SISTEMAS
INFORMÁTICOS
2005 - 2006

Aplicación de métodos de Inteligencia Artificial para la toma de decisiones en simulación de móviles.

Almudena Jiménez Sierra
Fco. Javier Muñoz Rodríguez
Gonzalo Torres Porta

Dirigido por:
Prof. Gonzalo Pajares Martinsanz
Dpto. Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

SISTEMAS INFORMÁTICOS

Proyecto de Sistemas Informáticos

ÍNDICE DE CONTENIDOS

0 RESUMEN DEL PROYECTO

0.1 Resumen

Este proyecto, realizado en la Universidad Complutense de Madrid para la asignatura de Sistemas Informáticos a petición de la empresa Eads-Casa, consiste en la simulación mediante técnicas de inteligencia artificial del comportamiento de un UAV (Vehículo Aéreo no Tripulado), capaz de tomar decisiones sobre su trayectoria y encontrar un camino óptimo entre dos puntos frente a un conjunto de obstáculos y teniendo en cuenta diversos factores limitantes. El algoritmo principal está implementado en java con una interfaz gráfica en 3-D en Matlab.

0.2 Abstract

This project, carried out in the “Universidad Complutense de Madrid” for the subject of Computers Systems at request of the EADS-CASA company, consists in the simulation by means of techniques of artificial intelligence of the behaviour of an UAV (Unmanned Air Vehicle), capable of taking decisions about his path and find an optimum way between two points facing a group of obstacles and taking into account different restricted factors. The principal algorithm is implemented in java, with a graphical interface in 3-D in Matlab.

0.3 Palabras claves

Estrategia, avión, autopilotado, 3D, A*, camino, nodo, móvil, inteligente, UAV.

1 AUTORIZACIÓN

Se autoriza a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales, y mencionando expresamente a sus autores, tanto la memoria, como el código, la documentación y/o el prototipo desarrollado.

Almudena Jiménez Sierra Gonzalo J. Torres Porta Fco. Javier Muñoz Rguez.

2 INTRODUCCIÓN

2.1 Presentación

Este proyecto nació tras diversas reuniones realizadas con el profesor Gonzalo Pajares de la Universidad Complutense de Madrid, desde Junio de 2005 para asignar, durante el curso 2005-2006, un trabajo para la asignatura de Sistemas Informáticos.

Se nos presentó un proyecto, en colaboración con la empresa EADS-CASA y tras la decisión de aceptarlo en Octubre comenzamos a desarrollar.

Cuando echó a andar otro grupo de tres personas también colaboraría para el mismo departamento y proyecto con lo cual nos pusimos de acuerdo para desarollarlo en paralelo.

Durante el desarrollo del proyecto se han usado técnicas de Ingeniería del Software, tanto para especificaciones, gestiones y planificación como para el diseño haciendo uso de patrones.

Las técnicas primordiales en nuestro proyecto fueron técnicas de Inteligencia Artificial y tras tratarse de algoritmos las asignatura Metodología y Tecnología de la Programación era imprescindible su conocimiento.

2.2 Descripción del proyecto

El proyecto trabaja en la investigación de un avión autodirigido. Principalmente la rama en la que se está desarrollando es la militar, aunque posteriormente se analice la civil.

Nuestra labor ha consistido en investigación-formación-desarrollo de una aplicación que basada en algoritmos heurísticos encontrara el camino que debería llevar al avión autodirigido a través de un marco en el cual hubiera la posibilidad de encontrarse con obstáculos a evitar; ya fueran de tipo enemigo o simplemente del relieve del mapa por el que atravesara.

Este camino debe ser el más corto y más óptimo. Se nos propusieron una serie de variables que nos condicionaban la ruta del avión como el combustible, radares, montañas … que ya describiremos más adelante.

La evolución de nuestro proyecto se puede dividir en cuatro períodos:

- El primero, constituyó la formación - documentación del trabajo que teníamos que realizar así como las primeras versiones de diseños de la aplicación inicial.

- El segundo, marcó la primera versión del proyecto final en el cual el algoritmo realizado era en 2D con java, incluyendo una simple interfaz para depuraciones. Se entregó al profesor una memoria inicial con lo que llevábamos hecho, también incluimos dificultades, dudas y líneas siguientes a realizar de puntos que no estaban del todo especificados.

- El tercero, una vez construido ese algoritmo se realizaron optimizaciones y se comenzó a desarrollar en 3D. En este momento, los dos grupos se dividieron para trabajar en paralelo; el otro grupo se encargó de la interfaz y nosotros del algoritmo además de modificaciones de la anterior interfaz para poder depurar y afianzarse con la línea de desarrollo que llevábamos.

- El cuarto período se caracterizó por la integración de las dos partes hechas por los distintos grupos y su posterior optimización. En cuanto a coordinación fue la etapa más difícil de llevar a cabo debido al número elevado de componentes para desarrollar la aplicación pedida.

2.3 Organización de la memoria

La estructura de la memoria la hemos organizado según los ámbitos de trabajo que nos repartimos los dos grupos.
Una primera parte de la memoria trata de las especificación del proyecto: requisitos, objetivos, recursos …
A continuación de la especificación se habla de lo que hemos llamado cerebro de la aplicación: el algoritmo (funcionalidades, entornos de desarrollo, ejemplos de ejecución, diagramas de interacción, …)
Una vez desglosado el cerebro de la aplicación describimos la interacción exterior, es de cir, la interfaz, donde se puede observar el usuario el funcionamiento de la aplicación.
En la parte de resultados se pueden visualizar capturas de ejecución de la aplicación y objetivos cumplidos.

Una parte especial de la memoria hemos querido dedicarla a la integración que tuvimos que realizar con el otro grupo. Ya no solo es la integración sino la coordinación constante que hemos tenido que llevar a cabo, no exenta de dificultades, para poder lograr el resultado final.

Por último, se pueden ver líneas futuras que quedan abiertas para continuar con lo desrrollado hasta el momento.
Como documentación complementaria se han añadido un manual de usuario para poder hacer uso de la aplicación y una bibliografía detallada con motivo de las líneas futuras anteriormente dichas.

2.4 Objetivos propuestos

- Documentación y análisis en profundidad de distintos tipos de algoritmos heurísticos. Especialmente, el A*.

- Desarrollo de un algoritmo óptimo tanto en 2D como en 3D implementado en java que simule un avión autodirigido desde un nodo originario hasta un destino indicado.

- Propuesta, estudio y posterior inclusión de obstáculos que afecten a la dirección a tomar por el avión un camino u otro.

- Desarrollo de una interfaz 2D en java que simule el funcionamiento del algoritmo e incluya un menú para el control y manipulación del mismo.

- Posterior creación de una interfaz 2D y 3D en el entorno Matlab. Con la consiguiente interacción con java para el algoritmo.

- Análisis de las distintas variables condicionantes de un avión, centrándonos en el combustible.

- Integración, coordinación y trabajo en equipo de los dos grupos integrantes del poryecto para la asignatura de Sistemas Informáticos.

- Realización de la memoria como guía instructiva del trabajo realizado y para usos posibles de futuros avances en esta línea.

2.5 Empresa EADS-CASA

- *Acerca de EADS*

EADS es un líder global de la industria aeroespacial, de defensa y servicios relacionados. El Grupo incluye al fabricante de aviones Airbus, a Eurocopter, el mayor proveedor de helicópteros del mundo, y a la empresa conjunta MBDA, líder internacional en la producción de misiles. EADS es el socio mayoritario del consorcio Eurofighter, es el contratista principal del lanzador Ariane, desarrolla el avión de transporte militar A400M y es el socio industrial mayoritario para el sistema europeo de navegación por satélite Galileo. EADS surgió en el año 2000 de la fusión de DaimlerChrysler Aersopace AG de Alemania, la francesa Aerospatiale Matra y la española CASA.

EADS cuenta con una plantilla de unos 113.000 empleados repartidos entre más de 70 centros de producción, mayormente en Francia, Alemania, Gran Bretaña y España, así como en Estados Unidos y Australia. Una red global de 29 Oficinas de Representación mantiene el contacto con los clientes.

EADS tiene, en su sede de París, una central integrada con las funciones de Estrategia, Marketing y Asuntos Jurídicos, y en la de Munich, las de Finanzas, Adquisiciones y Comunicación.

La Dirección operativa de EADS es responsabilidad de los Chief Executive Officers Tom Enders y Noël Forgeard. Están al frente del Executive Committee (Comité Ejecutivo), que es el órgano central de dirección en el ámbito operativo. Como Chief Operating Officers (COO), Jean-Paul Gut y Hans Peter Ring son responsables del desarrollo estratégico y financiero respectivamente.

- Proyecto

Nuestra colaboración con EADS se ha reducido a realizar en el sector de inteligencia artificial el desarrollo de posibles algoritmos iniciales para la investigación del proyecto sobre el cual están trabajando ellos.

2.6 Proyecto

Es un demostrador tecnológico de un avión no tripulado de combate que tiene previsto su primer vuelo en 2010. El programa está liderado por Dassault Aviation y en él participan importantes industrias aeronáuticas de varios países europeos.

EADS CASA y Dassault Aviation firmaron el 19 de mayo, en las instalaciones de Getafe, el contrato que establece la participación de la parte española de EADS en el programa. Con este contrato EADS CASA continúa progresando en su estrategia de ser en España la empresa de referencia en los programas nacionales e internacionales de vehículos aéreos no tripulados, apoyándose para ello en las considerables capacidades tecnológicas que ha desarrollado con la integración de sistemas, comunicaciones y enlace de datos, estructuras complejas en fibra de carbono, sistemas de misión y estaciones de tierra, entre otras. Algunas de estas capacidades son las que EADS CASA aportará en los paquetes de trabajo de su responsabilidad.

Las negociaciones de este contrato han sido llevadas a cabo en estrecha coordinación con el Ministerio de Defensa Español. La división Defence and Security Systems de EADS ofrece soluciones de sistemas integrados para las nuevas misiones a desempeñar por las fuerzas armadas y las fuerzas de seguridad nacionales. Sus actividades se extienden a las áreas de aviones militares, sistemas de misiles, inteligencia, sistemas de vigilancia y reconocimiento (ISR) con vehículos aéreos tripulados y no tripulados (UAVs),

sistemas de gestión de combate, electrónica de defensa, sensores, aviónica y servicios relacionados.

El futuro sistema UAV de tipo MALE (Medium Altitude Long Endurance) con gran autonomía para todas las misiones ISTAR (Intelligence, Surveillance, Target Acquisition, Reconnaissance).

Figura 1 - *UAV Eagle 1*

*2.7 Algoritmo A**

Realizamos una pequeña introducción al algoritmo A* ya que nuestro trabajo se ha centrado en este algoritmo.

El algoritmo A* es un método de búsqueda en el cual se garantiza que siempre que haya al menos una solución, va a encontrar la mejor de ellas posible. El algoritmo en si no garantiza que la búsqueda sea mínima, pero siempre encontrará la solución óptima al problema propuesto.

La búsqueda que realiza este algoritmo se realiza a través de un espacio de estados en el que sólo se visitan posibles estados en los cuales se puede alcanzar una solución mejor que la última encontrada (en caso de ser la primera, valdría que pudiera llegar a ser solución).

Esto no quiere decir que se visiten todos ellos, sino, que se va seleccionando el nodo que parece mejor de cara a encontrar una solución óptima (basándonos en una heurística que nos determine cuánto de prometedor es un nodo) y así hasta que no haya un nodo no expandido que pueda mejorar la solución encontrada hasta el momento. Ésta es la principal diferencia con respecto a otros algoritmos de búsqueda que no tienen heurísticas, los nodos expandidos, serán en general, menos que en otros tipos de búsqueda.

2.8 Estado del arte

2.8.1 Punto de partida

Se trata de un proyecto que surge por mutuo acuerdo entre el profesor supervisor y el grupo de alumnos en el ámbito de las normas que regulan la asignatura de Sistemas Informáticos.

La idea comienza a fraguarse por la inquietud que nos suscita el mundo de la inteligencia artificial y sobre todo al ofrecernos un proyecto que nos motivó desde el principio por el hecho de su ámbito de investigación y colaboración con la empresa EADS-CASA. Madurada la idea, el proceso comienza con las reuniones pertinentes con el profesor del sector de la IA, que nos introduce en la materia, proporcionándonos principalmente la documentación y referencias pertinentes.

Existe en la literatura una gran cantidad de trabajos relacionados con algoritmos similares basados en A*, si bien dado que el objetivo principal del proyecto no es la investigación, nos centramos en las referencias básicas que aparecen en la sección bibliografía.

No obstante, este proyecto parte de cero, con su pertinente fase de análisis y documentación, y asentando las bases para posteriores ampliaciones o como integración de trabajos futuros.

2.8.2 Conocimientos previos

En el desarrollo del proyecto ha sido necesario el uso e integración de conocimientos adquiridos previamente en asignaturas impartidas a lo largo de la carrera de Ingeniera Informática Superior.
De esta forma el proyecto se ha fundamentado en conocimientos previos de asignaturas como Inteligencia Artificial, Ingeniería del Software, Metodología y Tecnología de la Programación, Estructura de Datos y de la Información y los distintos laboratorios de programación.

3 Especificación

3.1 Organización del terreno

El terreno sobre el que se va a ejecutar el algoritmo se organiza diferenciando una parte estática, que son los obstáculos infranqueables que determinan el terreno fijo y que es constante durante toda la ejecución (montañas, edificios, suelo raso…), y obstáculos variables, que se añaden al anterior, y que influyen en la toma de decisiones del algoritmo. Estos últimos influyen de forma que la decisión tomada pueda ser más o menos óptima, decidiendo si se expanden los nodos a través de ellos dependiendo de los diferentes pesos que puedan tener, mientras que los primero limitan la expansión de nodos haciéndose infranqueables.

- En esta imagen se muestra una parte del terreno estático, formado por diferentes relieves los cuales el objeto móvil no puede traspasar.

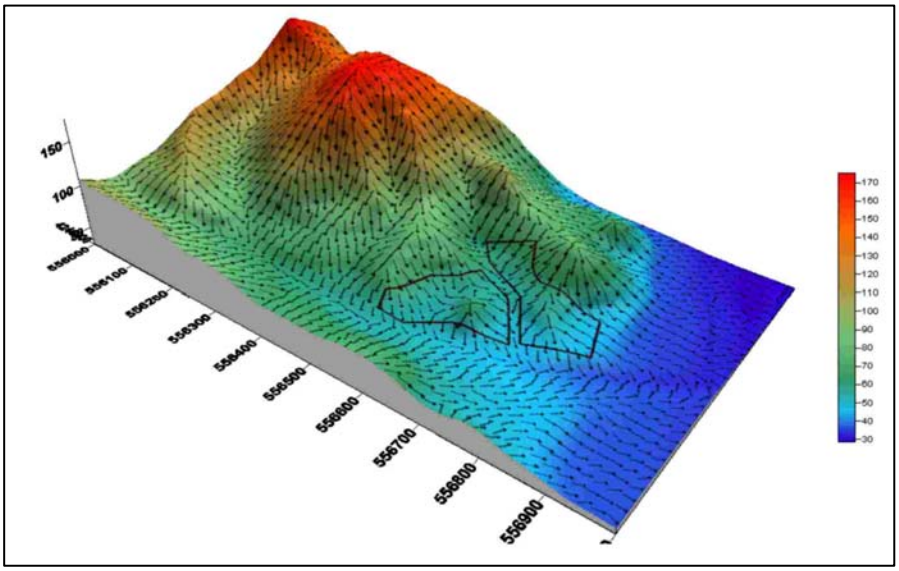


Figura 2 – *Terreno 3D*

- Sobre el terreno fijo se insertan elementos (radares) que influyen en la toma de decisión del camino elegido. Estos obstáculos se asientan sobre el anteriormente mencionado terreno estático y su función consiste básicamente en modificar los pesos de los nodos a los que les llega su radio de influencia, dependiendo de la cercanía al epicentro.

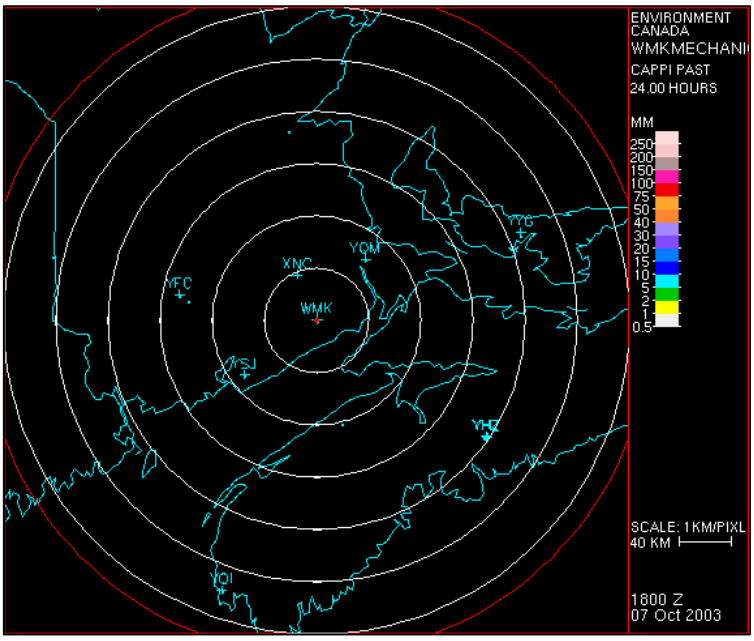


Figura 3 *- Ejemplo Radar*

3.2 Obstáculos

Como obstáculos definimos cualquier elemento de nuestro entorno que influya en el camino que decida tomar el avión para dirigirse a su destino u objetivo.
Hemos definido distintos tipos de obstáculos:

- Obstáculos con distintos pesos:

Cada obstáculo, dependiendo de su riesgo, tendrá mayor o menor coste asociado. Por ejemplo, un obstáculo infranqueable, como puede ser una montaña, tendrá un coste mayor que cualquier otro tipo de obstáculo ya que no puede ser atravesado; el algoritmo A* implementado lo que hará es meterlo directamente en la lista de cerrados para no expandir este tipo de nodos. Sin embargo, una zona alejada de un radar tendrá un coste proporcional al riesgo de

atravesar esa zona, el y en determinadas circunstancias, el algoritmo lo puede elegir para incluirlo dentro del camino.
Según esto tenemos los siguientes tipos de obstáculos según su peso. Éste está normalizado en la escala 0-1, con lo cual tendremos:

* Bajo = (0,0.25)
* Medio = [0.25,0.5)
* Alto = [0.5,0.75)
* Infranqueable = [0.75,1]
* Libre de obstáculo = 0

- Obstáculos con pesos variables :

En sí no son propiamente obstáculos, nos queremos referir a la zona de acción de los mismos. A la aplicación se le podrán incluir obstáculos con un rango de acción más grande que otros y cuyo coste irá bajando a medida que se aleje de su centro (Radar).

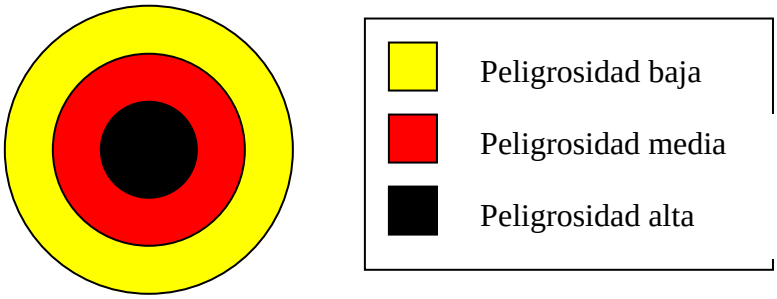


Figura 4 *– Ejemplo obstáculo radar*

En la figura 1, el centro del radar, representado con el color negro, hará que el avión sea descubierto y con el consiguiente peligro de ser derribado con mayor probabilidad que la zona marcada en color rojo y ésta a su vez mayor que la zona de exclusión amarilla.

3.3 Escenarios de Simulación

La falta de disponibilidad de herramientas software similares a la que se está desarrollando, hace imposible la validación de las soluciones por enfrentamiento de soluciones automáticas. Por este motivo, en este capítulo se pretende aportar una guía de validación para los casos de simulación resueltos por el software desarrollado. Esta guía no pretende ser rigurosa, pero sí pretende constituir al final del proyecto una guía de validación completa para poder ser utilizada a modo de metodología en futuros proyectos similares, con lo que este proceso deberá ser parte activa y cambiante del proyecto.

Como propuesta inicial, se apuesta por cuatro tipos de validación:

- Por enfrentamiento a cualquier software de cálculo de rutas óptimas en un escenario de obstáculos,
- Contra respuestas de un experto,
- Por enfrentamiento de casos, con el fin de comprobar la capacidad de reacción ante obstáculos (previstos o no) de los algoritmos seleccionados, y
- Por enfrentamiento contra la propia herramienta una vez establecido un nivel de conocimiento adecuado.

Todas las simulaciones realizadas deberán tener como objetivo principal el cumplimiento de su misión de acuerdo a los requisitos impuestos por el proyecto, teniendo en cuenta la medida de mérito especificada en mismo, y en la medida de lo posible criterios de mínimo combustible o tiempo. Cabe destacar dos características:

- Todas las simulaciones deberán considerar el cumplimiento de la misión desde el punto inicial al final aunque eso requiera pasar dos veces por la zona de amenazas.
- Las dimensiones del espacio de simulación podrán ser configuradas por el usuario.

A continuación se han establecido algunos ejemplos dentro de las categorías de validación mencionadas. En ellos, se utilizarán los siguientes elementos de representación:

Representan el radio de alcance del misil. Teniendo en cuenta este radio el vehículo deberá estimar (en el caso de conocer la composición del ADU), los radios, alturas y ángulos relativos a él, para poder realizar sus cálculos de estrategia de ruta teniéndolos en cuenta. Un código de colores indica el grado de riesgo por comparación entre unas y otras amenazas, de tal forma que los tonos claros indican bajo riesgo relativo a los tonos oscuros que indican alto riesgo.

Representa el mismo radio que en el caso anterior sólo que para un ADU que funciona en pop-up.

Representan zonas de aparición de posibles ADUs de forma aleatoria y desconocida a priori.

Ejemplo de ruta del vehículo que cumple la resolución del escenario.

X Objetivo de la misión como punto de localización al que debe llegar el vehículo.

