Sistemas Informáticos
Curso 2005-2006

Utilización de métodos de simulación basados en técnicas de Inteligencia Artificial aplicados a objetos móviles.

Diana Díaz Agrela
María Menéndez Blanco
Pedro Antonio Hernández Torres

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

Facultad de Informática
Universidad Complutense de Madrid

Prefacio

En este proyecto se ha producido un acercamiento hacia la inteligencia artificial mediante la creación de un algoritmo de búsqueda informado que proporciona una ruta óptima entre dos puntos.

Por algoritmo de búsqueda informado entendemos aquél que para realizar los cálculos que proporcionan la ruta óptima utiliza un cierto conocimiento previo que hace que la ruta sea en un cierto sentido guiada. No se puede hablar de búsqueda propiamente guiada porque en tal caso no habría que realizar ningún tipo de búsqueda, ya que el camino sería obtenido trivialmente.

El hecho de que sea informado quiere decir que tiene una cierta información que le permite intuir hasta cierto punto cual es el camino óptimo.

El algoritmo concreto que se ha utilizado ha sido el de búsqueda A* que proporciona la seguridad de que si hay camino, entonces va a encontrarlo y devolver el óptimo. Además, entre los algoritmos que garantizan esta propiedad, es el más eficiente.

La cuestión de la eficiencia es fundamental, ya que estos algoritmos consumen mucho tiempo si se aplican a sistemas grandes y resolver un problema de forma eficiente puede convertir un problema inabordable en factible.

Preface

In this project we have approached to artificial intelligence by means of the implementation of an informed search algorithm that produces an optimal path between a source point and a target point.

An informed search algorithm is an algorithm which makes the computations to provide the optimal path having a certain knowledge that makes it in some sense, to be guided to the optimal path. It is not exactly a guided search because in that point it would be unnecessary to search a path. It will be got immediately.

Informed search means that the algorithm is provided with certain clues that make it understand where it should go to find the best path.

The concrete algorithm that has been implemented is called A* algorithm which assure that if there is a path, the algorithm is going to find them and, overall, the optimal path is going to be found.
Moreover, among all the algorithms that fulfils this property, A* is the most efficient one. The efficiency problem is crucial. These kinds of algorithms need much time if they are asked to solve a complex system and the chance of solving a problem in an efficient way can transform it from an unsolvable problem to a resoluble one.

Lista de Palabras

- Algoritmo
- Heurística
- Camino
- Mínimo
- Completo
- Óptimo
- Inteligencia
- Simulación
- Trayectoria
- Terreno

Diana Díaz Agrela, María Menéndez Blanco y Pedro Antonio Hernández Torres, autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, la memoria desarrollada en este proyecto.

Madrid, a 30 de Junio de 2006

Fdo. Diana Díaz Agrela

Fdo. María Menéndez Blanco

Fdo. Pedro Antonio Hernández Torres

Agradecimientos

Queremos dar las gracias a nuestro tutor Gonzalo Pajares Martinsanz por darnos la oportunidad de realizar este proyecto y apoyarnos en la elección de realizarlo en colaboración con otro de los grupos de Sistemas Informáticos.

Agradecemos también su inestimable ayuda a Jesús Manuel de la Cruz, por encauzarnos en el proceso de desarrollo y ofrecernos sus conocimientos siempre que lo hemos necesitado.

Por último, agradecemos al otro grupo colaborador la buena compenetración que hemos tenido que ha hecho posible que, con la integración de los proyectos, se haya podido alcanzar un mejor resultado.

Introducción

Nuestro proyecto es una aportación a una investigación que se está realizando por el grupo ISCAR (Ingeniería de sistemas, control, automatización y robótica) de la Universidad Complutense de Madrid, el director de dicho grupo es el catedrático de Ingeniería de Sistemas y Automática Jesús Manuel de la Cruz. El grupo está alojado en la facultad de informática de la universidad.
El proyecto de investigación comienza con la firma de un contrato entre ISCAR y la empresa aeronáutica europea EADS-CASA, concebido con el título: Evaluación de Técnicas de Inteligencia Artificial para toma de decisiones y optimización de Estrategias.

El primer contacto que tuvimos con nuestro proyecto fue una conversación con el profesor Gonzalo Pajares Martinsanz, nuestro tutor a lo largo del proyecto, en un principio solo queríamos informarnos sobre los proyectos que iba a ofrecer para el curso 2005-2006. La primera descripción que nos motivó a la hora de elegir este proyecto era, investigar a cerca de la inteligencia artificial necesaria para un avión no tripulado. Además el hecho de ser un proyecto real para una empresa era un aliciente añadido.

Esta propuesta se fue matizando, nuestro trabajo sería centrarnos en el desarrollo de la inteligencia artificial para encontrar el camino mínimo que debería tomar el avión junto con ciertos requisitos, ya que en el mapa del terreno podrían aparecer distintos tipos de obstáculos, y no todos de la misma categoría. Se incluyeron algunos aspectos más como el problema del combustible, la posibilidad de reevaluar la situación al aparecer cambios en la escena y de crear misiones con más de un objetivo.

Todo el proceso se tenía que hacer en una primera aproximación en dos dimensiones, para poder comprobar fácilmente la eficacia del algoritmo, y una segunda versión final que debía estar en tres dimensiones.

Otro aspecto muy importante es la colaboración con otro de los grupos de Sistemas Informáticos, ambos grupos estábamos interesados en el proyecto y la cooperación conjunta nos proporcionaba unas expectativas más amplias para el desarrollo. Esta colaboración ha tenido que ser coordinada y planificada en todo momento, el resultado es uno de los factores más destacables del proyecto.

Planteamiento inicial

Es necesario tratar el planteamiento inicial realizado a la hora de afrontar el proyecto, para el cual es necesario matizar con detalle los requisitos deseados para poder ir introduciéndolos en las sucesivas iteraciones con el fin de llegar al resultado final esperado.

Para cumplir las especificaciones requeridas en un inicio, el profesor Gonzalo Pajares nos indicó que debíamos implementar el algoritmo A*, es el algoritmo de búsqueda del camino mínimo mas eficiente de los que nos garantizan encontrar una solución en el caso de que exista. Dicho algoritmo le estudiamos en detalle en la siguiente sección.

El lenguaje que decidimos utilizar en consenso con el profesor era Java, buscamos la herramienta más apropiada para el desarrollo, nuestra elección fue Eclipse por ser un software libre conocido por los integrantes de ambos grupos, y de fácil manejo. Además su conexión con el CVS, que comentaremos inmediatamente, es muy cómoda y rápida, factores imprescindibles para una buena implementación.

Al tener que realizar el desarrollo ente dos grupos, la coordinación fue uno de los primeros temas que tuvimos que concretar, la necesidad de herramientas para la comunicación (grupo de mensajería vía e-mail), y la unificación de versiones (utilización de un CVS Concurrent Versions System), así como una planificación detallada para cada iteración del desarrollo, apoyada por documentos creados para las reuniones, casos de uso y el reparto de tareas.

Aunque el proyecto se refiere a inteligencia artificial, para poder mostrar los resultados de dicha inteligencia, fue necesario pensar en una interfaz que cumpliera las expectativas. La primera aproximación, dos dimensiones, fue hacer una GUI en Java sencilla. La mayor complicación surge al desarrollar el programa en tres dimensiones, necesitábamos hacer una simulación suficiente para ver el resultado del algoritmo, no se trataba de hacer una simulación de vuelo, sino una interfaz gráfica en tres dimensiones en la que pudiéramos representar los elementos del terreno, el origen, destino, y el camino final calculado. Además de otros requisitos para una interacción con el usuario lo más intuitiva posible.

Contemplamos varias propuestas, como Java 3D, y C++ con OpenGL ambas soluciones aumentaban de una forma excesiva la complejidad del proyecto, finalmente optamos por la elaboración de un entorno en Matlab como recomendación del profesor Jesús Manuel de la Cruz. Más adelante veremos las ventajas e inconvenientes de nuestra elección.

La interfaz se convierte en este momento en una de las partes que, junto con la inteligencia, forman el grueso del proyecto, a partir de ahora denotaremos al programa Matlab que se encarga de la representación como *interfaz* y el programa java que implementa el algoritmo A* como *inteligencia.*
Con esta nueva división del proyecto se ve afectada la coordinación, y la distribución de tareas, que en un principio se hacía sobre los seis componentes, pasa a separarse en dos grupos, para realizar una posterior integración. Todos los detalles de la coordinación los expondremos más adelante.

Algoritmo A*

El algoritmo de búsqueda A* es un algoritmo de búsqueda inteligente que busca el camino de menor coste entre un estado inicial y otro destino. Para ello utiliza una heurística que debe ser óptima.

El algoritmo se basa en un espacio de estados en el que partiendo de un estado inicial, se va generando una lista de nodos estado a los que se puede llegar desde alguno de los nodos estado ya visitados, y que estarán más cerca del estado objetivo que el estado actual.

El mecanismo de este algoritmo consiste en escoger el que más promete de entre estos nodos, de tal forma que se acabe alcanzando el estado destino. La elección de ese nodo más prometedor, se basa en la heurística que se haya escogido y que deberá ser admisible.

El hecho de escoger en cada momento el nodo más prometedor no quiere decir que se puedan desechar todos los demás nodos que permanecían en la lista. Esto en ningún momento garantizaría que se eligiera el camino óptimo entre el estado origen y estado objetivo. El uso de la función heurística no es el de elegir el siguiente nodo de un camino(como sería el caso en un algoritmo de búsqueda voraz) sino guiar la búsqueda llevándola por donde parece mejor de tal forma que si desde un nodo no se puede mejorar el camino que se alcanza desde otro nodo hasta el objetivo, entonces no hace falta tratar a este primer nodo podando así el árbol de búsqueda y convirtiendo a este algoritmo en un algoritmo más eficiente que si sólo se realizara una búsqueda de fuerza bruta.

El problema de la eficiencia es fundamental en este tipo de problemas ya que el espacio de estados alcanza grandes dimensiones, y el hecho de podar soluciones que se sabe que no van a ser óptimas, nos puede llevar a no ser capaces de alcanzar la solución por el tiempo que nos llevaría calcularla, por lo tanto debe calcular una solución alcanzable en un tiempo permisible.

Para pasar de un estado a otro debemos utilizar operadores válidos, es decir, desde un estado sólo se puede ir a otros estados concretos, y no a cualquiera. Además, es posible que el nuevo estado deba cumplir alguna restricción para que pertenezca al espacio de estados. Por ejemplo, a la hora de buscar un camino entre dos puntos, hay lugares por los que no se puede pasar.

Para este algoritmo se utilizan tres funciones que llamaremos f',g y h'. La relación entre estas funciones es la siguiente:

$$f'(n) = g(n) + h'(n)$$

donde g es la función de coste de llegar al nodo actual n desde el nodo origen, y h´ es la función de estimación heurística que nos aproxima el coste que supone alcanzar el destino desde el nodo actual n.

Esquema básico del algoritmo de A*

1- Empezamos con una lista de nodos a la que llamamos ABIERTOS, inicialmente sólo contiene el nodo inicial. El valor g en ese nodo será de 0 y el de h' el que se obtenga realizando el cálculo con la función heurística (puede verse que f' valdrá h' en este primer nodo).
Tendremos otra lista CERRADOS que inicialmente estará vacía.

2- Repetimos el siguiente procedimiento hasta alcanzar un nodo objetivo:
Si no existen nodos en la lista ABIERTOS, informar del fallo. En otro caso, tomar el nodo de ABIERTOS con menor valor de f'. Lo llamamos MEJORNODO. Este nodo pasará de estar en ABIERTOS a CERRADOS.
Si MEJORNODO es un nodo objetivo, se informa de la solución. En caso contrario, generamos los sucesores de MEJORNODO, pero sin colocar MEJORNODO apuntando a ellos.

