);

            }

            //para beta +vision

            if (bv <= Math.PI / 2) {

                cateCont = u.distancia(coordActual, new Coordenada(this.maximaLat,
coordActual.getLongitud()));

                cateOpu = Math.tan((Math.PI / 2) - bv) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(this.maximaLat, coordActual.getLongitud()
+ cateOpu);

                this.puntosComprobacion.add(coordB);

            } else {

                cateCont = u.distancia(coordActual, new Coordenada(this.maximaLat,
coordActual.getLongitud()));

                cateOpu = Math.tan(bv - (Math.PI / 2)) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(this.maximaLat, coordActual.getLongitud()
- cateOpu);

                this.puntosComprobacion.add(coordB);

            }

        }
```

- UNA DE LAS LINEAS CHOCA CON LA PARED SUPERIOR DEL MAPA Y LA OTRA CON LA PARED IZQUIERDA

La primera línea estará entre 90 y 135 grados y la segunda entre 135 y 180 grados

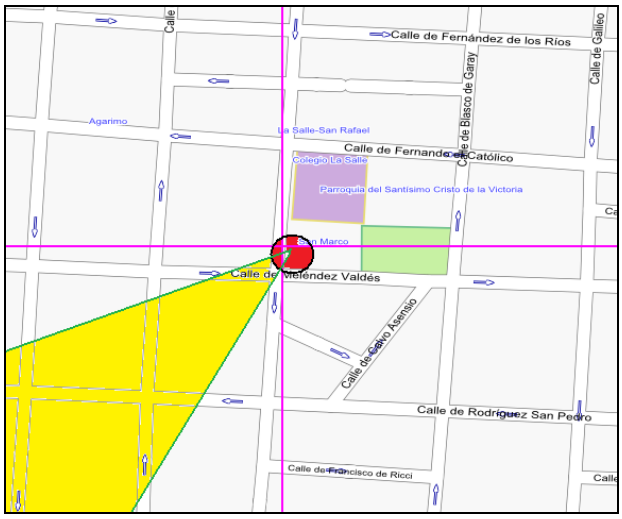



```
    //Corresponde con "/-"

    //beta toque arriba y (beta+vision)izqda

    else if(((beta>=anguloAux) && (beta<=(Math.PI-anguloAux)))&&
((bv>=Math.PI-anguloAux)&&(bv<=Math.PI+anguloAux)) )

  {

        //this.puntosComprobacion.add(coordActual);

        this.puntosComprobacion.add(new
Coordenada(this.maximaLat,this.minimaLon));

        //para beta

        if(beta<=Math.PI/2){

            cateCont= u.distancia(coordActual,new
Coordenada(this.maximaLat,coordActual.getLongitud()));

            cateOpu=Math.tan((Math.PI/2)-beta)*cateCont;

            //Hemos hallado el punto donde la recta corta con una frontera.

            coordA=new
Coordenada(this.maximaLat,coordActual.getLongitud()+cateOpu);

            this.puntosComprobacion.add(coordA);
```

```
        }

        else{

            cateCont= u.distancia(coordActual,new
Coordenada(this.maximaLat,coordActual.getLongitud()));

            cateOpu=Math.tan(beta-(Math.PI/2))*cateCont;

            //Hemos hallado el punto donde la recta corta con una frontera.

            coordA=new Coordenada(this.maximaLat,coordActual.getLongitud()-
cateOpu);

            this.puntosComprobacion.add(coordA);

       }

        //para beta +vision

        if(bv<=Math.PI){

            cateCont= u.distancia(coordActual,new
Coordenada(coordActual.getLatitud(),this.minimaLon));

            cateOpu=Math.tan((Math.PI)-bv)*cateCont;

            //Hemos hallado el punto donde la recta corta con una frontera.

            coordB=new
Coordenada(coordActual.getLatitud()+cateOpu,this.minimaLon);

            this.puntosComprobacion.add(coordB);

        }

        else{

            cateCont= u.distancia(coordActual,new
Coordenada(coordActual.getLatitud(),this.minimaLon));

            cateOpu=Math.tan(bv-(Math.PI))*cateCont;

            //Hemos hallado el punto donde la recta corta con una frontera.

            coordB=new Coordenada(coordActual.getLatitud()-
cateOpu,this.minimaLon);

            this.puntosComprobacion.add(coordB);

       } }
```

- UNA DE LAS LINEAS CHOCA CON LA PARED IZQUIERDA DEL MAPA Y LA OTRA CON LA PARED INFERIOR

Una de las líneas estará 180 y 225, la otra estará entre 225 y 270.

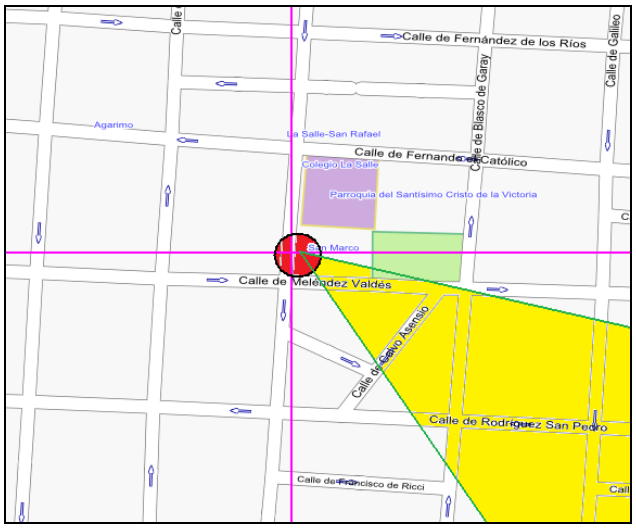



```
//Correspondiente con "-\"

        //beta toque izq y (beta+vision)abajo

        else if (((beta >= Math.PI - anguloAux) && (beta <= Math.PI + anguloAux)) && (((beta + this.vision) >= Math.PI + anguloAux) && ((beta + this.vision) <= (3 * (Math.PI / 2)) + angContRect))) {

            this.puntosComprobacion.add(new Coordenada(this.minimaLat, this.minimaLon));

            //para beta

            if (beta <= Math.PI) {

                cateCont = u.distancia(coordActual, new Coordenada(coordActual.getLatitud(), this.minimaLon));

                cateOpu = Math.tan((Math.PI) - beta) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordA = new Coordenada(coordActual.getLatitud() + cateOpu, this.minimaLon);

                this.puntosComprobacion.add(coordA);

            } else {

                cateCont = u.distancia(coordActual, new Coordenada(coordActual.getLatitud(), this.minimaLon));
```

```
                cateOpu = Math.tan(beta - (Math.PI)) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordA = new Coordenada(coordActual.getLatitud() - cateOpu,
this.minimaLon);

                this.puntosComprobacion.add(coordA);



            }

            //para beta +vision

            if (bv <= (3 * Math.PI) / 2) {

                cateCont = u.distancia(coordActual, new Coordenada(this.minimaLat,
coordActual.getLongitud()));

                cateOpu = Math.tan((3 * Math.PI / 2) - bv) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(this.minimaLat, coordActual.getLongitud() -
cateOpu);

                this.puntosComprobacion.add(coordB);

            } else {

                cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.minimaLon));

                cateOpu = Math.tan(bv - (3 * Math.PI / 2)) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(this.minimaLat, coordActual.getLongitud()
+ cateOpu);

                this.puntosComprobacion.add(coordB);

            }

        }
```

- UNA DE LAS LINEAS CHOCA CON LA PARED INFERIOR DEL MAPA Y LA OTRA CON LA PARED DERECHA

La primera línea está entre 270 y 315 grados y la otra de 315 a 360 grados.




```
//Correspondiente con "\-"

        //beta toque abajo y (beta+vision)dcha

        else if (((beta >= Math.PI + anguloAux) && (beta <= (3 * (Math.PI / 2)) +
angContRect)) && (((bv >= (2 * Math.PI - anguloAux)) && (bv <= (2 * Math.PI)))
|| ((bv >= 0) && (bv <= anguloAux)))) {

            this.puntosComprobacion.add(new Coordenada(this.minimaLat,
this.maximaLon));

            //para beta

            if (beta <= (3 * Math.PI) / 2) {

                cateCont = u.distancia(coordActual, new Coordenada(this.minimaLat,
coordActual.getLongitud()));

                cateOpu = Math.tan((3 * Math.PI / 2) - beta) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordA = new Coordenada(this.minimaLat, coordActual.getLongitud()
- cateOpu);

                this.puntosComprobacion.add(coordA);

            } else {

                cateCont = u.distancia(coordActual, new Coordenada(this.minimaLat,
coordActual.getLongitud()));
```

```
                cateOpu = Math.tan(beta - (3 * Math.PI / 2)) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordA = new Coordenada(this.minimaLat, coordActual.getLongitud()
+ cateOpu);

                this.puntosComprobacion.add(coordA);

            }

            if ((bv >= (2 * Math.PI - anguloAux)) && (bv <= (2 * Math.PI))) {

                cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.maximaLon));

                cateOpu = Math.tan((2 * Math.PI) - bv) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(coordActual.getLatitud() - cateOpu,
this.maximaLon);

                this.puntosComprobacion.add(coordB);

            } else {

                //Calculo el triangulo que contiene BETA

                cateCont = u.distancia(coordActual, new
Coordenada(coordActual.getLatitud(), this.maximaLon));

                cateOpu = Math.tan(bv) * cateCont;

                //Hemos hallado el punto donde la recta corta con una frontera.

                coordB = new Coordenada(coordActual.getLatitud() + cateOpu,
this.maximaLon);

                this.puntosComprobacion.add(coordB);

            }

        }
```

Una vez obtenido los puntos de nuestro polígono comprobamos que nodos están dentro de este polígono y de esto creamos una lista con los nodos disponibles. De esta lista de nodos se comprueba si el nodo pertenece a alguna calle o camino. Si satisface esta premisa, se comprueba que la calle o camino es la misma en la que se encuentra el nodo actual, cumpliendo esto el nodo permanece inalterable en la lista. Si por el contrario alguna de las condiciones anteriores no se cumplieran (no está situado en alguna calle, o

está situado en una calle y el nodo actual no está situado en dicha calle), el nodo queda descartado y se elimina de la lista. La comprobación de si está en una calle y si es la misma en la que el actual se encuentra, se realiza utilizando las tablas que se han rellenado al principio cuando se recibió el XML. En este punto tenemos los nodos que están en nuestra visión y además que están en la misma calle o camino en el cual el nodo actual se encuentra, por tanto son candidato a ser el punto siguiente al que me traslado. De todos estos nodos calculo cual es el que está más cerca mío, es decir, calculo la distancia de estos posibles nodos a mi nodo actual, y me quedo con el nodo que cumpla que su distancia es la mínima de entre las calculadas .La distancia no es más que la distancia euclídea entre dos puntos. Con esto ya obtengo el punto siguiente al cual me debo mover, ya que de entre los disponibles que están en mi calle y que puedo ver con mi visión, es el que está más cerca mío. La clave de este algoritmo es ir filtrando y descartar nodos a los cuales es imposible acceder bien porque el nodo no está en mi calle, o bien porque con mi visión actual y la inclinación de esta es imposible desplazarme hacia él, o bien porque cumpliendo ambas condiciones anteriores haya puntos más cercanos al actual.

Para realizar este algoritmo, no empezamos utilizando visión y moviéndonos de forma tan precisa, en una primera aproximación no utilizamos nada de ángulos, ni visión ni inclinación de esté, claro está tampoco construíamos ningún polígono, simplemente nos centramos en puntos cardinales (norte, sur, este y oeste). En nuestro proceso de selección de nodos nos quedábamos con los nodos que estuviesen hacia ese punto cardinal, de estos seleccionábamos los nodos que estuviesen en nuestra calle y como siguiente punto cogíamos el que estuviese más cerca. Esta aproximación era bastante imprecisa pero nos ayudo a derivar en lo que actualmente tenemos, con bastantes pasos de refinación en el camino. Esta aproximación tenía un gran problema que era la imprecisión, si todas las calles eran rectas y perpendiculares no había ningún problema, porque en un punto solo hay cuatro opciones(arriba, abajo, izquierda, derecha o norte, sur, este, oeste) pero en la realidad esto en raras ocasiones sucede, entonces podías estar situado en un cruce de calles con dos calles hacia el norte saliendo en forma de V desde el punto en el cual te encuentras y te moverías al punto más cercano de una de las calles, pero tú no podrías decidir hacia qué calle irías, simplemente se decidiría dependiendo de cuál de los puntos de esas dos calles estuviese más cerca de tu posición.

7. Implementación del Servidor

El servidor como bien se ha ido relatando en capítulos previos es el encargado de realizar todos los cálculos. Para llevar a cabo la implementación se ha divido las clases en paquetes con temáticas similares. La única clase no introducida en ningún paquete es la clase Main del Servidor. Dicha clase únicamente crea un objeto de la clase SuperServidor, sirve para arrancar éste. A continuación daremos un esquema de los paquetes con sus clases, y más abajo se explicarán en detalle.

Paquetes y clases:

- **Utilidades**: Es el paquete que concentra todas las utilidades que se utilizarán en otras clases.
    - **UtilidadesXML**: Es la clase encargada de ejecutar el algoritmo.
    - **Utilidades**: dispone de la mayoría de las funciones utilizadas para transformar unidades (como por ejemplo pasar de pixeles a coordenadas), calcular distancias entre puntos, etc.
    - **Jugador**: Como su nombre indica define a un jugador junto sus métodos y atributos.

- **Calibrar**: Es el encargado de pasar de coordenadas de los mapas a pixeles de nuestra pantalla. Tiene unos valores adecuados a nuestras circunstancias.

- **Coordenada**: Define lo que es una coordenada, sus valores de latitud y longitud como atributos y lo que podemos hacer con ella.

- **Pixel**: Identifica a los puntos de las pantallas.

- **PtosCardinales**: Esta clase define los puntos cardinales (norte, sur, este, oeste). Se creó en la primera aproximación al algoritmo.

- **Vector**: Define un vector sobre el plano. Tiene como atributos un punto y dispone de métodos para calcular su módulo, distancia a otro punto, etc.

- **Socket**: paquete con todo lo relacionado con la arquitectura cliente-servidor. También la comunicación entre ambas partes.

  - **SuperServidor**: Encargado de atender la primera conexión del cliente, y quedar a la espera de futuras peticiones.

  - **SocketServidor**: De ésta heredan todos los juegos. Se encarga de todas las comunicaciones con el cliente y va a ejecutar los juegos. Será una especie de controladora de que todo se ejecute pero sin involucrarse en la lógica propia de cada juego.

- **Juegos**: Aglutina todos los juegos. Todos ellos heredan de la clase Socket Servidor ya que comparten la comunicación mediante sockets con el cliente.

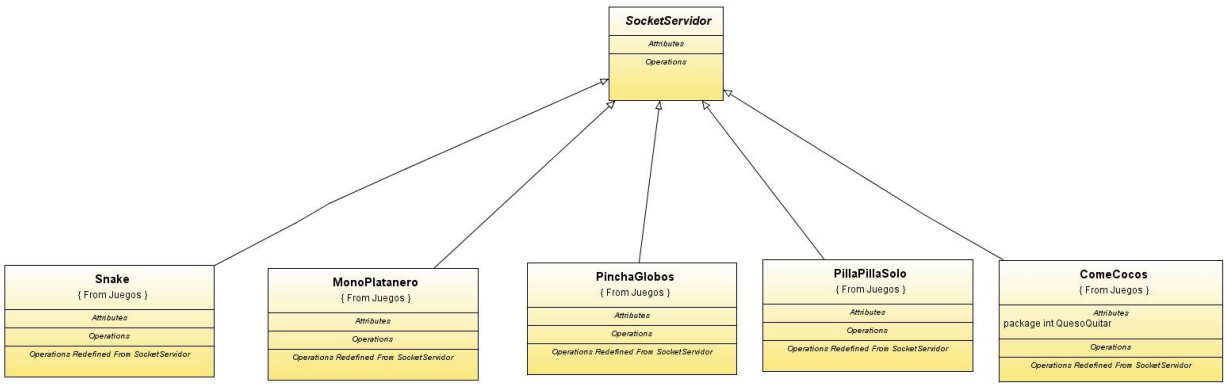
  - **ComeCocos**: Implementa la lógica propia del juego del comecocos.

  - **PillaPillaSolo**: Encargado del juego del pilla pilla.

  - **Snake**: Define en ella el juego de la serpiente.

  - **PinchaGlobos**: Es el juego del pincha globos.

  - **MonoPlatanero**: Juego del mono platanero.



Procedemos a introducirnos en materia en cuanto al desarrollo de cada clase:

- UTILIDADESXML

Define el algoritmo que se usa para el desplazamiento a través del mapa. Es una de las que mayor importancia tienen. Ha sido explicada con bastante detenimiento y esmero en capítulos anteriores por esta razón en este punto únicamente la citamos.

- UTILIDADES

Esta clase aglutina muchas de las utilidades que se usan en otras clases. No dispone de atributos, únicamente de métodos porque es verdaderamente para lo cual lo utilizamos. Los métodos de los que dispone son:

-Cambiar extensión de un fichero