A continuación, se muestra los distintos tipos de validación de rutas que puede tomar el ADU.

3.3.1 Por caminos geométricos

El escenario al completo será conocido. El vehículo aéreo conocerá las posiciones y especificaciones de las ADUs (radios de alcance) localizadas en el entorno de su misión y por lo tanto, podrá calcular las rutas geométricas fuera de riesgo tanto en planta como en alzado. El vehículo deberá encontrar la ruta óptima de acuerdo al objetivo de su misión, función de mérito y criterios de priorización (mínimo combustible o mínimo tiempo, por ejemplo). Ejemplos para este tipo de escenarios pueden ser los mostrados en las siguientes figuras.

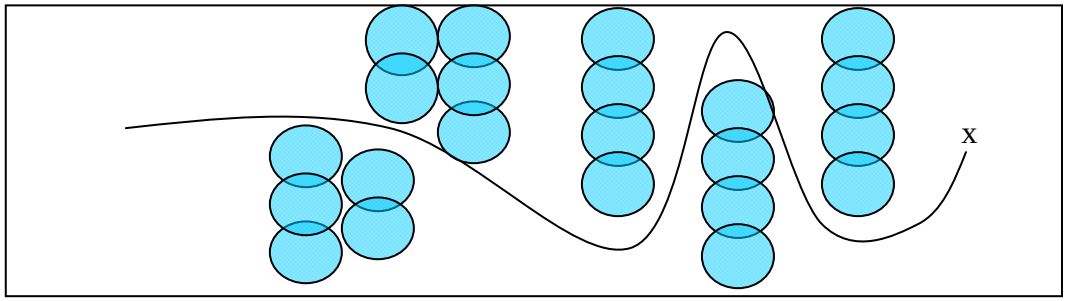


Figura 5 - *Ejemplo1 de camino*

El dispositivo móvil sortea los obstáculos, comprometiendo factores como el combustible, por ejemplo, a favor de encontrar un camino por la mejor ruta posible. Las figuras siguientes muestran como el avión sortea los obstáculos; si puede no los atravesará ya que es posible que alcance el objetivo por un camino alternativo, bien con coste mayor o menor pero sin arriesgar su pilotaje.

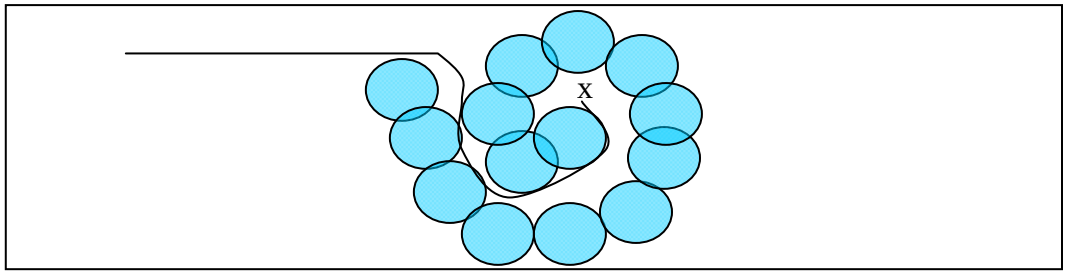


Figura 6 - *Ejemplo2 de camino*

En este segundo caso, al no tener alternativa el avión debe atravesar una zona de riesgo para poder alcanzar el objetivo. Que la atraviese o no depende de cómo de arriesgado sean dichos obstáculos, tipo de nivel de peligrosidad.

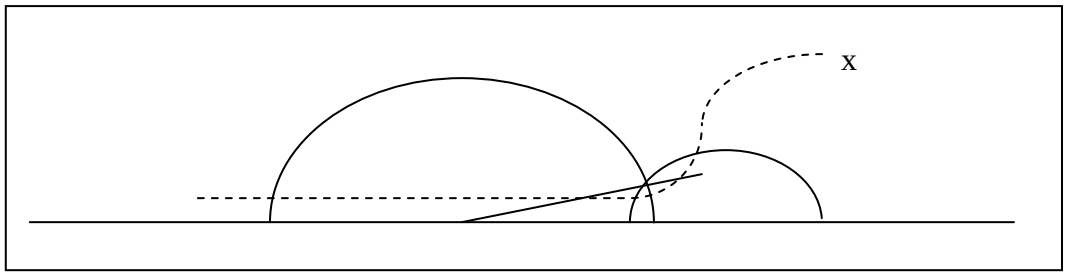


Figura 7 - *Ejemplo3 de camino*

3.3.2 Por lógica de decisión

En este caso el vehículo debe hacer uso de todo su conocimiento sobre las ADUs situadas en el entorno para tomar las decisiones en la construcción de su ruta óptima. Los escenarios para este tipo de pruebas deberán tener en cuenta la probabilidad de letalidad de cada uno de los ADUs en función de sus radios y alturas. Para obligar al vehículo tomar una decisión asumiendo un riesgo, no debe existir en este tipo de escenario ningún camino libre de amenazas, el vehículo deberá encontrar el camino de mínimo riesgo, maximizando su función de mérito y criterios de optimización.

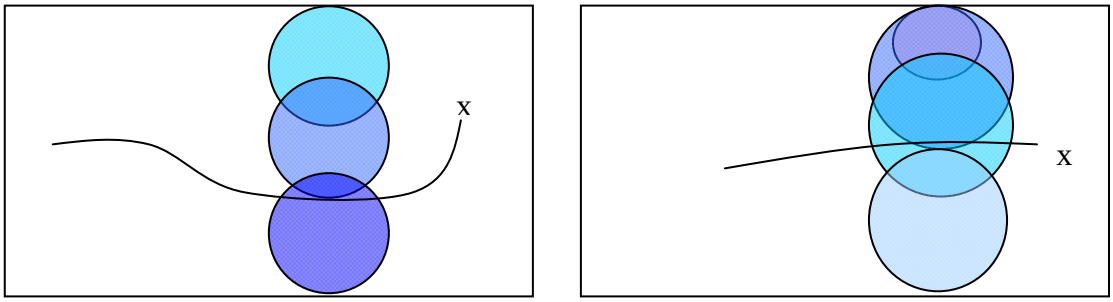


Figura 8 *- Ejemplo4 de camino*

Nota: *Estos ejemplos son sólo significativos como ejemplos, no aseguran que la ruta mostrada sea la óptima ya que dependerá del cálculo de riesgos particular de cada caso.*

3.3.3 Por enfrentamiento de casos

Por enfrentamiento de casos se pretende observar la reacción entre unos escenarios y otros al cambiar ADUs conocidas por ADUs aleatorias desconocidas (previamente) por el vehículo, o introducir pasillos de prohibición de paso.

A continuación se muestran algunos ejemplos (a la izquierda situación inicial, a la derecha situación nueva).

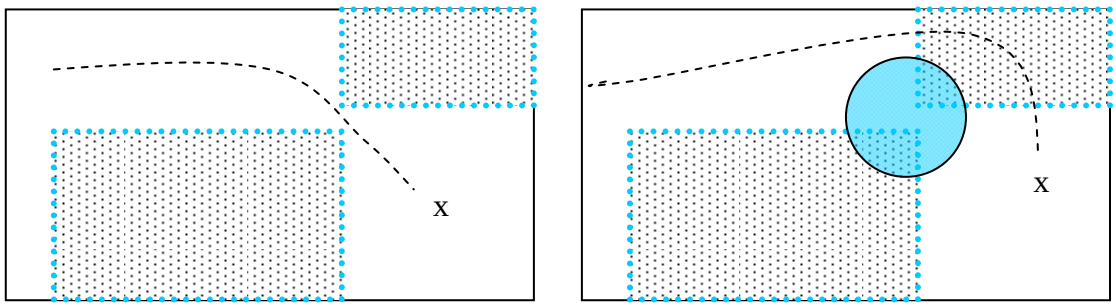


Figura 9 *- Ejemplo1 enfrentamiento de casos*

En un primer momento el móvil elegiría pasar por el pasillo libre que le conduce a su objetivo, pero tras introducir un obstáculo de mayor peso justo por ese camino, la toma de decisión cambiaría y rodearía este nuevo obstáculo por la zona que menos peligro representase (en este caso por su izquierda).

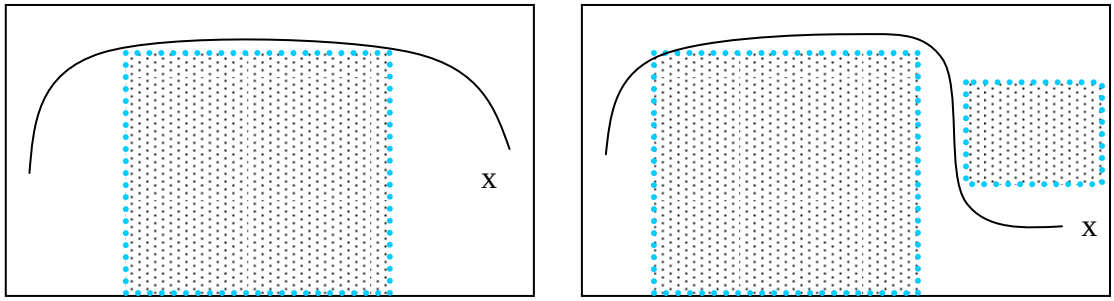


Figura 10 - *Ejemplo2 enfrentamiento de casos*

En este caso, al incluir un obstáculo justo antes del nodo objetivo, obliga al móvil a realizar un rodeo que alarga su camino hacia el destino.

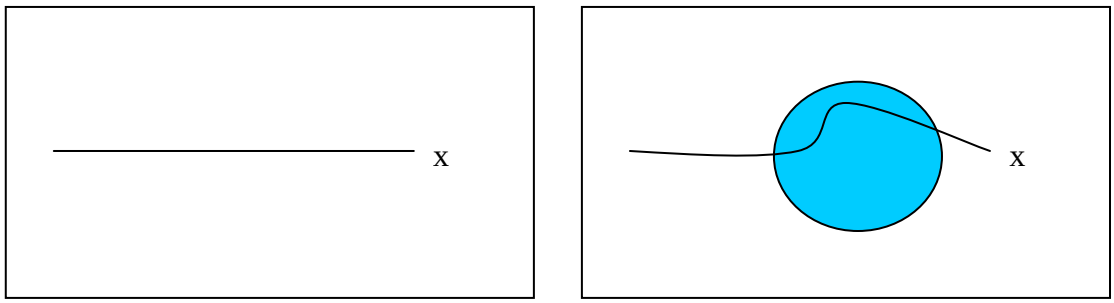


Figura 11 - *Ejemplo3 enfrentamiento de casos*

La inclusión de un obstáculo que en medio de la trayectoria del móvil obliga a rehacer el camino, sopesando diferentes factores como son la distancia al destino, la cantidad de combustible restante, el peso de la peligrosidad de los nodos, evitando en este caso el epicentro pero rodeándolo por zonas no completamente “libres”

3.4 Plataforma de desarrollo

En este apartado se muestran los lenguajes empleados para desarrollar la aplicación, comenzando por una visión general, pasando por sus orígenes, bases, evolución y finalizando en el uso de los mismos.

3.4.1 Orígenes de Java

Java tiene su origen en 1991 [W3UPM], cuando un grupo de investigadores de Sun Microsystems, liderado por James Gosling, trabajaba en técnicas para producir software capaz de ser incluido dentro de cualquier aparato electrónico (teléfonos, faxes, videos y electrodomésticos en general) para proveerlos de “inteligencia”. La reducida potencia de cálculo y memoria de estos aparatos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido. En principio, consideraron la posibilidad de utilizar lenguajes ya existentes como C++ o Smalltalk, pero pronto se hizo patente la necesidad de definir una máquina hipotética o virtual capaz de garantizar la portabilidad de las aplicaciones debido a la existencia de distintos tipos de CPU [W3UPM] y a los distintos cambios, a la vez que cumpliera los requisitos de seguridad derivados de su uso en dispositivos de uso común.

Por tanto, Java es el lenguaje de programación y, además, la definición de la máquina virtual (Java Virtual Machine) encargada de ejecutar las aplicaciones. A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje, por lo que iba a caer en desuso dado que los directivos de Sun no veían un mercado potencial y el grupo fue disuelto. Sin embargo, Gosling intuyó las posibilidades del lenguaje para proporcionar contenidos activos en las páginas Web y se embarcó en la construcción de un *browser* (navegador) basado en Java, HotJava, que constituyó la demostración general de las capacidades del lenguaje, comenzando las historia de Java como lenguaje íntimamente ligado a internet. Como lenguaje de programación se presentó a finales de 1995 y desde entonces ha sido uno de los temas que más interés ha despertado en el mundo de la informática, mereciendo, incluso, la atención de publicaciones no especializadas. La clave fue la incorporación de un intérprete Java en el programa Netscape Navigator, versión 2.0, produciendo una verdadera revolución en internet. La mayor parte de las aplicaciones Java se utilizaron para dotar de contenido dinámico e interactivo a las páginas del World Wide Web, mediante los llamados *applets*. Posteriormente el uso de Java se fue extendiendo a un gran número de sistemas y aplicaciones, en las que el modelo ofrecido por Java de un entorno distribuido y completamente transportable entre plataformas es enormemente atractivo.

Java 1.1 apareció a principios de 1997, mejorando sustancialmente la primera versión del lenguaje, encontrándose en el momento de escribir este documento por la versión 1.4.2.09

3.4.2 Principales características de Java

Java es un lenguaje de programación orientado a objetos, con una sintaxis similar a C o C++, pero ofreciendo una mayor simplicidad y robustez en el ciclo de desarrollo: las construcciones y características más complicadas de C y C++ han sido eliminadas y el lenguaje contiene mecanismos implícitos para garantizar la seguridad de las aplicaciones construidas con él.

También incorpora dos mecanismos a la hora de escribir programas simples, potentes y robustos: un tratamiento interno de multitarea y un sistema de excepciones que normaliza el procesado de errores por parte del programador. La principal característica de Java es que es independiente de la plataforma, pudiendo ejecutarlo sobre distintas arquitecturas y sistemas operativos sin que sea necesario modificar el código del programa. Esta independencia se logra debido a que el lenguaje está soportado por dos elementos fundamentales: el compilador y la máquina virtual. El compilador traduce los programas a un formato especial llamado *bytecodes*, que es el formato que se le pasa a la máquina virtual. Tanto el compilador Java como la máquina virtual son específicos para cada plataforma, por lo que para poder ejecutar un programa Java en una determinada plataforma debe existir previamente una máquina virtual para ella, por lo que Sun Microsystems dispone de un entorno de ejecución para la mayoría de las plataformas.

Otra ventaja es que cuenta con un gran número de clases preexistentes que no deja de aumentar, presentando una gran riqueza en cuanto al tipo de funciones que permiten realizar.

Sun describe el lenguaje Java como *"simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámico"* [W3SUN]:

• **Simple**: Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos. Los lenguajes más difundidos son C y C++, pero adolecen de falta de seguridad, característica muy importante para los programas que se usan en *Internet*, por ello Java se diseñó para ser parecido a ellos y así facilitar un rápido y fácil aprendizaje.

Java elimina muchas de las características de otros lenguajes [FRO00] para mantener reducidas las especificaciones del lenguaje y añadir características muy útiles, como el *garbage collector* (reciclador de memoria dinámica), que se encarga de liberar memoria no usada.

Java reduce en un 50% los errores más comunes de programación con lenguajes como C y C++ al eliminar muchas de las características de éstos, entre las que destacan que:

- no se admite aritmética de punteros
- no existen referencias
- no existe la definición de registros (struct)
- no existe la definición de tipos (typedef)

- no existe la definición de macros (#define)
- no existe la necesidad de liberar memoria (free)

• **Orientado a objetos:** Java implementa la tecnología básica de C++ con algunas mejoras y elimina algunas cosas para mantener el objetivo de la simplicidad del lenguaje. Java trabaja con sus datos como objetos y con interfaces a esos objetos.
Soporta encapsulación, herencia y polimorfismo. Hace uso de la definición de entidades formadas por métodos y variables que reciben el nombre de clases, la instancia de una clase recibe el nombre de objeto.

• **Distribuido:** Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como http y ftp. Esto permite a los programadores acceder a la información a través de la red con tanta facilidad como a los ficheros locales. Java en sí no es distribuido, sino que proporciona las librerías y herramientas para que los programas puedan ser distribuidos.

• **Interpretado:** La máquina virtual Java es un programa que se ejecuta sobre el sistema operativo del ordenador (por lo que es dependiente de la plataforma) y ejecuta directamente el código objeto mediante la interpretación de los *bytecodes*, aunque no se trata de un intérprete tradicional pues éstos ya han pasado por las etapas de validación del compilador Java.

• **Robusto:** Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador de la liberación o corrupción de memoria. Además, para asegurar el funcionamiento de la aplicación, realiza una
verificación de los *bytecodes*, que son el resultado de la compilación de un programa Java.

• **Seguro:** La seguridad en Java tiene dos facetas. Por una parte se eliminan características C y C++ para prevenir el acceso ilegal a la memoria como los punteros o el casting implícito. Por otra parte, el código Java pasa muchos tests antes de ejecutarse en la máquina virtual. Pasa a través de un verificador de *bytecodes* que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código que falsee punteros, viole derechos de acceso sobre objetos o intente cambiar el tipo o clase de un objeto. Si los *bytecodes* pasan la verificación sin generar ningún mensaje de error, entonces sabemos que:

- El código no produce desbordamiento de operandos en la pila.
- El tipo de los parámetros de todos los códigos de operación son conocidos y correctos.
- No ha ocurrido ninguna conversión ilegal de datos.
- El acceso a los campos de un objeto se sabe que es legal.
- No hay ningún intento de violar las reglas de acceso y seguridad.

El Cargador de Clases también ayuda a Java a mantener su seguridad, separando el espacio de nombres del sistema de ficheros local del de los recursos procedentes de la red. Esto limita cualquier aplicación del tipo Caballo de Troya, ya que las clases se buscan primero entre las locales y luego entre las procedentes del exterior. Las clases importadas de la red se almacenan en un espacio de nombres privado, asociado con el origen. Cuando una clase del espacio de nombres privado accede a otra clase, primero se busca en las clases predefinidas (del sistema local) y luego en el espacio de nombres de la clase que hace la referencia. Esto imposibilita que una clase suplante a una predefinida. En resumen, las aplicaciones de Java resultan extremadamente seguras.

• **Arquitectura neutral:** El compilador Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en que se ejecutará (este fichero con tiene los *bytecodes*). Cualquier máquina que tenga el sistema de ejecución (*run-time,* que sí es dependiente de la máquina) puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado.

• **Portable:** Más allá de la portabilidad básica por ser de arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros de 32 bits en complemento a 2 y las cadenas de caracteres utilizan Unicote (no ASCII). Además, Java construye sus interfaces de usuario a través de un sistema abstracto de ventanas de forma que las ventanas puedan ser implantadas en entornos diferentes (Unix, Pc, Mac, etc.).

• **Altas prestaciones:** Para los casos en que la velocidad del intérprete Java no resulte suficiente, existen mecanismos como los compiladores **JIT** (Just In Time), que se encargan de traducir, a medida que va siendo necesario, los *bytecodes* a instrucciones de código máquina. También existen otros mecanismos como los compiladores incrementales y sistemas dedicados para tiempo real.

• **Multitarea:** Java permite muchas actividades simultáneas en un programa. Los *threads* son pequeños procesos o piezas independientes de un gran proceso. Al estar los *threads* construidos en el lenguaje, son más fáciles de usar y más robustos que sus homólogos en otros lenguajes que no los soportan de manera nativa. Esta característica permite mejorar el rendimiento interactivo y el comportamiento en tiempo real.

• **Dinámico:** Java se beneficia todo lo posible de la tecnología orientada a objetos. Java no intenta conectar todos los módulos que comprenden una aplicación hasta el tiempo de ejecución. Las librerías nuevas o actualizadas no paralizarán las aplicaciones actuales (siempre que mantengan el API anterior). Esto permite actualizar el código “en caliente” y facilita el mantenimiento del software. Por otro lado, Java proporciona mecanismos para cargar dinámicamente clases desde la red, de manera que nuevos contenidos de información podrán ser tratados por manejadores específicos.

3.4.3 Descripción de Matlab

MATLAB es un entorno de cálculo técnico de altas prestaciones para cálculo numérico y visualización. Integra:

- Análisis numérico
- Cálculo matricial
- Procesamiento de señales
- Gráficos

En un entorno fácil de usar, donde los problemas y las soluciones son expresados como se escriben matemáticamente, sin la programación tradicional. El nombre *MATLAB* proviene de ``MATrix LABoratory'' (Laboratorio de Matrices). *MATLAB* fue escrito originalmente para proporcionar un acceso sencillo al software matricial desarrollado por los proyectos *LINPACK* y *EISPACK*, que juntos representan lo más avanzado en programas de cálculo matricial. *MATLAB* es un sistema interactivo cuyo elemento básico de datos es una matriz que no requiere dimensionamiento. Esto permite resolver muchos problemas numéricos en una fracción del tiempo que llevaría hacerlo en lenguajes como *C*, *BASIC* o *FORTRAN*. *MATLAB* ha evolucionado en los últimos años a partir de la colaboración de muchos usuarios. En entornos universitarios se ha convertido en la herramienta de enseñanza estándar para cursos de introducción en álgebra lineal aplicada, así como cursos avanzados en otras áreas. En la industria, *MATLAB* se utiliza para investigación y para resolver problemas prácticos de ingeniería y matemáticas, con un gran énfasis en aplicaciones de control y procesamiento de señales. *MATLAB* también proporciona una serie de soluciones específicas denominadas *TOOLBOXES*. Estas son muy importantes para la mayoría de los usuarios de *MATLAB* y son conjuntos de funciones *MATLAB* que extienden el entorno *MATLAB* para resolver clases particulares de problemas como:

- Procesamiento de señales
- Diseño de sistemas de control
- Simulación de sistemas dinámicos
- Identificación de sistemas
- Redes neuronales y otros.

Probablemente la característica más importante de *MATLAB* es su capacidad de crecimiento. Esto permite convertir al usuario en un autor contribuyente, creando sus propias aplicaciones. En resumen, las prestaciones más importantes de *MATLAB* son:

- Escritura del programa en lenguaje matemático.
- Implementación de las matrices como elemento básico del lenguaje, lo que permite una gran reducción del código, al no necesitar implementar el cálculo matricial.
- Implementación de aritmética compleja.
- Un gran contenido de órdenes específicas, agrupadas en *TOOLBOXES*.
- Posibilidad de ampliar y adaptar el lenguaje, mediantes ficheros de script y funciones .m.