Para cada SUCESOR, se hace lo siguiente:

a. Ponemos SUCESOR apuntando a MEJORNODO. Estos enlaces son los que permiten más adelante recuperar el camino una vez hallada la solución.

b. Calculamos g(SUCESOR) como g(MEJORNODO) + el coste de ir desde MEJORNODO a SUCESOR.

c. Miramos si SUCESOR está en ABIERTOS. En este caso, a este nodo se le llama VIEJO, ya que existía previamente. Como este nodo ya existe en el grafo se puede desechar SUCESOR y añadir VIEJO a la lista de sucesores de MEJORNODO.
Si el coste del camino encontrado hasta SUCESOR es de menor coste que hasta VIEJO, el enlace paterno de VIEJO debería apuntar a MEJORNODO. Para realizar esta comparación se comparan sus valores g. Si VIEJO tiene menor o igual coste, entonces no es necesario realizar ningún cambio. Si SUCESOR tiene menor coste, entonces se hace que el enlace paterno de viejo apunte a MEJORNODO y que se grabe el nuevo camino óptimo en g(VIEJO) y que se actualice f'(VIEJO).

d. Si SUCESOR no se encontraba en ABIERTOS, miramos si se encuentra en CERRADOS. si es así, llamamos VIEJO al nodo de CERRADOS y añadirlo a la lista de sucesores de MEJORNODO. Mirar si el nuevo camino es mejor que el viejo tal y como se hizo en el paso 2(c) y actualizar apropiadamente el enlace paterno y los valores de g y f'. Si se acaba de encontrar un camino mejor a VIEJO, se debe propagar la mejora a los sucesores de VIEJO. VIEJO apunta a sus sucesores. Cada sucesor, a su vez, a sus sucesores y así hasta que cada rama termine en un nodo que está en ABIERTOS o no tiene sucesores.

Es posible que al propagar el nuevo valor de g hacia abajo, el camino que se esté siguiendo pueda volverse mejor que el camino a través del antecesor actual, por lo que deben compararse los dos. Si el camino a través del antecesor actual es mejor, debe detenerse la propagación. Si el camino a través del cual se propaga es mejor, se inicializa el antecesor y se continúa con la propagación.

e. Si SUCESOR no estaba ya en ABIERTOS ni CERRADOS, ponerlo en ABIERTOS y añadirlo a la lista de sucesores de MEJORNODO. Calculamos f'(SUCESOR) = g(SUCESOR) + h'(SUCESOR).

Hay algoritmos que son pequeñas modificaciones del A* que se utilizan con la intención de encontrar una solución en un periodo de tiempo más corto.

El problema que presentan estos algoritmos es que pese a mejorar el coste en tiempo respecto al algoritmo de A*, no garantizan su completitud, es decir, no garantizan encontrar solución aunque la haya.

Una de las propiedades del algoritmo A* es el de la completitud. Además, también garantiza que ningún algoritmo que sea óptimo y completo expande menos nodos que el algoritmo de A*. Es decir, se expande el mínimo número de nodos que garantiza su corrección (salvo quizá en caso de empates en el valor de la función f' en más de un nodo que obliga a expandir alguno más de los necesarios).

Funciones heurísticas:

Las funciones heurísticas son estimaciones de coste que nos permiten aproximar el coste de llegar a un estado objetivo desde cualquier estado del espacio de estados del problema que se intenta resolver.

Una función heurística es admisible si nunca sobreestima el coste entre el nodo sobre el que estoy estimando el coste y el nodo objetivo, es decir, si la función heurística es una cota inferior del coste real que supondría partir desde el nodo actual al nodo objetivo.

Nos interesa, por tanto, que el valor que nos devuelva la función heurística sea lo más próximo al valor real del coste entre ambos nodos. Si esto es así el número de nodos que se podan será muy alto y la búsqueda del camino será prácticamente directa, es decir, se visitarán pocos nodos fuera del camino óptimo antes de calcularlo.

Aplicación de A*

La aplicación del algoritmo A* en nuestro proyecto es la de proporcionar un camino óptimo entre dos puntos sabiendo que entre ellos puede haber obstáculos que impidan llegar de origen a objetivo en línea recta. A parte de las dificultades del terreno (no se puede pasar por un punto si allí hay una montaña) se presenta otro tipo de obstáculos: los radares, que intentan descubrir el avión. Por todo ello necesitamos buscar una ruta óptima.

Como función heurística se ha considerado la distancia que hay entre el nodo actual y el nodo destino considerando que se puede llegar en línea recta. Se puede comprobar que esta heurística es admisible, ya que en ningún momento se podrá encontrar una ruta con un coste menor que el estimado por la heurística (el menor camino entre dos puntos siempre es una recta).

Desarrollo

Fase 1: 2D

Para esta primera fase el desarrollo del proyecto fue conjunto, esta característica era necesaria para conseguir una buena implementación y para que todos los componentes pudiéramos tener una visión global del sistema, imprescindible a nuestro modo de ver para un correcto desarrollo del proyecto de fin de carrera, y así en la división que, como comentaremos más tarde, fue necesaria en una segunda fase, ambos grupos tendríamos pleno conocimiento de los detalles principales del proyecto aunque profundizáramos más en aspectos más concretos de implementación.

La primera fase se centra principalmente en el análisis en dos dimensiones, en todo momento del desarrollo del proyecto vamos a tratar dos partes muy diferenciadas el algoritmo y la interfaz, aunque esta diferencia se aprecia más en la segunda fase, ya que en esta la interfaz es únicamente un medio sencillo y a bajo nivel para poder probar la validez del algoritmo implementado.

Los requisitos conseguidos se dividen en dos grandes clases:

Requisitos de la Inteligencia Artificial

Inicialmente pensamos en un sistema sencillo:

Objetos del sistema:

- Origen: punto(x,y) del mapa del que parte el avión.
- Destino: punto(x,y) del mapa al que queremos llegar, al ser un avión tiene una misión que cumplir, en este caso es llegar a su destino por el camino mínimo.
- Obstáculos infranqueables: puntos(x,y) del mapa por los que el avión no puede pasar, representan los obstáculos reales que aparecerían en el vuelo de un avión, como montañas o edificios.

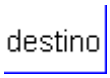
Algoritmo, dado un origen, un destino y el mapa en dos dimensiones (en cada punto(x,y) tenemos la información que nos dice si hay un obstáculo, un origen un destino o si no hay nada y se puede pasar), calcula el camino mínimo que debe seguir el avión. El algoritmo elegido es el A*, como detallamos anteriormente.

Requisitos para la interfaz gráfica de usuario

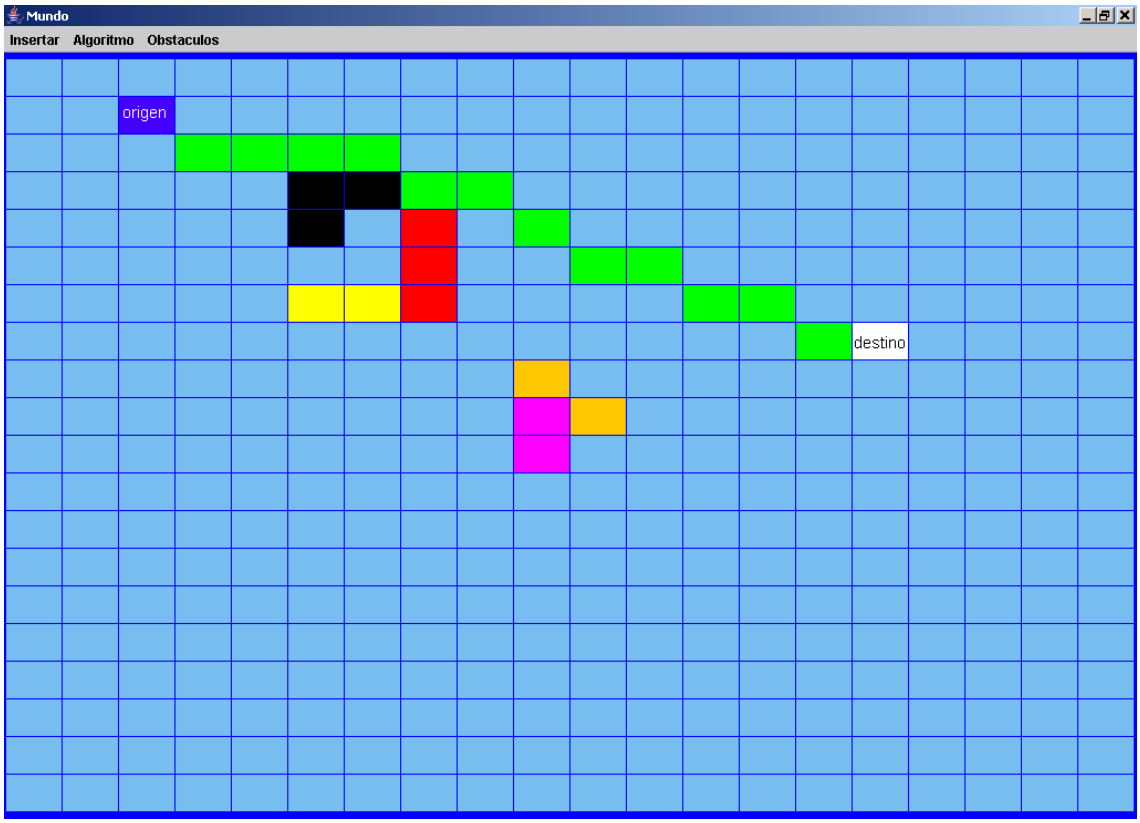
Para la primera versión de la interfaz gráfica en dos dimensiones, pensamos en una interfaz fácil de implementar e intuitiva, en la que origen, destino y obstáculos se seleccionaban interactuando con el ratón y la trayectoria simulada del avión se representaba coloreando las celdas que pertenecían a la trayectoria del algoritmo.

Imagen de la interfaz de entrenamiento de la representación en 2D:

Las casillas verdes representan el camino elegido desde el origen 

al destino 

Las casillas coloreadas con los restantes colores son los distintos tipos de obstáculos.



Esta versión preliminar constaba de las siguientes funcionalidades:

- Seleccionar con el ratón el origen y el destino de la trayectoria
- Seleccionar con el ratón las casillas donde había obstáculo.
- Cambiar el origen y el destino
- Incluir más obstáculos
- Recalcular la trayectoria

Crear una nueva simulación

Restricciones del diseño en la interfaz dos dimensiones:

Inserción de objetos:

- Solo puede haber un origen y un destino, su situación se selecciona con el ratón, y puede cambiarse de la misma forma.
- Los obstáculos son todos infranqueables.
- Una vez insertado un obstáculo no se puede modificar su posición.
- Los obstáculos se tienen que introducir casilla a casilla.

La ejecución:

- No se pueden hacer cambios en el origen y destino, o introducir nuevos obstáculos cuando el camino se está calculando, hay que esperar a que finalice el cálculo así como la representación, y hacer entonces los cambios oportunos y volver a ejecutar el algoritmo.
- No hay posibilidad de hacer modo de ejecución con "wait points", es decir, no es posible introducir una misión con varios destinos.

Fase 2: 3D

Una vez desarrollado un prototipo inicial en el que ya habíamos resuelto el problema básico que queríamos abordar, comenzamos a introducir las mejoras, tanto en la inteligencia como en el diseño de la interfaz.
Para la inteligencia introducimos nuevas características como distintas tipos de obstáculos, decisiones sobre el vuelo según el combustible, poder recalcular dinámicamente una trayectoria o poner puntos de paso obligatorios.

El desarrollo en esta fase se dividió en dos grupos interfaz e inteligencia, en base a una planificación previa en la que se especificaron correctamente todas las partes con las que iban a interactuar uno con el otro, de esta manera cada grupo podía trabajar por separado, respetando el modelo común, para después poder integrarlo y así poder avanzar más rápidamente en cada una de las partes.

Requisitos de la Inteligencia Artificial

Obstáculos. Existen tres tipos de obstáculos:

- *Radares*, son obstáculos cuya área de acción tiene distintos niveles de peligrosidad. El valor que toma la peligrosidad en un punto de acción es indirectamente proporcional a la distancia del mismo al epicentro del radar. Cuando un avión encuentra a un obstáculo de este tipo debe calcular, en función de los distintos parámetros que conoce, como riesgo de atravesar el radar en ese punto, combustible restante, distancia al destino, distancia al siguiente waitpoints…, si es mejor atravesarlo o buscar una trayectoria alternativa.