```
String cambiarExtension(String filename){
     return filename.replaceAll("osm", "xml");
 }
```

-Calcular la distancia entre dos puntos

```
public double distancia (double x, double y){
    if (x>0&&y>0||x<0&&y<0)
       return Math.abs(Math.abs(x)-Math.abs(y));
    return Math.abs(x)+Math.abs(y);
 }
```

-Transforma la latitud de una coordenada a metros

```
public double latitudAmetro(double gradosLat){
     return 111133*gradosLat;
 }
```

-Transforma la longitud a metros

```
public double longitudAmetro(double gradosLat, double gradosLong){
```

```
    return   111320*Math.cos(gradosLat)*gradosLong;

 }
```

-Método que pasa una coordenada dada en metros a pixeles

```
public Coordenada MetrosAPixel(Coordenada coordAct,double minLat,double
minLon,double maxLat,double maxLon,int altoPant,int anchoPant){

  Coordenada coordPix=new Coordenada();

 double diferenciaLat= this.calcularDistanciaCoordenadasLat(minLat,maxLat);

 double
diferenciaLong=this.calcularDistanciaCoordenadasLong(maxLon,minLon,minLat,m
axLat);

 double relLat=altoPant/diferenciaLat;

 double relLong=anchoPant/diferenciaLong;

 double latAm=latitudAmetro(coordAct.getLatitud());

 double lonAm=longitudAmetro(coordAct.getLatitud(), coordAct.getLongitud());

 double latesquinam=latitudAmetro(minLat);

 double lonesquinam=longitudAmetro(minLat,maxLon);

 double distPtoOrigenlat=distancia(latAm, latesquinam);

 double distPtoOrigenlon=distancia(lonAm, lonesquinam);

 double pixX=(distPtoOrigenlon*relLong);

 double pixY=(distPtoOrigenlat*relLat);

 coordPix.setLatitud(pixY);

 coordPix.setLongitud(pixX);

 return coordPix;

}
```

Únicamente hemos expuestos los que nos parecían más interesantes. El resto se pueden consultar en el código fuente.

JUGADOR

Es la clase que define a nuestro jugador. Tiene como atributos: la *posición* donde esta, *inclinación* de su visión, si es *visible* en un momento ya que puede haber jugadores que

no sea posible verlos, *puntos* ganados, imagen de su icono, un *suceso* de lo que le ha ocurrido. Los métodos que dispone son accesores y modificadores de sus atributos. En su constructora se inicializan éstos.

- CALIBRAR

Es el encargado de hacer el calibrado, pasar puntos de la imagen real a puntos en la pantalla, y aún siendo diferentes escalas conseguir que todo se mantenga en el mismo lugar. Se necesitará los valores de máximas y mínimas longitudes y latitudes. El proceso en sí está explicado detalladamente con anterioridad.

- COORDENADA

Como su nombre indica, define una coordenada. Tiene dos atributos de tipo double que son la longitud y latitud, los valores que identifican el punto en la Tierra. Dispone de métodos básicos para acceder a las variables y modificarlas. Se ha redefinido el método para identificar si son dos coordenadas iguales. También se redefinió el método hashcode.

- PIXEL

Es un punto de la pantalla, del eje de cartesianas de dos dimensiones. Queda identificado por dos enteros. Tiene métodos básicos para usar la clase, accesores y modificadores. Igual que en el caso anterior se ha redefinido el método equals y hashcode.

- VECTOR

Es un vector centrado en el origen. Lo podemos usar como punto ya que únicamente tiene dos atributos de tipo double y los métodos para manejar estos atributos. Además tiene métodos propios de los vectores como el cálculo de su módulo, producto vectorial entre dos de ellos.

```
double ProductoEscalar(Vector v1,Vector v2) {

      return ((v1.getX() * v2.getX()) + (v1.getY()* v2.getY()));

 }
```

```
   //Función que haya el módulo del producto vectorial de vectores con signo

     double ModuloDelProductoVectorialConSigno(Vector v1,Vector v2) {

      return ((v1.getX()* v2.getY()) - (v1.getY() * v2.getX()));

   }
```

```
  //Método que haya el módulo de un vector
```

```
double modulo(){

  return (Math.sqrt(Math.pow(this.x,2)+Math.pow(this.y,2)));

}
```

- SUPERSERVIDOR

El objeto de esta clase cuando se crea entra en un bucle esperando clientes. Es la encargada de atender la primera vez a cada cliente. Y luego delega en socketservidor para futuras conexiones del cliente. Antes de esta delegación del trabajo, se ocupa de asignar un número de puerto diferente a cada juego nuevo que entra. La ejecución del juego se va a comunicar a partir de entonces con este nuevo número de puerto. Así conseguimos que haya varios juegos en máquinas diferentes utilizando como servidor esta clase simultáneamente.

- SOCKETSERVIDOR

De esta clase heredarán todos los juegos, es una de las clases claves en la interconexión. Una vez que el SuperServidor ha hecho su trabajo, crea un objeto de esta clase que va a implementar el juego.

Para poder estar a la escucha de futuras peticiones de nuestro cliente, un cliente ya fijo, entrará en un bucle infinito. Cuando alguno de los dos necesite comunicarse abrirán conexión por el puerto que nos había indicado el SuperServidor. Este puerto es únicamente para comunicarnos durante este concreto juego. Si entrase un nuevo cliente para iniciar un juego accedería a SuperServidor y con un número de puerto conocido.

Para obtener más información sobre esta clase se puede consultar en el apartado (ver capítulo 4) de la comunicación entre cliente y servidor donde ha sido explicada detenidamente, ya que juega un papel importantísimo.

- PAQUETE DE JUEGOS

Cada una de las clases que conforman este paquete implementa un juego diferente. Todas ellas heredan de SocketServidor. Este paquete ha sido explicado extensamente en un apartado de este texto exclusivamente.

Cada clase implementa el juego que indica su propio nombre. La clase comecocos implementa el juego del comecocos, la clase Snake implementa el juego de la serpiente (“snake” en inglés) y así para PinchaGlobos, MonoPlatanero y PillaPillaSolo.

8. Implementación del cliente

En este capítulo se hará hincapié en lo relativo al cliente de nuestra aplicación, destacando las características más importantes de la misma, junto con los detalles de las partes en que se descompone el cliente de nuestro proyecto.

Desde un principio hemos querido que nuestra aplicación tuviera una vocación hacia internet, es decir, estuviera accesible, como hemos comentado anteriormente, desde la red de redes para todo el mundo. Por todo esto se pensó en implementar el cliente de tal forma que pudiera ser accesible a través de un navegador web, ya que este es el instrumento más utilizado y popular hoy día para el uso de internet. Como resultado de una ardua investigación pensamos utilizar un componente llamado Applet para llevar a cabo nuestra implementación del cliente.

8.1 Applet

Un Applet es un componente de una aplicación que es capaz de ejecutarse en un navegador web, como es en nuestro caso, o en el contexto de cualquier otro programa. Cuando decimos que es capaz de ejecutarse dentro de un navegador web significa que el propio navegador web es capaz de enviar diversas señales a éste para que se inicie, se pare o se cierre la ejecución que realiza.

Como toda nuestra aplicación usa el lenguaje Java, tuvimos la fácil decisión de utilizar los objetos JApplet de Java para implementar el Applet correspondiente a nuestro cliente.

Estos objetos corresponden a una clase abstracta de la que debemos implementar diversos métodos para poder utilizarla. Estos métodos son rutinas que se lanzan en diversas circunstancias que explicaremos para cada caso:

```
public void init()
```

Esta rutina se lanza cuando abrimos la página web y se inicia el Applet.

En nuestro caso hacemos labores de inicialización de parámetro como el tamaño del Applet o la creación de objetos que nos serán necesarios más adelante.