3.4.4 Historia de Matlab

Cleve Moler escribió el *MATLAB* original en *FORTRAN*, durante varios años. Los algoritmos matriciales subyacentes fueron proporcionados por los muchos integrantes de los proyectos LINPACK y EISPACK. El *MATLAB* actual fue escrito en *C* por The Mathworks. La primera versión fue escrita por:

- Steve Bangert, que escribió el intérprete parser
- Steve Kleiman que implementó los gráficos
- John Little y Cleve Moler que escribieron las rutinas de análisis, la guía de usuario y la mayoría de los ficheros .m.

Desde la primera versión muchas otras personas han contribuido al desarrollo de *MATLAB*.

4 PARTE 1 – Cerebro de la aplicación

4.1 Plataforma de desarrollo

Como entorno de desarrollo hemos utilizado la herramienta de libre distribución Eclipse. El Eclipse Project (en inglés) es un proyecto de desarrollo de software de fuente abierta dedicado a proporcionar una plataforma de mercado robusta, con recursos completos, para el desarrollo de herramientas altamente integradas. Parte del proyecto Eclipse es el Eclipse Modeling Framework (EMF), que ofrece una serie de capacidades esenciales, notablemente generación de código, administración de metadatos, serialización de default y soporte de editor.
Por lo tanto hemos elegido esta herramienta, muy utilizada actualmente en sectores del desarrollo software, por su robustez en los sistemas que implementa y por las posibilidades y versatilidad que nos aporta.

4.2 Evolución durante el desarrollo del proyecto

Al ser un proyecto que parte de cero, la primera parte consiste en una labor de documentación y análisis del problema propuesto y cómo abordar las diferentes problemáticas que se nos presentan. Ésta es una labor tediosa y lenta, donde nosotros mismos hemos de dejar especificado completamente el trabajo a realizar antes de empezar a desarrollar. Una de las partes fundamentales en esta parte es cómo abordar el algoritmo A* para este problema en concreto y las variables que van a limitar su admisibilidad, de tal forma que optimicemos su funcionalidad.

Ya teniendo especificados todos los requisitos y con una idea sobre como modular las diferentes partes, y abordado el problema de la implementación del algoritmo principal, comenzamos el desarrollo, con una interfaz gráfica básica en dos dimensiones que utilizamos básicamente como plataforma de pruebas y depuración.

Una vez conseguido modelar el “cerebro” y a falta de añadir variables que consideren diferentes tipos de obstáculos y su adaptación a tres dimensiones, se comienza el desarrollo de una interfaz final para la simulación completa del algoritmo. Por lo tanto en esta fase es donde más se trabaja en paralelo, por una parte la implementación de la interfaz, donde se incluirán los diferentes tipos de obstáculos y una visión “realista” y por otra la depuración del algoritmo y su adaptabilidad a dicha interfaz. Durante esta fase es necesaria una comunicación entre ambas partes y donde se han utilizado técnicas aprendidas en asignaturas de la carrera como es Ingeniería del Software.

Terminado el proceso de desarrollo, llegamos al momento más complicado de la integración de ambas partes, donde hizo falta realizar pequeñas modificaciones para la adaptación, implementadas en diferentes plataformas de desarrollo. La ventaja que habíamos previsto era que Matlab con Java son

compatibles y se facilita mucho en este sentido el proceso de integración de una a la otra, dando como resultado el cerebro final de la aplicación.

4.3 Algoritmo empleado

Para el desarrollo de nuestro proyecto elegimos el algoritmo A* debido a las características que fueron expuestas en el apartado 1.7 de la memoria.

El uso de este algoritmo es fundamental para la implementación, ya que toda ella tiene como núcleo el A*. En el proyecto se utiliza para buscar un camino entre dos puntos del espacio a considerar (origen/destino del avión) que sea el que suponga un coste mínimo, teniendo en cuenta las restricciones tanto del terreno como de variables externas o internas al avión.

El algoritmo A* garantiza encontrar el camino óptimo siempre que lo haya y de manera más eficiente en general que otros algoritmos.

El algoritmo A* va acompañado de una función heurística. La heurística será la responsable de guiar al algoritmo para que expanda uno u otro nodo. La heurística por tanto está relacionada directamente con la eficiencia.

El funcionamiento global del algoritmo es el siguiente:

- Inicialmente cogemos como nodo a expandir el origen (ya que no hay otro).

- Expandimos este nodo generando así todos sus hijos (en nuestro caso a donde puede ir en un solo movimiento el avión) que cumplen que puede haber un camino al destino mejor que el hasta ahora encontrado. Destacar que en la versión 3D que explicaremos más adelante el número de nodos a expandir será lógicamente muy superior al algoritmo 2D.

- Elegimos el que supuestamente vaya a tener un coste menor de entre todos los nodos a los que podemos llegar desde los nodos expandidos y que no han sido, a su vez, expandidos. Fundamental el uso de una buena heurística para saber que nodo debemos escoger en cada momento.

- Si es destino, comprobamos si es mejor camino que el mejor encontrado hasta el momento, sino, seguimos el mismo proceso.

- Así hasta que no haya nodos que puedan mejorar el camino creado.

- Hay que tener en cuenta también que la heurística sea admisible, es decir, nunca se debe tener un valor heurístico mayor que el coste mínimo de ese nodo y esto ha de cumplirse para cualquier nodo.

A continuación trataremos de explicar cómo hemos llevado las anteriores características a la implementación, tanto en 2D como en 3D. Será una visión general de la estructura del algoritmo como de sus componentes básicas

(estructuras de datos) y sus accesos. El desglose de la implementación se realizará en el apartado siguiente 3.4 de funcionalidades.

Implementación

El algoritmo se puede dividir en cuatro partes bien diferenciadas:

- Inicialización.
- Expandir nodos.
- Elección de nodos.
- Construcción del camino.

***Inicialización**:*

Hablaremos principalmente de cómo se inicializa la parte del modelo de nuestra aplicación(estructura de datos usadas).

Nuestra aplicación necesita almacenar un grupo de datos en un sólo objeto.

Los arrays sirven bien para este propósito, pero algunas veces necesitamos
incrementar o reducir dinámicamente el número de elementos del array, o hacer que contenga distintos tipos de datos.

Para esta clase de grupos de datos crecientes y menguantes, podemos usar la clase Vector, o la reciente clase ArrayList del paquete java.util. Nosotros elegimos ArrayList.

Un ArrayList contiene tantos objetos como necesitemos. ArrayList tiene varios constructores, dependiendo de cómo necesitemos construir el ArrayList.
Un objeto ArrayList sólo contiene referencias a objetos. Para almacenar tipos primitivos como double, long, o float, usamos una clase envoltura.

Si necesitamos circular a través de los elementos del ArrayList, usamos la clase Iterator y sus métodos hasNext y next.

ArrayList es una de las muchas clases del **Collection Framework**, que proporciona un conjunto de interfaces y clases bien-diseñados para almacenar y manipular grupos de datos como una sola unidad, una coleccion.

La estrutura de paquetes:

java.util
Class ArrayList
java.lang.Object
└ java.util.AbstractCollection
└ java.util.AbstractList
└ **java.util.ArrayList**

Presentamos cuatro variables para el almacenamiento de datos en estructuras tipo ArrayList:

- Opened : lista en la que se guarda los nodos que vamos visitando
- Closed: nodos que hemos expandido o que jamás deben expandirse (obstáculos infranqueables).
- ExpArray: lista de nodos sucesores de otro nodo que ha sido expandido en un momento dado.
- OptimalPath: lista de nodos que devuelve l algoritmo y que corresponde al camino óptimo encontrado.

- **Implementación 2D**

Lo primero que realiza el algoritmo es la inicialización tanto de las variables anteriores como de variables del sistema:

- xStart, yStart: coordenadas del nodo origen.
- xTarget, yTarget: coordenadas del nodo objetivo.
- xNode, yNode: coordenadas del nodo actual en proceso.
- xVal, yVal: coordenadas auxiliares de ayuda para el recorrido de la lista de cerrados(closed) una vez que queremos construir el camino.
- parentX, parentY: coordenadas del nodo padre de un nodo en consideración.
- pathCost: coste del camino acumulado.
- existPath: variable booleana para el control de si existe camino o no.

La parte de inicialización acaba con un bucle que lo que realiza es encontrar aquellos nodos pertenecientes a obstáculos infranqueables y añadirlos a la lista de cerrados (closed). Esto es debido a que en esta lista almacenamos todos aquellos nodos los cuales no queremos expandir de nuevo ya sean porque son nodos camino u obstáculos por donde no podemos atravesarlos.

Respecto al contenido de las lista son objetos de las siguientes clases:
- ClosedNode: define los nodos de la lista de cerrados, closed.
- OpenedNode: define los nodos de la lista de cerrados, closed. Se diferencia de la anterior en que esta tiene los parámetros necesarios para el cálculo de las heurísticas y el coste del camino.
- ExpandedNode: define los nodos de la lista expandArray.
- Node: clase padre de todas las anteriores.

El mapa del terreno se representa a través de una clase que contenía un parámetro que era una matriz donde las filas representaban la coordenada x y las columnas la y.

- **Implemetación 3D**

El salto a 3D supuso una serie de cambios a nivel del algoritmo que exponemos a continuación:

- Todas las variables que definen coordenadas tendrán una nueva para la tercera dimensión caracterizada por el símbolo z seguido del tipo de nodo que representa.
- El manejo de la tercera dimensión obliga a tener un número mayor de nodos tanto a expandir como a recorrer en el mapa del terreno, con lo que el coste en tiempo y espacio supuso un problema. Para mejorar la eficiencia se cambió la estructura del mapa. De tener una matriz pasamos a tener un arrayList de objetos tipo EspacioNode que tienen a su vez los tres parámetros que representan las tres coordenadas correspondientes a las tres dimensiones.
- EspacioNode: clase que define los nodos del mapa que recorremos. La lista que se devuelve, optimalPath, es una lista de objetos de esta clase.
- Espacio: implementa el mapa, con sus tamaños para cada dimensión y la estructura de los nodos.

Expandir nodos:

Se realiza a través de la función *expandArray* de la que se tratará posteriormente.

Lo fundamental en esta función es el cálculo de la heurística: el valor heurístico resulta de calcular la distancia entre el punto donde se encuentra actualmente y el punto destino.

Existen tres tipos de distancias entre dos puntos A(x1, y1) y B(x2, y2) que consideramos:

a) Euclídea = $[(x2 – x1)^2 + (y2 – y1)^2]^{1/2}$ Es la distancia que elegimos.
b) Manhattan = abs(x-xpunto)+abs(y-ypunto)
c) Cuadrada o ajedrez = max(abs(x-xpunto),abs(y-ypunto))

Como cada nodo del entorno tiene un peso asociado correspondiente a su tipo (a mayor peligrosidad el peso será mayor) la distancia vendrá influida por un compromiso entre menor distancia y una seguridad máxima o la que le indiquemos.

También nos hemos preocupado que cumpla otras propiedades como la de consistencia. La consistencia nos dice que para todo nodo n y su sucesor m, el coste estimado de alcanzar el objetivo desde n, no es mayor que el coste de alcanzar m más el coste de alcanzar el objetivo desde m.

Destacar que en el algoritmo final 3D los recorridos tanto en esta función como en todas se modificaron para tener en cuenta esta tercera coordenada.

Elección de nodos

Esta elección se hace a través de la función implementada *minFn*. Lo que hace es elegir aquel nodo de los existentes en la lista de nodos abiertos, opened, el cual tenga menor coste desde el nodo padre.

Construcción del camino

Una vez terminado el bucle principal del A*, si existe camino posible, tenemos en la lista de closed todos los nodos expandidos y que formarán el camino óptimo (a excepción de los obstáculos infranqueables). Se trata pues de construir este camino a través de enlaces de padres a hijos en la lista de closed. Se recorre esta lista hasta alcanzar las coordenadas del nodo origen empezando a con el nodo objetivo que fue el último insertado en la lista de cerrados.
El enlace se hace a través de los atributos del objeto closedNode (parentX, parentY, parentZ) que nos guardan el enlace al nodo padre.

4.4 Funcionalidad

En esta apartado desglosaremos la aplicación a nivel más bajo, es decir, a nivel de programación: funcionalidades.

La implementación de la parte algorítmica se divide en cuatro paquetes correspondientes a los distintos niveles que podemos encontrar en un modelo estructural vista controlador:

DOCUMENTACIÓN

Paquetes	
Algorithm	Paquete que contiene la algoritmo.
Control	Paquete de control para lanzamiento de la aplicación.
Gui	Interfaz Gráfica de Usuario. Vistas.
Model	Estructura de datos usada.

En la aplicación final, los paquetes gui y control dejan de existir puesto que las funcionalidades de estos paquetes se harán desde la implementación y enlace con Matlab. Se desarrollaron para depuraciones y desarrollo de interfaz 2D en un primer incremento de la aplicación.

A continuación pasamos a explicar cada paquete:

ALGORITHM

Es el paquete que implementa el algoritmo A* y que contiene la clase Algorithm()

Algorithm
Class Algorithm
java.lang.Object
└ **Algorithm.Algorithm**

---

public class **Algorithm**
extends java.lang.Object

Constructor	
Algorithm()	

Métodos utilizados	
(package private) static float	**distance**(int xt,int yt,int zt,int xn,int yn, int zn)
(package private) static java.util.ArrayList	**expandArray**(int xn,int yn,int zn,float pathCost, int xt,int yt,int zt,java.util.ArrayList closedList ,int mx, int my, int mz, Espacio map)
static float	**min**(float a, float b)
static int	**minFn**(java.util.ArrayList opened, int openedCount, int xTarget, int yTarget, int zTarget)
static int	**nodeIndex**(java.util.ArrayList closed, int xVal, int yVal, int zVal)
java.util.ArrayList	**runAlgorithm**(Espacio map, float combustibleTotal, float gastoCombustibleMedio)

Métodos inherentes de la clase java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait

Detalles del constructor
Algorithm

public **Algorithm**()

Detalles de los métodos

Distance : calcula la distancia Euclídea entre (xt,yt,zt) y (xn,yn,zn)
static float **distance**(int xt,int yt,int zt,int xn,int yn, int zn)

---

expandArray: función que haya los sucesores del nodo (xn,yn,zn)
staticjava.util.ArrayList
expandArray(int xn,int yn,int zn,float pathCost,int xt,int yt,int zt,
java.util.ArrayList closedList,int mx,int my,int mz, Espacio map)

Throws: lanza excepción si no encuentra nodos sucesores posibles
ControlError

Min: **halla el mínimo entre dos valores a y b**
public static float **min**(float a,float b)

---

minFn: función para elegir que nodo va ser el siguiente a expandir de entre todos lo nodos en la lista de abiertos,opened.
public static int **minFn**(java.util.ArrayList opened,int openedCount,
int xTarget,int yTarget,int zTarget)

---

nodeIndex
public static int **nodeIndex**(java.util.ArrayList closed, int xVal,
int yVal,int zVal)

---

runAlgorithm: implementa el A*
public java.util.ArrayList **runAlgorithm**(Espacio map,
float combustibleTotal,
float gastoCombustibleMedio)
throws ControlError
Throws: lanaza excepción si no hay camino posible
ControlError

Se le pasa el mapa del terreno en forma objeto del tipo Espacio que es quien contiene el array con los nodos de nuestro mapa. Además para el manejo de la variable interna del avión condicionante combustible se le pasa las dos variables combustibleTotal que, como su nombre indica, es el combustible que inicialmente lleva el avión autodirigido, y la otra variable que mide lo que consume el avión por nodo recorrido.
El bucle principal de la función es el del A*, recorrer los nodos para hallar el camino a través de ver si estaban o no ya expandidos y si estaban en abierta o no para reconsiderar costes.

CONTROL

PAQUETE CONTROL: contiene dos clases para el control de la aplicación	
ControlError	Clase de control de errores y excepciones.
Main	Clase del programa principal quien lanza la aplicación.

Comenzaremos explicando las funcionalidades de la clase ControlError():

Class ControlError
java.lang.Object
  └java.lang.Throwable
    └**Control.ControlError**

All Implemented Interfaces:
java.io.Serializable

---

public class **ControlError** extends java.lang.Throwable

Atributos	
private static long	**serialVersionUID**

Constructor
ControlError(java.lang.String error)

Métodos inherentes de la clase java.lang.Throwable
fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

Métodos inherentes de la clase java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait

Atributos

serialVersionUID
private static final long **serialVersionUID**

Constructor

ControlError: se le para como argumento un string que indica el error que

queremos visualizar
public **ControlError**(java.lang.String error)

Veamos ahora la clase **Main**:

Control
Class Main
java.lang.Object
└ **Control.Main**

---

public class **Main**
extends java.lang.Object

Atributo	
(package private) static Mundo	**mundo**

Constructor	
Main()	

Método	
static void	**main**(java.lang.String[] args) Inicializa el entorno en el que trabajamos, tamaño del mapa, a la vez que las variables de gestión del combustible. Inicializa el atributo mundo que es la interfaz con la que trabajamos para depuraciones.

Métodos inherentes de la clase java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait

GUI: Interfaz Gráfica de Usuario

Este paquete recordamos que se explica a modo referencial en el apartado 4 de Interfaz ya que la aplicación final no contiene este paquete ya que la interfaz de usuario que se usa está desarrollada en Matlab. Debido al desarrollo en paralelo con el otro grupo se tuvo que manterner esta interfaz inicial y sus posteriores versiones mejoradas para poder depurar y controlar el desarrollo continuo del algoritmo.

MODEL

Clases	
ClosedNode	Clase que define los nodos que pertenecen a la lista de cerrados
Espacio	Clase que define el espacio sobre el cual interactuamos en la simulación
EspacioNode	Clase que implementa los nodos del terreno
ExpandedNode	Clase que define los nodos que pertenecen a la lista de expandidos
Node	Clase padre de todo tipo de nodos
OpenedNode	Clase que define los nodos que pertenecen a la lista de abiertos

Veamos cada clase:

Clase ClosedNode

java.lang.Object
  ∟Model.Node
    ∟**Model.ClosedNode**

---

public class **ClosedNode**
extends Node

Constructor
ClosedNode(int fila, int columna, int altura, java.lang.String tipo, int padreFila, int padreColumna,int padreAltura)

Métodos	
int	**getPadreAltura**()

int	**getPadreColumna**()
int	**getPadreFila**()
void	**setPadreAltura**(int padreAltura)
void	**setPadreColumna**(int padreColumna)

Métodos inherentes a la clase Model.Node
getAltura, getColumna, getFila, getTipo, setAltura, setColumna, setFila, setTipo

Métodos inherentes a la clase java.lang.Object
equals, getClass, hashCode, notify, notifyAll, toString, wait

Detalles del constructor

ClosedNode
public **ClosedNode**(int fila,
int columna,
int altura,
java.lang.String tipo,
int padreFila,
int padreColumna,
int padreAltura)

Clase Espacio

java.lang.Object
└ **Model.Espacio**

---

public class **Espacio**
extends java.lang.Object

Constructor
Espacio(int maximoF,int maximoC,int maximoA)

Métodos	
void	**borraCamino**()
int	**busqueda**(int x,int y,int z)

java.util.ArrayList	**getCubo**()
int	**getMaxA**()
int	**getMaxC**()
int	**getMaxF**()
int	**getObjetivo**()
int	**getOrigen**()
void	**setCubo**(java.util.ArrayList cubo)
void	**setMaxA**(int maxA)
void	**setMaxC**(int maxC)
void	**setMaxF**(int maxF)
void	**setObjetivo**(int objetivo)
void	**setOrigen**(int origen)

Clase EspacioNode

java.lang.Object
  └ **Model.EspacioNode**

---

public class **EspacioNode**
extends java.lang.Object

Constructor	
EspacioNode()	
EspacioNode(int fila,int columna,int altura,float peso)	

Métodos	
int	**getAltura**()

int	**getColumna**()
int	**getFila**()
float	**getPeso**()
void	**setAltura**(int altura)
void	**setColumna**(int columna)
void	**setFila**(int fila)
void	**setPeso**(float peso)

Clase ExpandedNode

java.lang.Object
  └ Model.Node
    └ **Model.ExpandedNode**

---

public class **ExpandedNode**
extends Node

Constructor
ExpandedNode(int fila, int columna, int altura, java.lang.String tipo)
ExpandedNode(int fil, int colum, int altura, java.lang.String tip, float hheur, float gheur, float fheur)

Métodos	
float	**getF**()
float	**getG**()
float	**getH**()
void	**setF**(int nuevoF)
void	**setG**(int nuevoG)

void	**setH**(int nuevoH)

Clase Node

java.lang.Object
  └ **Model.Node**
Direct Known Subclasses:
ClosedNode, ExpandedNode, OpenedNode

---

public class **Node**
extends java.lang.Object

Constructor	
Node()	
Node(int fila,int columna,int altura, java.lang.String tipo)	

Métodos	
int	**getAltura**()
int	**getColumna**()
int	**getFila**()
java.lang.String	**getTipo**()
void	**setAltura**(int altura)
void	**setColumna**(int columna)
void	**setFila**(int fila)
void	**setTipo**(java.lang.String tipo)

Clase OpenedNode

java.lang.Object
  └ Model.Node
    └ **Model.OpenedNode**

---

public class **OpenedNode**
extends Node

Constructor
OpenedNode(int fila, int columna, int altura, java.lang.String tipo, int padreFila, int padreColumna, int padreAltura, float h, float g, float f)

Métodos	
float	**getF**()
float	**getG**()
float	**getH**()
int	**getPadreAltura**()
int	**getPadreColumna**()
int	**getPadreFila**()
void	**setF**(float f)
void	**setG**(float g)
void	**setH**(float h)
void	**setPadreAltura**(int padreAltura)
void	**setPadreColumna**(int padreColumna)
void	**setPadreFila**(int padreFila)

4.5 Diagramas UML del núcleo

En este apartado, nos encargaremos de visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software a través del Lenguaje de Modelamiento Unificado (UML). Se mostrarán los diagramas uml de la aplicación incluyendo la interfaz de Matlab en diagrama de interacción y en los de clases la interfaz desarrollada inicialmente en 2D.