- *Obstáculos* infranqueables, tales como montañas o edificios. En estos obstáculos el nivel de peligrosidad es el máximo por lo que la decisión siempre va a ser no atravesarlo y buscar una trayectoria alternativa.

Este nuevo algoritmo, con nuevas opciones y características sigue desarrollándose en Eclipse.

Requisitos para la interfaz gráfica de usuario

Para poder implementar las necesidades de la nueva definición de la inteligencia artificial, la mayor complicación surge al desarrollar el programa en tres dimensiones, necesitábamos hacer una simulación suficiente para ver el resultado del algoritmo, no se trataba de hacer una simulación de vuelo, sino una interfaz gráfica en tres dimensiones en la que pudiéramos representar los elementos del terreno, el origen, destino, y el

camino final calculado. Además de otros requisitos para una interacción con el usuario lo más intuitiva posible.

Requisitos funcionales

Recalcular camino:

Es un opción que ofrece esta nueva implementación, en todo momento de la ejecución es posible colocar un nuevo obstáculo, el programa de la interfaz debe conservar la trayectoria seguida hasta el momento de la interrupción y a partir volver a llamar al algoritmo, para recalcular el camino mínimo con las variaciones que se han introducido sobre el mapa del terreno.

Esta opción es útil porque hay que tener en cuenta que puede haber objetos que aparezcan dinámicamente. Puede ocurrir que un radar calcule una trayectoria y durante el tiempo de vuelo el radar que estaba inactivo se conecta, es este momento hay que capturar la interrupción y recalcular la trayectoria teniendo en cuenta los nuevos elementos.

Waitpoints:

Una nueva funcionalidad añadida es la posibilidad de crear misiones con varios destinos, es decir, puntos intermedios obligatorios para la trayectoria del avión.

Requisitos gráficos

Las partes en la que se divide la interfaz gráfica son:

- Barra de menú
- Visualización del terreno
- Visualización de variables

Barra de menú

Es el elemento que contiene las acciones que se pueden realizar sobre los mapas del terreno, las opciones de que existen son:

- *Terrain*, es un menú desplegable que contiene :
  - *Terrain→Select terrain*, abre un navegador en el que podemos elegir entre las distintas imágenes tipo que tiene el programa por defecto. Estas imágenes están clasificadas según el tipo de terreno que contienen:
    - *Muy bajo,* es una imagen cuya matriz tiene valores que están contenidos en el intervalo [200,255], la varianza de la distribución es pequeña por eso, gráficamente, se traduce en una imagen prácticamente plana, aunque elevada sobre el nivel del mar.
    - *Bajo,* las celdas de la matriz que representa a esta imagen toman valores dentro de intervalo [203,249], que gráficamente se

traduce a un terreno casi plano, aunque elevado sobre el nivel del mar, con algún monte de poca altura.

- *Medio*, es una imagen con valores pertenecientes al intervalo [212,249], que gráficamente se traduce en un relieve con curvas suaves y de alturas bajas.

- *Cordillera*, es una imagen cuya matriz se caracteriza porque existe una región con valores altos, rodeados por un borde imaginario de celdas con valores significativamente más pequeños, es decir, la varianza de la distribución de los valores de las celdas es alta. El intervalo de valores es [29, 255]. Gráficamente esta imagen simula una cordillera.

- *Montañoso*, la matriz que representa a esta imagen contiene valores que pertenecen al intervalo [86,255], gráficamente es una imagen con montañas de altura media repartidas uniformemente por todo el mapa.

- *Muy montañoso*, es una matriz con valores pertenecientes al intervalo [5, 255], gráficamente se traduce a una imagen muy montañosa con curvas escarpadas.

Estas son imágenes de prueba para intentar clasificar algunos de los tipos más comunes de terreno existentes, pero evidentemente, el algoritmo funciona con cualquier tipo de mapa.

La información que se obtiene al crear un terreno es:

→ Una matriz que representa a la nueva imagen que el usuario quiere establecer como mapa de terreno.
→ Una estructura terreno con los valores de sus campos inicializados.
→ Una nueva ventana en la que están contenidos todos los objetos.

- *Map*, en el mapa existen diferentes tipos de objetos tales como:

  - *Map→Source*, es el punto de partida de la trayectoria. Cuando el usuario pulsa el botón "Source" del menú "Map", se produce un evento que hace una llamada a una función que permite al usuario introducir, utilizando el ratón sobre el mapa en 2D, la posición deseada para el origen de la trayectoria. La aplicación almacena las coordenadas (X,Y,Z) (como ya hemos indicado, este mapa es realmente un mapa en 3D mostrado desde arriba, por eso es posible obtener las tres coordenadas) y muestra un mensaje emergente para consultar el valor deseado por el usuario para la altura.
    Señalamos que en esta ventana aparece un valor por defecto para la altura que se corresponde con la altura del mapa del terreno en el punto del

plano indicado por el usuario. Es recomendable que si se modifica este valor, se cambie por uno mayor que el mostrado por la aplicación.
Cuando el programa posee todos los datos necesarios, dibuja un origen sobre el mapa de 2D y en el 3D.

La información que se obtiene sobre el origen es:

→ Posición del epicentro de la elipse (X,Y,Z) sobre el terreno de la imagen.

→ Una matriz que representa los puntos de la superficie de la elipse

- *Map→Target*, es el punto de llegada de la trayectoria. Cuando el usuario pulsa el botón "Target" del menú "Map", se produce una llamada a una función que obtiene las coordenadas (X,Y,Z) de la posición indicada por el usuario mediante un evento de ratón sobre la imagen en 2D y muestra un mensaje emergente para consultar la altura deseada para el destino.
  Cabe destacar que en esta ventana se muestra por defecto un valor para la altura que se corresponde con la coordenada Z obtenida por la aplicación. Es recomendable que la altura modificada sea mayor que el valor indicado en el mensaje.
  Una vez que la aplicación posee todos los parámetros necesarios, dibuja un objeto destino con las características indicadas sobre el mapa de terreno de 2D y 3D.

  La información que sobre los objetos destino es:

  → Posición del epicentro de la elipse (X,Y,Z) sobre el terreno de la imagen.

  → Una matriz que representa los puntos de la superficie de la elipse

  Una variable NumDestinos que indica el número de destinos que hemos colocado sobre el mapa. Esta variable es necesaria para recomponer el camino cuando el usuario elige el modo de ejecución con waitpoints.

- *Map→Obstacle*, son objetos que el avión puede encontrar en su camino. Dependiendo de su naturaleza los obstáculos pueden ser:

  - *Map→Obstacle→Radar*, son obstáculos no infranqueables. Cuando se pulsa el botón surge un mensaje emergente donde el usuario debe introducir el número de radares que quiere colocar, esto es necesario para no tener que pulsar repetidas veces sobre el botón.
    Una vez introducido el número de radares deseados, se invoca a la función que los dibuja, el usuario elige interactivamente, pulsando sobre el mapa del terreno en 2D, los epicentros para cada uno de los radares.
    Gráficamente son elipses con distinto nivel de peligrosidad en cada punto de su volumen. El nivel de riesgo en un punto es indirectamente proporcional al valor del radio en el mismo.
    Estos niveles se aprecian por un coloreado más intenso a medida que se acercan al epicentro.

Como observación, señalamos que cuando el usuario indica, interactivamente, un punto del terreno donde desea situar el epicentro del obstáculo, la aplicación calcula dinámicamente el valor máximo entre los valores de los píxeles situados a una distancia menor o igual que R/2 (R=radio).
Esta modificación es útil porque cuando el usuario coloca un obstáculo sólo indica el punto donde desea situar el epicentro del obstáculo, que puede no tener una altura representativa del área de acción del mismo. Si la altura de ese píxel discreto es significativamente más pequeña que la de los píxeles adyacentes provoca que el radar se esconda entre las montañas, disminuyendo realismo a la simulación. Para evitar esta situación, se elige como altura la coordenada Z máxima de los píxeles que pertenecen al conjunto de adyacentes.

La información que se obtiene por cada uno de los radares es:

→ Una matriz que representa los puntos de la superficie de la elipse, se utiliza para dibujar en los mapas, y para rellenar el cubo que le pasamos al algoritmo.

→ Otras dos matrices auxiliares para representar las distintas capas del radar, solo se utilizan para dibujar en los mapas.

- *Map→Obstacle→Impassable*, son obstáculos infranqueables.

- *Parameters*, en este menú están contenidas las variables externas que afectan al comportamiento del sistema. En la implementación actual, la única variable que afecta al sistema es el combustible.
  Esta opción pretende simular de manera más fiel el comportamiento real de un avión de las características deseadas. De esta manera, dado un origen, un destino y la cantidad de combustible disponible, el algoritmo intenta optimizar el combustible que le queda al avión, esto puede implicar arriesgarse a pasar por zonas con más peligrosidad en los radares. Si a pesar de ello no es posible alcanzar el destino deseado con el combustible disponible, muestra un mensaje de atención para indicar que no se puede realizar terminar ese trayecto.
  Cuando el usuario pulsa el botón *Parameters→Fuel* se genera un mensaje emergente donde se introducen los litros de combustible que le quedan, una vez introducidos y aceptado aparece otro mensaje consultando el consumo medio por celda.

  La información sobre los parámetros que se obtiene es:

  → Una variable combustible que indica el combustible disponible, el valor de esta variable actualiza el que tiene el campo combustible de la estructura global Terreno.

  → Una variable consumo que indica el consumo medio del avión por celda. El valor de esta variable es el que toma el campo consumo de la estructura global Terreno.

- *Run*, este menú contiene las dos opciones de ejecución que ofrece el sistema. Estos dos posibles modos son:

    - Ejecución estándar, es la ejecución normal de la aplicación. Se selecciona un origen, un destino y obstáculos y la aplicación devuelve una trayectoria óptima. Cuando se pulsa el botón *Run→Run Standar* se invoca una función que ejecuta el algoritmo con los datos dados por el usuario. Este método obtiene la matriz tridimensional que se extrae del origen, la del obstáculo y el vector de matrices con los radares, realiza una modificación para normalizarlo con las estructuras establecidas para la integración de la inteligencia con el algoritmo, esta matriz transformada en un vector de objetos, que son instancias de la clase EspacioNode, es la entrada del algoritmo implementado en JAVA, que es la inteligencia de la aplicación.
      La inteligencia aplica el algoritmo a estos datos y devuelve un vector de EspacioNode que sólo contiene los nodos que pertenecen al camino óptimo.
      Una vez que la interfaz gráfica posee el camino que es la salida del algoritmo, dibuja sobre el mapa de terreno, tanto en 2D como en 3D, la trayectoria que forman los nodos pertenecientes al camino óptimo.
      De esta manera se simula el comportamiento de un avión en las condiciones planteadas.
      Ejecución con waitpoints, es la ejecución con puntos intermedios obligatorios en la simulación. Cuando se pulsa el botón *Run→Run Waitpoints* se invoca una función que ejecuta el algoritmo con los datos del usuario, teniendo en cuenta que en la trayectoria tienen que estar los puntos especificados.
      Estos puntos se almacenan en una estructura de datos de la forma: $(destino_1, destino_2, ..., destino_n)$ y se ejecuta el algoritmo con la especificación inicial $\{origen = origen\}$ y $\{destino = destino_1\}$, se obtiene una trayectoria óptima que se pinta sobre el mapa del terreno y se vuelve a llamar al algoritmo con $\{origen = destino_1\}$ y $\{destino = destino_2\}$, se vuelve a obtener el camino óptimo para este fragmento de la trayectoria y se pinta en el mapa de terreno. La función repite el mismo proceso hasta que $\{origen = destino_{n-1}\}$ y $\{destino = destino\}$, se pinta el último fragmento del trayecto y así se obtiene la trayectoria óptima simulada de un avión con las condiciones iniciales propuestas y aplicando el algoritmo de la inteligencia.

Para esta parte del desarrollo utilizamos MATLAB, inicialmente intentamos utilizar la herramienta de MappingTool de la aplicación.
Desarrollamos una versión preliminar con ella, en la que dibujábamos los mapas, objetivos y obstáculos mediante la interacción con el usuario, pero como la representación de los mapas es real, es decir, se expresa en términos de altitudes y

latitudes, era muy complicado cumplir los requisitos especificados para poder realizar la integración.
Pensamos que era más importante una integración completa y sencilla que una interfaz gráfica más real pero con más problemas para la integración con la inteligencia artificial, por eso decidimos no utilizar MappingTool e implementarlo con las funciones clásicas que nos ofrece MATLAB para dibujos en 3D.