```
public void start()
```

Esta rutina se lanza cuando después de parar el Applet por algún motivo, este se vuelve reiniciar.

Esta rutina se lanza cuando el Applet es destruido, es decir, cuando la página web es cerrada por el navegador.

En nuestro caso cuando destruimos el Applet una de las cosas que realizamos es cerrar las conexiones abiertas con el servidor y la comprobación de que todo ha terminado correctamente

Cuando nosotros abrimos una página web, el navegador se comunica con el servidor donde se encuentra alojada. Acto seguido, el navegador se descarga el código de la página web junto con sus diversos componentes, Applets, fotos,… Una vez descargados, el navegador interpreta el código, dependiendo del lenguaje de programación utilizado en la página web, y lo ejecuta. Esta ejecución la vería el usuario por pantalla si el código de la página así lo requiere. Todo esto parece muy correcto, pero en realidad no lo es. Pongamos por ejemplo que realizamos un applet que al ejecutarse borre todo el disco duro de la máquina donde se ejecuta el applet. Estaríamos ante un pseudo-virus que hemos ejecutado inconscientemente al entrar en una web de la que no conocíamos su contenido.

Para evitar este tipo de problemas, los applets disponen de varias restricciones por defecto para garantizar la seguridad del usuario que los ejecuta. Una de ellas, quizá la más importante, es la prohibición del applet de acceder y hacer uso de los distintos recursos de la máquina donde se ejecuta, por ejemplo, no podrá consultar, crear, modificar o borrar carpeta alguna. Todo lo contrario ocurre con la consulta, creación, modificación o borrado de carpetas en la máquina donde se encuentra alojado, es decir, en el servidor. En este caso no existe ninguna restricción y es en la posibilidad que utilizamos en nuestro proyecto

Existe una alternativa, eliminando la anterior restricción, la llamada firma digital del applet. Una firma digital es un consentimiento que se le pide al usuario que ejecuta el applet para que éste pueda hacer uso de los recursos de la máquina donde se ejecuta. De

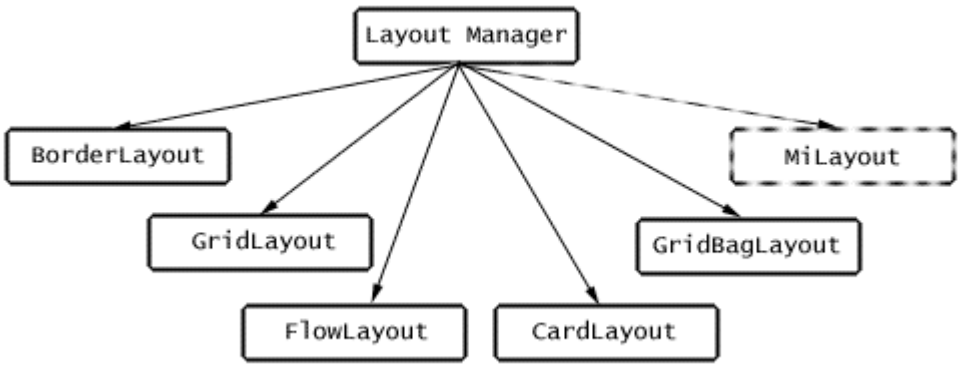
esta forma al inicio de la ejecución del applet saldrá un cuadro de diálogo con la petición y quién lo ha firmado. De aquí el internauta sacará sus conclusiones y permitirá o no eliminar la restricción de la que hablamos.

En nuestro proyecto decidimos no realizar firma digital alguna ya que en cierta parte no nos era indispensable. Nos evitamos de este modo realizar la firma digital aunque por el contrario sobrecargamos un poco más la comunicación cliente-servidor. De esta forma no almacenamos nada en la máquina donde se ejecuta el applet, todos los datos están guardados en el servidor de la aplicación.

8.2 CardLayot

El Applet por implementación sólo dispone de un contenedor donde se pueden alojar diferentes componentes del swing de Java. Para el desarrollo de nuestra aplicación esto no nos era útil ya que para el transcurso de la misma nos hacían falta un gran número de pantallas diferentes para que el usuario pudiera visualizar todas las posibilidades que nuestro proyecto albergaba.

Para poder encajar las necesidades básicas de nuestro proyecto en las características del Applet pensamos en utilizar una herramienta proporcionada por Java propia de los contenedores. Tanto el JFrame de Java como el Applet disponen de un llamado contenedor. El contenedor es un espacio donde se colocará todo lo que queramos que esté disponible, visible o no, cuando el usuario ejecute la aplicación y está ante ellos. Para ello el contenedor dispone de unas reglas que determinarán la posición de los componentes agregados en el mismo. Existen varios tipos de conjuntos de reglas, en general se les llama *layout managers* o *manejadores de composición*. A continuación se mostrará los diferentes manejadores de composición que propone Java, aunque cualquier usuario puede caracterizar el suyo propio.



Por ser el más similar a nuestras necesidades, para nuestro Applet se ha utilizado el CardLayout. Este manejador de composición dispone de varios JPanel de Java, en definitiva son paneles o pantallas, en los que nosotros agregaremos componentes a nuestro gusto. Por ejemplo, podremos tener un panel o pantalla que tenga un solo botón y otro panel que tenga varias JLabel o etiquetas. Lo característico del CardLayout es que simula el comportamiento de una baraja de cartas. Tendremos un mazo de cartas

que será todo el conjunto de paneles que hayamos creado pero, aquí viene lo importante, el usuario solo podrá ver el panel o carta que se encuentre en la cima del mazo de cartas.

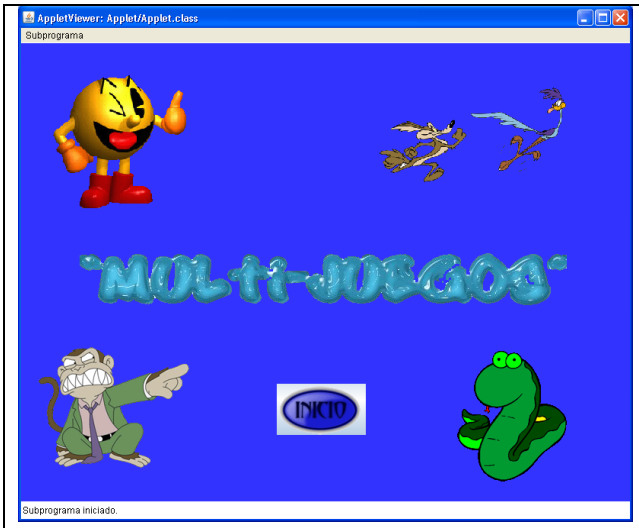
En el caso del dibujo, solo podría ver el panel correspondiente al *As de picas* y no el resto. El layout manager permite cambiar el panel a mostrar y por lo tanto, por símil, cambiar la carta que estaría en la cima del mazo.

En el caso de nuestra aplicación hemos creado diferentes paneles respetando las necesidades de nuestra aplicación que posteriormente hemos introducido en el CardLayout creado para así poder tener a nuestro alcance la posibilidad de cambiar el panel a mostrar a nuestro antojo. Todos estos paneles se encuentran en el paquete GUI de nuestro proyecto, en la parte cliente que es el que soporta toda la parte gráfica del mismo.

Para crear un CardLayout debemos realizar:

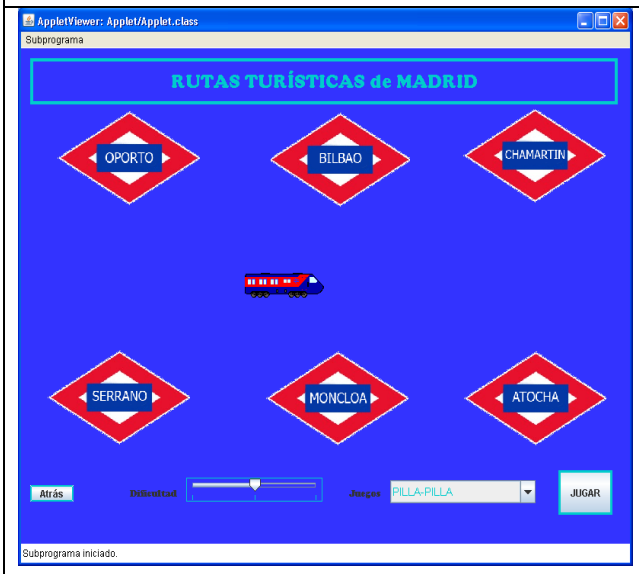
```
CardLayoutPanel card = new CardLayoutPanel();

card.addComponentToPane(this.getContentPane())
```

A continuación se mostrarán los diferentes paneles creados y que están inmersos en este proceso.



Panel de inicio

Panel de elección Tipo



Panel Mapas Fijos

Panel Juegos

Panel elección Lugar

Panel Juego

Una vez creados los paneles y el objeto de tipo CardLayout pasamos a añadir estos paneles en el CardLayout:

```
cards.add(panelEleccion, PANEL_ELECCION);

cards.add(panelInicio, PANEL_INICIO);

cards.add(panelJuego, PANEL_JUEGO);

cards.add(panelJuegos, PANEL_JUEGOS);

cards.add(panelEleccionTipo, PANEL_ELECCION_TIPO);

cards.add(panelMapasFijos,PANEL_MAPAS_FIJOS);
```

Para terminar, mostrar la órden a la que obedece el layout manager para cambiar el panel a mostrar, en el ejemplo se podrá ver el panel de inicio de la aplicación.

```
CardLayout cl = (CardLayout) (cards.getLayout());

cl.show(cards, PANEL_INICIO);
```

8.3 Sonidos

Otra de las características propias de los applets y de la que hemos sacado un buen uso es la de los sonidos. En una inmensa mayoría, no en nuestro caso, los applets son utilizados en aplicaciones más bien pequeñas y de limitada capacidad. Muchas de ellas son aplicaciones simples e interactivas que se basan en el uso de formas y sonidos dibujados simplemente sobre el contenedor del applet sin llegar a usar ningún layout

manager. Para hacer más fácil el uso de estas aptitudes la clase JApplet trae implementado un sistema sencillo para poder llevarlas a cabo.

En nuestro caso para una mejor visión del proyecto decidimos enfrascar todas las características relacionadas con el proyecto en una clase con los diferentes métodos para poder realizar los diferentes sonidos. En esta clase, llamada Sonidos, se disponen de todas las rutas de los archivos de audio y de los métodos a utilizar para poder llegar a ejecutar los mismos.

Para poder ejecutar un sonido sobre un applet de Java solo es necesario crear un objeto AudioClip de la siguiente manera.

```
AudioClip sonido;
```

Cargar posteriormente el sonido, comunicando la ruta donde se encuentra alojado el archivo, en este caso se encontrará situado en el proyecto cliente, es decir, asociado al applet y no al servidor.

```
sonido = getAudioClip( getDocumentBase(),"sonido.au" );
```

Una vez cargado y creado el sonido solo tenemos que ejecutarlo cuando nos plazca, para ello tenemos diferentes comandos para ejecutar o parar los sonidos.

```
sonido.play();
```

- Para reproducir el clip de sonido.

```
sonido.loop();
```

- Para reproducir el clip de sonido pero entrando en un bucle de repetición.

```
sonido.stop();
```

- Para parar la reproducción del clip.

En nuestra aplicación hemos utilizado sonidos para diferentes partes de la misma, siendo escuchados en diferentes menús o incluso en el transcurso de algún juego.

8.4 Pintado

Como hemos comentado en el apartado anterior, los applet de Java tienen diversos mecanismos implementados preparados para aplicaciones sencillas, entre ellos el de dibujar formas geométricas o imágenes sobre su contenedor. Para ello solo basta con introducir en su método paint los elementos que se quieren dibujar, bajo unas determinadas funciones, para obtener el resultado esperado. Tener claro, para este objetivo, que la esquina superior izquierda del applet corresponde a las coordenadas cartesianas (0,0) y la esquina inferior derecha con (n,n).

En nuestro caso tuvimos un mayor problema a la hora de poder pintar diferentes formas sobre el applet de Java. Necesitábamos pintar imágenes y formas geométricas para representar, dentro de una partida, los diferentes elementos de un juego (las imágenes del mono y los plátanos, por hablar de uno de los juegos que explicaremos posteriormente). Al principio utilizamos el método descrito anteriormente aunque al colocar el layout manager para poder realizar el cambio de paneles se superposicionaban resultando nula la visión de los mismos.

Cuando el proyecto avanzó, el siguiente paso que dimos fue la colocación, por el método anteriormente mencionado, de la imagen de un mapa de fondo con los diferentes elementos del juego encima. Este método fue útil durante esos días, dado que todavía no habíamos realizado interfaz gráfica ninguna y nuestro único objetivo era llegar a probar el funcionamiento del algoritmo y jugabilidad de la aplicación.

Por último, una vez implementada la interfaz gráfica de la aplicación, tuvimos la necesidad de colocar los diversos elementos de una partida y de un juego encima de una imagen de un plano. La situación era que la imagen del plano se encontraba dentro de un panel perteneciente al CardLayout antes descrito y más concretamente dentro de PanelJuego. La imagen estaba alojada en forma de JLabel dentro de PanelJuego.

Cuando nosotros cargábamos una imagen sobre esta JLabel lo hacíamos de forma estática, es decir, una vez cargada la imagen para un juego no se volvía a cambiar, lo cual resultaba bastante sencillo. Sin embargo, cuando nosotros colocábamos los diferentes elementos antes descritos, no los podíamos colocar de forma estática, ya que estos se debían mover en función de los parámetros explicados en capítulos anteriores. Es por ello que no podíamos utilizar el mismo método.

Después de unos días de reflexión y estudio de la situación concluimos que la mejor idea para resolver el problema era realizar un objeto que heredara de la clase JLabel y siendo la única diferencia entre ambos el método paint. Este método paint se comportaría igual que el de la clase de la que heredaba salvo en que cada vez que se ejecutara pintara además de la imagen del mapa los diferentes elementos almacenados en una estructura posteriormente explicada.

En nuestro proyecto hemos realizado una clase que se encarga del pintado de los elementos y la propia imagen dentro de la clase anteriormente descrita, su nombre es clase Pintado. Para ello, dentro del paint antes mencionado, se realizada una llamada a un método de esta clase y es ésta la que pinta sobre el objeto que hereda de JLabel. La información que tiene que pintar la obtiene de la clase Conexión. Esta clase, aunque posteriormente se explicará con más profundidad, es la encargada del intercambio de datos con el servidor. Por lo tanto, llegamos al resultado deseado.

8.5 Conexión

Desde los primeros pasos en la construcción del cliente hemos querido que éste estuviera bien dividido y estructurado para poder tener una visión más amplia del mismo delimitando sus partes más características y especializadas. Como consecuencia

de este motivo, decidimos realizar una clase encargada exclusivamente de la comunicación con el servidor.

Esta clase de la que hablamos no es más que un mero interlocutor que habla con el servidor haciendo uso del idioma de éste utilizando la clase SocketCliente.

La clase SocketCliente es la encargada de llevar a cabo la comunicación entre el cliente y el remoto servidor. Para empezar a explicar esta clase introduciremos la apertura de conexión. Con respecto a esto diferenciaremos dos métodos, ambos para abrir la comunicación. Uno de los métodos es el encargado de abrir la primera conexión en la que existe un número de puerto conocido que no cambia. Esta conexión se lanzará al SuperServidor. Cuando se produce esta apertura el servidor nos comunica el número de puerto con el cual, a partir de ahora debemos comunicarnos. Este número le almacenamos como atributo de la clase. Las futuras comunicaciones con el servidor no se realizarán ya con el SuperServidor sino con el SocketServidor, para ello es necesario el anterior número de puerto. Como atributo imprescindible para todo debemos tener el socket.

Con número de Puerto fijo, para el SuperServidor:

```
PUBLIC VOID ABRIRCONEXION1() THROWS IOEXCEPTION {

    SOCKET = NEW SOCKET("LOCALHOST", 45560);

    SOCKET.SETSOLINGER(TRUE, 10);

    SYSTEM.OUT.PRINTLN("CONECTADO");

 }
```

Con número de Puerto que nos ha enviado el SuperServidor, comunicaciones con socket servidor:

```
PUBLIC VOID ABRIRCONEXION() THROWS IOEXCEPTION {

    SOCKET= NEW SOCKET ("147.96.114.211",PUERTO );

     SOCKET.SETSOLINGER(TRUE, 10);

    SYSTEM.OUT.PRINTLN("CONECTADO");

 }
```

También tenemos los métodos propios para adquirir cada uno de los buffers necesarios para las comunicaciones. Los buffers los tenemos como atributos de la clase y disponemos de los métodos para adquirirlos. Disponemos de buffers de entrada y salida tanto de tipos básicos (enteros, booleanos, string) como para tipos no básicos, instancias de clases. Dichas clases deben ser serializables para que no ocurran problemas.

Atributos, buffers de entrada y salida:

```
ObjectOutputStream bufferObjetosOut;
    DataInputStream bufferIn;
    DataOutputStream bufferOut;
    ObjectInputStream bufferObjetosIn;
```

Los siguientes son los métodos para la adquisición de los buffers, los nombres de los métodos son bastantes intuitivos para saber a lo que está destinado cada uno.

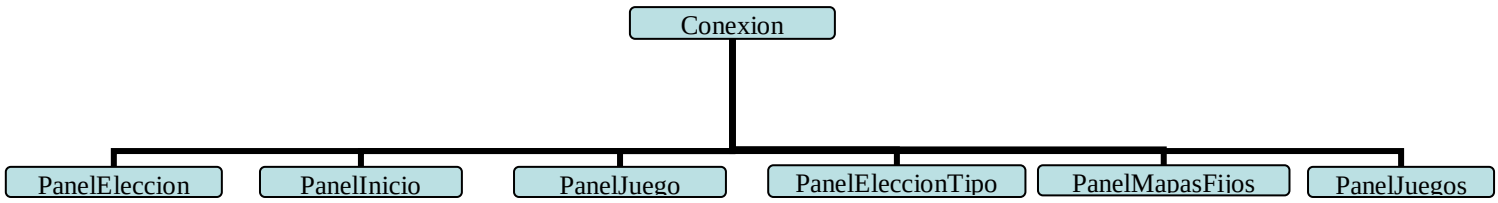
```
public void obtieneFlujoDatosIn() {
        try {
            if (socket != null) {
                bufferIn = new DataInputStream(socket.getInputStream());
            }
        } catch (IOException ex) {
            Logger.getLogger(SocketCliente.class.getName()).log(Level.SEVERE, null,
ex);
        }
    }
    public void obtieneFlujoDatosOut() {
        try {
            if (socket != null) {
                bufferOut = new DataOutputStream(socket.getOutputStream());
            }
        } catch (IOException ex) {
            Logger.getLogger(SocketCliente.class.getName()).log(Level.SEVERE, null,
ex);
        }
    }
    public void obtieneFlujoObjetosIn() {
        try {
```

```
            if (socket != null) {
                bufferObjetosIn = new ObjectInputStream(socket.getInputStream());
            }
        } catch (IOException ex) {
            Logger.getLogger(SocketCliente.class.getName()).log(Level.SEVERE, null,
ex);
        }
    }
    public void obtieneFlujoObjetosOut() {
        try {
            if (socket != null) {
                bufferObjetosOut = new ObjectOutputStream(socket.getOutputStream());
            }
        } catch (IOException ex) {
            Logger.getLogger(SocketCliente.class.getName()).log(Level.SEVERE, null,
ex);
        }
    }
```

Para finalizar las comunicaciones debemos realizar el cierre de ésta. Para ello, cerramos el socket.