Nuestro objetivo a conseguir es entregar un material de apoyo que le permita al lector poder definir diagramas propios como también poder entender el modelado de diagramas ya existentes.

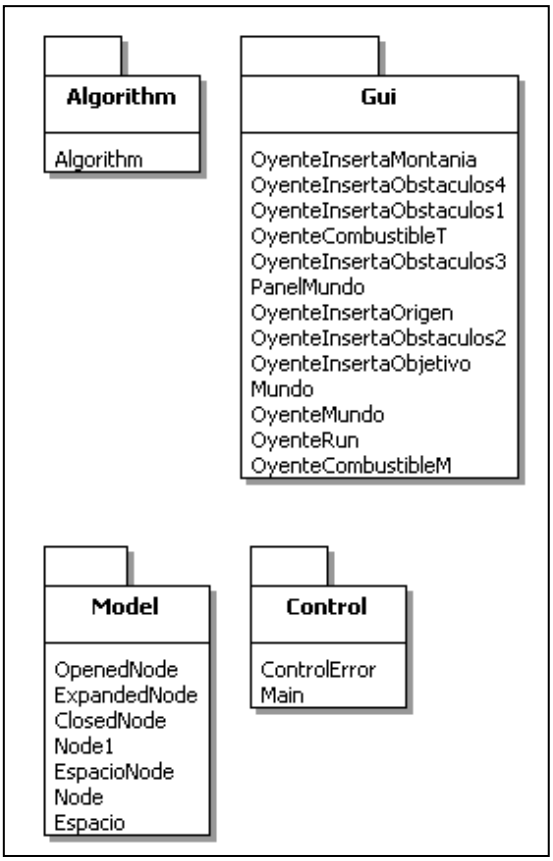


Figura 12 *– Diagrama de clases de los paquetes*

4.5.1 Modelado de clases

Paquete Control

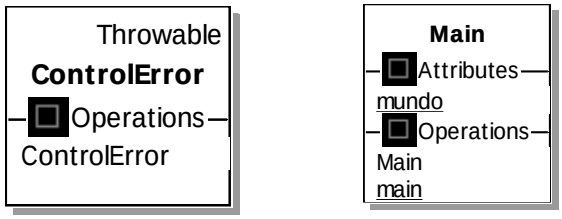


Figura 13 *– Diagrama de clases del paquete control*

Paquete Algorithm

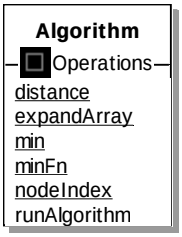


Figura 14 *– Diagrama de clases del paquete Algorithm*

Paquete Model

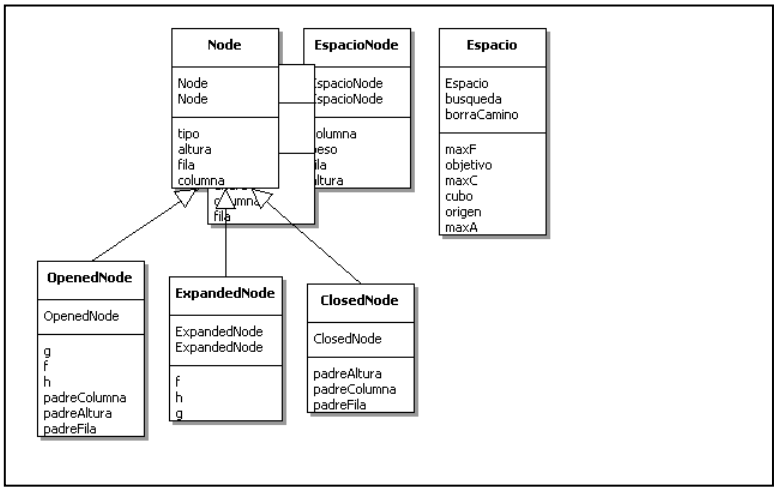


Figura 15 *– Diagrama de clases del paquete Model*

Paquete Gui

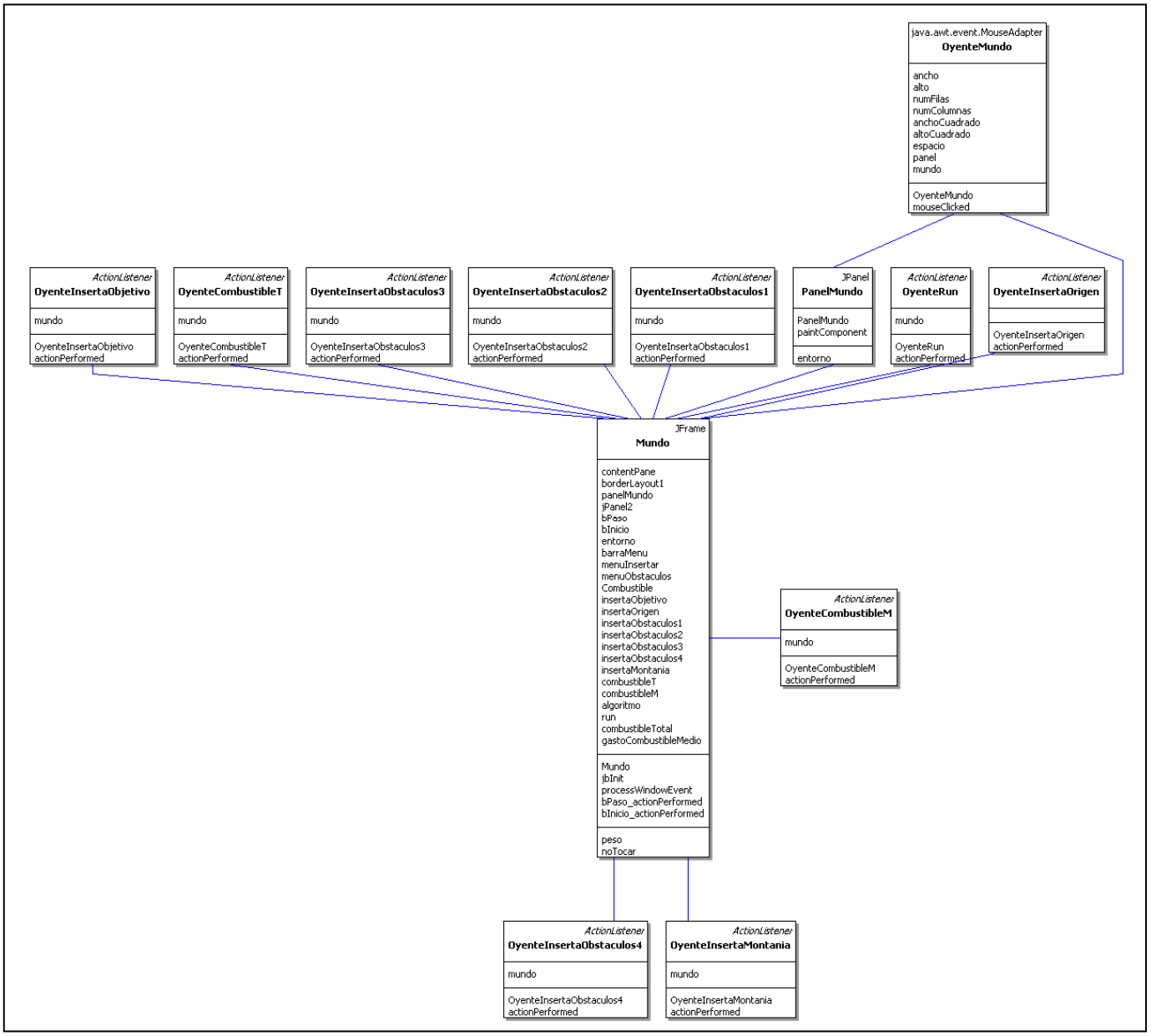


Figura 16 *– Diagrama de clases del paquete Gui*

En este apartado, describiremos el conjunto de secuencias de acciones, incluyendo variantes, que ejecuta la aplicación para producir el resultado observable para el usuario.

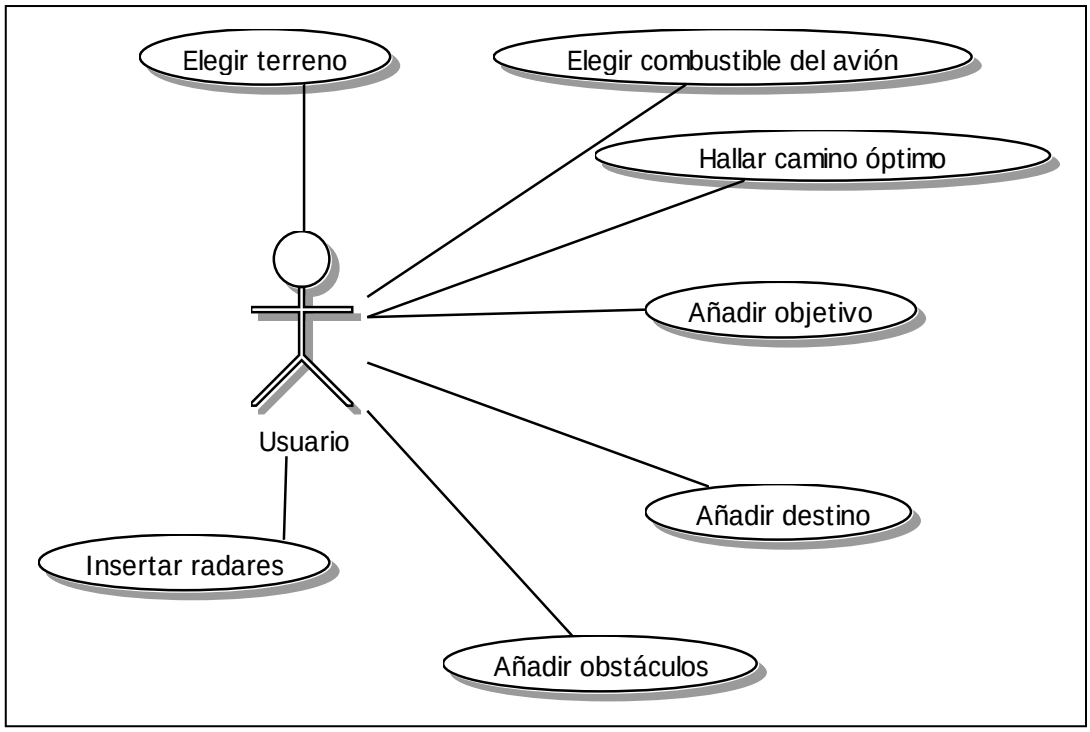


Figura 17 *– Diagrama de casos de uso*

4.5.3 Diagramas de interacción

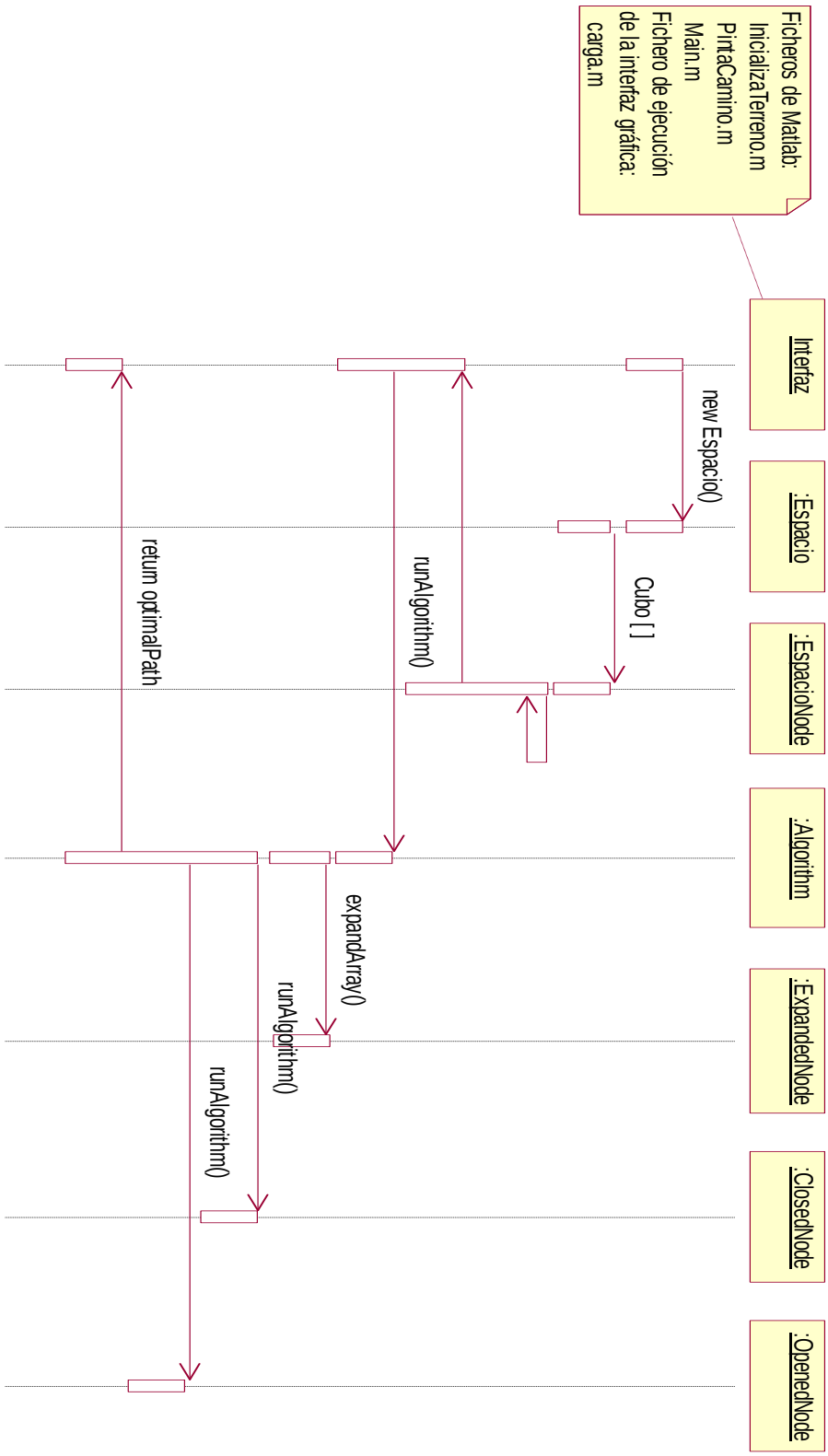


Figura 18 *– Diagrama de interacción*

5 PARTE 2 – Interfaz Gráfica

5.1 Plataforma de desarrollo

En un primer momento la implementación de toda la aplicación se desarrolló en java, con el entorno de desarrollo Eclipse 3.1, con el kit de desarrollo j2sdk 1.4.2_09.

Para la versión final utilizamos el Matlab 7.0 al ser una herramienta más potente en el cálculo de matrices y para la simulación 3D. Para la conexión de Matlab con los ficheros java se importan a través de la instrucción javaclasspath (dirección). A partir de ahí se pueden crear objetos de las clases java realizadas dentro de los ficheros .m de matlab.

5.2 Evolución durante el desarrollo del proyecto

En cuanto a la plataforma de desarrollo para la interfaz gráfica, hay que diferenciar entre las usadas en diferentes partes durante el proceso de desarrollo del proyecto. En un primer momento creamos una interfaz temporal en java mientras se desarrollaba el algoritmo inicial, en dos dimensiones y destinada principalmente a pruebas, ensayos y depuración.

5.2.1 Implementación 2D inicial

Esta interfaz está integrada dentro del paquete **gui** explicado en el apartado anterior. Sus funcionalidades son las siguientes:

- **GUI**

Clases que contiene	
Mundo	Interfaz para depuraciones hecha en 2D
OyenteCombustibleM	Controlador del combustible promedio
OyenteCombustibleT	Controlador del combustible
OyenteInsertaMontania	Controlador para inserciones de obstáculos tipo montañas
OyenteInsertaObjetivo	Controlador para inserciones del objetivo
OyenteInsertaObstaculos1	Controlador para inserciones de obstáculos de nivel 1 de peligrosidad
OyenteInsertaObstaculos2	Controlador para inserciones de obstáculos de nivel 2 de peligrosidad

OyenteInsertaObstaculos3	Controlador para inserciones de obstáculos de nivel 3 de peligrosidad
OyenteInsertaObstaculos4	Controlador para inserciones de obstáculos de nivel 4 de peligrosidad
OyenteInsertaOrigen	Controlador para inserciones del origen
OyenteMundo	Controlador para cambios en la interfaz
OyenteRun	Controlador para el lanzamiento de la ejecución del algoritmo
PanelMundo	Clase auxiliar para pintar el mapa

A continuación iremos explicando cada clase:

Clase Mundo

java.lang.Object
  └ java.awt.Component
    └ java.awt.Container
      └ java.awt.Window
        └ java.awt.Frame
          └ javax.swing.JFrame
            └ **Gui.Mundo**

---

public class **Mundo**
extends javax.swing.JFrame

Clases anidadas

Clases inherentes de la clase javax.swing.JFrame
javax.swing.JFrame.AccessibleJFrame

Clases inherentes de la clase java.awt.Frame
java.awt.Frame.AccessibleAWTFrame

Clases inherentes de la clase java.awt.Window
java.awt.Window.AccessibleAWTWindow

Clases inherentes de la clase java.awt.Container
java.awt.Container.AccessibleAWTContainer

Clases inherentes de la clase java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Atributos	
(package private) javax.swing.JMenu	**algoritmo**
(package private) javax.swing.JMenuBar	**barraMenu**
(package private) javax.swing.JButton	**bInicio**
(package private) java.awt.BorderLayout	**borderLayout1**
(package private) javax.swing.JButton	**bPaso**
(package private) javax.swing.JMenu	**Combustible**
(package private) javax.swing.JMenuItem	**combustibleM**
(package private) javax.swing.JMenuItem	**combustibleT**
(package private) float	**combustibleTotal**
(package private) javax.swing.JPanel	**contentPane**
(package private) Espacio	**entorno**
(package private) float	**gastoCombustibleMedio**
(package private) javax.swing.JMenuItem	**insertaMontania**
(package private) javax.swing.JMenuItem	**insertaObjetivo**
(package private) javax.swing.JMenuItem	**insertaObstaculos1**
(package private) javax.swing.JMenuItem	**insertaObstaculos2**
(package private)	**insertaObstaculos3**

javax.swing.JMenuItem	
(package private) javax.swing.JMenuItem	**insertaObstaculos4**
(package private) javax.swing.JMenuItem	**insertaOrigen**
(package private) javax.swing.JPanel	**jPanel2**
(package private) javax.swing.JMenu	**menuInsertar**
(package private) javax.swing.JMenu	**menuObstaculos**
(package private) boolean	**noTocar**
(package private) PanelMundo	**panelMundo**
(package private) float	**peso**
(package private) javax.swing.JMenuItem	**run**
private static long	**serialVersionUID** Esta interfaz es a modo de pruebas depuraciones del algoritmo, la interfaz final está implementada en Matlab

Atributos inhererentes de la clase javax.swing.JFrame
accessibleContext, EXIT_ON_CLOSE, rootPane, rootPaneCheckingEnabled

Atributos inhererentes de la clase java.awt.Frame
CROSSHAIR_CURSOR,DEFAULT_CURSOR,E_RESIZE_CURSOR, HAND_CURSOR, ICONIFIED, MAXIMIZED_BOTH, MAXIMIZED_HORIZ, MAXIMIZED_VERT, MOVE_CURSOR, N_RESIZE_CURSOR, NE_RESIZE_CURSOR, NORMAL, NW_RESIZE_CURSOR, S_RESIZE_CURSOR, SE_RESIZE_CURSOR, SW_RESIZE_CURSOR, TEXT_CURSOR, W_RESIZE_CURSOR, WAIT_CURSOR

Atributos inhererentes de la clase java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Atributos inhererentes de la interfaz javax.swing.WindowConstants
DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE

Atributos inhererentes de la interfaz java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor
Mundo(Espacio entorno, float combustibleTotal, float gastoCombustibleMedio)

Métodos	
(package private) void	**bInicio_actionPerformed**(java.awt.event.ActionEvent e)
(package private) void	**bPaso_actionPerformed**(java.awt.event.ActionEvent e)
boolean	**getNoTocar**()
float	**getPeso**()
private void	**jbInit**()
protected void	**processWindowEvent**(java.awt.event.WindowEvent e)
void	**setNoTocar**(boolean tocar)

Detalles de atributos

serialVersionUID
private static final long **serialVersionUID**
Esta interfaz es a modo de pruebas depuraciones del algoritmo, la interfaz final está implementada en Matlab

---

contentPane
javax.swing.JPanel **contentPane**

---

borderLayout1
java.awt.BorderLayout **borderLayout1**

---

panelMundo
PanelMundo **panelMundo**

---

jPanel2
javax.swing.JPanel **jPanel2**

---

bPaso
javax.swing.JButton **bPaso**

---

bInicio
javax.swing.JButton **bInicio**

---

entorno
Espacio **entorno**

---

barraMenu
javax.swing.JMenuBar **barraMenu**

---

menuInsertar
javax.swing.JMenu **menuInsertar**

---

menuObstaculos
javax.swing.JMenu **menuObstaculos**

Combustible
javax.swing.JMenu **Combustible**

---

insertaObjetivo
javax.swing.JMenuItem **insertaObjetivo**

---

insertaOrigen
javax.swing.JMenuItem **insertaOrigen**

---

insertaObstaculos1
javax.swing.JMenuItem **insertaObstaculos1**

---

insertaObstaculos2
javax.swing.JMenuItem **insertaObstaculos2**

---

insertaObstaculos3
javax.swing.JMenuItem **insertaObstaculos3**

---

insertaObstaculos4
javax.swing.JMenuItem **insertaObstaculos4**

---

insertaMontania
javax.swing.JMenuItem **insertaMontania**

---

combustibleT
javax.swing.JMenuItem **combustibleT**

---

combustibleM
javax.swing.JMenuItem **combustibleM**

---

algoritmo
javax.swing.JMenu **algoritmo**

---

run
javax.swing.JMenuItem **run**

---

peso
float **peso**

---

noTocar
boolean **noTocar**

---

combustibleTotal
float **combustibleTotal**

---

gastoCombustibleMedio
float **gastoCombustibleMedio**

Detalles del Constructor

Mundo
public **Mundo**(Espacio entorno,
float combustibleTotal,
float gastoCombustibleMedio)