Restricciones de diseño

Terrain
Las restricciones que hay que tener en cuenta son:

- Restricción en los formatos, los archivos que se seleccionan en el explorador sólo pueden ser de alguno de los formatos que soporta la aplicación. En esta primera versión las imágenes aceptadas por el programa son mapas topográficos en blanco y negro.
  Como consejo para una visualización más clara se recomienda que los mapas no pertenezcan a áreas extensas de un territorio.

Las restricciones se han establecido en base a criterios de optimización del espacio, de eficiencia o mejora en la apariencia de la interfaz.
Existen restricciones para diferentes submenús de la barra principal:

Map→ Source
Las restricciones que hay que tener en cuenta:

- Restricción en el número de orígenes, sólo se permite la elección de un origen por cada ejecución del algoritmo por razones de realismo de la aplicación.

- Restricción de implementación, para introducir un origen en el mapa, es necesario elegir la situación de dicho elemento mediante un evento de ratón sobre el mapa en 2D, situado en la mitad derecha de la visualización del terreno.

- Restricción gráfica, el epicentro de un origen sólo se puede situar en una posición en la que, con el radio establecido por defecto, se pueda asegurar que no sobrepase los límites del mapa del terreno. Además, la altura del origen nunca podrá ser inferior a la altura del terreno en ese punto.

Map→Target
Las restricciones que hay que tener en cuenta dependen del modo de ejecución que haya sido elegido:

- Restricción en la ejecución estándar, se corresponde con el botón *Run→RunStandar*. Si el usuario elige está opción sólo puede existir un objeto Destino por cada ejecución sobre un mapa de terreno.

- Restricción en la ejecución con waitpoints, cuando el usuario elige la opción *Run→RunWaitpoints*, no existe restricción sobre la unicidad del destino, se pueden poner tantos destinos como la naturaleza del mapa del terreno permita.

- Restricción de implementación, para introducir un origen en el mapa, es necesario elegir la situación de dicho elemento mediante un evento de ratón sobre el mapa en 2D, situado en la mitad derecha de la visualización del terreno

- Restricción gráfica, la aplicación sólo posiciona en el mapa del terreno aquellos orígenes cuyos límites, que se calculan utilizando el epicentro, elegido interactivamente por el usuario, y el radio, establecido por defecto, no sobrepasen los límites del mapa del terreno. Tampoco serán posicionados aquellos cuya altura proporcionada por el usuario no supere a la altura del terreno para las coordenadas de anchura y profundidad del epicentro del origen.

Map→Obstacle
Las restricciones a tener en cuenta sobre los obstáculos son las siguientes:
- Restricción en el número de obstáculos, no existen restricciones sobre el número de obstáculos en un mapa del terreno, pero cabe señalar que sólo se podrán situar tantos obstáculos como se haya indicado previamente en el dialogo emergente. Si el usuario necesita incluir más obstáculos debe volver a pulsar el botón e indicar en la pantalla emergente los obstáculos que quiere insertar.
- Restricción gráfica, no se pueden colocar obstáculos cuyo volumen haga que la figura sobrepase los límites del mapa del terreno.

Parameters→Fuel
Las restricciones que hay que tener en cuenta son:

- Restricción en el dominio del combustible, por razones evidentes sólo se permite cantidades positivas de combustible. Los errores son recuperables, por eso si se introduce un número negativo el sistema devuelve un mensaje de error y permite volver a introducir cantidades hasta que se introduzca una correcta.

- Restricción en el dominio del consumo, igualmente sólo se permite que el consumo sea positivo. De la misma manera que el combustible, los errores son recuperables y se pueden introducir cantidades hasta que consiga una que sea correcta.

Run
Las restricciones implícitas del botón de ejecución son:

- Restricción sobre el origen, la aplicación necesita conocer el origen de la trayectoria para poder calcular el camino mínimo, por ello para poder ejecutar el programa se necesita que el usuario haya indicado, mediante un evento de ratón, la posición deseada para el punto de origen. No se puede dar la situación de que existan más de un origen sobre el mapa del terreno, porque está restricción ya ha sido contemplada en el diseño de los objetos origen

- Restricción sobre el destino, de la misma manera que para el origen, es necesario que el usuario haya situado, interactivamente, un destino dentro del mapa del terreno.

Visualización del terreno

Dentro de la interfaz gráfica, la parte más importante es la zona de visualización del terreno. Si sitúa en la mitad superior de la pantalla y muestra los mapas de terreno sobre los que se va a realizar la exploración. Existen dos partes diferenciadas:

- Visualización en 2D, muestra el terreno en 2D. Esta visualización es necesaria para conseguir una vista aérea de la trayectoria del avión. En este mapa es donde el usuario especifica, interactivamente, la posición del origen, destino y obstáculos deseados. Cuando el usuario pulsa el botón *Run*, en este mapa también se dibuja la trayectoria del camino óptimo seguido por el avión.
  Realmente la imagen en 2D es una vista superior de un mapa en 3D, por lo tanto todos los puntos en este mapa están definidos mediante sus coordenadas (X,Y,Z)

- Visualización en 3D, se puede visualizar el terreno en 3D en dos ventana distintas:

  - Visualización en 3D en la ventana principal
  El mapa del terreno se muestra en la mitad izquierda de la zona de visualización del terreno de la ventana principal.
  El usuario no puede hacer ninguna operación, mediante eventos de ratón, sobre esta imagen.
  Esta visualización es necesaria para obtener una simulación más real de la trayectoria. Cuando un usuario define un origen, destino y obstáculos sobre la imagen en 2D, estos también aparecen en la imagen en 3D, de la misma manera que se dibuja la trayectoria del camino óptimo que devuelve a la interfaz el algoritmo de inteligencia artificial.

  - Visualización en 3D en una ventana secundaria
  Como funcionalidad añadida, se ha incorporado un botón *Full screen* dentro de la zona de visualización del terreno que permite mostrar al usuario la imagen en 3D en pantalla completa.
  Cuando se incluye un elemento en el mapa del terreno en la visualización de la pantalla principal, los cambios se producen dinámicamente en la imagen mostrada en la pantalla completa.
  Esta opción es necesaria para obtener una vista de la simulación más clara del mapa en 3D que es demasiado pequeño en la visualización en la ventana principal

  Además el usuario puede realizar operaciones sobre esta imagen como:
    - Salvar simulación, la aplicación proporciona la opción de salvar la simulación con origen, destino, obstáculos y la trayectoria del camino óptimo calculado por la inteligencia artificial.
    - Imprimir simulación

- Seleccionar imágenes, cada uno de los objetos que componen la simulación puede ser seleccionado y mediante el botón derecho del ratón, realizar operaciones de modificación y consulta de sus propiedades.
- Zoom in
- Zoom out
- Desplazar la imagen, la imagen completa de la simulación puede ser trasladada en la ventana de pantalla completa.
- Rotar la imagen, la imagen en 3D puede ser rotada, esto es útil para observar la trayectoria desde todos los puntos.
- Conseguir información de los objetos, de está manera se puede obtener gráficamente los datos sobre las coordenadas del origen, destino y obstáculos.
- Insertar una barra de color, es una figura intuitiva que muestra la leyenda de los colores utilizados en el mapa del terreno para simular los distintos niveles de altura.
- Mostrar/esconder las variables de la aplicación

Visualización de las variables

La tercera componente en la interfaz gráfica es la visualización de las variables. En la parte inferior izquierda se visualizan las características del terreno actual y los parámetros que actúan sobre la trayectoria.

Las variables que pueden ser interesantes para visualizar son:

- Punto de origen, en este campo se observa la posición del epicentro de la elipse que el usuario ha situado como origen de la trayectoria. Cuando se cambia de terreno o se comienza una nueva simulación, estos valores se actualizan porque no es necesario almacenar la historia de los sucesos anteriores.

- Punto de destino, en este campo se muestra el valor del epicentro de la elipse que el usuario ha elegido como destino del viaje, o los valores de los epicentros de las elipses que el usuario ha elegido como paradas obligatorias en el transcurso del viaje.

- Combustible, inicialmente en este campo se muestra el valor del combustible que el usuario ha introducido en los mensajes emergentes al pulsar el botón *Parameters→Fuel* o en su defecto el que ha sido declarado en la inicialización de la estructura global Terreno, por lo que este valor se modifica dinámicamente a lo largo de la trayectoria, según la relación $\frac{Combustible_{disp}}{Consumo_{celda}}$ introducida por el usuario al pulsar el botón *Parameters→Fuel* o en su defecto la definida en la inicialización de la estructura global Terreno.

Fase 3 Integración

En esta fase ambos grupos volvimos a reunirnos para conectar ambos programas, aunque la puesta a punto final se hizo principalmente en la interfaz, ya que es la encargada de gestionar toda la ejecución. Surgieron problemas de eficiencia, ya que la transformación del cubo que genera Matlab a un objeto java suponía un incremento de tiempo considerable, por lo que tuvimos que intentar disminuirlo con diversas técnicas. Es este el mayor inconveniente de la utilización de Matlab para la interfaz, frente a las ventajas que nos proporciona.

Para una correcta integración fue necesario hacer una buena planificación al comienzo de la fase 2, los requisitos de ambas partes (algoritmo e interfaz) necesarios son:

Inteligencia:

- Recibe:
    1. El cubo que representa el mapa del terreno, pero transformado ya en un objeto java, esta transformación se hará desde Matlab.

    2. El combustible del que dispone el avión introducido por el usuario y la proporción que consume por espacio recorrido.

- Ejecuta para la situación del terreno recibida el algoritmo A*, teniendo en cuenta las restricciones de combustible, y calcula el camino mínimo.

- Devuelve el camino mínimo como un objeto java, será necesario transformarlo como casillas de la interfaz, tarea de la que se encarga la interfaz.

Interfaz:

- La interfaz necesita atributos como son el mapa, origen, destino, radares y combustible, que facilita el usuario.

- Ejecución:
    1. Primera labor, convertir la información introducida por el usuario en un cubo para representar el mapa del terreno.

    2. Traducir el cubo generado a un objeto java que entienda el algoritmo.

    3. Llamar al programa java, ejecución del algoritmo con el objeto anterior.

    4. Convertir el camino mínimo calculado por el algoritmo, que es un objeto java, en casillas del cubo de la interfaz.

    5. Visualizar dicho camino.

- Como requisitos añadidos a la interfaz ya comentados anteriormente, tenemos:

    1. Posibilidad de introducir nuevos obstáculos y recalcular el camino, para lo cual será necesario repetir los pasos detallados en el punto de ejecución.

    2. Ejecutar el programa con otro modo de ejecución "wait points", para lo cual tendremos que hacer sucesivas llamadas al algoritmo, pero esta vez solo será necesario actualizar el valor del origen y el destino.

Para manejar los elementos mencionados implementamos funciones que se detallarán el la explicación de la implementación.

Implementación

En este apartado vamos a describir la implementación de las distintas partes en las que se divide el proyecto, como ya hemos comentado anteriormente, hay dos grandes partes Inteligencia e Interfaz, para cada una de ellas ilustraremos las principales características de la implementación.

Inteligencia

Es el programa desarrollado en Java, encargado de la ejecución del algoritmo A*. Hacemos un estudio más detallado de las características principales, y completamos dicha información con sus correspondientes diagramas de dependencia (UML), así como la descripción de sus clases estructurada en paquetes y las librerías de java utilizadas:

Descripción de los principales atributos de la inteligencia:

- Espacio:
  - Contiene la información de las casillas del cubo, se organiza en forma de lista, para lo cual se crea un objeto de la clase ArrayList de java, cada uno de sus elementos es un EspacioNode, es en las casillas por tanto donde diferenciamos la posición, es decir donde representamos las tres coordenadas de un punto del mapa.
  - El espacio contiene además del cubo una serie de atributos, merece la pena mencionar entre ellos el origen y el destino. Son enteros que guardan en que posición de la lista están dichos puntos. Facilita la implementación del algoritmo.