```
  public void cerrarConexion() {
        try {
            this.socket.close();
            System.out.println("cerrada conexion");
        } catch (IOException ex) {
            Logger.getLogger(SocketCliente.class.getName()).log(Level.SEVERE, null,
ex);
        } }
```

Para que en cualquier parte del juego, hilado siempre en los paneles de la aplicación, pudiéramos hacer uso de la comunicación cliente-servidor está disponible un objeto Conexión capaz de interactuar cuando se le pida con el servidor estableciendo o no comunicación con él o simplemente transfiriendo los datos necesarios en cada momento.



Dentro de este objeto se encuentran almacenados los diferentes datos e informaciones relacionadas con el juego y que son estrictamente necesarios. Estos datos tienen como fuente al servidor en unos casos y al cliente en otros, por ejemplo, en el tema relacionado con la personalización que pueda hacer el usuario del juego. A continuación se explicarán algunos de los más importantes.

```
private ArrayList jugadores;
```

Este array dispondrá de la información de la posición de los elementos que debe mostrar el cliente por pantalla y que es utilizado únicamente para pintar los elementos sobre el JLabel, anteriormente explicado.

```
private String nombreJuego;
```

Indica el nombre del juego y es utilizado, además de estéticamente, para diferenciar diferentes características de un juego u otro y así realizar distintas rutinas de trabajo.

```
private double maxLat, minLat, minLon, maxLon;
```

Tamaño del mapa que es recibido desde el servidor y que es utilizado en el cliente para realizar el calibrado.

```
private ImageIcon imagenFondo;
```

Recibida por el servidor, después de hacer este los mecanismos necesarios, es alamacenada aquí para poder ser mostrada por la clase Pintado en el applet.

```
private int dificultad;
```

Seleccionada por el usuario.

```
private boolean finJuego;
```

Es una variable enviada por el servidor que señala al cliente si debe proceder a ejecutar el proceso de terminación de partida, con todo lo que implica eso: cambiar pantalla, cerrar conexión,…

```
private boolean jugandoTeclado o  jugandoRaton;
```

Estas variables se utilizan para saber si el juego al que se está jugando necesita el ráton o el teclado. De esta forma se habilitan un u otro dependiendo de la elección. El habilitado está directamente relacionado con la activación o no de los eventos de los mismos.

Una vez explicados las diferentes informaciones que almacena el objeto conexión en forma de atributos se pasará a los métodos implementados en el mismo para la realización de sus funciones características de interlocutor con el servidor.

```
public void iniciarSuperServidor(String nombreJuego)
```

Este método inicia la conexión al SuperServidor enviando a éste el nombre del juego elegido por el usuario. Siempre se conectará con el SuperServidor a través de un puerto concreto en el que permanece a la escucha. Una vez contactado, recibe como respuesta el número de puerto que a partir de ese momento se usará siempre para la transmisión de información.

```
public void iniciarParametros(String mapa)
```

Este método se encarga de adquirir los datos necesarios para comenzar a jugar una partida de un juego dado, en definitiva, sería el procedimiento cuya responsabilidad radica en el relleno de la mayor parte de los atributos anteriormente descritos.

Los pasos a seguir una vez ejecutado este método resultarían:

- Abriría la conexión con el puerto obtenido con el método iniciarSuperServidor.

- Una vez establecida la conexión se le enviaría un número, en este caso un uno, como parte del protocolo de comunicación implementado en este proyecto y explicado en capítulos anteriores.

- Se envía la dificultad seleccionada por el usuario.

- A continuación se selecciona si se quiere un mapa personalizado por el usuario o bien un mapa fijo, indicando cuál en este caso.

- El cliente recibe la imagen del mapa anteriormente seleccionada.

- Una vez obtenido el mapa, se pasa a la recepción del ArrayList de jugadores que contendrá objetos de tipo Jugador, que definen tanto las posiciones como las características de los elementos que se deben ver por la pantalla dentro de una partida.

- Recepción de las diferentes longitudes, en cuanto a coordenadas cartesianas, del mapa recibido anteriormente para poder realizar correctamente el calibrado del mapa.

- Seguidamente se pasa a la selección del modo de juego: ratón o teclado.

- Este es el último paso se activa la rutina del *timer* para poder añadir funcionalidad a la partida en la que se está jugando. A continuación se explicará más en profundidad su motivo.

Una vez completados los pasos asociados a este método, hemos sido capaces de rellenar todos los atributos anteriormente mostrados. De esta forma la partida estaría totalmente preparada para comenzar.

```
private void activarTiempo()
```

Esta función activa una rutina que se ejecuta cada quinientos milisegundos una vez ejecutado el método anteriormente descrito.

Esta rutina tiene como utilidad intercambiar información respecto a los jugadores o elementos que se disponen por la pantalla. De esta forma cada medio segundo se actualizan estos valores para poder actualizarlos de la misma forma a los ojos del usuario. A la vez que se realiza este proceso, también se recibe del servidor si la partida ha terminado o no. Si hubiera terminado se invocará otro método que resuelve el fin del juego, sin embargo, si no ha terminado se seguirá con la ejecución normal de la partida.

8.6 Eventos

Otra forma de actualizar las posiciones de los elementos de la partida es cuando el usuario pulsa una tecla, que sea un cursor. En este momento, el cliente capta este evento del teclado, a través de una función alojada en esta clase, e invoca una rutina que trata la misma. El evento en cuestión se llama *KeyListener*. En nuestro caso, esta rutina se encarga de conectarse con el servidor, enviarle la etapa en la que está trabajando el cliente, esto tiene que ver con el protocolo descrito anteriormente. Una vez enviada ésta, el cliente recibe los datos actualizados del ArrayList de jugadores y además comprueba si se ha llegado al final del juego.

Para hacer más eficiente el intercambio de información este evento solo se activa cuando el usuario suelta la tecla pulsada y no cuando se pulsa. Este hecho es así ya que al tener pulsada la tecla se activaba la rutina en un intervalo muy corto muchas veces, por lo que saturaba la red de comunicación y producía fallos de sincronización en el servidor al acceder varias llamadas de la rutina a un mismo recurso. Se demostró más eficiente la alternativa finalmente implementada.

Por otro lado, también existe otra rutina de tratamiento de eventos del ratón para las partidas de juegos enmarcadas en esta posibilidad, por ejemplo, *el pincha-globos*, que más adelante se mostrará más en profundidad.

Cuando nos encontramos jugando al juego anteriormente nombrado y clickeamos sobre el applet, se evoca una rutina que capta la posición pulsada dentro del applet. En nuestra implementación se calcula si esta posición se refiere a alguna cuya ubicación se encuentre dentro de la imagen del mapa, es decir, dentro de la etiqueta. Esto es así porque las pulsaciones del ratón fuera de ella no nos son importantes de tratar ya que carecen de utilidad.

Una vez seleccionadas las pulsaciones válidas se procede a la conexión con el servidor enviándole la etapa del protocolo adecuada para este caso, por ejemplo, la etapa tres. Además se adjunta la información en forma de Pixel asociada al clickeo del ratón, para posteriormente en el servidor transformarla a coordenada UTM y comprobar jugador a jugador si coinciden ambas. Poniendo como ejemplo el caso del *pincha-globos* sería causa de que se hubiera explotado un globo y se pasaría al tratamiento de este caso: sumar puntos, colocar otro globo,…

8.7 Estructura de clases

Como se ha ido explicando en los capítulos anteriores, el cliente es la cara de nuestra aplicación que se muestra al usuario. En definitiva es el encargado de recibir las peticiones del cliente, trasladar estos deseos al servidor y mostrar por pantalla el resultado de su procesamiento. Salvo la interfaz gráfica y diversas rutinas de tratamiento, sobre todo durante la partida, el cliente no procesa ningún dato.

El cliente se estructura en cinco paquetes, dividiendo estos las partes más significativas del cliente de la aplicación. En cada uno de ellos tenemos diversas clases con sus respectivos atributos y métodos que explicaremos a continuación.

Paquetes y clases:

- **Applet:** Es el paquete donde se encuentra todo lo que tiene una relación directa con el Applet.
    - **Applet:** Es la clase que hereda de JApplet y es la entidad donde se encuentra verdaderamente el Applet. Como hemos comentado antes tiene cinco métodos dado que implementa una clase abstracta.
    - **Conexión:** Es la clase que se encarga de la comunicación entre el cliente y el servidor. Todo lo relacionado con la conexión entre ambos se encuentra aquí.
    - **PanelImagenFondo:** Es un objeto que hereda de JLabel pero redefine el método paint de la misma para poder pintar elementos sobre la imagen del mapa.
    - **Pintado:** Es la clase que se encarga del pintado de los elementos sonbre PanelImagenFondo.

  - **Sonidos:** En esta clase se encuentran alojados todos los sonidos que deben sonar durante la ejecución del Applet.

- **Utilidades**: Es el paquete que concentra todas las utilidades que se utilizarán en otras clases.
  - **UtilidadesXML**: Es la clase encargada de ejecutar el algoritmo.
  - **Utilidades**: Dispone de la mayoría de las funciones utilizadas para transformar unidades (como por ejemplo pasar de pixeles a coordenadas), calcular distancias entre puntos, etc.
  - **Jugador**: Como su nombre indica define a un jugador junto sus métodos y atributos.
  - **Calibrar**: Es el encargado de pasar de coordenadas de los mapas a pixeles de nuestra pantalla. Tiene unos valores adecuados a nuestras circunstancias.
  - **Coordenada**: Define lo que es una coordenada, sus valores de latitud y longitud como atributos y lo que podemos hacer con ella.
  - **Pixel**: Identifica a los puntos de las pantallas.
  - **PtosCardinales**: Esta clase define los puntos cardinales (norte, sur, este, oeste). Se creó en la primera aproximación al algoritmo.
  - **Vector**: Define un vector sobre el plano. Tiene como atributos un punto y dispone de métodos para calcular su módulo, distancia a otro punto, etc.
- **Archivos:** En este paquete se alojan todos los archivos de imágenes que utiliza el Applet durante su ejecución.
- **GUI:** En este paquete se encuentra todo lo relacionado con la interfaz gráfica.
  - **CardLayoutPanel:** Se encuentra la clase que implementa la disposión de los paneles dentro de la aplicación, su motivación ya se explicó con anterioridad.
  - **Paneles:** Se encuentran todos los paneles de la aplicación.
- **Socket:** Paquete con todo lo relacionado con la arquitectura cliente-servidor. También la comunicación entre ambas partes.
  - **SocketCliente:** Es la clase que tiene todos los métodos para poder enviar y recibir información por parte del cliente hacia el servidor.

Procedemos a introducirnos en materia en cuanto al desarrollo de cada clase:

- APPLET

Como se ha explicado anteriormente, el Applet es el pilar fundamental del cliente, capaz de mostrar todos los resultados de nuestro proyecto por pantalla. Se caracteriza por contener distintos paneles distribuidos gracias al layout manager.

- CONEXIÓN

Es la encargada de canalizar todo el flujo de datos entre el cliente y el servidor. Contiene distintas rutinas de comunicación que sirven para enviar y recibir datos según el protocolo establecido. Es la única clase que usa método de la clase SocketCliente. Además en ella se almacena toda la información intercambiada entre ambos para una mayor seguridad.

- PANELIMAGENFONDO

Es aquí donde se coloca la imagen del mapa. Es una clase que hereda de la clase JLabel redefiniendo el método paint para poder pintar y colocar los distintos elementos sobre el plano.

- PINTADO

La clase encargada del pintado de los elementos anteriormente nombrados sobre el mapa. Para ello, obtiene dicha información de los atributos de conexión, que es la clase que contiene toda la información..

- SONIDOS

Esta clase es la encargada de almacenar las rutas de los diferentes archivos de sonidos y de contener los diferentes métodos para la reproducción de dichos archivos. Para poder hacer que suenen se utiliza la tecnología de los Applet propia de los sonidos, como anteriormente se ha explicado.

- PAQUETE ARCHIVOS

Como se ha comentado anteriormente contiene todas las imágenes utilizadas dentro del Applet. Esto es así para poder reducir el tráfico entre el cliente y el servidor.

- CARDLAYOUTPANEL

Esta clase implementa el layout manager concreto, en este caso el que da nombre a esta clase. Para ello tiene que estar en posesión de los objetos de los paneles que se quieren alojar en éste.

- PANELES

Son los conjuntos de paneles dispuestos dentro del Applet y que se muestran al usuario para hacer más sencilla y agradable su estancia en la aplicación. Existen diferentes, cada uno para una fase del juego.

- SOCKETCLIENTE

Esta clase se encarga de implementar los métodos para poder hacer efectiva la comunicación entre el cliente y el servidor. Para ello hace uso de sockets a una implementación bastante baja. La única clase que usa ésta es Conexión.

- PAQUETE UTILIDADES

Por último las clases incluidas en el paquete utilidades ya se encuentran explicadas en el capítulo 7: *Implementación del Servidor*.

9. Juegos arcade

Nuestro objetivo una vez implementado los algoritmos de búsqueda y una vez que hemos conseguido descargarnos los mapas de la Web (todo esto ha sido comentado en anteriores capítulos), lo que nos quedaba por hacer eran los juegos en sí. Para ello, nos basamos en nuestras vivencias personales y pensamos en adaptar algunos juegos clásicos que jugamos en nuestra infancia, como pudiera ser un rescate o pilla pilla, un comecocos, juegos que tuvieron un gran éxito en plataforma de móviles como la serpiente y juegos que surgieron en la facultad, como son el pincha globos y el mono platanero y adaptarlos a nuestro proyecto ya que nos parecía curioso, poder jugar a todos ellos en nuestras propias calles y barrios. Como hemos dicho, la idea de estos juegos surgió en el proceso de rememorar nuestras propias vivencias, las noches de verano jugando con los amigos, etc. por estas razones nos decidimos a implementarlos.

A continuación se describirá cada uno de los juegos de manera detallada, con todos los algoritmos usados comentados (como pueda ser la inteligencia artificial que se ha usado en cada uno de los juegos que la tengan).

9.1 Pilla Pilla

Este juego consiste en lo siguiente, uno de los personajes (el que maneja el usuario), tiene que intentar atrapar a todos sus adversarios, que irán huyendo de él a través del

mapa de juego, en un determinado tiempo. Si transcurrido este espacio de tiempo el usuario no ha sido capaz de atrapar a todos sus oponentes, se acabará la partida mandando el correspondiente mensaje, en caso contrario, cuando atrape a todos, se le mostrará un mensaje de felicitación. En ambos casos, se saldrá a una nueva pantalla en la que se mostrará la puntuación final y podrá regresar de nuevo al menú de selección de juegos para que pueda elegir otro para seguir jugando.

Este juego hereda de la clase socketServidor2, que a su vez es un hilo creado a partir del SuperServidor que controla a todos los clientes (comentaremos estas clases más adelante). Decir que la herencia es un mecanismo que nos permite crear clases derivadas a partir de una clase base, permitiendo compartir automáticamente métodos y datos entre clases, subclases y objetos. Por ejemplo: si declaramos una clase globo de helio derivada de una clase globo general, todos los métodos y variables asociadas con la clase globo general, son automáticamente heredados por la subclase globo de helio. Entre estos métodos heredados, se encuentran los que permiten saber si nos hemos comido a un oponente, inicializar valores, etc. El juego se pondrá en marcha una vez que el SuperServidor sea capaz de discernir cuál es el juego al que el usuario quiere meterse a jugar. A continuación se crea un nuevo hilo (que es un subprograma) que será el que se encargue de atender las distintas peticiones que le lleguen del cliente (se le asignará un nuevo puerto al que dirigirse, que se le pasará como parámetro inicial a la constructora del juego) y ya por último, se le cederá el control de la partida a este nuevo hilo (así el SuperServidor podrá seguir atendiendo peticiones de más usuarios que quieran conectarse al servidor, sin necesidad de que el usuario ya conectado tenga que abandonar su partida).

Una vez unido al juego, la clase que implementa el pilla pilla irá al método que hace que comience, añadiendo elementos de tipo jugador (dependiendo de la dificultad que se haya seleccionado por parte del usuario) al ArrayList que los contendrá (aquí sólo se añaden jugadores vacíos, que se rellenarán en otro método). A continuación adjuntamos el código correspondiente.