Detalles de los Métodos

jbInit
private void **jbInit**()
throws java.lang.Exception
Throws:
java.lang.Exception

---

processWindowEvent
protected void **processWindowEvent**(java.awt.event.WindowEvent e)

---

bPaso_actionPerformed
void **bPaso_actionPerformed**(java.awt.event.ActionEvent e)

---

bInicio_actionPerformed
void **bInicio_actionPerformed**(java.awt.event.ActionEvent e)

---

setPeso
public void **setPeso**(float tipoEntrada)

---

getPeso
public float **getPeso**()

---

setNoTocar
public void **setNoTocar**(boolean tocar)

---

getNoTocar
public boolean **getNoTocar**()

Clase PanelMundo

java.lang.Object
  └ java.awt.Component
    └ java.awt.Container
      └ javax.swing.JComponent
        └ javax.swing.JPanel
          └ **Gui.PanelMundo**

---

public class **PanelMundo**
extends javax.swing.JPanel

Clases

Clases inherentes a la clase javax.swing.JPanel
javax.swing.JPanel.AccessibleJPanel

Clases inherentes a la clase javax.swing.JComponent
javax.swing.JComponent.AccessibleJComponent

Clases inherentes a la clase java.awt.Container
java.awt.Container.AccessibleAWTContainer

Clases inherentes a la clase java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Atributos	
private Espacio	**entorno**
private static long	**serialVersionUID**

Atributos inherentes a la clase javax.swing.JComponent
accessibleContext, listenerList, TOOL_TIP_TEXT_KEY, ui, UNDEFINED_CONDITION, WHEN_ANCESTOR_OF_FOCUSED_COMPONENT, WHEN_FOCUSED, WHEN_IN_FOCUSED_WINDOW

Atributos inherentes a la clase java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT,

TOP_ALIGNMENT

Atributos inherentes a la interfaz java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor	
PanelMundo(Espacio entorno)	

Métodos	
Espacio	**getEntorno**()
void	**paintComponent**(java.awt.Graphics g)
void	**setEntorno**(Espacio entorno)

Métodos inherentes a la clase javax.swing.JPanel
getAccessibleContext, getUI, getUIClassID, paramString, setUI, updateUI

Class OyenteInsertaOrigen

java.lang.Object
  └ **Gui.OyenteInsertaOrigen**

All Implemented Interfaces:

java.awt.event.ActionListener, java.util.EventListener

---

class **OyenteInsertaOrigen**
extends java.lang.Object
implements java.awt.event.ActionListener

Atributos	
private Mundo	**mundo**

Constructor	
OyenteInsertaOrigen(Mundo mundoMarco)	Recibe como argumento el entorno

Método	
void	**actionPerformed**(java.awt.event.ActionEvent ev)

Métodos inherentes a la clase java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait

Class OyenteInsertaObjetivo

java.lang.Object
  └ **Gui.OyenteInsertaObjetivo**

All Implemented Interfaces:
java.awt.event.ActionListener, java.util.EventListener

---

class **OyenteInsertaObjetivo**
extends java.lang.Object
implements java.awt.event.ActionListener

Atributos	
private Mundo	**mundo**

Constructor
OyenteInsertaObjetivo(Mundo mundoMarco)

Método	
void	**actionPerformed**(java.awt.event.ActionEvent ev)

Métodos inherentes a la clase java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait

Class OyenteInsertaObstaculos1

java.lang.Object
  └ **Gui.OyenteInsertaObstaculos1**
All Implemented Interfaces:
java.awt.event.ActionListener, java.util.EventListener

---

class **OyenteInsertaObstaculos1**
extends java.lang.Object
implements java.awt.event.ActionListener

Atributos	
private Mundo	**mundo**

Constructor
OyenteInsertaObstaculos1(Mundo mundoMarco)

Métodos	
void	**actionPerformed**(java.awt.event.ActionEvent ev)

Métodos inherentes a la clase java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait

Las demás clases tiene una estructura similar.

Ejemplos de ejecución y visualización de la interfaz:

Como muestra la siguiente figura el espacio de recorrido se diseñó inicialmente como una cuadrícula de nodos por donde el avión puede atravesar.

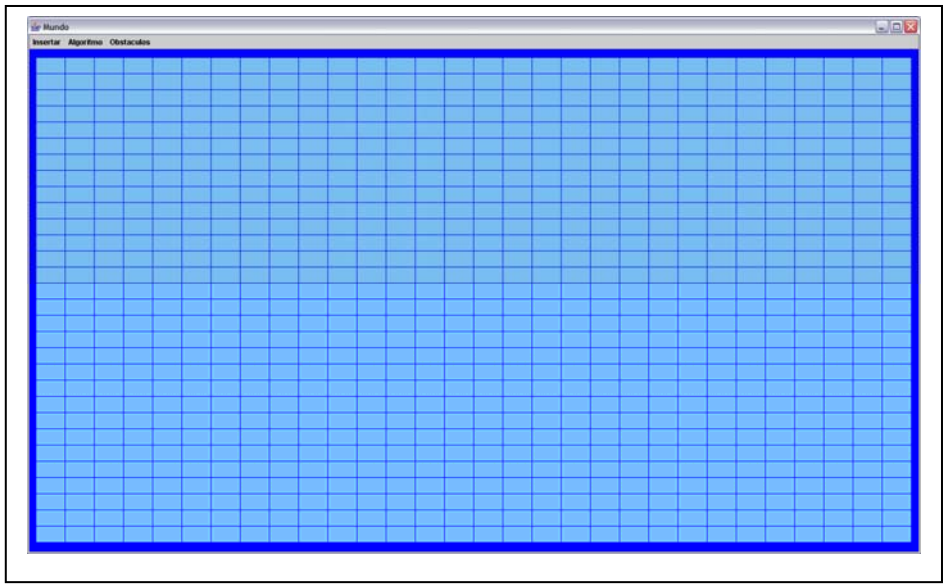


Figura 19 *–Tamaño: 30x30x1*

O en otra resolución:

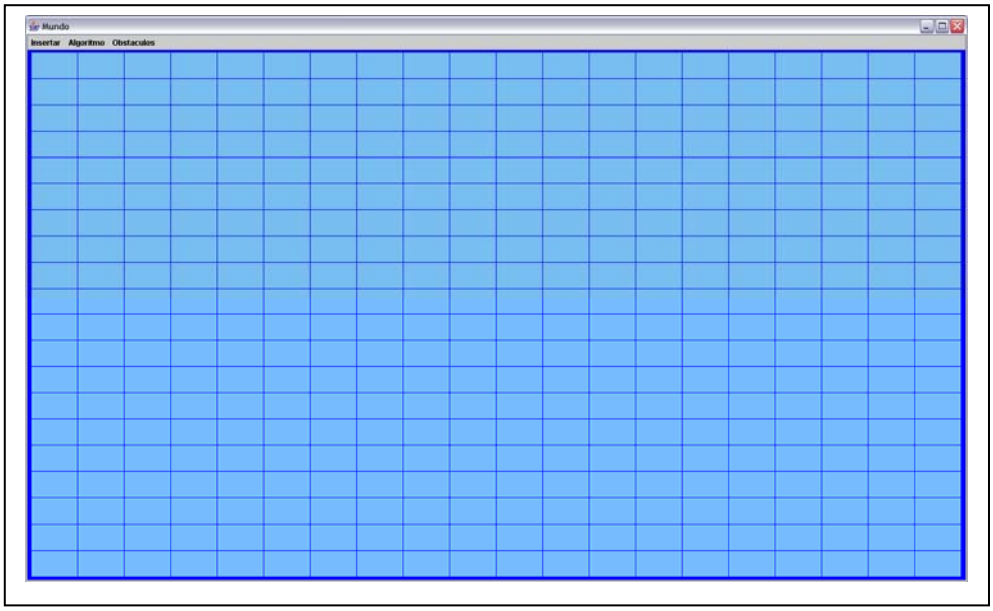


Figura 20 *– Tamaño: 20x20x1*

El usuario, a través del menú, tiene las siguientes opciones:

1. Menú Insertar: origen, objetivo

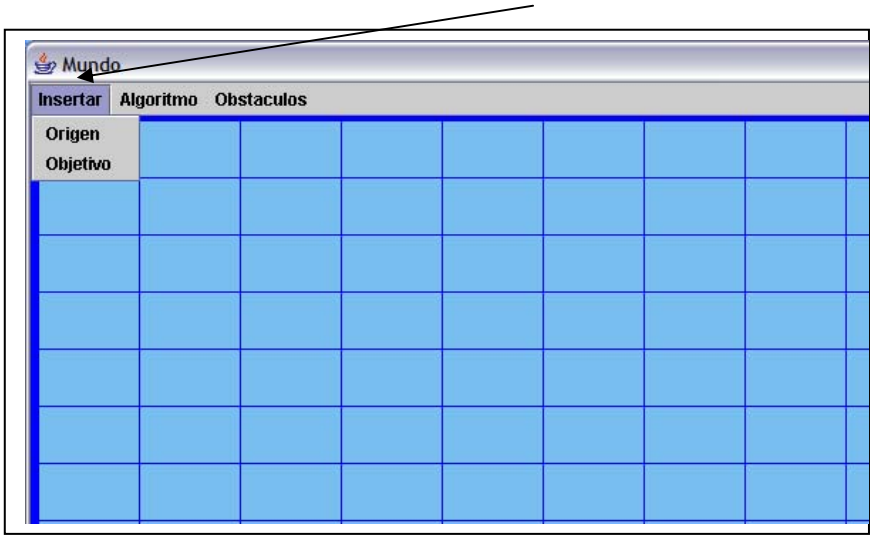


Figura 21 *–Menú Insertar*

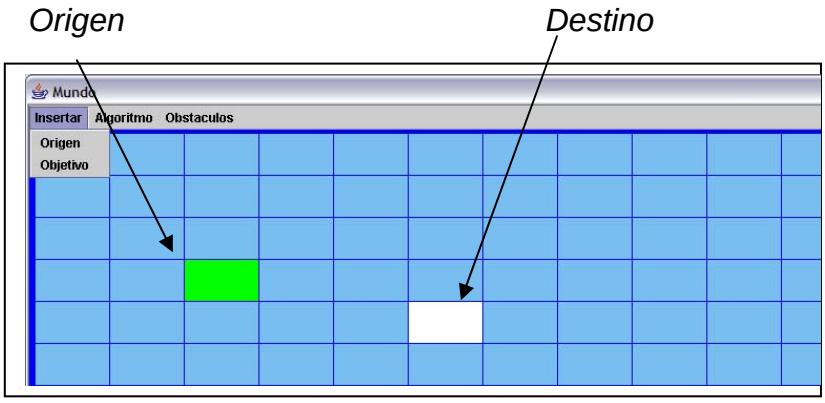
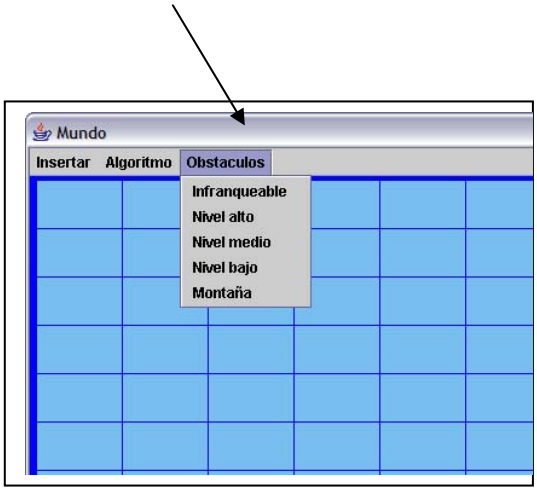
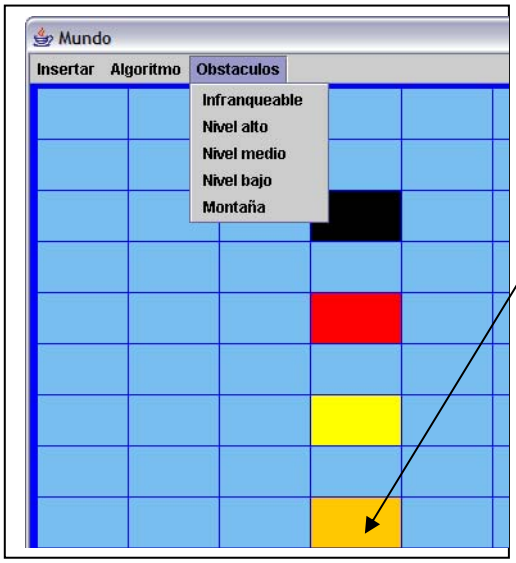


Figura 22 *–Origen - Destino*

Si el origen y destino indicados no son los que deseamos se puede volver a pinchar en el menú para elegir otros distintos.

2. Menú Obstáculos





- ***Infranqueable***: añade un nodo donde le indiquemos con el ratón el cual no pueda ser atravesado el avión. Corresponde a un nivel de peligrosidad o peso de [0.5,0.75]

- ***Nivel alto***: corresponde a un nivel de peligrosidad o peso de [0.5, 0.75)

- ***Nivel medio***: corresponde a un nivel de peligrosidad o peso de [0.25, 0.5)

- ***Nivel bajo*****:** corresponde a un nivel de peligrosidad o peso de [0,0.25)

- ***Montaña*****:** en 3D correspondería a una altura determinada rellena de nodos infranqueables.

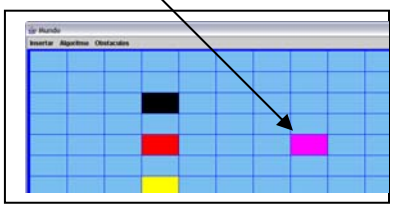


Figura 23 *–Obstáculos*

3. Menú Algoritmo: desde donde se puede ejecutar el algoritmo, una vez insertado el origen, destino u obstáculos.

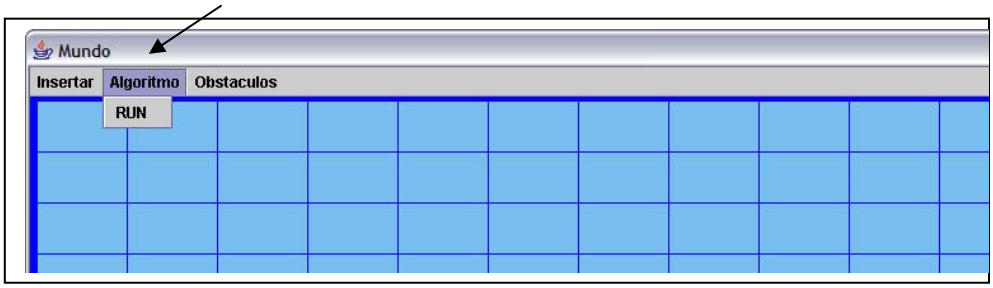


Figura 24 *– Menú Algoritmo*

A continuación, mostramos ejemplos de caminos que puede adoptar el algoritmo dependiendo de los riesgos o dificultades del terreno:

1. Sin ningún tipo de obstáculos:

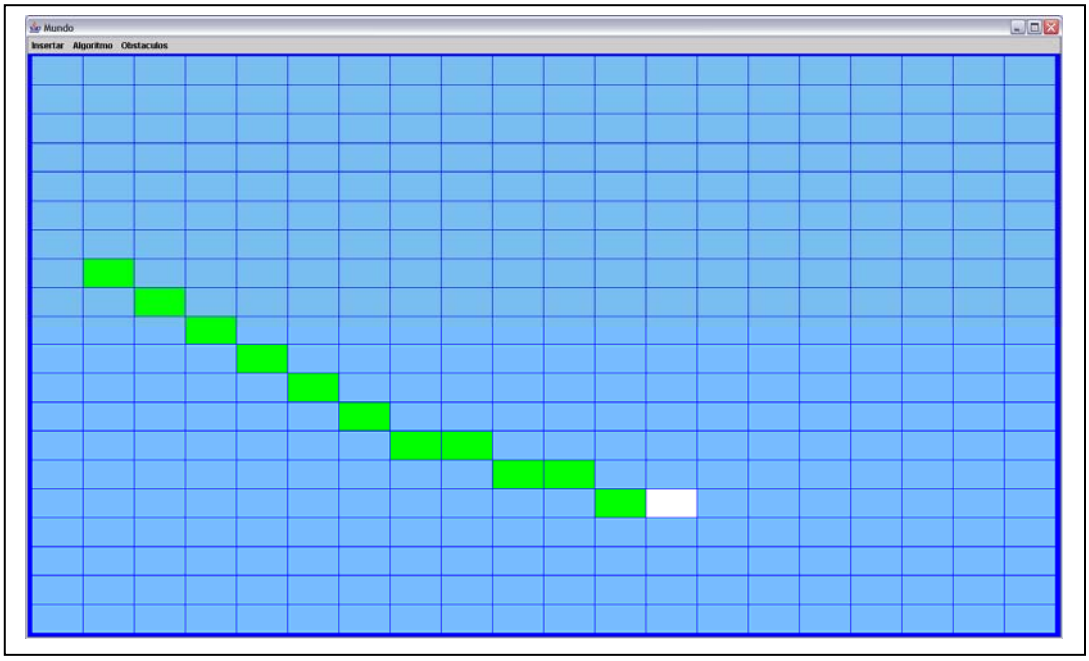


Figura 25 *– Camino óptimo*

2. Con obstáculos de distintos niveles de dificultad:

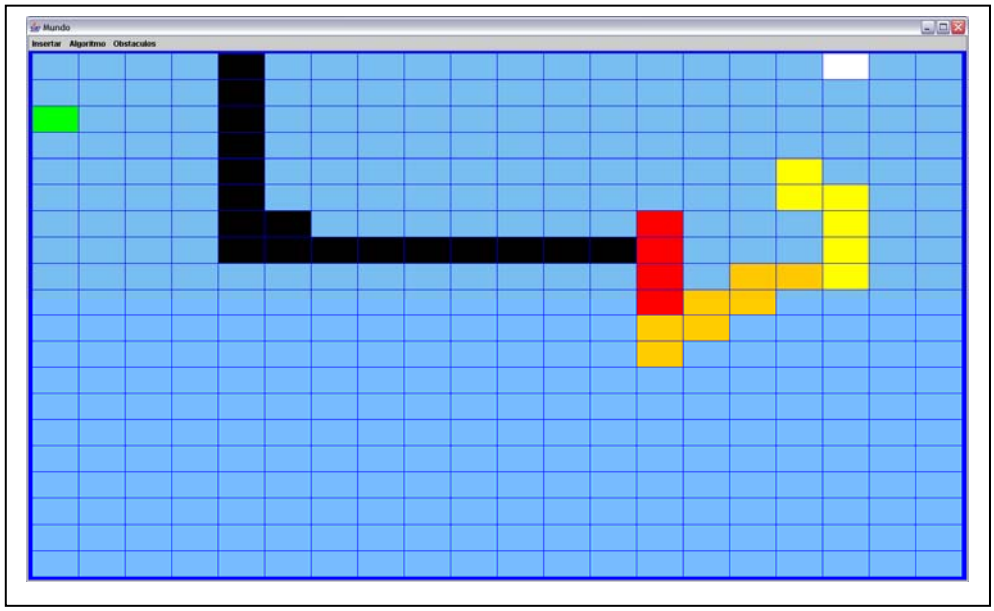


Figura 26 *– Ejemplo1 de camino*

Se puede ver que atraviesa un nodo de obstáculo de nivel de peligrosidad bajo debido a dos posibles razones:

1. Coste de atravesar obstáculo es menor que dar un mayor rodeo sin atravesar ningún obstáculo. Hay que tener en cuenta también el nivel de combustible, si será o no suficiente.
2. El obstáculo de nivel bajo atravesado tiene un valor de peso demasiado bajo. El peso de cada obstáculo se puede variar ya que los valores están normalizados en una escala de 0 a 1.