- EspacioNode:
  - Representa la información para una casilla del mapa, como hemos visto es necesario que este objeto contenga las coordenadas de ese punto. A continuación veremos los principales atributos:

    - *Fila, columna y altura*: representan las coordenadas en el mapa en tres dimensiones de esa casilla.

    - *Peso*: cada uno de los posibles tipos de casilla va a tener un peso distinto. Hay cinco tipos de casilla, los listamos y hacemos mención al valor del peso correspondiente:
      - Origen: le etiquetamos con *peso = -1*.
      - Destino: le etiquetamos con *peso = -2*.
      - Nodos del camino: les etiquetamos con *peso = -3*.
      - Obstáculos: Para los obstáculos el peso mide el nivel de peligrosidad que representan, una montaña por ejemplo, al ser un obstáculo infranqueable tendrá el nivel de

peligrosidad máximo. Para obstáculos como los radares dependiendo de lo lejos que estemos del núcleo central será varía el nivel de peligrosidad. Este nivel es un número comprendido entre [0,1] donde el 1 representa la máxima peligrosidad. Así mismo es necesario clasificar dichos niveles de la siguiente forma:

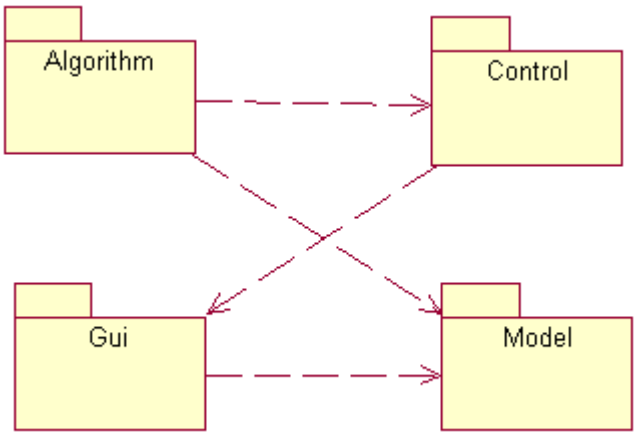
Valor del Peso	Peligrosidad
[0 , 0.25)	Bajo
[0.25 , 0.5)	Medio
[0.5 , 0.75)	Alto
[0.75 , 1]	Infranqueable

Diagramas UML:

1. Relaciones entre paquetes:

Vamos a ver las relaciones presentes entre las clases que forman parte de la aplicación. Primero observemos la separación por paquetes y posteriormente en el interior de cada uno de estos.

Esta es la relación encontrada entre los distintos paquetes de la aplicación:



Podemos ver como el paquete Algorithm necesita, por un lado el contenido del paquete control y por otro lado el del paquete Model. Podemos observar esta relación de forma simple, ya que en el paquete Control se produce la gestión de errores y en el paquete Model, está el modelo del diseño.

Por su parte el paquete Control necesita del paquete Gui porque al contener la clase Main en este paquete, necesita la referencia de la interfaz gráfica que es donde se producen los cambios.

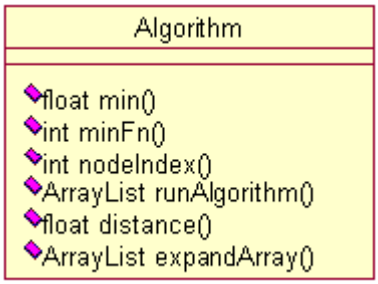
Finalmente comprobamos como el paquete Gui depende del paquete Model. El motivo es que el modelo que se utiliza es necesario que se conozca para su representación gráfica.

2. Relaciones entre clases

a. Paquete Algorithm

En este paquete hay una única clase, la clase Algorithm que lógicamente no tiene relaciones de dependencia con otras clases de su mismo paquete, pero sin con clases de Control y Model.

Veamos la representación de este paquete:

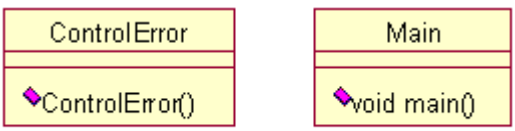


Vemos que entre sus métodos está runAlgorithm que es el que calcula el algoritmo de búsqueda A* utilizado para la búsqueda del camino.

b. Paquete Control

Es junto con el paquete Algorithm, el más sencillo de los que se encuentran en la aplicación. En este caso sólo aparecen dos clases que tienen relaciones de dependencia con clases de otros paquetes pero no entre ellas.

Podemos ver la representación de las clases de este paquete:

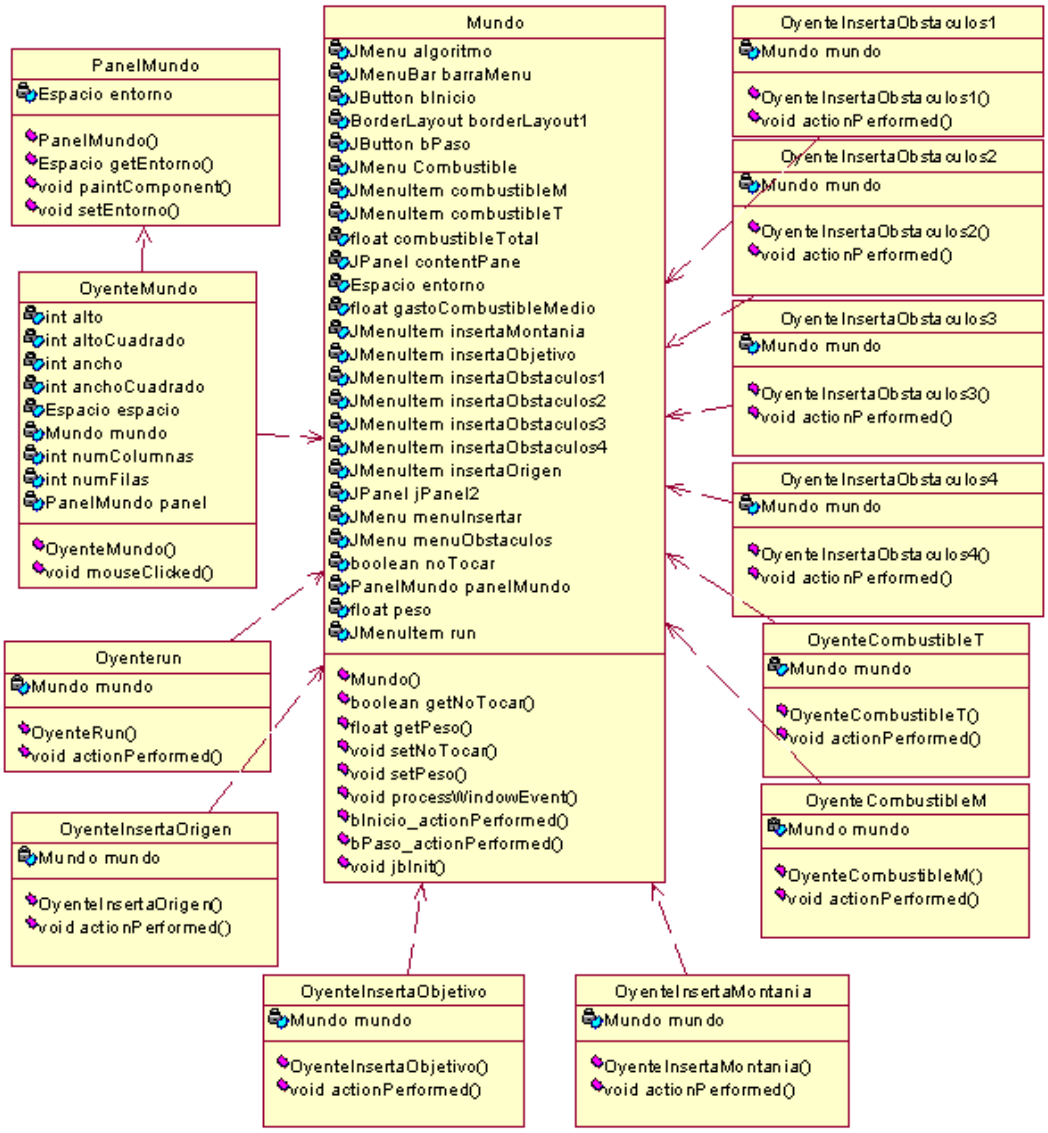


Como se puede comprobar son clases sencillas con un solo método cada una de ellas. ControlError, que está relacionada con la gestión de excepciones y Main que es la clase Main de la aplicación.

c. Paquete Gui

Este paquete contiene las clases necesarias para la interfaz en 2D que nos sirve para hacer pruebas a bajo nivel del algoritmo, obviando la parte gráfica en 3D.
Este paquete presenta una mayor complejidad que los dos anteriores. En él hay trece clases por lo que la complejidad es mucho mayor.

Su diagrama de relaciones de dependencias sería como sigue:

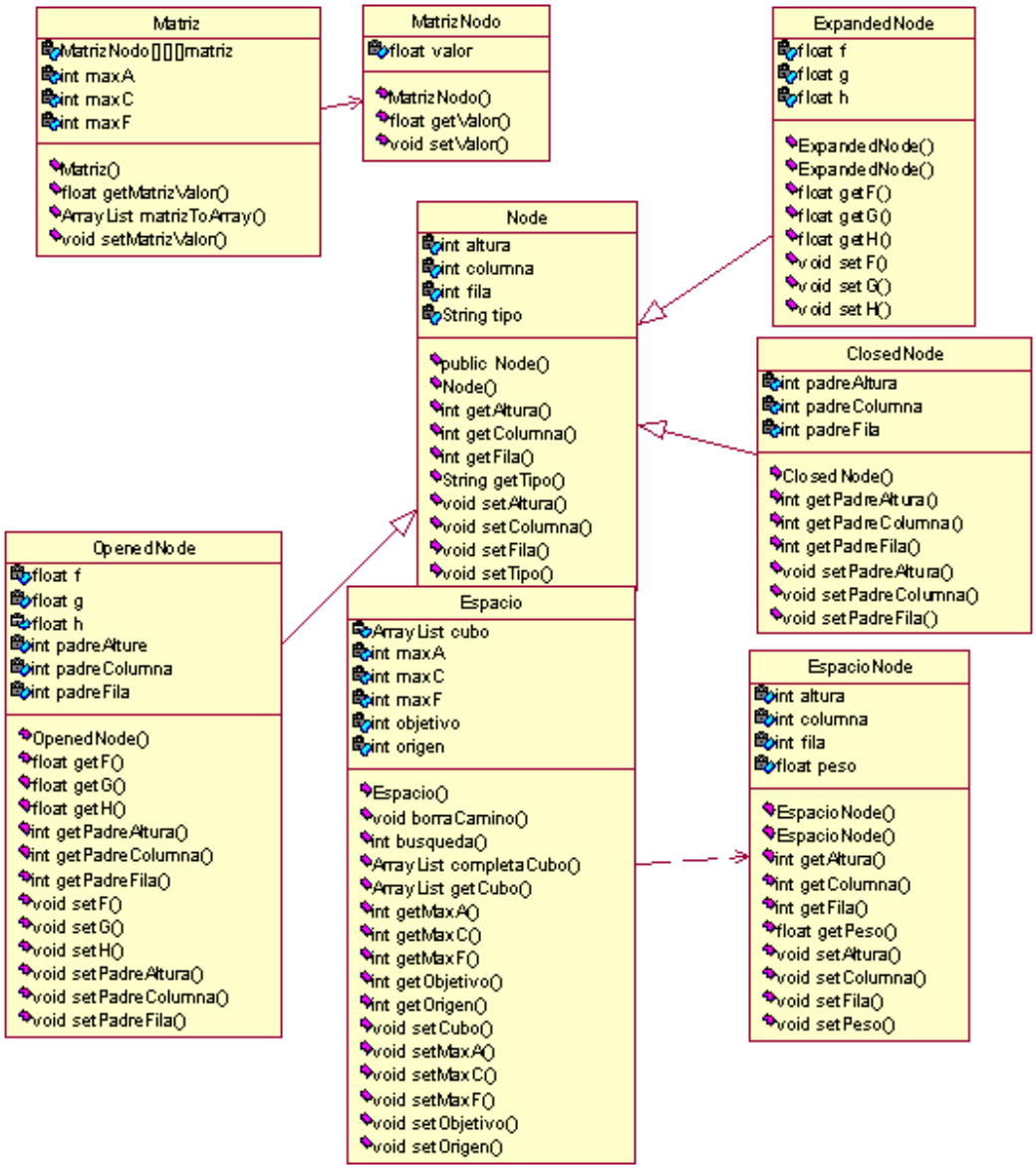


Como se puede observar, aquí si aparecen relaciones de dependencia entre las clases de un mismo paquete. Concretamente, se puede comprobar como todas las clases de tipo oyente dependen de la clase Mundo. Además, la clase OyenteMundo depende de PanelMundo.

d. Paquete Model

El paquete Model es el que contiene las clases relacionadas con la estructura de los elementos de la aplicación. Podemos ver como se relacionan en grupos de clases.