```
public void comenzarEspecifico()
        {
         jugadores.add(new Jugador());
        //Creará tantos oponentes como dificultad tenga el juego
         for (int i = 0; i < this.dificultad; i++) {
               jugadores.add(new Jugador());
           }
        }
```

Luego, se pasará al método que inicializa las posiciones en el cual sí que se irán rellenando los jugadores anteriormente creados según sus características, dependiendo básicamente si el jugador que tenemos entre manos es el usuario o la máquina. Para

ello, se le asignará un icono (en formato GIF), un valor inicial de beta de cero grados (su visión) y una coordenada de partida (que se comprobará que no coincida con otras coordenadas de otros jugadores anteriormente fijados). El siguiente comando nos permitirá asignar un icono al jugador

```
jug.setGif(new ImageIcon(dameGif(jug.getBeta())));
```

donde jug es el jugador en cuestión que estamos tratando, dameGif es un método que dependiendo del valor de beta, nos asignará un icono y la constructora ImageIcon nos permitirá crear un icono que pertenezca a dicha clase. Cada cierto tiempo, desde el SocketServidor asociado al juego, saltará una interrupción que irá actualizando las posiciones de todos los participantes en el juego. Primero intentará mover al usuario dependiendo de que un punto caiga dentro del polígono de visión del jugador, si esta operación tiene éxito, se puede mover, en caso contrario, nos quedaremos en donde estábamos desde un principio. Luego comprobaremos si me puedo pillar a un oponente, para ello veremos si tenemos la misma coordenada que algún enemigo, actualizaremos el contador de puntos y notificaremos al usuario los cambios que han sucedido y miraremos si nos hemos comido a todos, ya que en ese caso el juego se dará por acabado de manera definitiva. Ya por último nos queda saber cómo se mueven cada uno de los adversarios. Para ello tuvimos dos versiones. En la versión inicial movíamos a todos los oponentes cada vez (comprobábamos cuál era el punto más alejado del usuario y nos movíamos hacia ese punto). Nos dimos cuenta que esta opción siendo la mejor en cuanto a inteligencia proporcionada a la máquina, no lo era en el caso de la carga de trabajo, imaginemos que nuestro ArrayList de jugadores constara de 100 enemigos y que cada vez tuviéramos que mover a cada uno de ellos de manera individual y luego tuviéramos que pasárselo al cliente, lo más seguro es que hubiera datos que se perderían durante la transmisión provocando un comportamiento incorrecto del sistema y en el ralentizado del juego, haciéndolo menos atractivo al usuario. Por ello, nos decantamos por una segunda opción más factible, sólo movemos aquellos enemigos que se encuentren demasiado cerca del usuario, sin tocar al resto. Como podemos observar, la carga de trabajo del sistema en el caso peor, es exactamente la misma que en el caso anterior pero en el resto de casos, no tendríamos que mover tantos adversarios de haciendo que el sistema funcione correctamente sin tener bajadas de rendimiento provocadas por la merma de velocidad en el juego, haciéndolo mucho más atractivo al usuario final que jugará a él.

A continuación adjuntamos el código de la inteligencia que hemos adoptado como definitiva y que hemos expuesto anteriormente de manera más detallada.

```
ArrayList posibles=new ArrayList();

 //añadimos los que estén muy cerca

 for (int i = 1; i < this.jugadores.size(); i++) {

        jug2 = (Jugador) jugadores.get(i);

        if(this.uXML.getU().distancia(jug.getPosicion(), jug2.getPosicion())<0.002){

              posibles.add(i);
```

```
            }

  }
```

Tenemos un ArrayList de candidatos a moverse, serán todos aquellos oponentes que se encuentren a una distancia inferior a 0.002 con respecto al muñeco que maneja el usuario. Una vez hecho esto, comprobamos qué jugadores forman parte del conjunto de candidatos a moverse

```
for (int i = 1; i < this.jugadores.size(); i++)     {

            if (posibles.contains(i)){

               jug2 = (Jugador) jugadores.get(i);

               calcularSiguientePosicion(jug, jug2);



}
```

y los movemos haciendo uso del siguiente código.

```
while (i < 12) {

Coordenada aux = uXML.ptoSiguiente(oponente.getPosicion(),
oponente.getBeta());

  if (aux != null) {

            auxDist = this.uXML.getU().distancia(aux, jug.getPosicion());

            if (auxDist > distanciaMin) {

                     distanciaMin = auxDist;

                     mejorCoordenada = aux;

                     mejorBeta = oponente.getBeta();

                    this.posiciones.add(0, mejorCoordenada);

            }

          }

  oponente.setBeta(((oponente.getBeta() + 10) % 360));

   i++;

   }
```

```
  oponente.setPosicion(mejorCoordenada);

  oponente.setBeta(mejorBeta);

}
```

comprobamos todas las opciones de movimiento y escogemos aquella que aleje más del usuario, una vez elegida la posición se le asignará la nueva coordenada y el nuevo beta.

Ya por último nos queda por decir que existen dos formas de terminar el juego, una de ellas es que se haya comido a todos los enemigos, con lo que se felicitará al usuario por su hazaña (se irá comprobando periódicamente al entrar en el método que hace que se muevan los jugadores) y la otra cuando salta el temporizador (que habremos activado previamente con el tiempo que queremos que dure la partida), síntoma de que se ha perdido antes de salir del juego borramos todas aquellas variables intrínsecas en el juego para que una vez volvamos a entrar todo funcione de manera correcta.

El siguiente juego que vamos a presentar es del mono platanero, juego basado en un ejercicio de una asignatura de años anteriores, para que fuese posible incluirlo dentro de nuestro servidor, tuvo que ser modificado por completo quedándonos solamente la esencia del problema (hay que ayudar a nuestro mono a capturar el mayor número de plátanos posibles).

9.2 Mono Platanero

En este juego, debemos ayudar a nuestro amigo el mono Chispita a comerse todos los plátanos que le aparezcan por pantalla en un tiempo determinado. Una vez transcurrido este periodo de tiempo, se pasará a una pantalla en dónde se visualizará los puntos conseguidos por parte del usuario y si así lo deseamos, vamos a poder regresar a la ventana principal de la aplicación para escoger un nuevo juego y así poder continuar la diversión iniciada aquí.

Este juego hereda de la clase SocketServidor que a su vez es un hilo que se lanza a ejecución por parte de la clase SuperServidor (no volveremos a mencionar qué es una clase heredada puesto que ya lo hemos definido en el juego anterior). Como en el caso anterior, hemos decidido lanzar hilos puesto que así se permite asignar un puerto a cada petición de los clientes pudiendo establecer más de una conexión de manera simultánea, es decir, permitimos que varios usuarios se puedan conectar a la aplicación a la vez sin tener que esperar a que uno de ellos acabe con su partida.

A la constructora de la clase sólo se le pasará como parámetro el número de puerto asignado y en ella se inicializará alguna de las variables que necesitaremos para desarrollar nuestro juego. Tenemos un método que sirve para comenzar la variable que contendrá los jugadores llamado comenzarEspecifico en el cual añadiremos tantos jugadores como valor tenga la constante número de plátanos (se van generando plátanos de manera periódica pero sólo mostraremos unos cuantos por pantalla para no llenar el mapa con un montón de bananas dificultando el uso del juego por parte del usuario final). Una vez realizada dicha operación, vamos a inicializar todos estos jugadores en

un método llamado inicializarPosiciones, en el cual vamos a asignar las imágenes correspondientes al jugador así como su coordenada y un valor beta inicial, además comprobaremos que no se solapen dos jugadores en la misma posición del mapa, en caso de que ocurra asignaremos otra coordenada hasta que el valor obtenido no se monte con el resto de jugadores. Aquí también iniciaremos el temporizador que nos servirá para marcar la finalización del juego. El siguiente código muestra cómo se crea un temporizador y cómo se pone en marcha.

```
tiempo2 = new Timer (60000, new ActionListener ()
      {
         public void actionPerformed(ActionEvent e)
         {
           tiempo2.stop();
           finJuego=true;
           vivoApplet=false;
           ((Jugador)jugadores.get(0)).setSuceso("¡ FIN DEL JUEGO !");
         }
      });
      tiempo2.start();
```

En la parte del SocketServidor existe otro temporizador que cada cierto tiempo irá comprobando que nos hayamos movido de sitio y si al movernos nos hemos comido un plátano. Para realizar esta comprobación, llamaremos a un método llamado actualizar posiciones, en el, comprobaremos si nuestro personaje se puede mover a una posición determinada, en caso de que así fuese nos moveremos de manera automática sin pedir confirmación de ningún tipo al usuario final. Comprobaremos si nos hemos comido una banana y por último, en el caso de que el número de plátanos en pantalla sea menor que el elegido como constante en el número de plátanos, añadiremos más al conjunto de jugadores, impidiendo que el número de plátanos totales descienda por debajo de dicho umbral.

A continuación adjuntamos el código que nos permite comprobar si nos hemos comido un plátano que se encuentra en el mapa.

```
public void pillaOponente() {
        Jugador jug=((Jugador)jugadores.get(0));
        for (int i=1;i<jugadores.size();i++){
          Jugador jug2=((Jugador)jugadores.get(i));
```

```
            if(mismaPosicion(jug.getPosicion(),jug2.getPosicion())){
              this.jugadores.remove(i);
              jug.setPuntos(jug.getPuntos()+5);
              this.numPlatanosComidos++;
              jug.setSuceso("¡ PLATANO COMIDO !");
            }
          }
        }
```

Como podemos observar, es un bucle for que comprueba si alguna de las coordenadas de las bananas coincide con la posición en la que se encuentra el mono en un instante determinado, si es así, se borra uno de los plátanos sumamos cinco puntos a la puntuación total del usuario, sumamos uno a la variable que controla el número de plátanos comidos y mostramos un mensaje al usuario. Decir que este juego no tiene ningún tipo de inteligencia ya que los plátanos no se van a mover de la posición que les haya tocado inicialmente y que no existe un método que nos indique el fin de juego en sí ya que es el temporizador del que anteriormente hemos hablado el que se encargará de ver cuándo finaliza la partida. Por último concluir, que existe un método que limpia las variables que son específicas de este juego cada vez que terminemos la partida, así no mostraremos valores incorrectos cada vez que iniciemos una nueva. A la hora de realizar este juego, han surgido algunos problemas sobre todo con el error de concurrent modification exeption, consistente en que por ejemplo dos hilos en ejecución intentan modificar a la vez una variable que comparten. Para solucionar el problema, colocamos la cláusula synchronized en el método que controla el temporizador en SocketServidor. Esta cláusula impide que los hilos entren a modificar a la vez la variable compartida, haciendo que uno de los hilos pueda modificarla pero el otro se quede a la espera sin poder realizar acción alguna hasta que el primero de los hilos termine su operación. Hay que tener mucho cuidado en el uso de synchronized ya que al parar uno de los hilos, provoca que la aplicación vaya más lenta por lo que hay que usarla solamente en trozos pequeños de código.

Para el tercero de nuestros juegos, nos basamos en uno que marcó una época, la serpiente. A éste se jugaba sobre todo en plataformas móviles extendiéndose muy rápidamente, sobre todo entre la gente joven. Dados estos antecedentes, no podíamos dejar escapar la oportunidad de homenajearlo haciendo una versión completamente diferente a las anteriores, ya que jugaremos con ella en nuestras propias calles y barrios, ganando en emoción y una mayor adicción.

9.3 Snake

Nuestra adaptación de este clásico consiste en una serpiente que se va moviendo a través del mapa de juego con el objetivo primordial de ir comiéndose al mayor número de conejos posibles durante un tiempo preestablecido. Una vez transcurrido el tiempo, se le mostrará al usuario una pantalla de puntuación y si lo desea podrá regresar al menú principal para poder seguir jugando a alguno de los otros juegos que componen nuestro servidor.

La clase que implementa este juego, hereda de SocketServidor, que a su vez es un hilo que se creará en la clase SuperServidor (ya comentamos anteriormente el significado de herencia en la programación orientada a objetos). Lo creamos en forma de hilo para que dos o más usuarios puedan estar accediendo a la vez tanto a este juego como a juegos distintos sin necesidad de tener que esperar a que uno de ellos termine su partida para que el otro usuario pueda jugar.

La constructora del juego recibirá como parámetro de entrada el número de puerto al que se debe dirigir para atender las peticiones que le lleguen por parte del cliente, siendo éste la parte activa de la comunicación. Una vez creado el juego, se irá al método comenzarEspecifico que lo único que hará será añadir todos los personajes que participarán en el juego (aunque cuando te comas a un conejo, se añadirán otros nuevos pero para no inundar el campo de juego con conejos haremos que vayan apareciendo en escena poco a poco facilitando los movimientos de usuario). Una vez que abandonamos este método, tenemos que rellenar los actores de nuestro teatro para saber quiénes son qué posiciones ocupan, etc. De esta parte se encarga inicializarPosiciones, tendremos un bucle que irá recorriendo las posiciones de ArrayList que contendrá a los jugadores y les irá asignando uno a uno la posición inicial (comprobando que dicha coordenada no coincida con otras que se estén utilizando para otro personaje) que ocuparán en el mapa así como su beta inicial y el icono correspondiente a cada uno de ellos. En este método también iniciaremos el reloj que desencadenará el final de la partida una vez que haya transcurrido un cierto tiempo, el código siguiente ilustra el modo de poner en funcionamiento el temporizador.