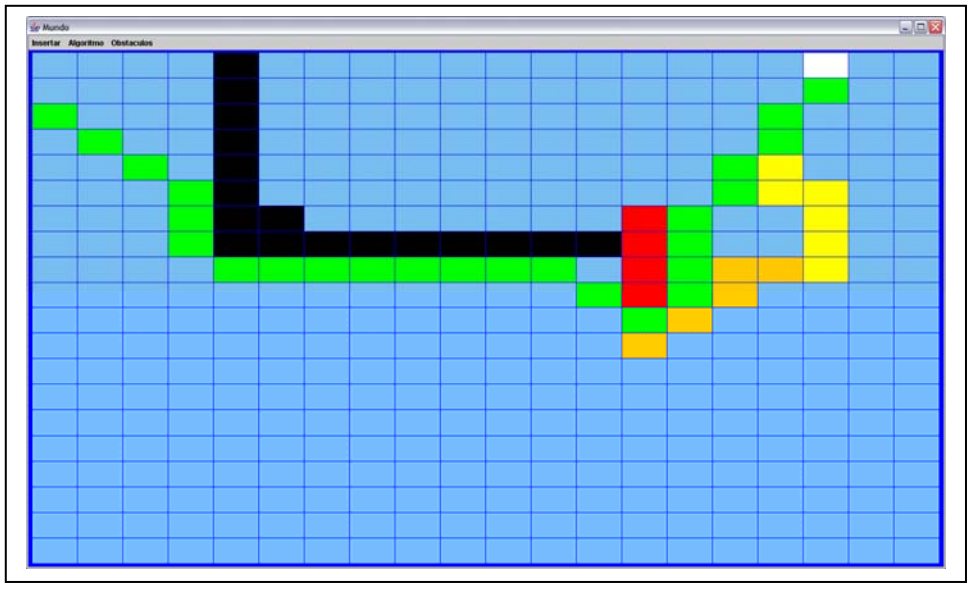


Figura 27 *– Ejemplo2 de camino*

3. Si construimos una barrera infranqueable:

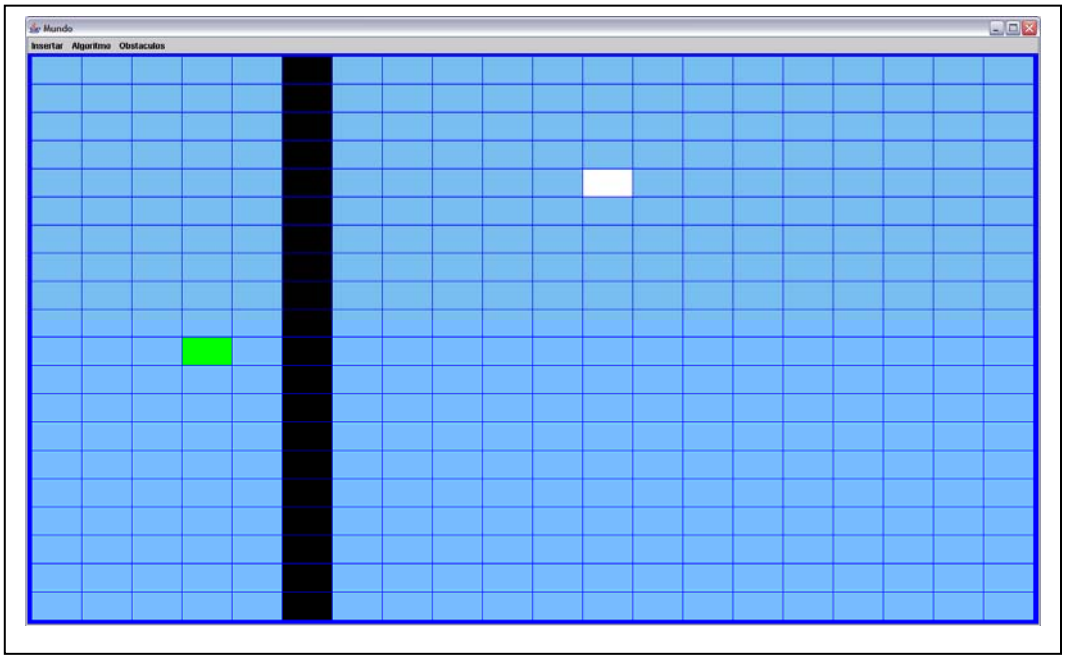


Figura 28 *– Ejemplo3 de camino*

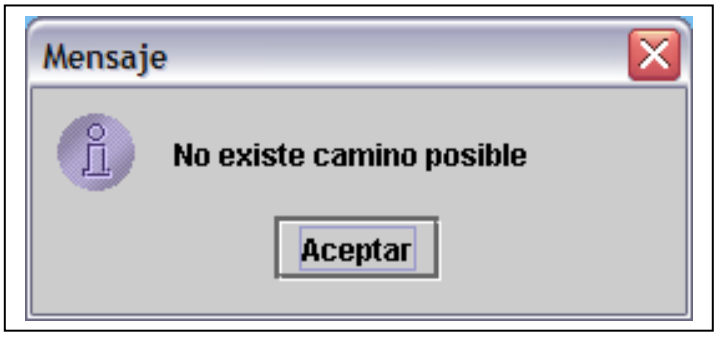


Figura 29 *– Pantalla error*

4. Si no diera el combustible:

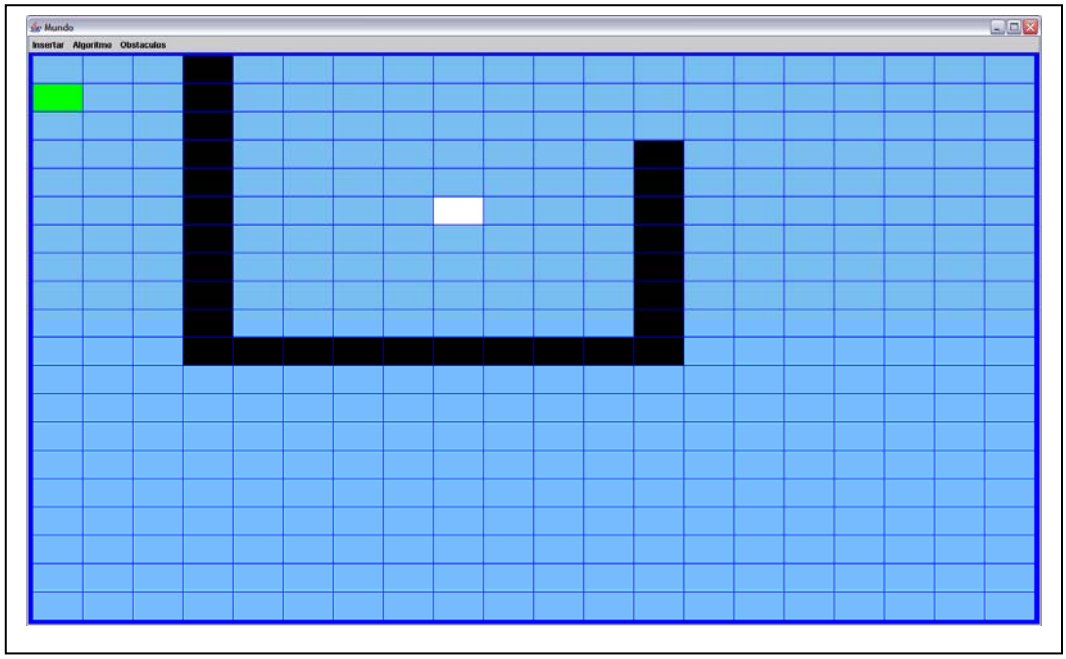


Figura 30 *– Avión sin combustible*

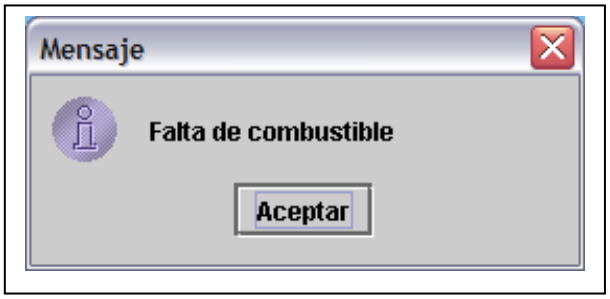


Figura 31 *– Pantalla errror*

Y mostraríamos el camino que recorreríamos si tuviéramos combustible suficiente:

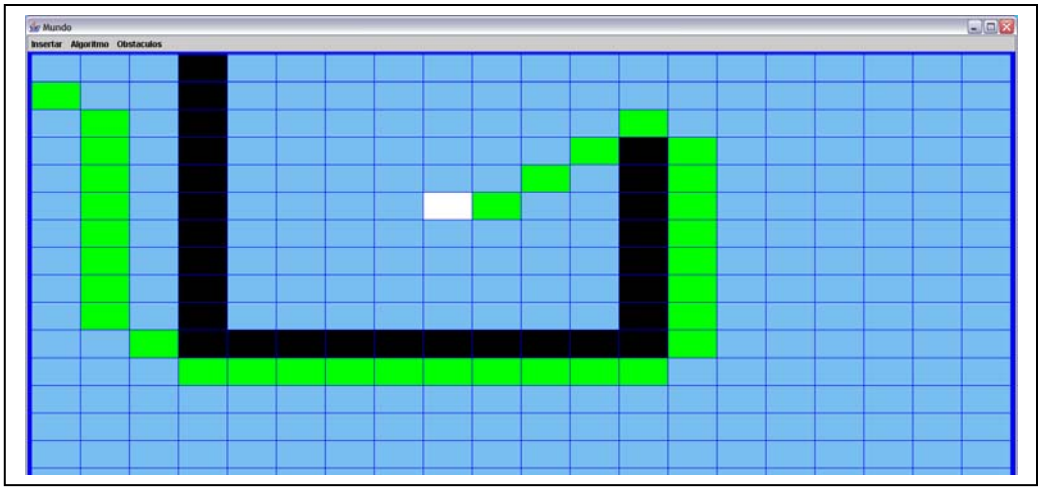


Figura 32 *– Camino*

Destacar que los ejemplos anteriores simulan una interfaz 2D de tamaño

5. Ejemplo de camino infranqueable en 3D:

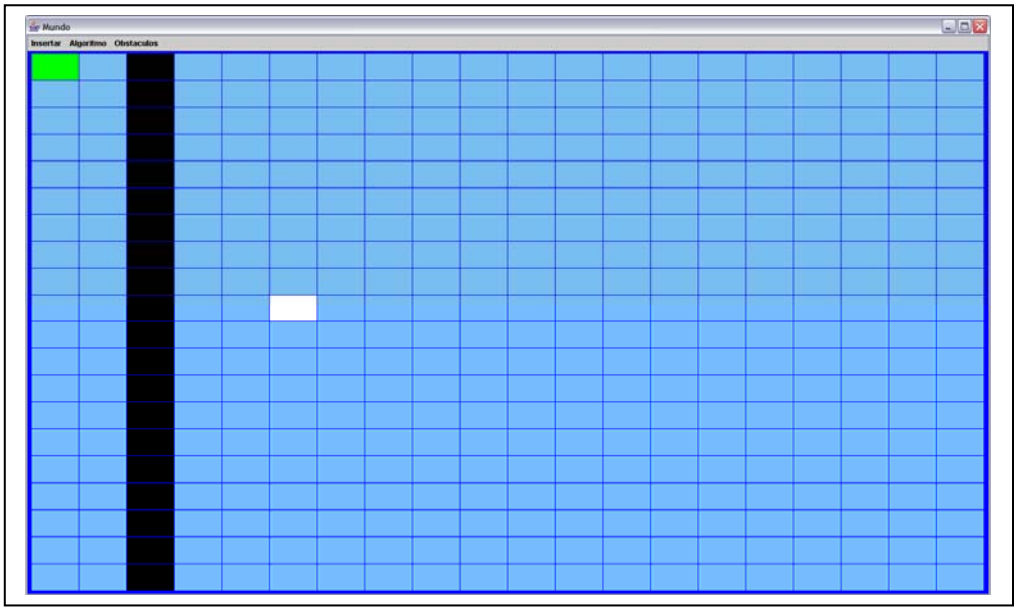


Figura 33 *–Barrera Infranqueable en 3D*

Debido a que tenemos 3D podría alcanzar el objetivo subiendo un nivel en altitud. Estos cambios se muestran en un distinto color.

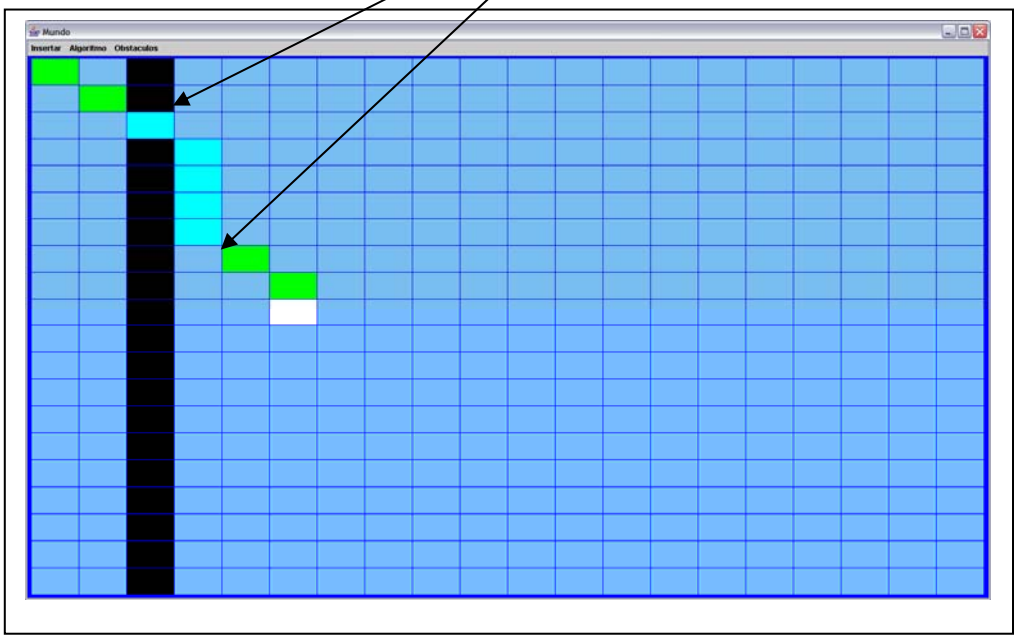


Figura 34 *–Camino en 3D*

A mediados del proyecto, la interfaz adquirió una mayor relevancia y se destinaron mayores recursos a su implementación en tres dimensiones en Matlab, con relieves y obstáculos, que se pasarían posteriormente al algoritmo, el cual decidiría un camino a seguir y se representaría en una vista amigable e intuitiva.

5.2.2 Implementación 2D en Matlab

Un segundo paso antes de generar la interfaz final en 3D se realizaron una serie de pruebas con simulación 2D utilizando el programa Matlab. Esto se llevó a cabo también para aprendizaje de conexiones java – archivos matlab y para seguir depurando código del algoritmo. A continuación se muestra la interfaz:

1. En un primer paso, a la hora de lanzar la aplicación habrá que elegir el objetivo:

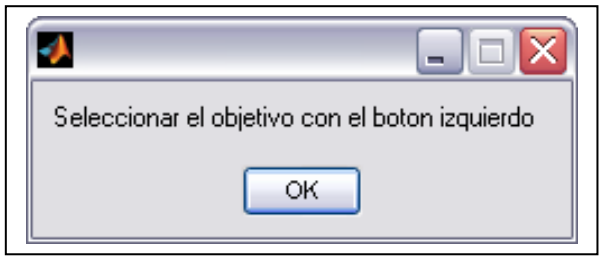


Figura 35 *– Selección del objetivo*

2. Un segundo paso consiste en elegir los obstáculos:

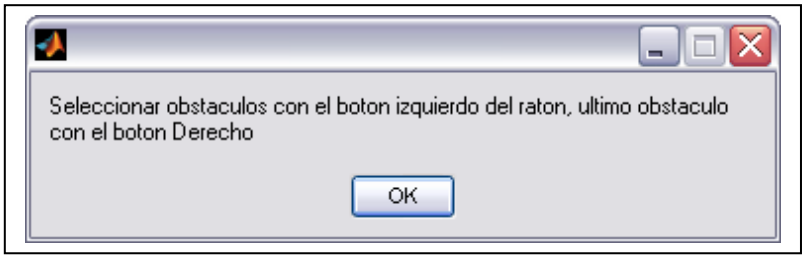


Figura 36 *– Selección de obstáculos*

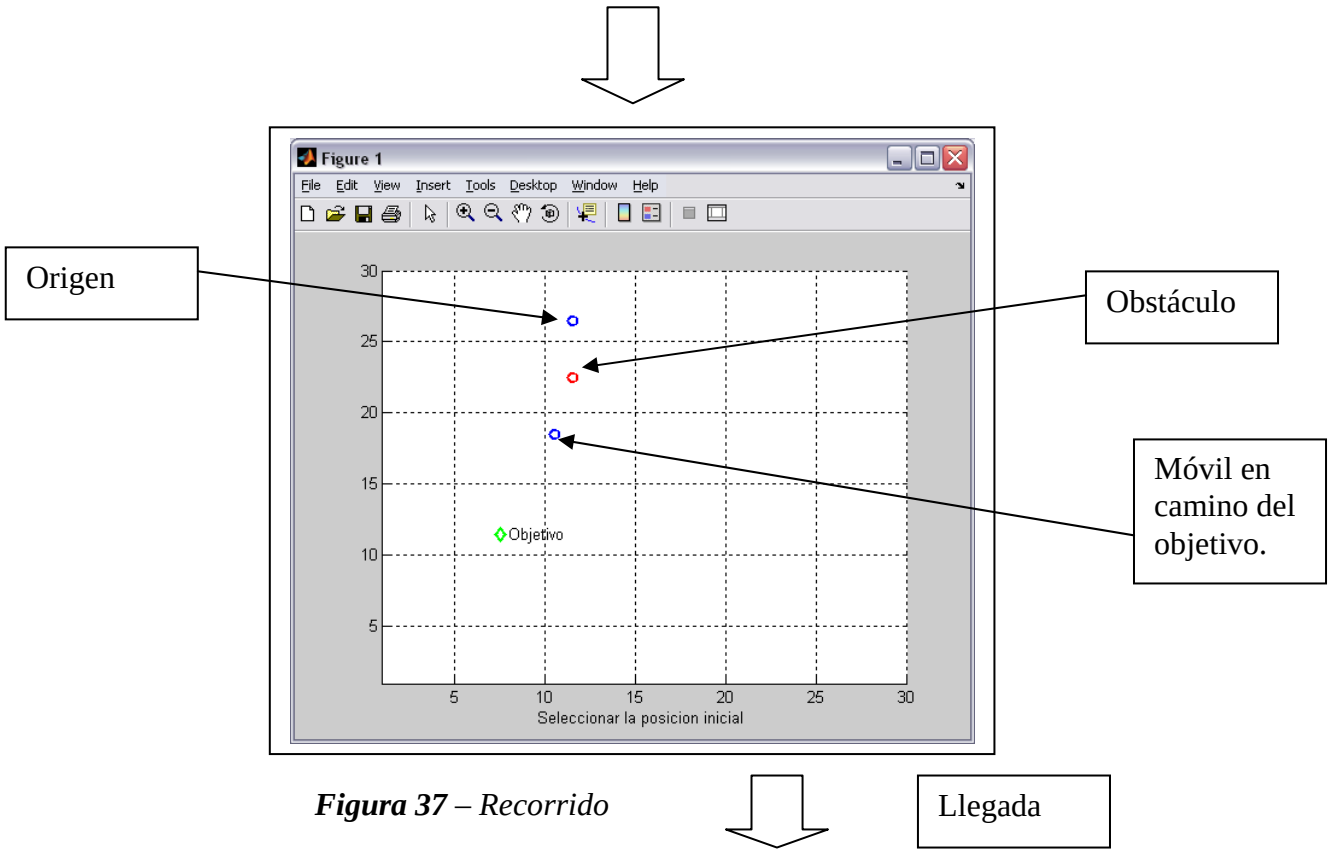


Figura 37 *– Recorrido*

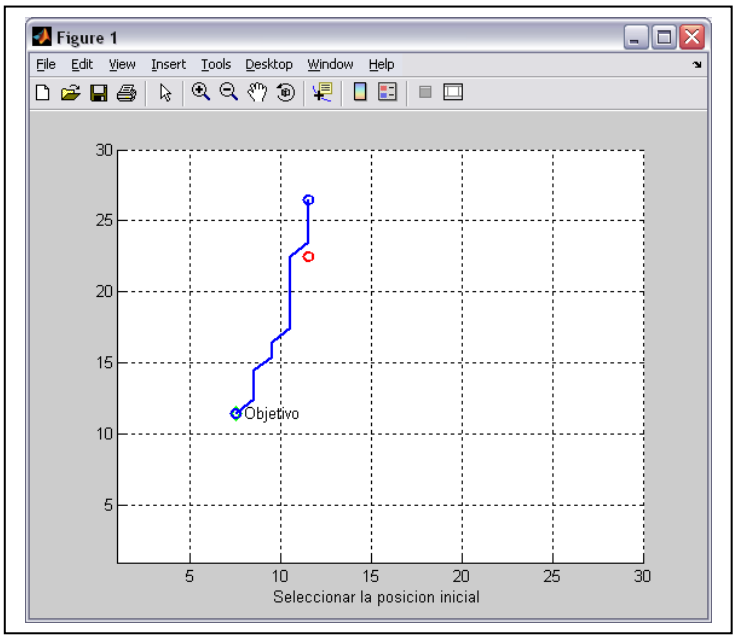


Figura 38 *– Camino óptimo 2D*

5.3 Interfaz gráfica final

Como se ha comentado a lo largo de la memoria, el proceso de creación del proyecto tuvo varios incrementos o fases. En la penúltima fase, se dedicó mayoritariamente a la implementación 3D. Debido a la gran versatilidad y potencia de Matlab para trabajar con aplicaciones que requieren gran potencia de cálculo y a su adaptación a gráficos diferentes tanto en 2D como en 3D, se decidió implementar el salto a 3D de la aplicación con Matlab. La conexión con la aplicación desarrollada en java fue a través de la funcionalidad javaclasspath.

5.3.1 Estructura

La estructura de la interfaz gráfica de usuario puede englobarse en tres sectores:

- Carpeta Images: conjunto de imágenes usadas por la aplicación para la simulación de distintos terrenos (planos, altos, bajos, montañosos) por los que puede ir el avión. Todas son formato .jpeg.
- Carpeta Classes: carpeta donde se alberga todos aquellos archivos .java y .class que forman la aplicación desarrollada en java y que serán utilizados por Matlab para ejecutar la aplicación.
- Archivos Matlab: conjunto de archivos .m donde se implementa la interfaz gráfica de usuario. Recordar que la aplicación se lanza desde matlab, y es ésta quien se encarga de llamar al algoritmo desarrollado en java.