Lo vemos de forma gráfica:



Vemos que los distintos tipos de nodo dependen de Node, mientras que Matriz depende de MatrizNode y Espacio de EspacioNode.

Descripción de sus clases estructurada en paquetes:

1. Paquete Algorithm

Paquete con una única clase aunque muy importante pues contiene el método que resuelve el algoritmo que se aplica para la búsqueda de la ruta óptima entre dos puntos.

Clase Algorithm

Esta es la clase más importante de toda la aplicación, ya que se encarga de resolver el algoritmo de búsqueda.

- Método "distance" calcula la distancia entre dos nodos (nodo actual y nodo destino). Es el cálculo de la heurística, esta distancia se mide en línea recta, es decir, suponiendo que no hay ningún obstáculo. Este método calcularía el h(n) en el algoritmo de A*.
- Método "expandArray" que genera una lista con los sucesores de un nodo.
- Método "min" para calcular el mínimo entre dos números (para ver cual de los dos tiene un menor coste heurístico).
- Método "minFn" que hace lo mismo que el método anterior pero no entre dos valores, sino entre los elementos de una lista (lista de nodos abiertos).
- Método "nodeIndex", para calcular el índice de un nodo en la lista de nodos cerrados.
- Y finalmente, "runAlgorithm" que es el algoritmo propiamente dicho

2. Paquete Control

En este paquete se encuentran las clases referentes al control de la aplicación. Sólo aparecen dos: una de control de errores y la clase Main.

Clase ControlError

- Clase que muestra un mensaje de error (muestra un JOptionPane) en caso de producirse una excepción.
- Hereda de la clase Throwable por lo que se permite lanzar excepciones.

Clase Main

Clase principal de la aplicación que básicamente realiza lo siguiente:

- Crea una matriz nueva del tamaño que se le especifique.
- Establece un combustible total y un gasto de combustible promedio.
- Crea un mundo con los datos especificados en los dos puntos anteriores.

3. Paquete Gui

En este paquete están contenidas todas las clases referentes a la parte gráfica. La representación gráfica en Java es únicamente en dos dimensiones. El algoritmo se puede

llamar de forma genérica con un mayor número de dimensiones, pero no se representan más que dos.

Clase OyenteCombustibleM

Clase que actúa de oyente sobre el combustible que se gasta en promedio durante el trayecto.

Clase OyenteCombustibleT

Clase semejante a la anterior pero que hace referencia al combustible total del avión.

Clase OyenteInsertaMontaña

Implementa la interfaz ActionListener actúa de oyente. Cuando el usuario selecciona en la representación gráfica insertar una montaña, se crea un objeto de esta clase.

Clase OyenteInsertaObstaculo1, 2, 3 y 4

La función de esta clase es similar a la de la montaña, simplemente se diferencia en que en lugar de seleccionar una montaña, se selecciona un obstáculo denominado de tipo 1.

Clase OyenteInsertaOrigen

Similar a las clases anteriores con la diferencia de que en este caso actúa cuando se quiere insertar un nuevo origen en el mapa.

Clase OyenteInsertaObjetivo

Equivalente al caso anterior, pero en esta ocasión consideramos el objetivo.

Clase OyenteMundo

Esta clase es utilizada cuando sucede un evento de ratón. Es decir, cuando a través del ratón se produce un cambio en el mundo.

Clase OyenteRun

Se busca crear un objeto de esta clase cuando se selecciona "run" en el menú. Será el encargado de mandar ejecutar el algoritmo con las condiciones actuales.

Clase PanelMundo

Crea un panel de nuestro mapa. Esta clase extiende a JPanel y establece un entorno (que es un espacio como el definido en el paquete Model).
En el método paintComponent se establecen las dimensiones del panel y las coordenadas de origen del nodo a pintar. Se pinta todos los nodos que aparezcan en la lista del espacio.

Clase Mundo

Clase que extiende a JFrame y por lo tanto hace de marco de nuestra interfaz gráfica. En ella se encontrará un objeto de la clase PanelMundo, los botones de la interfaz y los menús de opciones.

4. Paquete Model

Este paquete contiene las clases referentes al modelo del sistema, es decir, a las estructuras que se han utilizado para la desarrollar la parte central de la aplicación que es el algoritmo propiamente dicho.

Clase Node.

- Es un tipo genérico de nodo que contiene la información básica sobre el mismo. Sólo contiene información respecto a su posición (fila, columna y altura) y al tipo de nodo que es.
- Será una clase de la que extiendan la ClosedNode y OpenedNode que completarán su información.
- Sólo hay métodos de acceso y modificación de sus atributos.

Clase ClosedNode.

- Clase que extiende de Node. Sus nodos contendrán la información necesaria para un nodo de la lista de nodos cerrados. Por lo tanto completa la información de la clase Node mediante la incorporación de la posición del padre (fila, columna y altura).
- Sus métodos se reducen a métodos de acceso y modificación.

Clase OpenedNode.

- Clase cuyos objetos serán elementos de la lista de nodos abiertos.
- Esta clase extiende a la clase Node y además incorpora la información necesaria sobre la posición de su padre y los valores f, g y h que determinan su valor heurístico.
- De nuevo, sólo encontramos métodos de acceso y modificación.

Clase ExpandedNode.

- La clase ExpandedNode crea objetos correspondientes a los que hay en la lista de nodos expandidos. Esta clase también extenderá de la clase Node, y además incorporará información sobre los valores de f, g y h útiles para conocer su valor heurístico estimado.
- Los únicos métodos pertenecientes a esta clase serán de acceso y modificación sobre sus atributos.

Clase MatrizNodo.

- Los objetos de esta clase son los nodos de los que estará compuesta la matriz que representa el estado del terreno.
- Un objeto de esta clase contiene un valor que se corresponde con lo que cuesta pasar por ahí.
- Los métodos que encontramos en esta clase se reducen a acceso y modificación de este valor.

Clase Matriz

- Es una matriz tridimensional cuyos elementos son objetos de la clase MatrizNodo.
- Tiene como atributos las dimensiones de la matriz (número de filas, columnas y altura).
- Aparte de los métodos de modificación y acceso (para modificar un elemento de la matriz), también existe un método para pasar la matriz a un objeto ArrayList.

Clase EspacioNode

- Objetos definidos por su posición (fila, columna y altura) y su peso (un valor que representa lo que te encuentras en la posición que representa).
- De nuevo encontramos sólo métodos de acceso y modificación.

Clase Espacio

- Es la clase más compleja de este paquete. Sus atributos son un ArrayList que representa una lista con la información del problema (parecido a la matriz) y cuyos elementos son objetos de la clase Espacio, la posición en la que se encuentra el origen y el destino dentro de esta lista, y el tamaño máximo del espacio de búsqueda en el que nos encontramos (tamaño de fila, columna y altura).
- Además de los métodos de acceso y modificación encontramos un método llamado “completacubo” que añade los nodos al cubo correspondientes a las posiciones en las que el peso es 0, es decir, en las que no hay nada.
- Hay otro método llamado “busqueda” que se encarga de encontrar el elemento de la lista que corresponde con unas determinadas coordenadas en el mapa.
- Por último hay un método llamado “borraCamino” que restablece el peso correcto a los elementos de la lista que formaban parte del camino, es decir, se pone a cero.

Librerías JAVA

En el transcurso de este proyecto, se han utilizado varias clases ya predefinidas en las librerías de Java. Vamos a ver una pequeña descripción de cada una de ellas, de este modo a cualquier desarrollador que quisiera utilizar el código, le sería útil esta sección, al igual que su análoga en la parte de Matlab como veremos más adelante.

1. java.util

ArrayList

En esta librería se encuentra la clase ArrayList. La clase ArrayList nos proporciona una lista de objetos (cualquier tipo de objeto puede pertenecer a un ArrayList). Se le puede asignar un tamaño al crear el objeto aunque esto no es necesario, lo que permite mantener lista tan largas como los recursos disponibles nos permitan.
Los métodos que nos proporciona la clase ArrayList más comúnmente utilizados son los siguientes:

- add. Añade un objeto al ArrayList creado. Se tienen diferentes opciones: se puede añadir un elemento en la lista, se puede decir dónde queremos que lo añada (posición) y se puede añadir una colección de objetos.

- clear. Borra todos los elementos de la lista.

- remove. Elimina el elemento que se quiera. Se le pasa el índice en el que se encuentra y lo elimina.

- size. Devuelve el tamaño de la lista

- get. Devuelve el elemento que se quiera. Se le pasa como argumento el índice donde se encuentra dicho elemento.

1. java.lang

Math

Aquí encontramos la clase Math. De ella se utiliza el método sqrt, que calcula la raíz cuadrada de dos números.

Throwable

También encontramos la clase Throwable que es una superclase de los errores y excepciones. Sirve precisamente para lanzar excepciones.

2. javax.swing

JOptionPane

A esta librería pertenece la clase JOptionPane que extiende a JComponent e implementa a Accesible. Sirve para lanzar al usuario una ventana en la que se le pide un valor o para ofrecerle información.
Hay distintos tipos de mensaje (CANCEL_OPTION, CLOSED_OPTION, WARNING_MESSAGE, etc.) para utilizar el más adecuado en cada momento.

3. Java.awt

Event

Encontramos la clase Event. La clase Event es una clase que encapsula eventos de la parte gráfica de un usuario. Esta clase se mantiene sólo por compatibilidad. La información proveída en esta clase sirve para permitir a los programadores pasar de programas en Java 1.0 al nuevo modelo de evento.

Toolkit

La clase Toolkit es una superclase abstracta de todas las implementaciones de Abstract Window Toolkit. Las subclases de Toolkit son usadas para enlazar las diversas componentes a implementaciones nativas de Toolkit particulares.

JPanel

La clase JPanel nos proporciona un panel en el que se añaden componentes de la interfaz.

BorderLayout

La clase BorderLayout nos proporciona un contenedor preparando y redimensionando sus componentes en cinco partes (norte, sur, este, oeste y centro). Cuando se añade un nuevo componente al contenedor, se le dice en que parte se debe añadir.

JButton

La clase JButton crea un botón al que se le puede asociar un nombre, un método, etc.

JMenuBar

La clase JMenuBar (que extiende de JComponent) nos permite crear un menú añadiendo a ella componentes de la clase JMenu.

JMenu

La clase JMenu extiende a JMenuItem. Una implementación de JMenu necesita de objetos de JMenuItem que son mostrados cuando el usuario selecciona un camino en el JMenuBar.

JMenuItem

La clase JMenuItem extiende a AbstractButton y en esencia un objeto de esta clase es un botón emplazado en una lista. Cuando el usuario usa este "botón" se realiza la acción asociada a dicho "botón".

Dimension

La clase Dimension encapsula el ancho y el largo de un elemento en un único objeto.

MouseAdapter

La clase MouseAdapter es una clase abstracta utilizada para recibir eventos del ratón. Los métodos de esta clase están vacíos. La clase existe por conveniencia para crear objetos oyentes.

ActionListener

La interfaz ActionListener extiende a EventListener. Utilizada para recibir eventos de acciones.

ActionEvent

La clase ActionEvent extiende a AWTEvent. Es un evento semántico que informa que una acción definida ha ocurrido.

Graphics

La clase Graphics es la clase básica abstracta para todos los contextos gráficos que permite a una aplicación dibujar sobre componentes que son realizadas por diversos dispositivos, así como visualizar imágenes.
Un objeto de esta clase contiene la información necesaria para realizar las operaciones básicas que Java permite.