```
tiempo2 = new Timer (maxTimeJuego, new ActionListener ()

      {

         //entrara por aqui y habra perdido

         public void actionPerformed(ActionEvent e)

         { finJuego=true;

           vivoApplet=false;

((Jugador)jugadores.get(0)).setSuceso("HA     PERDIDO, SU TIEMPO SE HA
AGOTADO");
```

```
            tiempo2.stop();

            tiempo.stop();

           System.out.println(" ¡¡¡¡¡¡ FIN DEL JUEGO !!!!!!");

          }

       });

 tiempo2.start();
```

El código que está encerrado en actionPerformed es el que se ejecutará cuando se haya agotado el tiempo preestablecido para el reloj. Lo que hará será poner en el valor correcto a aquellas variables que sean necesarias para finalizar la partida en curso. Después de crear el temporizador debemos activarlo, cosa que hacemos con el método start. Una vez creado todos los componentes básicos de la partida, nos queda comprobar el movimiento de la serpiente a lo largo del mapa. Para ello, en la clase SocketServidor, saltará cada cierto tiempo una interrupción que llamará al método encargado de comprobar el movimiento (actualizarPosiciones). Aquí lo que haremos será comprobar básicamente si nos hemos comido a nosotros mismos dando por concluida la partida automáticamente (al igual que el juego de versión móvil). El siguiente trozo de código muestra esta opción

```
if(numCuerpos!=0){


if(mismaPosicion(coordConsulta,((Jugador)jugadores.get(1)).getPosicion())&&((Ju
gador)jugadores.get(1)).isVisible()){

              System.out.println("hello");

                 this.finJuego=true;

                 this.vivoApplet=false;

                 tiempo.stop();

                 tiempo2.stop();

                 ((Jugador)jugadores.get(0)).setSuceso("HA MUERTO, SE HA
CHOCADO CONSIGO MISMA");

       }

  }
```

Para este caso lo único que hacemos es concluir la partida poniendo el correspondiente valor a las variables que permiten acabar con ella. En el caso que sólo tengamos la cabeza de la serpiente y no nos hayamos comido ningún conejo, lo único que haremos será movernos a la siguiente posición. En cambio si por el contrario sí que ha comido

uno de los conejos, además de moverse a la siguiente posición, le crecerá un cuerpo detrás de su cabeza.

```
else if (numCuerpos==0 && haPillado){
                auxj=new Jugador();
                auxj.setBeta(jug.getBeta());

auxj.setGif(newImageIcon("C:\\hlocal\\GOPIRO\\Iconos\\Snake\\cuerpo.GIF"));
                auxj.setPosicion(jug.getPosicion());
                auxj.setPuntos(jug.getPuntos());
                auxj.setSuceso(jug.getSuceso());
                auxj.setVisible(true);


                jug.setPosicion(coordConsulta);
                jug.setGif(new ImageIcon(dameGif(jug.getBeta())));
                numCuerpos=numCuerpos+1;
                jugadores.add(1, auxj);



          }
```

Como podemos comprobar, lo que hacemos es crear un nuevo jugador (que será el cuerpo de la serpiente) le asignamos una posición (la ocupada anteriormente por la cabeza) a la cabeza se le asignará la siguiente posición y se añadirá al arrayList que lleva el conjunto de jugadores en curso, y por último se incrementa el número de cuerpos que tiene la serpiente. Otro caso, es aquel en el que hay algún cuerpo y no se ha comido a un enemigo, lo que haremos será mover cada uno de los cuerpos hacia la posición que ocupa el cuerpo o la cabeza que se encuentra delante y se moverá la cabeza a la siguiente posición. En el caso de que sí hayamos comido y tengamos uno o más cuerpos, repetiremos la acción del caso en que no teníamos ningún cuerpo. Por último si la siguiente coordenada es nula no nos moveremos pero si hay un enemigo en nuestra posición tendremos que incrementar el número de cuerpos de la serpiente.

Además existe un método que sirve para comprobar si me he comido a un rival, lo que haremos será recorrer el ArrayList para comprobar si la cabeza de la serpiente coincide en posición con alguno de los objetos restantes. En el caso de que caigamos en la

posición enemiga nos lo comeremos y haremos que dicho enemigo se ponga como no visible, lo que provocará que en la pantalla de juego no vuelva a aparecer en el caso de que no sea un enemigo es que te has chocado y por tanto la partida deberá concluir.

```
if (frutasRestantes>=3){

                ((Jugador)this.jugadores.get(i)).setPosicion(uXML.coordAleatoria());

               //   ((Jugador)this.jugadores.get(i)).setVisible(true);}

                  int j = 0;

                   Jugador jug2;

                   while (j < jugadores.size() ) {

                   if(j!=i){

                    //Comprobar si la posicion no coincide con las posiciones anteriores

                    jug2 = (Jugador) jugadores.get(j);

                    if (mismaPosicion(((Jugador)this.jugadores.get(i)).getPosicion(),
jug2.getPosicion())) {


((Jugador)this.jugadores.get(i)).setPosicion(uXML.coordAleatoria());

                      ((Jugador)this.jugadores.get(i)).setVisible(true);

                    } else {

                     j++;

                    }

                    }

                      else{

                    j++;

                    }

                 }

            }
```

Este trozo de código ilustra lo que hemos hablado con respecto a lo que hacemos cuando nos comemos a un enemigo. Existe un método de fin de juego que sólo se usará en el caso que nos hayamos comido a nosotros mismos iniciando el mecanismo que finaliza la partida. Y por último comentar la existencia de otro método de limpieza de

variables específicas para que al salir de la partida y al volver a entrar, ésta se cree de manera correcta.

Los problemas que hemos tenido a la hora de programar se deben, como en el caso del mono platanero, a la existencia del error de concurrent modification exeception resuelto de la misma manera que en el mono.

El cuarto juego que realizamos es el pincha globos. La idea de este juego surgió de una de las prácticas de programación que tuvimos en años anteriores en la cuál debíamos ir pinchando globos durante un tiempo así que decidimos adaptarlo a nuestras especificaciones e integrarlo dentro de nuestro portal de juegos online.

9.4 Pincha Globos

La adaptación de este juego consiste en ir pinchando los globos que aparecen por el mapa en un tiempo determinado. Para ello el usuario deberá pulsar el ratón encima de los objetos que irán mostrándose. Como se ha podido ver, el manejo de este juego es diferente al resto de juegos que hayamos implementado ya que introducimos el ratón como manejador en vez del teclado como sucedía en los juegos anteriores. Los globos no se moverán por una calle si no que irán saltando de calle en calle haciendo más difícil su captura.

Este juego hereda de la clase de SocketServidor que será a su vez, un hilo que se formará en la clase SuperServidor. La ejecución está en forma de hilo ya que así permitimos que dos usuarios que quieran jugar, no tengan que esperar a que uno de ellos termine con su partida si no que los dos puedan conectarse a la vez para poder jugar. Como anteriormente hemos visto, tenemos un método que se encargará de añadir los jugadores a la partida de manera vacía, es decir, sin especificar su posición inicial ni demás parámetros que hacen falta. Añadimos tantos enemigos como dificultad sea seleccionada por parte del usuario. Una vez que se hayan introducido los participantes en el ArrayList, comenzará el proceso de inicialización de éstos, esta parte se realiza en el método inicializarPosiciones. Para ello, recorreremos el ArrayList e iremos seleccionando uno a uno cada uno de los jugadores que forman parte de él y rellenaremos sus atributos de manera correcta (metiendo una coordenada inicial, comprobaremos que dos jugadores distintos no tengan la misma coordenada, añadiendo un icono y poniendo su beta inicial) así como la puesta en marcha del reloj que nos indicará cuando ha finalizado la partida una vez transcurrido un cierto período de tiempo. Una vez realizadas estas operaciones el juego está listo para comenzar. En la parte del SocketServidor, tendremos otro temporizador que nos irá marcando cuándo debemos actualizar las posiciones de todos los jugadores, de esto se encargará la función actualizarPosiciones. En ella, recorreremos todo el ArrayList de jugadores (desde la posición 1 en vez de la posición 0, puesto que esta última posición está reservada únicamente para el muñeco que representa a la persona en el juego, cosa que en este juego no existe), seleccionando una posición aleatoria del mapa (que caiga dentro de una calle) y comprobando que dicha posición no está siendo ocupada por alguien en cuyo caso se volvería a calcular otra vez otra coordenada. A continuación adjuntamos el código de esta función.

```
for (int i = 1; i < this.jugadores.size(); i++) {
        jug = (Jugador) jugadores.get(i);
        jug.setPosicion(uXML.coordAleatoria());
        int j = 1;
        while (j < i) {
           jug2 = (Jugador) jugadores.get(j);
           if (mismaPosicion(jug.getPosicion(),     jug2.getPosicion())) {
              jug.setPosicion(uXML.coordAleatoria());
           } else {
              j++;
           }
        }
     }
```

Aquí las funciones que permiten calcular la siguiente posición y la que comprueba que nos hemos comido un oponente no nos harán falta ya que para la primera, ya hemos realizado el movimiento de los globos en la función anterior al seleccionar coordenadas aleatorias para ellos (ver el código anterior). En cambio la segunda función no podemos usarla ya que nuestro juego no se maneja con el teclado si no por ratón. Para poder comprobar donde hemos pinchado en la pantalla, existe otro método llamado clickeado que nos permitirá saber cuando hemos alcanzado un globo.

Decir que este método tiene un parámetro de entrada que es la coordenada (en píxeles) de la posición que ha sido seleccionada. Lo único que debemos hacer es ir mirando cada una de las coordenadas de los globos existentes y comparar con el parámetro de entrada, decir que para alcanzar un globo no hace falta pinchar justo en el centro si no que damos un margen de error (dentro de unos límites) para que sea algo más fácil. Para poder comparar ambas coordenadas debemos pasar las del globo actual a coordenadas píxel (están en UTM) para así poder comprobarlas con las coordenadas del ratón que están en píxel. Si hemos pinchado dentro del rango, habremos acertado y por lo tanto mostraremos un mensaje de acierto amén de incrementar el número de puntos actuales que llevamos, en caso contrario, se mostrará un mensaje de fallo. Para ilustrar mejor nuestras explicaciones, vamos a mostrar la codificación de la que hemos estado hablando.

```
public void clickeado(Pixel p) {

      Jugador jug;

      Pixel aux;

      for (int i = 1; i < this.jugadores.size(); i++) {

         jug= (Jugador) jugadores.get(i);


aux=this.uXML.getC().UTMaPixel(jug.getPosicion().getLongitud(),jug.getPosicion(
).getLatitud());

         if ((p.getPixX()-10<aux.getPixX()&&
p.getPixX()+10<aux.getPixX())&&(p.getPixY()-
10<aux.getPixY()&&p.getPixY()+10<aux.getPixY())){

            ((Jugador)jugadores.get(0)).setSuceso("Ha ACERTADO!");
((Jugador)jugadores.get(0)).setPuntos(((Jugador)jugadores.get(0)).getPuntos()+5);

               this.actualizarPosiciones();

            }

            else{

            ((Jugador)jugadores.get(0)).setSuceso("Ha FALLADO!");

            }

             }

        }
```

Por último comentar que una vez transcurrido el tiempo estipulado de partida, saltará la interrupción que habíamos puesto al iniciar la partida, poniendo la variable correspondiente al fin de juego como cierta, lo que provocará que al entrar al método que comprueba si se ha llegado al final, se ejecute poniendo el valor correspondiente a las variables encargadas de terminar con el juegos así como de mostrar un mensaje de fin de juego al usuario. Una vez finalizado el juego, se mostrará una pantalla con la puntuación del usuario y éste podrá volver a jugar con otro de los excitantes juegos que conforman nuestra plataforma multijuegos. Comentar que la dificultad máxima a la hora de realizar este juego, consistía en poder tomar el valor de la posición pinchada por el ratón, ya que el evento que controla al ratón no funcionaba si se quería ver los mensajes que se iban mostrando, ya que el cursor de selección se quedaba en el panel correspondiente impidiendo que se volviera a pinchar sobre el panel principal. Este contratiempo se solucionó poniendo el panel de los mensajes como no seleccionable, así aunque pincháramos en él, el cursor no se quedaba metido en él pudiendo volver a pinchar sobre el panel del juego.

Para culminar nuestra obra, decidimos que el último juego que implementaríamos sería uno de los grandes clásicos arcade de todos los tiempos, jugado tanto por padres como por niños haciéndoles pasar más agradablemente su tiempo. Este juego debía aunar tanto pericia como reflejos como estrategia y concluimos que el único juego capaz de reunir todas estas características no era otro que el come cocos.

9.5 Come Cocos

Este juego es la adaptación del clásico Pacman que aún sigue causando furor entre los jugadores arcade. En él debemos comernos todos los quesitos que salen en el mapa evitando que nos capturen los malvados fantasmas guardianes, para nuestra ayuda contamos con las bolas de poder que nos permitirán amedrentar a estos malvados seres durante un breve período de tiempo así como darnos superpoderes que nos permitirán comérnoslos, pero cuidado si te capturan tres veces los fantasmas, el juego se habrá acabado quedando tu misión inconclusa.

La clase comecocos hereda de SocketServidor3 que a su vez es un hilo que se lanza desde SuperServidor permitiendo que dos o más usuarios entren a la vez a jugar sin tener que esperar a que uno de ellos termine con la partida en curso. Para ello, cada vez que le llegue una petición, se le asignará un puerto y se lanzará el hilo a ejecución que será el encargado de ir atendiendo a las peticiones del cliente.

Lo primero que haremos a la hora de construir el juego será asignar el número de jugadores, el primero representará al jugador persona, los cuatro siguientes a los fantasmas y las últimas cuatro posiciones a las bolas de poder e iremos introduciéndolos en el ArrayList que contiene a los jugadores. Luego iremos introduciendo en otro ArrayList los quesos que hay en el mapa (tantos como puntos tenga), más adelante explicaremos los motivos que nos llevaron a realizar esta división. Una vez realizada esta operación, nos tocará iniciar las posiciones de cada uno de los personajes así como sus iconos y demás (comprobaremos que no caigan uno encima de otro), para los quesos iremos asignando todas las posiciones del mapa sin tener que comprobar si solapan con otros jugadores (ya que al ser muchos puntos los que hay, tendríamos una gran sobrecarga de trabajo impidiendo al juego su correcto funcionamiento). Una vez realizadas estas operaciones estamos listos para empezar a jugar. En el SocketServidor3 existe un reloj que permitirá actualizar las posiciones de los jugadores en el mapa cada cierto tiempo llamando a la función actualizarPosiciones. Primero comprobaremos que el muñeco que representa al usuario esté mirando hacia la siguiente posición, en este caso lo moveremos de manera automática en caso contrario nos quedaremos en la posición actual. Más tarde iremos introduciendo en un ArrayList de candidatos aquellos fantasmas que se encuentren a distancia menor de 0,002, para ilustrar esto pondremos el código correspondiente.