A continuación, explicaremos las funcionalidades de los archivos que componen la interfaz:

- *Carga.m:* es el archivo desde donde se arranca la aplicación o la interfaz. Contiene la inicialización de las variables de la interfaz y la llamada a *main.m*

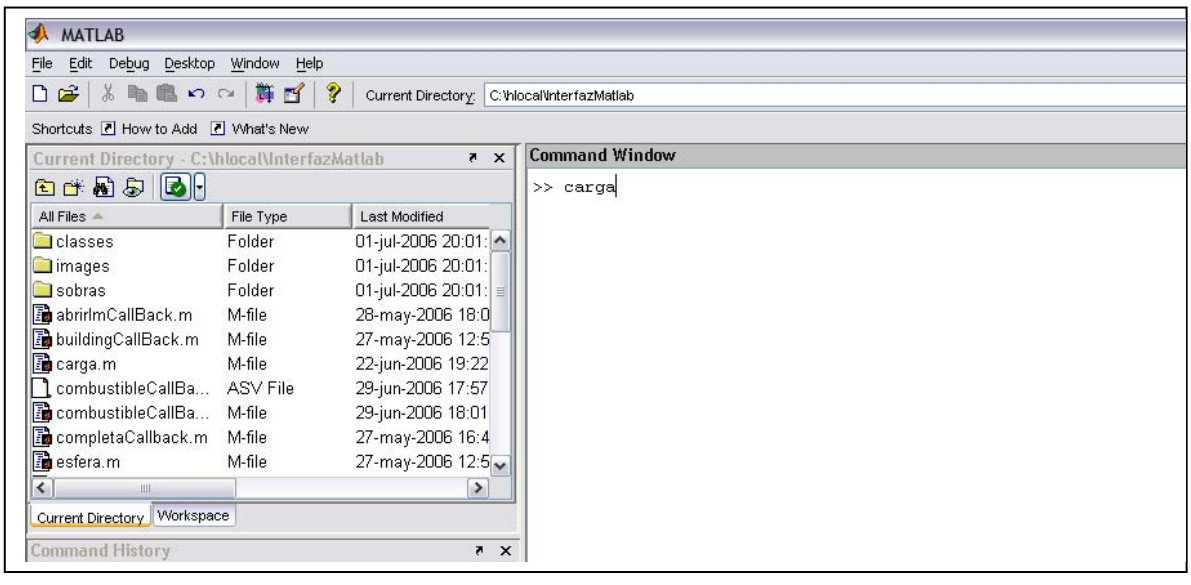


Figura 35 *– Cargar la aplicación*

- *main.m:* inicialmente crear el diálogo para que el usuario pueda seleccionar el tipo de terreno. Se puede incrementar los terrenos simplemente incluyendo los .jpeg en la carpeta de images del proyecto. Después pinta la imagen y el terreno a través de los archivos pintaImagen.m y inicializaTerreno.m

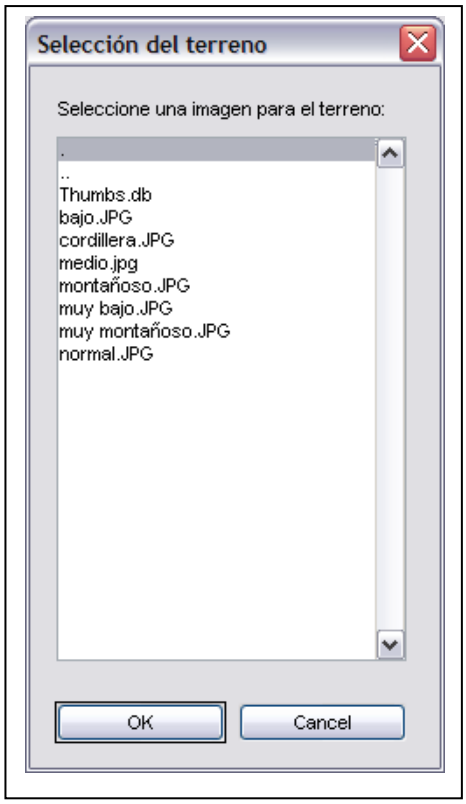


Figura 36 *– Diálogo elegir terreno*

- *PintaImagen.m:* se encarga de pintar el entorno elegido anteriormente. Lo dibujará tanto en 2D como en 3D.
- *inicializaTerreno.m:* inicializa las variables de obstáculos, radares, nodos origen, destino, así como todas aquellas variables que formen parte del terreno.

A su vez, una vez elegido el terreno, se encarga prueba25.m de crear el menú de operaciones que puede hacer el usuario para configurar su propio terreno. Se Veamos esto con más detenimiento:

Ejemplo de terreno (montañoso):

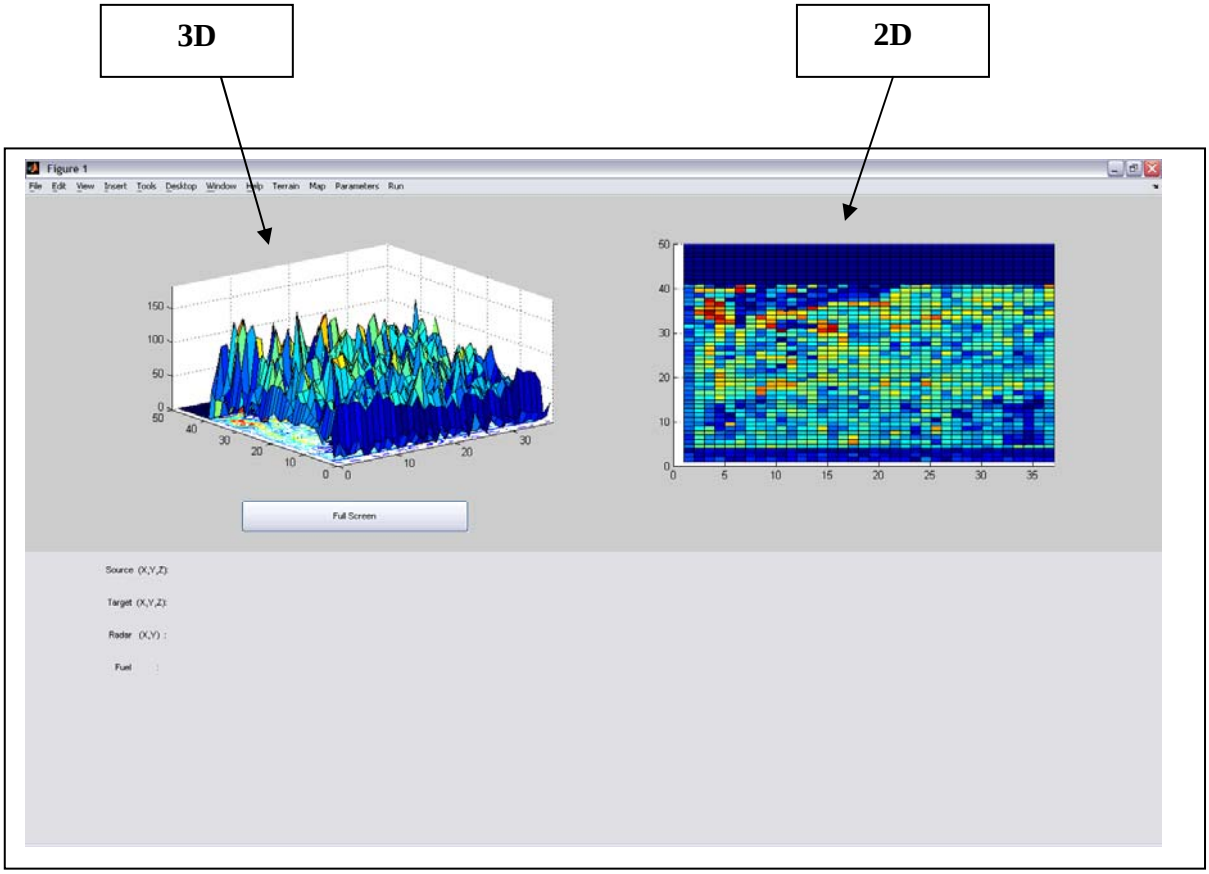


Figura 37 *– Diseño interfaz*

La barra de menú como puede verse en la siguiente figura se compone de menús para manejo de la imagen como guardar, editar, abrir, etc

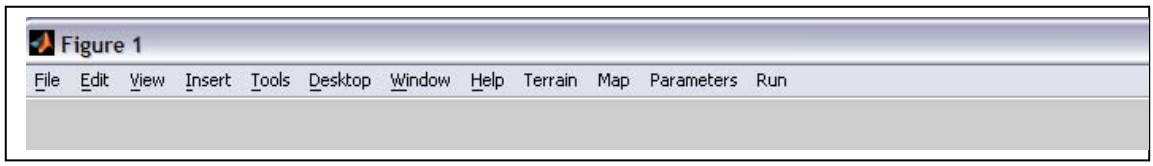
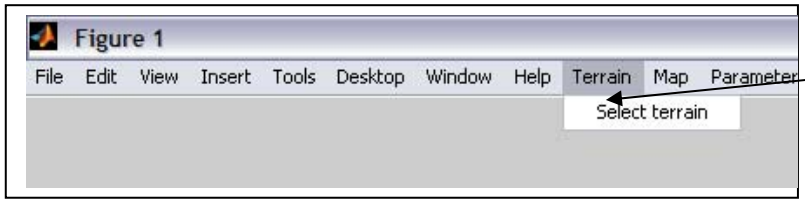


Figura 38 *– Barra de menús*



Terrain nos sirve para poder cambiar de terreno.

Figura 39 *– Menú Terrain*

Map es el menú para elegir los puntos clave: origen (Source), destino (Target) y obstáculos (Obstacle), que puede ser bien un radar o un nodo infranqueable (Impassable):

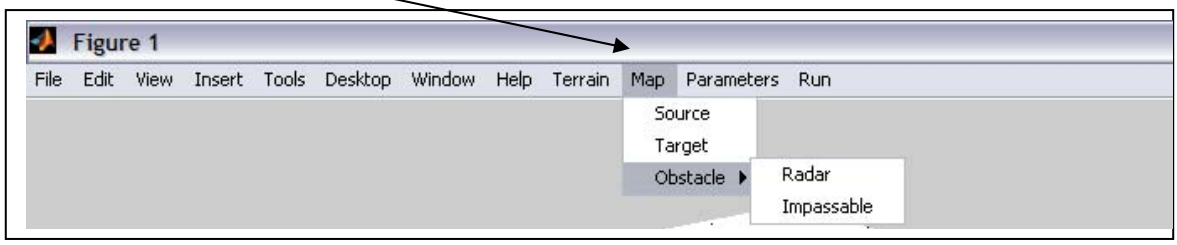


Figura 40 *– Menú Map*

Parameters: variables que influyen en la toma de decisiones. En nuestro caso, está solamente el combustible, pero se deja como posibles incrementos en líneas futuras más adelante explicadas.

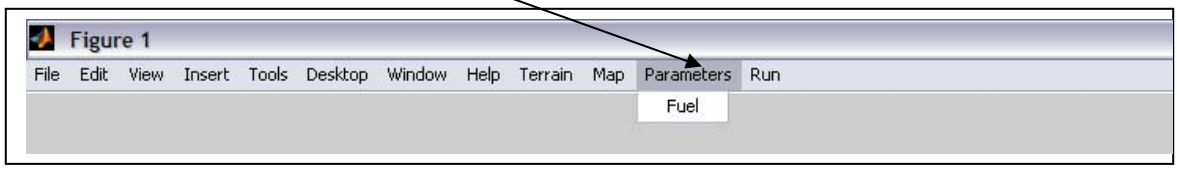
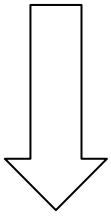


Figura 41 *– Menú Parameters*



Una vez seleccionado, podemos introducir los litros de combustible que le quedan.

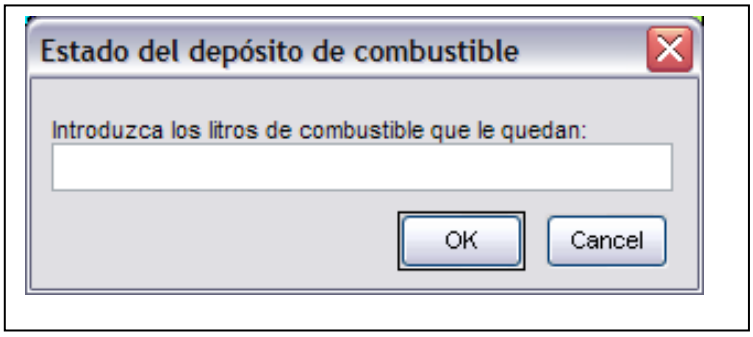


Figura 42 *– Insertar combustible*

Run: para ejecutar la simulación una vez configurado el terreno. Podemos ejecutarla bien en modo normal o con waitpoints (puntos obligatorios por donde tiene que pasar el avión).

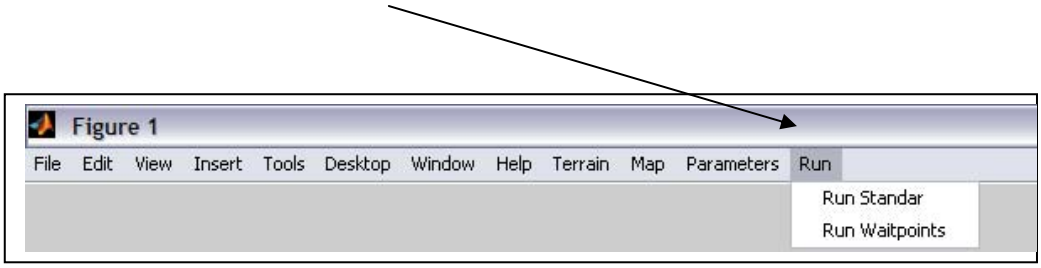


Figura 43 *– Menú para ejecutar*

Los demás archivos Matlab utilizados son para simular la interacción con el usuario con todas las componentes del menú que hemos explicado. Haremos especial mención al archivo:

- ***pintacamino.m :*** es el archivo de enlace con las clases java. Quien llama al algoritmo.

Full screen: botón para poder verlo en pantalla completa el terreno 3D

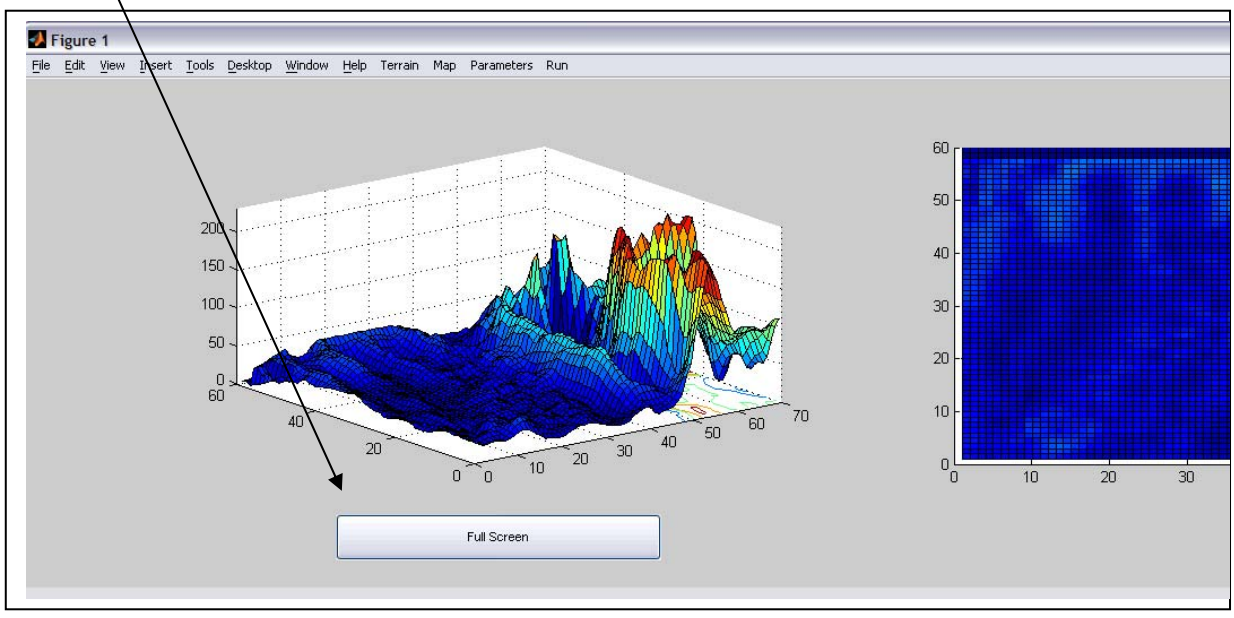


Figura 44 *– Full Screen*

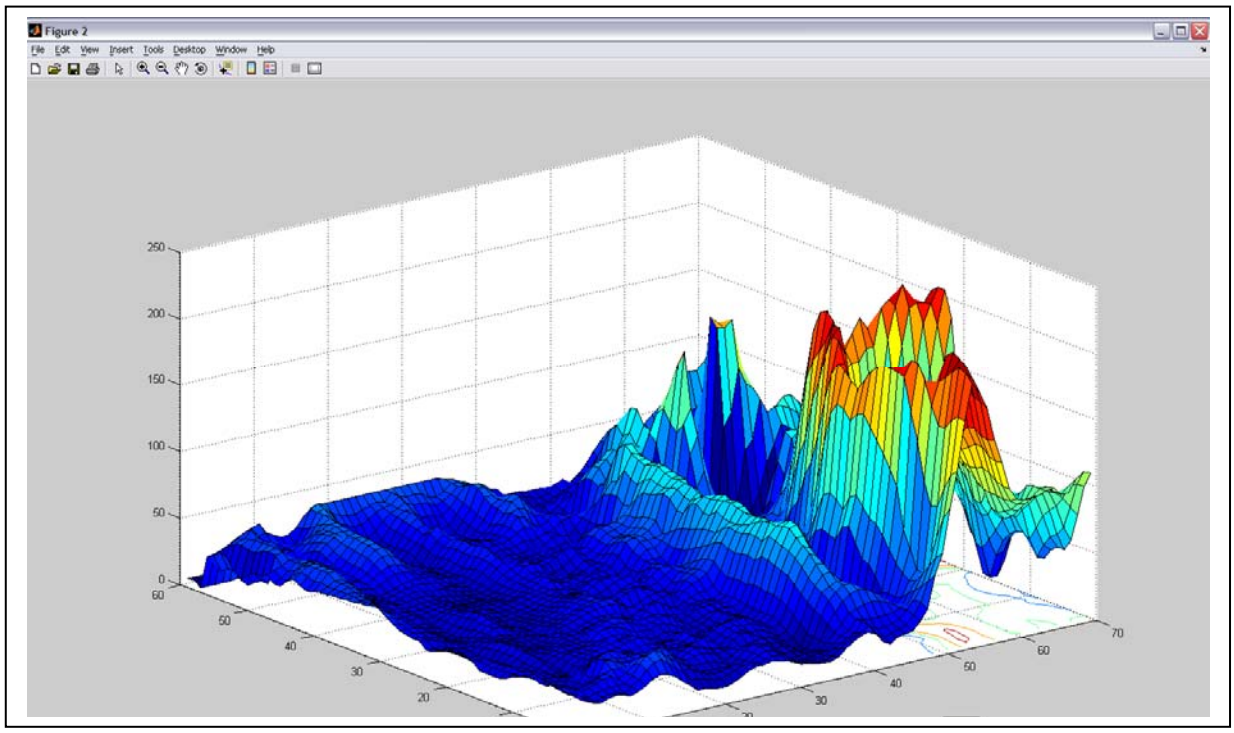


Figura 45 *– Pantalla completa*

5.4 Complicaciones

Las principales complicaciones a la hora de crear una interfaz final válida para el algoritmo o cerebro de la aplicación han sido el modo de desarrollar en paralelo ambas partes, con especial cuidado para hacer posible una posterior integración, con numerosas reuniones, siguiendo diversas pautas y estándares, y rehaciendo trabajo a causa de restricciones impuestas tanto por parte de la interfaz como por parte del algoritmo.

6 DESARROLLO SOFTWARE

Para la aplicación del proyecto, hemos llevado un enfoque sistemático, disciplinado y cuantificable al desarrollo para el funcionamiento del software.
Para tal efecto se han utilizado técnicas de Ingeniería del Software para llevar a la práctica el conocimiento científico en el diseño y construcción de programas de computadora y asociando la documentación requerida para dicho fin.

En el proyecto realizado, la evolución del software va a estar ligada al hardware, dado que este ha sido implementado con algoritmos potentes que requieren mucha capacidad de cálculo lo que implica una mayor complejidad de hardware para un funcionamiento óptimo. No obstante en la actualidad es difícil que el software explote las posibilidades plenas del hardware.

- En la primera fase, se realizó la definición del producto:
  - centrada en el *qué*
  - se identificaron los requisitos del sistema y software:
    - Información a procesar.
    - Función y rendimiento deseados.
    - Comportamiento del sistema
    - Interfaces establecidas
    - Restricciones del diseño

Para llevar a cabo esta parte, se desarrollaron tres tareas principales:

- Planificación del proyecto software.
- Ingeniería de información
- Análisis de requisitos.

- En la segunda fase, se comenzó el desarrollo
  - Centrada en el *cómo*
  - En este apartado se definieron:
    - Las estructuras de datos
    - Implementación de las funciones
    - Caracterización de las interfaces
    - Traducción del diseño a un lenguaje de programación
    - Pruebas a realizar

Para llevar a cabo las tareas de esta fase consistieron en:

- Diseño del software
- Generación del código
- Pruebas del software

- También se realizó una fase de mantenimiento:

  - centrada en el *cambio asociado a*

    - Corrección de errores
    - Adaptaciones requeridas por la evolución del entorno software
    - Cambios en los requisitos

- Modelo de proceso:

Se eligió el modelo de proceso basándonos en las siguientes características:

- Entendibilidad
- Visibilidad
- Soportabilidad
- Aceptabilidad
- Fiabilidad
- Robustez
- Mantenibilidad
- Rapidez

Se intentó seguir un modelo evolutivo ya que el desarrollo del proyecto implicaba construcciones de software cada vez más complejas. También consideramos importante este en este tipo de modelos su adaptabilidad a los cambios de requisitos y a las especificaciones parciales del producto, ya que cuando se comenzó, estas no estaban definidas en su totalidad.

El modelo final se basó en un modelo evolutivo en espiral, más específicamente el *modelo evolutivo en espiral de Boston*, el cual comprende las siguientes partes:

- Comunicación con el cliente

En nuestro caso se trata de las reuniones realizadas con el director de proyecto Gonzalo Pajares.

- Planificación
- Análisis de riesgos
- Ingeniería
- Construcción y adaptación.
- Evaluación por el cliente.

Figura 45 *– Ingeniería del Software*

➢ Gestión del proyecto:

1. **Participantes**: En este apartado se hace mención a las personas directamente implicadas en el desarrollo del proyecto. Como desarrolladores, estamos incluidos los tres miembros del grupo del proyecto; Como cliente, el anteriormente citado director asignado a la asignatura de sistemas informáticos; Como usuarios finales, empresas aeronáuticas que se dediquen a realizar proyectos en el entorno de inteligencia artificial sobre móviles capaces de toma de decisiones de forma autónoma.

2. **Equipo de desarrollo**: Optamos por llevar una organización de tipo Descentralizado Democrático (DD) debido al pequeño número de miembros del grupo (tres), intentando que cada uno acometiera distintas tareas. Las decisiones, problemas, enfoques, se llevan a consenso dentro del grupo. La comunicación entres los miembros es horizontal.

6.1 INTEGRACIÓN

Durante el desarrollo del proyecto hemos usado un modelo-vista-controlador (MVC) separando en capas el modelo de la vista y de la lógica de negocio, haciendo lo máximo posible modulable cada una de dichas partes. Gracias a este proceso de desarrollo software, se ha realizado de una manera más sencilla y en menos tiempo la integración de los módulos de la interfaz gráfica con el núcleo de la aplicación, lo cual ha sido una de las partes cruciales del desarrollo del proyecto.