Color

La clase Color se utiliza para encapsular los colores en el espacio de colores por defecto RGB o en espacios de colores arbitrarios identificados por un ColorSpace.
Implementa las interfaces Saint y Serializable.

Interfaz

Es el programa desarrollado en Matlab, encargado de la representación gráfica del sistema. Hacemos un estudio más detallado de las características principales, y completamos dicha información con sus correspondientes diagramas de dependencia, para poder ver de una forma más intuitiva la estructura desarrollada, dado que al no tener una estructura de clases como en el caso de Java, no podemos aplicar UML propiamente dicho, pero su representación nos facilita la explicación de las distintas componentes del programa. Proseguiremos con una descripción de sus archivos principales y las funciones predefinidas de Matlab utilizadas:

Descripción de aspectos fundamentales

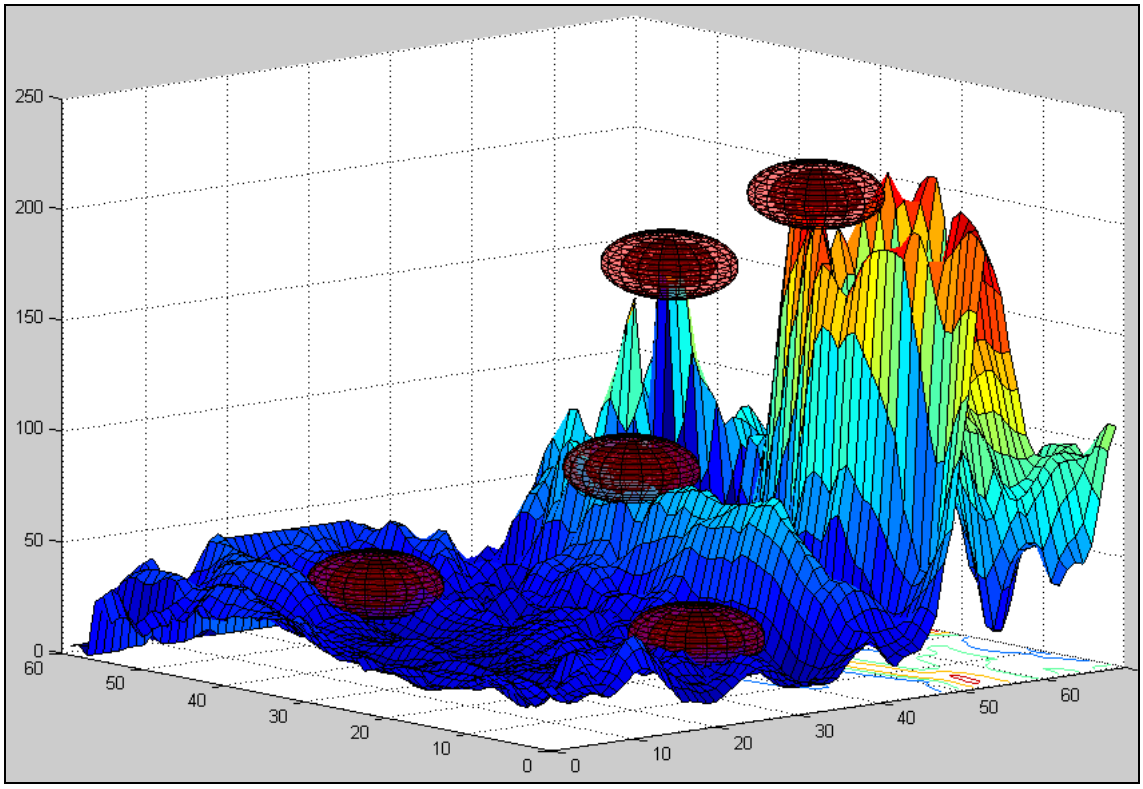
Principales atributos de la interfaz en Matlab:

1. **Imagen:**

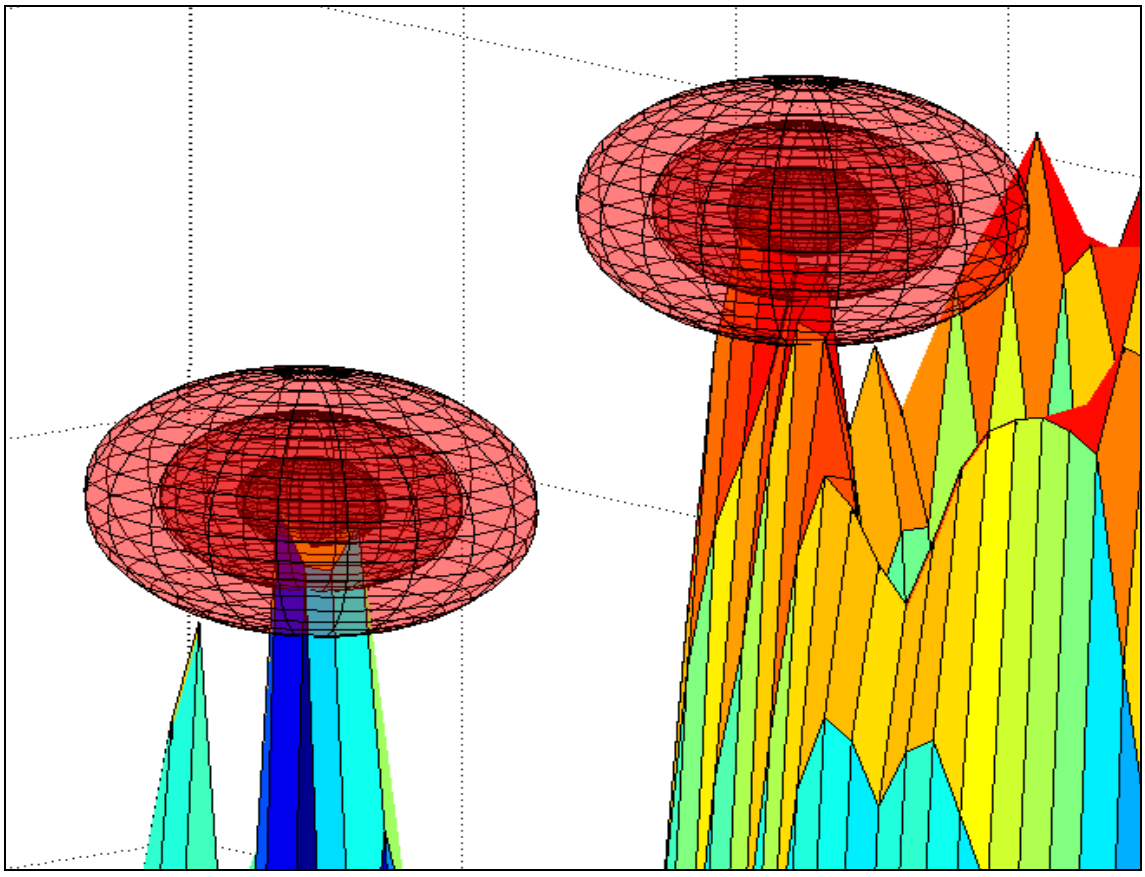
- Escogemos una imagen guardada en un archivo. La imagen que escogemos se representa en dos y tres dimensiones. Hay que tener en cuenta que la imagen en dos dimensiones es una representación como una vista aérea de la imagen que estamos considerando, es decir es una imagen 3D con la cámara situada sobre ella. La representación de las casillas de la imagen es en forma matricial.
- La imagen en ambos casos se visualiza en diferentes colores, variando estos en función de la altura, de tal forma que es totalmente observable la diferencia de altura en el mapa de dos dimensiones. Los colores oscilan entre el azul y el rojo simbolizando el azul la altura más baja y el rojo son las alturas más altas.

2. **Rádares:**

- Los radares representan un tipo de obstáculo en el camino.
- Gráficamente, un radar se visualiza como tres esferas concéntricas sobre el mapa con color de intensidad decreciente, para simular el alcance de un radar, ya que cuanto más nos alejemos de él más posibilidades tenemos de no ser detectados.
- En cuanto a su implementación, la existencia del radar queda almacenada en una estructura que almacena toda la información concerniente al estado del mapa. Los rádares se almacenan como una lista.
- El efecto de un radar no es local a ese punto, como ya hemos comentado, sino que se distribuye en círculos concéntricos en torno a ese punto perdiendo intensidad. Para ello se ha definido una función en la que se divide la zona de influencia del radar en zona de máxima peligrosidad, zona media y zona baja.



Observando más de cerca se pueden ver las elipses concéntricas que representan los niveles de peligrosidad, la intensidad del color es directamente proporcional al nivel de peligrosidad de los mismos:



3. Origen:

- El origen es el punto de inicio a partir del cual se empieza a calcular el camino óptimo hasta un objetivo.

- Debe ser único, ya que no tiene sentido empezar en más de un punto diferente. Visualmente lo vemos como una esfera en el mapa de la imagen.

- A nivel de implementación su información está contenida en una estructura, esta información son sus coordenadas (X,Y,Z) del epicentro, y la matriz que describe el contorno de la esfera para poder dibujarla en los mapas.

- El origen puede cambiar de posición siempre y cuando el algoritmo no esté calculando un camino. En caso contrario, se deberá esperar.

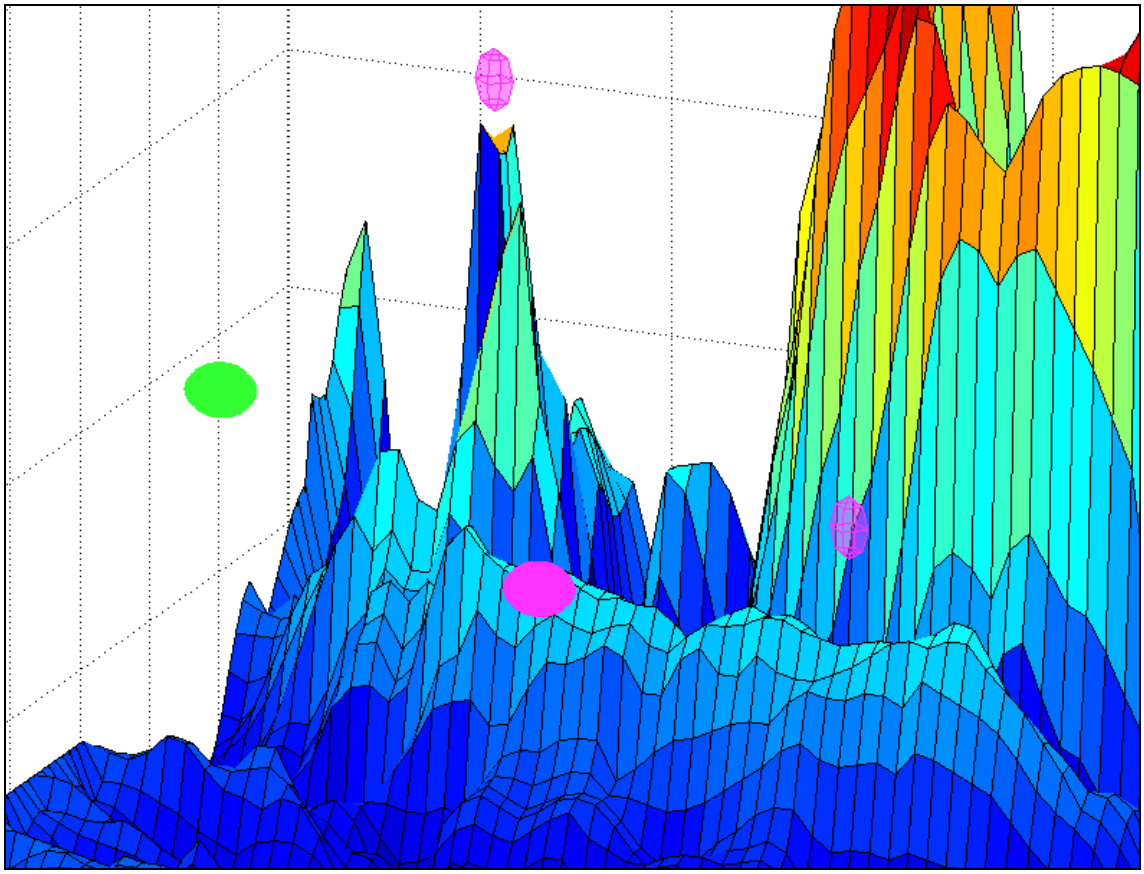
4. Destino:

- El destino es el punto que se quiere alcanzar cuando se calcula un camino.