```
ArrayList posibles=new ArrayList();

            for (int i = 1; i < 5; i++) {

                jug2 = (Jugador) jugadores.get(i);

              if (this.uXML.getU().distancia(jug.getPosicion(),
jug2.getPosicion())<0.002){

                    posibles.add(i);

               }

            }
```

Una vez realizada esta operación, comprobaremos cuál es la siguiente posición que debería ocupar el fantasma. Esto viene determinado por dos factores fundamentales, el primero de ellos es que si tiene o no una bola de poder, ya que en el caso de que la tenga deberá huir del usuario moviéndose a la posición más alejada que éste ocupe

```
while (i<18) {

                Coordenada aux =
uXML.ptoSiguiente(Oponente.getPosicion(),Oponente.getBeta());

                if (aux!=null){

                    auxDist=this.uXML.getU().distancia(aux,jug.getPosicion());

                     if ( auxDist>distanciaMin ){

                        distanciaMin=auxDist;

                        mejorCoordenada=aux;

                        mejorBeta=Oponente.getBeta();

                              }

                 }

                Oponente.setBeta(((Oponente.getBeta() + 20) % 360));

                i++;

            }
```

Como podemos observar, lo que hacemos es un barrido de todas las posibles posiciones que se puede mover y seleccionamos aquella que se encuentre más alejada de la actual posición del jugador principal. La cláusula i<18 es porque el ángulo de visión del jugador se mueve 20 grados 18*20=360 así evitamos ciclos y cálculos innecesarios que

llevarían a un menor rendimiento del sistemas congestionándolo. La segunda alternativa es que no tengamos en nuestro poder la bola de poder por lo que los fantasmas deberán perseguirnos, el bucle que maneja esto es similar al anterior sólo que cambiamos la condicicion auxDist>distanciaMin por auxDist<distanciaMin, ya que ahora los fantasmas deberán acercarse lo más posible al objetivo (comernos). Dentro de este procedimiento comprobaremos además si se ha comido a algún oponente o algún queso (van en procedimientos separados).

Para el caso de pillar oponente, barreremos el ArrayList de jugadores para ver si nuestra posición coincide con la de otro, podemos tener tres casos. En el primero, nuestra posición es la misma que la de un fantasma pero no tenemos bola de poder, tras lo cual se nos restará una vida que tengamos (como máximo tendremos 3), y se nos asignará otra posición del mapa para empezar (que no coincida con otro jugador). La segunda alternativa consiste en que nuestra posición es la de un fantasma pero en este caso sí que tenemos la bola de poder, tras lo cual se nos sumarán veinte puntos extras y el fantasma será descartado del juego. Y la última alternativa que nos queda es que hayamos caído en la posición de una bola de poder, con lo que activaremos el reloj que nos sirve para controlar el tiempo en el que los fantasmas huyen de nosotros así como para cambiarles la imagen usada, que nos indicará que podemos comernos a los fantasmas (transcurrido ese tiempo los fantasmas volverán a la normalidad).

El otro procedimiento mencionado es aquel que nos permite saber si hemos caído en la posición de un queso. Para ello iremos barriendo las posiciones del ArrayList que contienen a los quesos y en caso de coincidir con alguno de ellos y además que no tenga su atributo visible a falso, restaremos uno al número de quesos que nos quedan por comer, sumaremos cinco puntos extras y pondremos el atributo de visible a falso e incluiremos en la última posición del ArrayList de jugadores el número de queso que nos hemos comido. Por último tenemos un método que comprueba si hemos finalizado el juego (bien porque nos hayamos comido todos los quesos y tengamos más de cero vidas, hemos ganado, o bien porque hemos gastado todas nuestras vidas, hemos perdido).

Como hemos mencionado con anterioridad, esta forma de división (jugadores y quesos separados) no la teníamos en un principio, todo estaba junto pero nos dimos cuenta que el juego iba muy lento ya que cada cierto tiempo muy pequeño teníamos que pasar del cliente al servidor y viceversa todos estos datos, lo cual constituía una gran sobrecarga para el juego. La solución encontrada fue separar, los quesos de los jugadores, pasar al principio el ArrayList de puntos del servidor al cliente y tener una copia en ambos. Sólo modificaríamos la parte del cliente en el caso de que la última posición del ArrayList de jugadores fuera distinto de -1, lo que nos indicaría que uno de los quesos ha sido comido además de la posición que ocupa el queso, esto provoca que la parte del servidor también tiene que ir modificándose para no tener valores incoherentes pero estas operaciones siempre serán de menor coste que ir pasando todo el rato el ArrayList completo. Una segunda modificación fue la inteligencia artificial que teníamos, en un principio movíamos todos los fantasmas, provocando también una gran congestión impidiendo el normal funcionamiento del juego, ahora sólo moveremos aquellos enemigos que se encuentren muy cerca del usuario, esta operación tiene un coste menor o igual al coste de la primera operación con lo que mejoramos el rendimiento obtenido. Otro de los problemas aquí encontrados es la excepción concurrent modification

exception que fue solucionada introduciendo los synchronized correspondientes (explicados con anterioridad).

Por último mencionar el hecho de tener más de un SocketServidor, esto es así ya que hay juegos para los que no son necesarios la cláusula synchronized y dado que retarda la ejecución del programa, sería absurdo tenerla para estos juegos. En el resto de casos que sí hace falta, también puede ser que tengamos otros SocketServidor ya que el synchronized puede hacer falta en lugares distintos de código para uno y otro juego así como la necesidad de introducir más, en los juegos que no haga falta esto, provocaría una gran sobrecarga de trabajo y puesto que no se va a utilizar sería ilógico conservarlos. Pasaremos a comentar la estructura de SocketServidor (uno de ellos, ya que en el resto sólo cambiamos los synchronized) y cada uno de los métodos que aquí se implementan (y que luego heredarán cada uno de los juegos).

Primeramente, tendremos un método run que se ejecutará cuando el Superservidor genere el hilo y lo lance a ejecución. Dentro de él tendremos las órdenes de intercambio de atributos del servidor al cliente. A continuación mostramos un fragmento de dicho código.

```
obtieneFlujoObjetosOut();

synchronized(bufferObjetosOut){

this.bufferObjetosOut.writeObject(this.jugadores);

}
```

Como podemos observar, hemos puesto la cláusula sychronized para evitar que dos hilos, por ejemplo, accedan a la vez a la variable bufferObjetosOut, modificándola y provocando con ello un error de sincronización. En este método tenemos tres opciones a la hora de enviar y recibir datos. En la primera, recibiremos el nombre del mapa en cuestión y la dificultad y enviaremos los datos relativos a coordenadas máximas y mínimas del mapa, el mapa en sí y la posición inicial de los jugadores (que previamente habremos inicializado). El segundo caso es cuando el cliente ha pulsado la tecla o ha pulsado con el ratón, recibiremos la tecla pulsada o bien el píxel sobre el que se ha pinchado con el ratón y devolveremos los jugadores ya modificados. El último caso se da cuando ha saltado el timer y no se ha pulsado ninguna tecla, entonces enviamos si es o no fin de juego y enviamos las posiciones de los jugadores modificadas. Otro de los métodos que componen esta clase es comenzar. Aquí discriminamos entre si el mapa está o no personalizado, crearemos la estructura que contiene a las posiciones de los jugadores y llamaremos al comenzar específico del juego en cuestión así como al método que inicializa las posiciones. Otro de los métodos es el de limpieza de variables, que se encargará de destruir cada una de las variables que componen esta clase así como las variables que componen la clase del juego actual. El siguiente método que compone la clase es el que se refiere a la creación y aceptación de la conexión por parte del socket, para ello usaremos las siguientes primitivas.

```
socket = new ServerSocket(puerto);

cliente = socket.accept();
```

donde puerto es la variable que almacena el número de puerto en el cual el servidor se debe poner a la escucha y accept(), es la primitiva que nos permite establecer la comunicación entre el cliente y el servidor. Para poder mandar/recibir información a través de los sockets, debemos primeramente obtener los flujos de entrada o salida (según el caso), para ello utilizamos las siguientes primitivas (y usaremos una distinta en el caso que queramos mandar/recibir datos primitivos, enteros, etc. o bien queramos mandar objetos definidos por nosotros).

```
bufferOut = new DataOutputStream(cliente.getOutputStream());

bufferObjetosIn = new ObjectInputStream(cliente.getInputStream());
```

En el primer caso, sirve para obtener los flujos de salida para datos predefinidos y en el segundo caso, es para datos que hayamos definido nosotros. El siguiente método es el de mover, que dada una tecla gira el ángulo de visión del personaje en diez grados y le asigna un nuevo icono que lo represente. Sólo moverá el personaje derecha, izquierda y abajo (aunque en este último caso, no se hará nada), moverse hacia delante se realizará de manera automática a la hora de actualizar la posición en cada uno de los juegos. A continuación mostramos el código que implementa el movimiento a la izquierda del personaje, sólo mostraremos éste, porque el movimiento a la derecha es similar a éste.

```
switch (tecla) {

          //37->IZQDA, 38 ARRIBA, 39->DCHA, 40->ABAJO

           case 37:

              System.out.println("tecla pulsada :IZQUIERDA");

              jug.setBeta((jug.getBeta() + 10) % 360);

              jug.setGif(new ImageIcon(dameGif(jug.getBeta())));

               break;

}
```

El último método que nos queda por comentar de esta clase es el que se refiere al paso del tiempo y a su creación (ponerTiempo). Lo que haremos será crear un temporizador que salte cada quinientos milisegundos. Cada vez que salte, actualizará las posiciones de cada jugador de manera automática y volverá a restablecer la cuenta.

```
periodo=500;

    tiempo = new Timer (periodo, new ActionListener ()

    {

          public void actionPerformed(ActionEvent e)

          {
```

```
        try{
          synchronized(jugadores){
          if (vivoApplet ){
              tiempo.stop();
              actualizarPosiciones();
              tiempo.restart();
          }                }
          else{
            tiempo.stop();
          }
        }
         }
         catch (Exception f){
            System.out.println(f.getMessage());
         }