Para poder realizar la unión con los menos recursos posibles de personas y en un periodo de tiempo razonable, se decidió crear dos clase java cuyos objetos optimizaran los recursos que nos proporciona la plataforma matlab.

Exponemos estas clases a continuación:

EspacioNode: Esta clase identifica los nodos que configuran el espacio de simulación siendo accesible a cada uno de ellos las tres coordenadas del espacio en el que esta situado, y el peso (o riesgo) que tiene cada uno para la simulación.

Espacio: Esta clase representa el espacio 3D de simulación. En ella se configuran las dimensiones del mapa del terreno, el origen y el destino que ha de recorrer el UAV, y un atributo cubo, el cual guarda todas las posiciones posibles de nodos EspacioNode, que contiene la clase Espacio.

El motivo por el cual se ha implementado de esta forma es para sacar el máximo rendimiento a la potencia de matlab, respecto al cálculo con matrices.

Exponemos a continuación los principales ficheros ***.m** en los que se han creado objetos java de las clases anteriormente descritas:

PintaCamino: Este fichero es fundamental ya que es donde se realiza la importación de las clases java anteriores. Para este cometido, hay que añadir manualmente mediante la intrucción "*javaaddpath (dir)"* incluyendo la dirección completa del directorio donde se encuentran las clases *EspacioNode* y *Espacio.* Para crear objetos de estas clases, utilizaremos la instrucción *javaObject (paquete_clase, parámetros de entrada)* donde indicamos el pequete donde se encuentra la clase a la que pertenece el objeto que vamos a crear, seguido de los parámetros de entrada. También hemos utilizado librerías java que posee ya Matlab como las que hemos accedido mediante la instrucción *import java.util.** para la creación de ArrayList necesarias para la implementación.

IncializaTerreno: En este fichero se minimiza el espacio de acción del UAV para optimizar y mejorar el coste espacial y temporal de la aplicación. En él también se especifican todas las variables necesarias en la simulación.

7 Resultados

7.1 Testing y Capturas Finales

Una vez seleccionado el terreno vamos añadiendo el origen y destino.

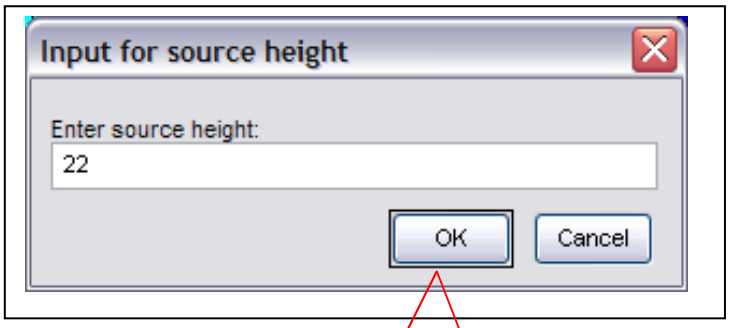
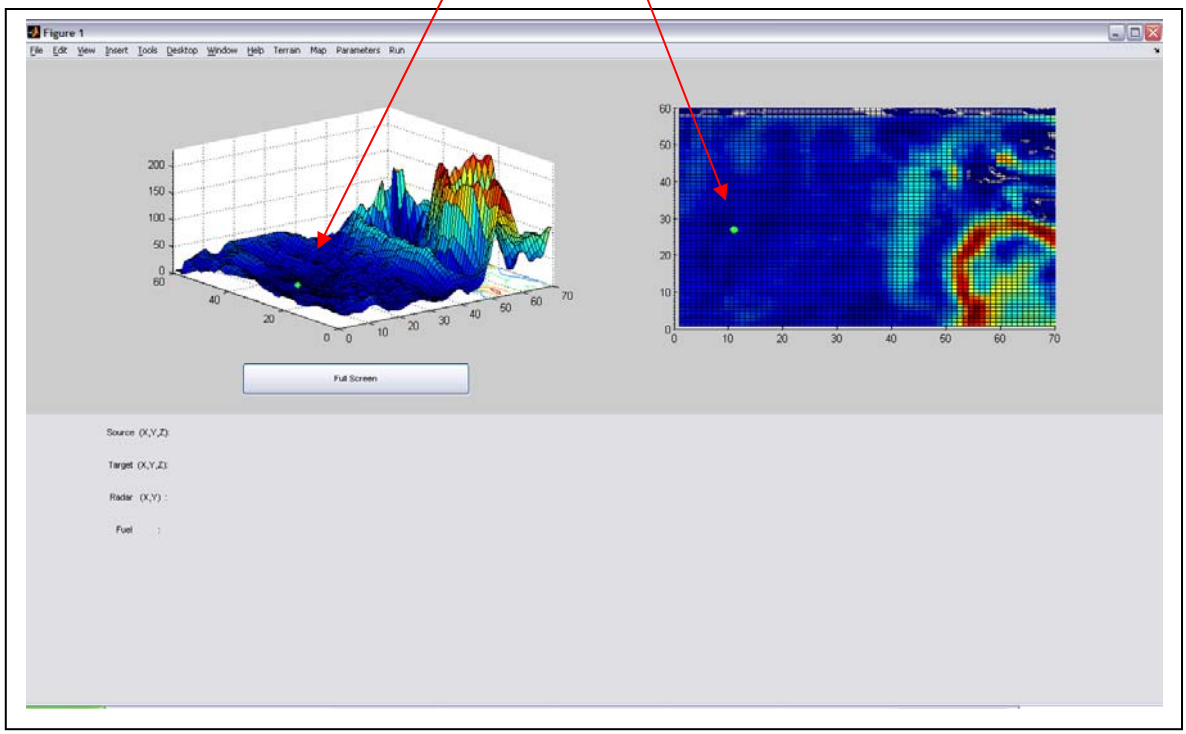


Figura 46 *– Origen*

ORIGEN



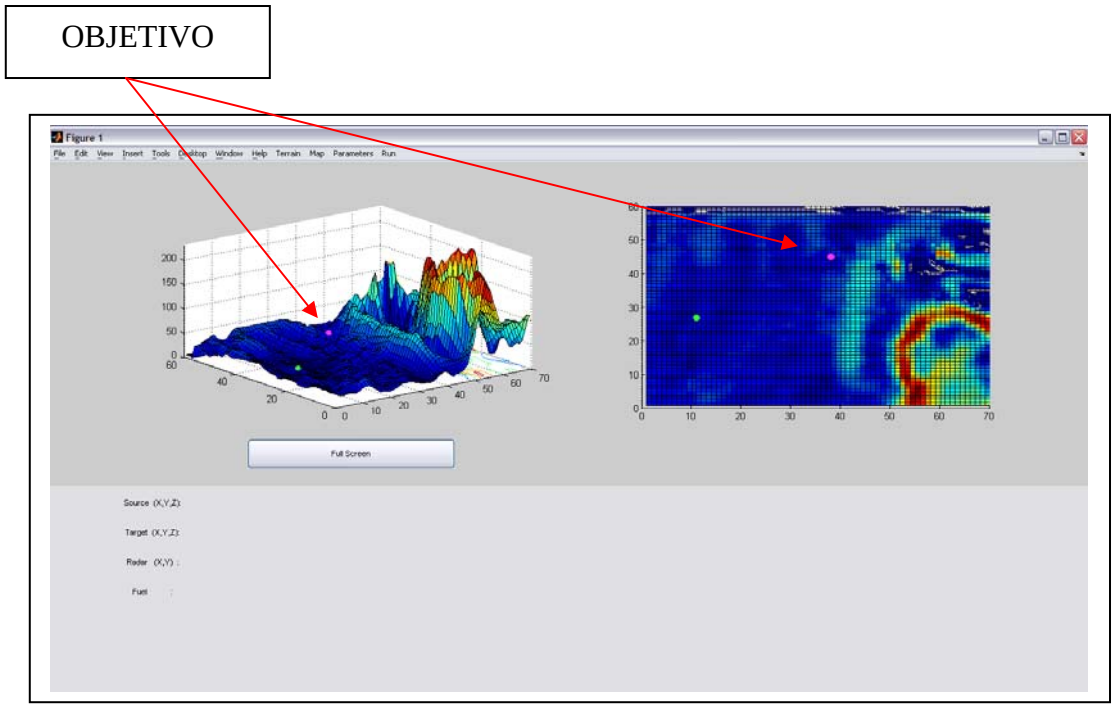


Figura 47 *– Objetivo*

Camino resultante:

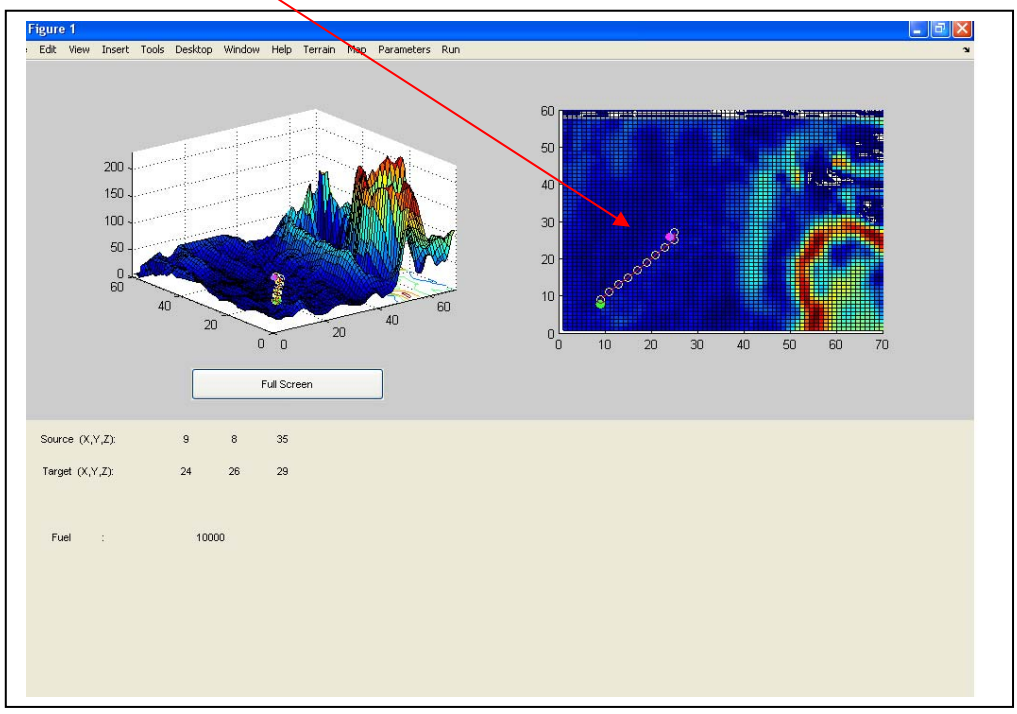


Figura 48 *– Camino*

7.2 Objetivos Completados

Aplicación de métodos de Inteligencia Artificial para la toma de decisiones en simulación de móviles.

La realización del proyecto se ha realizado en varias fases:

- **FASE 1:**

  - Realización del algoritmo A Estrella, en lenguaje Java, para el cálculo del camino óptimo de un vehículo autotripulado en un escenario de simulación en dos dimensiones (2D) hostil.
  - Propuesta, estudio y posterior inlusión de obstáculos que afecten en la toma de decisiones del UAV.
  - Creación de distintos tipos de obstáculos con distinto peso (hostilidad) que representan las diferentes amenazas o zonas de imposible acceso (obstáculos naturales, edificios,…).
  - Inclusión de obstáculos variables que representan los “Radares” de detección de aeronaves, siendo su peso mayor en el centro y menor en la periferia.
  - Implementación en Java de una interfaz de usuario para la visualización y configuración de las diferentes simulaciones que se pueden probar en ella.

- **FASE 2:**

  - Ampliación del algoritmo realizado en la *Fase 1* para la simulación en tres dimensiones (3D).
  - Consideración de la superposición de obstáculos del tipo “Radar”.
  - Modificación de la interfaz gráfica realizada en Java para las pruebas en 3D clasificando los obstáculos mediante un código de colores.

- **FASE 3:**

  - Inclusión de variables de vuelo: Gasto de combustible. Si el UAV no puede llegar al objetivo con el combustible que posee saldrá un mensaje por pantalla y mostrará el camino si hubiera tenido el combustible suficiente.
  - Creación de una interfaz gráfica en 2D realizada en Matlab para las pruebas de ejecución de Java desde Matlab.

- **FASE 4:**

  - Realización de un interfaz gráfica realizada en Matlab pero con capacidad de simulación 3D.
  - Pruebas realizadas con distintos escenarios de simulación.
  - Memoria explicativa del proyecto.

8 Trabajo Futuro

La complejidad de los escenarios y de las soluciones obtenidas deberá ir aumentando hasta la finalización del programa.

Consideraciones generales:

- La evolución de los modelos software.
- La realización de una interfaz gráfica que mejore la visualización y la simulación mediante la inserción de todas las variables de vuelo.
- La sustitución del modelo actual de avión por otro más complejo en el que por ejemplo el gasto de combustible dependerá de las maniobras del avión y no como en el actual modelo en el que se considera un gasto de combustible lineal con el tiempo, incorporación de contramedidas, etc.
- La incorporación del terreno, lo cual podría provocar, por ejemplo, enmascaramientos de los radares creando rutas alternativas para el vehículo.
- Las restricciones de vuelo como el tiempo de llegada al objetivo lo que implica el hecho de poder aumentar o disminuir la velocidad.
- La existencia de obstáculos dinámicos como pueden ser los radares. Éstos pueden activarse en cualquier momento haciendo al avión vulnerable por lo que deberá recalcular un nuevo camino al destino teniendo en cuenta las restricciones de vuelo impuestas.
- La capacidad de vuelos en escuadrilla en formación y realización de misiones conjuntas tanto de aviones autotripulados en su totalidad como mezcla de aviones pilotados y autopilotados.
- Si el vehículo es amenazado por las defensas antiaéreas y posteriormente atacado con misiles del tipo “tierra-aire”, éste debe calcular una ruta de escape y maniobras evasivas recalculando después el nuevo camino al objetivo.
- La posibilidad de envío en tiempo real desde un puesto avanzado de control aéreo los mapas del terreno con las coordenadas de las “zonas calientes” a las que se debe enfrentar y sortear en lo posible según las especificaciones de la misión.
- La completa capacidad “aire-tierra” y “aire-aire” con lo que el avión podrá enfrentarse también a amenazas que tengan capacidades de vuelo como puede ser un avión enemigo.

8.1 Algoritmo de búsqueda

Un sistema que no cambia es un sistema estático (es decir, determinístico.) Muchos de los sistemas son sistemas dinámicos, los cuales cambian a través del tiempo. Cuando nos referimos a que cambian a través del tiempo es de acuerdo al comportamiento del sistema. Cuando el desarrollo del sistema sigue un patrón típico decimos que tiene un patrón de comportamiento. El sistema será estático o dinámico dependiendo del horizonte temporal que se escoja y de las variables en las cuales se está concentrado. El horizonte temporal es el período de tiempo dentro del cual se estudia el sistema. Las variables son valores cambiables dentro del sistema. En los modelos determinísticos, una buena decisión es juzgada de acuerdo a los resultados. Sin embargo, en los **modelos probabilísticos**, no solamente nos preocupamos por los resultados, sino que también con la cantidad de riesgo que cada decisión acarrea.

El concepto de probabilidad ocupa un lugar importante en el proceso de toma de decisiones. En muy pocas situaciones de toma de decisiones existe información perfectamente disponible – todos los hechos necesarios.- La mayoría de las decisiones son hechas de cara a la incertidumbre. La probabilidad entra en el proceso representando. Los modelos probabilísticos están ampliamente basados en aplicaciones estadísticas para la evaluación de eventos incontrolables (o factores), así como también la evaluación del riesgo de sus decisiones.

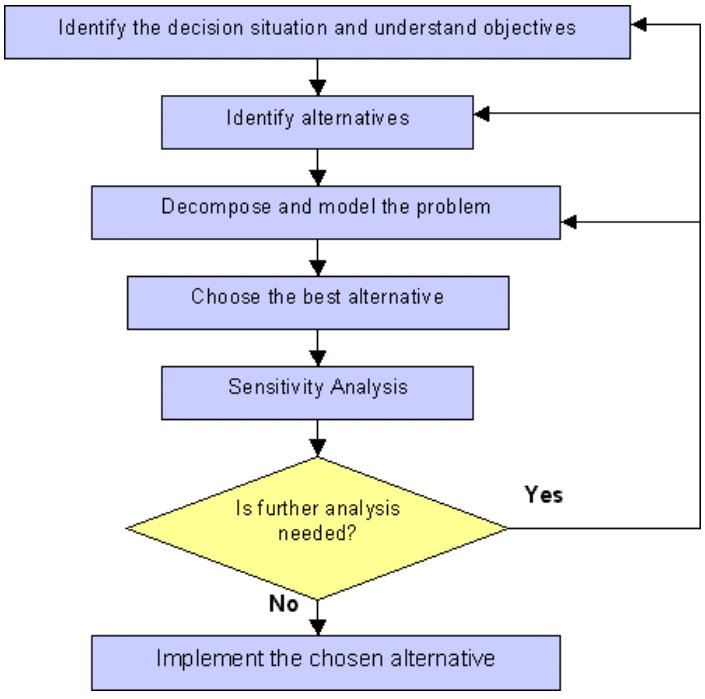


***Figura 49** – Diagrama probabilísticos*

El estudio sistemático de la toma de decisiones proporciona el marco para escoger cursos de acción en situaciones complejas, inciertas o dominadas por conflictos. La elección entre acciones posibles y la predicción de resultados esperados resultan del análisis lógico que se haga de la situación de decisión.

Las continuas decisiones que el UAV debe tomar para conseguir el camino óptimo y más seguro hasta el objetivo lleva consigo un riesgo lógico debido a la concepción militar del proyecto. Este riesgo implica cierto grado de incertidumbre y la habilidad para controlar plenamente los resultados o consecuencias de dichas acciones. Sin embrago, en algunos casos la eliminación de cierto riesgo podría incrementar riesgos de otra índole. El manejo efectivo del riesgo requiere la evaluación y el análisis del impacto subsiguiente del proceso de decisión. Este proceso permite al UAV evaluar las estrategias alternativas antes de tomar cualquier decisión.

El proceso de decisión se describe a continuación:

1. El problema esta definido y todas las alternativas confiables han sido consideradas. Los resultados posibles para cada alternativa son evaluados.

2. Los resultados son discutidos de acuerdo al riesgo que se toma y las restricciones de vuelo.

3. Varios valores inciertos son cuantificados en términos de probabilidad.

4. La calidad de la estrategia óptima depende de la calidad con que se juzgue. El UAV deberá examinar e identificar la sensitividad de la estrategia óptima con respecto a los factores cruciales.

Ahora el algoritmo escogerá un nodo u otro dependiendo de la amenaza que exista en un determinado punto sino de la probabilidad que existe de pasar por ese punto y el UAV ser derribado. La heurística se calculará mediante ecuaciones de probabilidad y teniendo en cuenta las variables cambiantes del sistema como el combustible, tiempo de vuelo o contramedidas que tenga el avión en ese momento.

8.2 Interfaz de simulación

Actualmente la IU dispone de dos versiones de visualización de la simulación del UAV:

1. La vista por satétilite (2D)
2. La vista 3D

Se debe considerar:

- La posibilidad de dotar a los mapas del terreno de un mayor realismo.
- La visualización de la simulación en distintas ventanas:
  - Vista satélite.
  - Vista exterior a nivel del UAV.
  - Vista interior en la cabina del UAV.
  - Vista según los obstáculos por los que va pasando el UAV.
- Configuración de todas las variables del sistema de vuelo en una ventana distinta a la visualización de la simulación.
- La capacidad de introducir nuevos obstáculos en tiempo de ejecución para comprobar las capacidades del UAV en la toma de decisiones.
- Que el usuario pueda introducir nuevos terrenos en la simulación.
- Poder guardar las simulaciones realizadas para estudiarlas más tarde y construir estudios estadísticos según las capacidades que vaya desarrollando el UAV.
- La construcción de reglas para la toma de decisiones y poder guardarlas en una base de datos para utilizarlas dependiendo de las características de la misión del UAV, mapa del terreno y número y tipo de amenazas.
- La creación de una ventana con la configuración de las características del UAV:
  - Potencia de los motores.
  - Defensas: contramedidas lanzables (libreas antirradar chaff y bengalas), contramedidas electrónicas, etc.
  - Defensas “aire-aire” o “tierra-aire”
  - Mimetizado del fuselaje.
  - Configuración automática o manual.

9 AGRADECIMIENTOS

- Gracias al director del proyecto y profesor de la Facultad de Informática de la Universidad Complutense de Madrid, Don Gonzalo Pajares Martinsanz, por aceptarnos en su proyecto para la asignatura de Sistemas Informáticos y por prestarnos la ayuda que le hemos pedido durante el transcurso de su realización.

- Gracias a Don Jesús Manuel de la Cruz García , profesor de la Facultad de Informática de la Universidad Complutense de Madrid por animarnos y orientarnos en el desarrollo de la interfaz gráfica implementada en Matlab.

- Gracias a nuestras familias por ayudarnos en establecer un lugar tranquilo de trabajo, por su apoyo en todo momento y haciendo más llevadero el desarrollo del proyecto.

10 Bibliografía

Para la documentación sobre este proyecto nos hemos basado principalmente en libros de Inteligencia Artificial, fuentes de Internet y tutoriales.

Sobre Eads-Casa.
Para la documentación se ha hecho referencia en la siguiente web oficial:

http://www.eads.com

Sobre eclipse:

http://www.eclipse.org/
http://www.emagister.com/ (tutorial de eclipse)

Sobre Matlab:

http://www.mathworks.com

Sobre A*:

http://club2.telepolis.com/ohcop/aasteris.html

[1] Gonzalo Pajares y Matilde Santos; *Inteligencia Artificial e ingeniería del Conocimiento;* RA-MA, 2005 (Primera Edición en español) ISBN: 84-7897-676-0;

[2] Russell, S. y Norvig, P. ; *Artificial Intelligence: A Modern Approach ;* Prentice Hall, New Jersey, Edición del 2004 en español;