       }
    });
```

El resto de sockets servidores son similares a éste cambiando o añadiendo synchronized en dónde sea oportuno.

La realización de estos juegos ha puesto a prueba todos nuestros conocimientos que hemos ido adquiriendo a lo largo de la carrera así como nuestra pericia a la hora de mejorarlos para que fuesen bien y no se atascasen debido a una merma de rendimiento provocada por una sobrecarga del sistema provocando que los supuestos clientes de esta aplicación la desechasen. De esta manera consideramos que hemos conseguido que nuestra aplicación sea más atractiva.

10. Manual usuario

En la esta sección comentaremos los pasos que deberá seguir el usuario para poder jugar, desde la instalación de la aplicación a cómo usar la aplicación para poder comenzar a jugar, etc. Como primer paso comentaremos cómo funciona cada parte del sistema (cliente y servidor) y cómo arrancar cada una de estas partes.

El servidor es una aplicación que estará alojada dentro de un servidor (Apache Tomcat) implementada con Web service (comentado con anterioridad). Primeramente se deberá ejecutar esta parte para poder dar cabida a nuestra aplicación. Una vez que hayamos arrancado el servidor, ya podemos poner en marcha al cliente. Decir que la parte del cliente es una aplicación HTML (Applet) que irá pasando los datos proporcionados por el usuario (opción seleccionada, tecla pulsada o parte del mapa seleccionada con el ratón) al servidor, esta parte es la que se encargará de realizar los cálculos necesarios para calcular la siguiente posición del jugador. Una vez que haya realizado los cálculos oportunos, se pasarán los resultados otra vez al cliente que se encargará de mostrarle por pantalla al usuario dichos resultados, así iremos haciéndolo hasta que nos desconectemos definitivamente del sistema. Accederemos al sistema a través del puerto 8080, una vez dentro del sistema, se nos redigirá a través de otros puertos, por ejemplo, el súper servidor tiene de número de puerto del 45560 y como pueden jugar varios jugadores a la vez se nos irá dirigiendo a puertos de numeración sucesiva.

A continuación comentaremos de manera exhaustiva, los pasos a realizar en cada una de las pantallas y las opciones que tenemos para cada una así como las implicaciones que conllevan hasta poder empezar a jugar de manera definitiva. También comentaremos qué hacer una vez que terminemos nuestra partida para volver a jugar o para desconectarnos definitivamente del sistema.

Una vez accedido al sistema, se nos presentará una primera pantalla a modo de presentación en la que se mostrará los dibujos que representan a nuestros juegos así como uno a modo de botón, pulsando sobre dicha imagen, nos dará paso a la siguiente pantalla. Cada uno de estos iconos están modificados y retocados en PhotoShop permitiéndonos introducir efectos llamativos para cada una de ellas.

Imagen de la Pantalla de Inicio del sistema

La siguiente pantalla que se nos presentará nos dará a elegir entre dos posibilidades, elegir unos mapas que estarán predefinidos y jugar en ellos o bien descargarnos nuestro propio mapa para jugar en él. Para la primera opción, seleccionaremos *fijos* mientas que para la segunda pulsaremos *personalizados*. Una vez que optemos por una de las dos, se nos mostrará una pantalla dependiendo de la selección hecha.

Imagen de la Pantalla de la elección del tipo de mapas a jugar

En el caso de que la opción de la pantalla anterior fuera la de mapas fijos se nos aparecerá la siguiente pantalla, en la cual se han preestablecido mapas a modo de ruta turística por Madrid.

Imagen de la Pantalla de Rutas Turísticas de Madrid

Como podemos observar, cada mapa se presenta como una parada de metro del suburbano madrileño, en total habrá seis mapas de las siguientes zonas madrileñas: Oporto, Bilbao, Chamartín, Serrano, Moncloa y Atocha. También escogeremos unos de los juegos disponibles (comecocos, pincha globos, mono platanero, snake y pilla pilla) así como la dificultad que queremos para nuestro juego (alta, media, baja) se presenta como un deslizador de cara a que sea más fácil la selección por parte del usuario. Dependiendo de la dificultad aparecerán más o menos enemigos o bien habrá más o menos tiempo de finalización. Una vez elegido todo esto, pulsaremos el botón de *Jugar* que nos llevará a la pantalla de juego. También podremos pulsar el botón de *Atrás* si queremos volver a la pantalla de elección del tipo de mapas.

En el caso de que nos hayamos decantado por la otra opción, se abrirá una pantalla en la cual debemos elegir primeramente el nombre del juego al que deseamos jugar. Aparecerán el nombre de todos los juegos de manera normal, al pasar ratón por encima de ellos se agrandará el nombre y será la hora de poder seleccionarlo, en la parte de debajo de la pantalla aparecerá una breve descripción del juego que vamos a seleccionar. También existe un botón (*Atrás*) para volver a la pantalla anterior por si nos arrepentimos.

Pantalla de Plataforma sin Seleccionar

Pantalla de Plataforma Juego Seleccionado

A continuación nos aparecerá la confirmación del juego, si deseamos jugar debemos pulsar sobre la opción si, y si no, sobre la opción no. Una vez dicho que sí procederemos a la descarga del mapa de la página de OpenStreetMap. Lo primero que debemos hacer será introducir el nombre del lugar en el cual deseamos jugar. El programa se conectará automáticamente y procederá a la búsqueda de todos aquellos lugares que coinciden con el nuestro. El segundo paso tenemos que hacer será proceder a la selección de uno de los lugares devueltos para proceder a la descarga definitiva del mapa. Hay que hacer notar que el sistema tardará cierto tiempo a la hora de mostrar el juego. La última opción que nos aparece será la de la elección de la dificultad del juego, funciona igual que en el caso de los mapas fijos. Una vez que tengamos todo preparado y al igual que en el caso anterior, nos faltará por introducir los datos relativos a qué juego deseamos jugar y la dificultad que éste debe tener, con todos estos datos pulsaremos el botón de continuar para pasar a la pantalla de juego.

Imagen de la Pantalla de Mapas Customizados

Una vez que hayamos pasado por todas y cada una de las pantallas, de haber rellenado cada uno de los campos y opciones de manera correcta (no se puede, por ejemplo, dejar espacios en blanco a la hora de hacer nuestro mapa customizado) aparecerá la última y definitiva pantalla, la pantalla de juego. Ésta estará formada por una parte en donde aparecerá el mapa a modo de área de juego, una zona para cada uno de los comentarios que tendrán lugar al realizar una opción (por ejemplo, se nos mostrará un comentario cuando nos comamos un fantasma en el juego del comecocos), un bocadillo que nos mostrará el número de puntos que llevamos actualmente así como otro con el tiempo que llevamos (hemos colocado un icono de una regla en el que va pasando una pelota a modo de reloj para recalcar mejor el paso del tiempo). También se ha incluido un botón que nos permitirá abortar definitivamente (*Terminar Juego*). Hay que resaltar que para poder movernos por el mapa debemos primeramente pinchar con el ratón en la zona de juego (esto es así porque los applet de java funcionan de esta manera), una vez realizada esta operación podremos movernos por el mapa de manera normal pulsando las teclas de dirección del teclado o pulsando directamente sobre los enemigos, en el caso de usar el ratón. Como título de la pantalla aparecerá el nombre del juego actual para que el usuario sepa en todo momento en qué juego está empleando su tiempo.

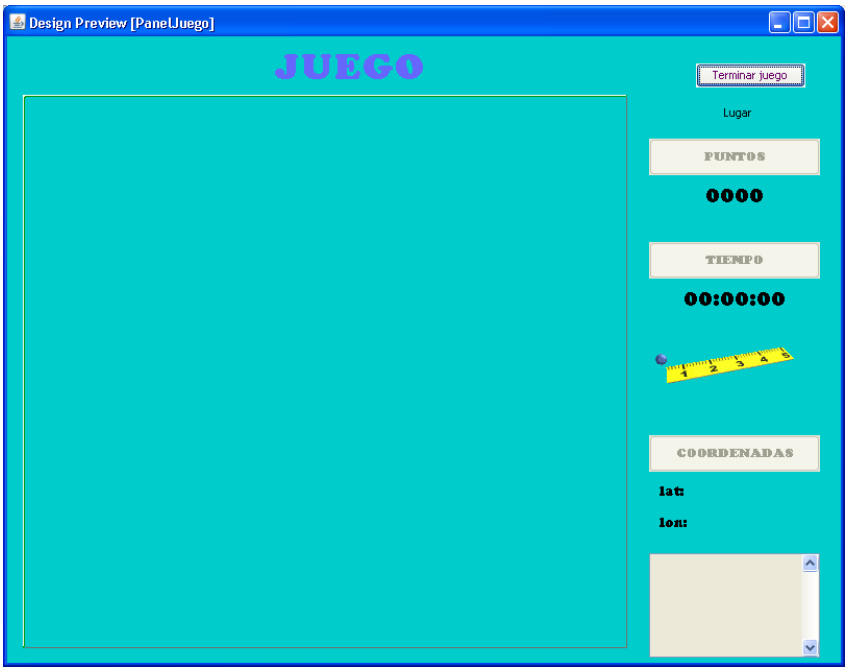


Imagen de la Pantalla de Juego

Una vez que hayamos terminado el juego (bien ganemos o perdamos) o tomemos la decisión de abortar, se nos mostrará la última pantalla del sistema. En ella se nos indicará el número de puntos que llevábamos y el tiempo empleado para conseguir dicha puntuación. Aparecerá asimismo el nombre del juego actual y un mensaje del por qué se ha finalizado el juego. El botón de A*ceptar* nos permitirá volver a la pantalla de elección de los mapas

Imagen de la Pantalla de Fin de Partida

Una vez que hemos descrito el funcionamiento de todas las pantallas que componen el sistema, hablaremos de los juegos que forman parte de nuestro servidor, dando indicaciones de cómo funcionan y el método de uso.

PILLA-PILLA

Como punto de partida, hablaremos del pilla pilla, en este juego debemos hacer que el coyote (el personaje que manejamos) logre capturar a todos los correcaminos que andan pululando por el mapa sin control. Una vez que se nos presenta el juego debemos pulsar sobre la zona de juego (una de las características intrínsecas al applet) para después movernos con las teclas de dirección del teclado. Una vez que hayamos pillado un oponente se nos sumará los puntos correspondientes y se nos mostrará por pantalla. Transcurrido un cierto tiempo, el juego concluirá. A continuación mostraremos una serie de imágenes que nos darán una idea del uso del juego.

Imagen del Juego

Imagen del Juego con Oponente Comido

Imagen Fin de Partida

MONO PLATANERO

El segundo juego que del que podemos disfrutar es el mono platanero, nuestro deber en este juego es que nuestra mascota pueda nutrirse. Para ello tenemos que guiar a nuestro mono hacia la comida con las teclas de dirección del teclado (primero debemos pulsar sobre la zona de juego para poder empezar a movernos). Una vez que hemos comido uno de estos sabrosos regalos, se nos sumarán los puntos correspondientes, transcurrido un cierto tiempo, el juego finalizará de manera automática. A continuación mostraremos un ejemplo de ejecución.

Imagen del Juego

Imagen del Juego con Plátano Comido

Imagen de fin de partida

SNAKE

El tercer juego implementado es el Snake, en este juego debemos ayudar a nuestra amada serpiente a alimentarse del mayor número de conejos posibles para evitar una plaga en la ciudad. Para ello, nos iremos moviendo a través del mapa con las teclas de dirección del teclado (primero debemos pulsar sobre la zona de juego para poder movernos). Hay que tener cuidado de no comernos a nosotros mismos ya que perderíamos la partida, de todas maneras, el juego acabará transcurrido un tiempo predeterminado. A continuación mostramos una simulación del juego.

Imagen del Juego Comiendo Enemigo

Imagen del Juego Después de Comer

Imagen Fin de Partida

PINCHA GLOBOS

El cuarto juego de los que integran nuestro servidor es el pincha globos. Nuestro objetivo será ir destruyendo el máximo número de globos posibles en un tiempo determinado. Para ello contaremos con la ayuda de nuestro ratón, ya que al contrario de los demás juegos, este se controla con dicho aparato en vez de con el teclado, pero aún así debemos pinchar la zona de juego para poder comenzar. A continuación mostramos una traza del juego en cuestión.

Imagen del Juego cuando Aciertas

Imagen del Juego cuando Fallas

COME COCOS

Por último, el juego siguiente es la versión del comecocos que hemos realizado para este trabajo. Al igual que la versión original debemos ir comiéndonos los quesitos (cuadros negros) sin que nos pillen los fantasmas guardianes. Para ello, contaremos con la inestimable ayuda de las bolas de poder que harán que dichos fantasmas huyan despavoridos debido a los nuevos poderes obtenidos. Para jugar a este juego, debemos usar las flechas de dirección del teclado, pinchando primeramente en la zona de juego para comenzar. A continuación, mostraremos imágenes de este clásico adaptado a nuestro proyecto.

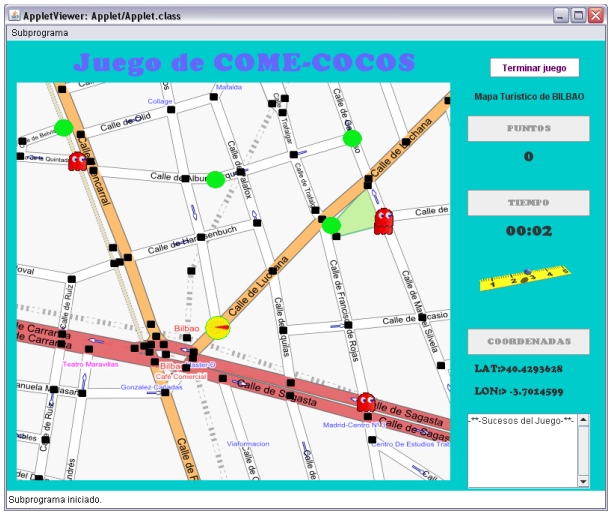


Imagen del Juego Sin Bola de Poder

Imagen del Juego Con Bola de Poder

11. Bibliografía

[1] Página oficial de Apache Tomcat http://tomcat.apache.org/

[2] Tutorial Apache Tomcat
http://www.terra.es/personal/tamarit1/instalacion_servidor/apache/index.html

[3] Para el manejo de applets http://www.programacion.com/html/tutorial/applets/

[4] http://portaljuegos.wikidot.com/tutoriales-de-tecnologias

[5] Página oficial de la fundación OpenStreetMap http://www.openstreetmap.org/

[6] Foro de OpenStreetMap http://forum.openstreetmap.org/

[7] Foro en español de OpenStreetMap http://www.nabble.com/OpenStreetMap--Spanish-List-f35211.html

[8] Wiki de OpenStreetMap http://wiki.openstreetmap.org/wiki/Main_Page

[9] Wiki de OpenStreetMap de España
http://wiki.openstreetmap.org/index.php/WikiProject_Spain

[10] Página para descargar el programa JOSM e instrucciones sobre éste.
http://wiki.openstreetmap.org/index.php/JOSM

[11] Página con informacion sobre programación, Java. Dispone de foro
http://chuidiang.com/

[12] Página web sobre coordenadas cartesianas
http://www.cartesia.org/foro/viewtopic.php?t=5949

[13] Páginas web con juegos sobre mapas de Google Maps
http://www.tomscott.com/realworldracer/

[14] http://geoquake.jp/en/webgame/DrivingSimulatorGM/

[15] Otros enlaces de interés
http://java.sun.com/docs/books/tutorial/networking/sockets/clientServer.html

[16] http://portaljuegos.wikidot.com/tutoriales-de-tecnologias

[17] http://www.geocities.com/xtr3m3_sc0rpi0/java/chat/

[18] http://www.programacion.com

[19] Ken Arnold, James Gosling, David Holmes. 2001. *El lenguaje de programación JAVA.* Pearson Educacion 3ª Edición.

[20] Todd Barron. 2001. *Multiplayer Game Programming Stary L. Hiquet Multiplayer Game Programming.*

[21] Jesús Carreter Perez, Félix García Carballeira, Pedro de Miguel Anasagasti, Fernando Perez Costoya. 2007. *Sistemas operativos. Una vision aplicada*.
Mc. Graw Hill 2ª Edición.

[22] Bruce Eckel. 2002. *Piensa en Java.* Pearson Educacion 2ª Edición.

[23] Hiroshi Maruyama, Kent Tamura, Urcmoto Nohiko. 1999. *XML and Java developing Web Applications.*Addison-Wesley 4ª Edición.

[24] Ian Millingston. 2006. *Artificial Intelligence for games.* Morgan Kaufman.

[25] Andrew Mulholland, Glenn Murphy. 2003. *Java 1.4 Game Programming.* Wordware Publishing.

[26] Elliotte Rusty Harold. 2009. *Java I/O.* O´Reilly 1ª Edición.