- A diferencia del origen puede ser múltiple, podemos definir la misión del avión como un conjunto de destinos que debe alcanzar, irá recorriendo desde el origen al primer destino y desde este último a los demás destinos hasta alcanzar el último de los destinos. Podemos ver los destinos no finales como destinos parciales.

- Visualmente se diferencian en su representación. Mientras que el destino final se asemeja a la representación del origen salvo que se realiza en otro color, los destinos parciales se representan como una elipse del mismo color que el destino final pero con otra estructura.

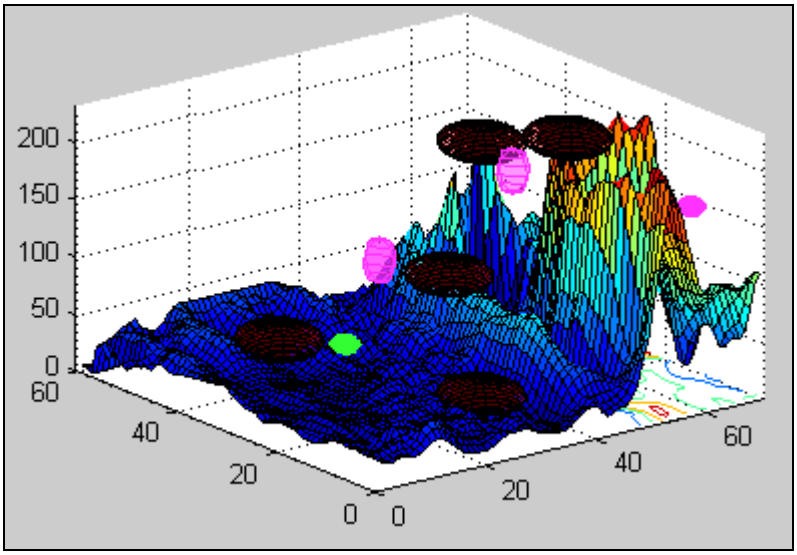
En la siguiente imagen se muestra una simulación con un origen (elipse opaca verde), un destino (elipse opaca magenta) y dos destinos intermedios (elipses translúcidas magenta):



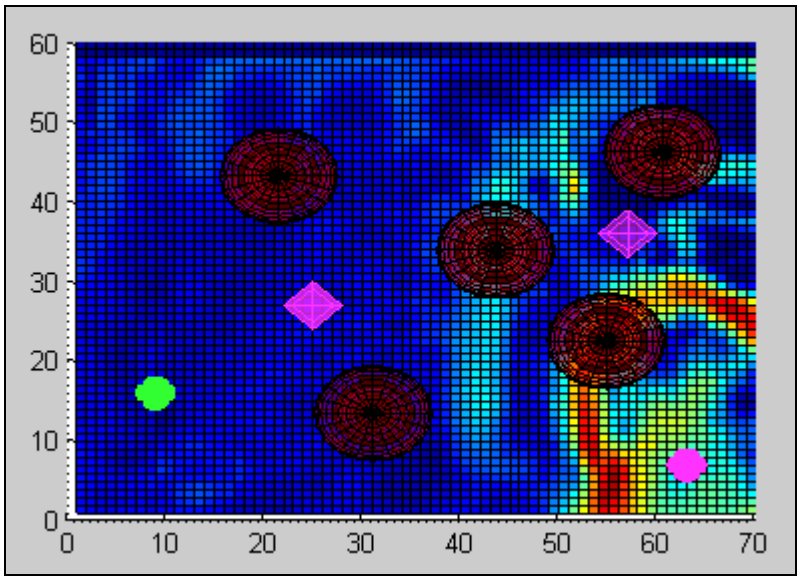
5. Pantallas

- En la interfaz aparecerán primariamente dos imágenes. Una de ellas será en dos dimensiones mientras que la otra se hará en tres dimensiones.
- Opcionalmente, se permite a través de un botón obtener una nueva pantalla en la que se ve la imagen de tres dimensiones en pantalla completa. Para ello, se necesita una función que mantenga actualizada la pantalla completa de tal forma que si se produce alguna novedad, esto es, se introduce un nuevo destino, se cambia en origen, etc. Al abrir de nuevo la pantalla con la imagen en tres dimensiones se mantenga con los últimos datos actualizados.

Visualización del terreno en tres dimensiones:



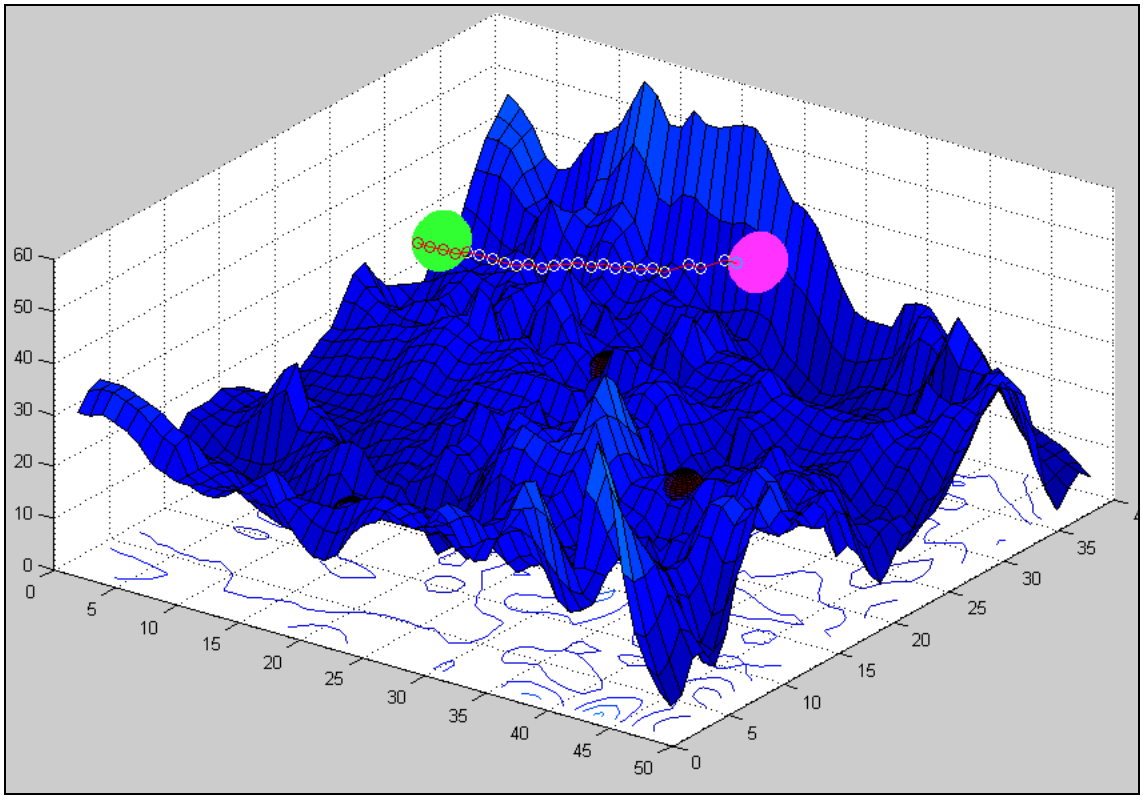
Visualización del terreno en dos dimensiones:



6. Camino

- El camino se representa mediante una animación que simula la trayectoria del avión partiendo del origen indicado, pasando por los destinos intermedios y alcanzando finalmente su destino, evitando los obstáculos según las decisiones que toma la inteligencia.

Visualizamos un ejemplo de trayectoria:



Descripción de los principales eventos del programa:

1. Eventos del Menú

Hay diferentes eventos asociados al menú. Son los siguientes:

- Cuando se selecciona abrir una nueva imagen se debe actualizar la estructura del terreno. Por un lado debe aparecer el menú que lista todas las imágenes disponibles. Por otro lado también es necesario que el terreno se inicialice de tal forma que la información que teníamos correspondiente a la imagen anterior desaparezca.

- Si se selecciona elegir un origen, hay que comprobar que no había ninguno previamente. En caso de existir, se debe eliminar del mapa (tanto visualmente como la información almacenada sobre él). Los valores del nuevo origen sustituirán a los del anterior.

- Por contra, si se selecciona un nuevo destino, éste se debe añadir a la lista de destinos disponibles, pero no eliminarlo. Visualmente, sin embargo, si se debe tener cuidado con el nuevo destino, ya que el aspecto del anterior destino debe cambiar para pasar a ser un destino parcial.

- El caso de los radares es similar al caso del destino, pero visualmente no hay que modificar el radar anterior.

- Cuando se selecciona la variable combustible entonces aparece un mensaje de diálogo en el que se le pide al usuario introducir el combustible disponible y posteriormente aparecerá otra ventana de diálogo en el que se le pregunta por el coste de combustible.

- Finalmente si se selecciona run, entonces el algoritmo debe ejecutarse y mostrar por pantalla el camino encontrado.

2. Eventos del ratón

Cuando se pulsa con el ratón sobre la imagen, se pueden tener distintos eventos asociados dependiendo de si se había seleccionado previamente introducir origen, destino o radar.

En los tres casos debe recoger las coordenadas del punto en el que se selecciona (coordenadas X e Y), si estamos en el caso de origen o destino, el usuario debe introducir la altura a través de una ventana de diálogo. Dependiendo de si es un origen, un destino o un radar debe pintarlo de una forma u otra.

Descripción de la información necesaria para poder aplicar el algoritmo:

- Por un lado necesitamos la posición del origen (fila, columna y altura).

- Debemos tener también al menos un destino, aunque podría haber más de uno, no puede haber cero.

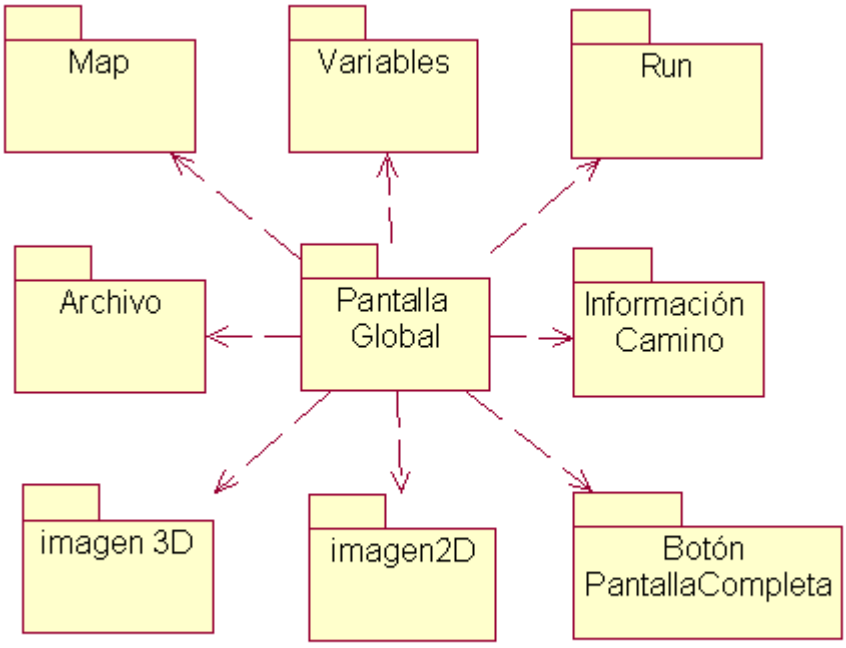
- El número de obstáculos es irrelevante. Puede haber cualquier cantidad incluyendo cero.

- También de forma opcional se puede pedir que se incluya información sobre el combustible disponible y el consumo promedio.

Diagramas de dependencias

Podemos ver el diseño de dos formas diferentes. Por un lado podemos ver la relación de objetos que se observan en la interfaz y por otro las relaciones que existen entre las funciones que ya han sido implementadas. Veamos ambas.

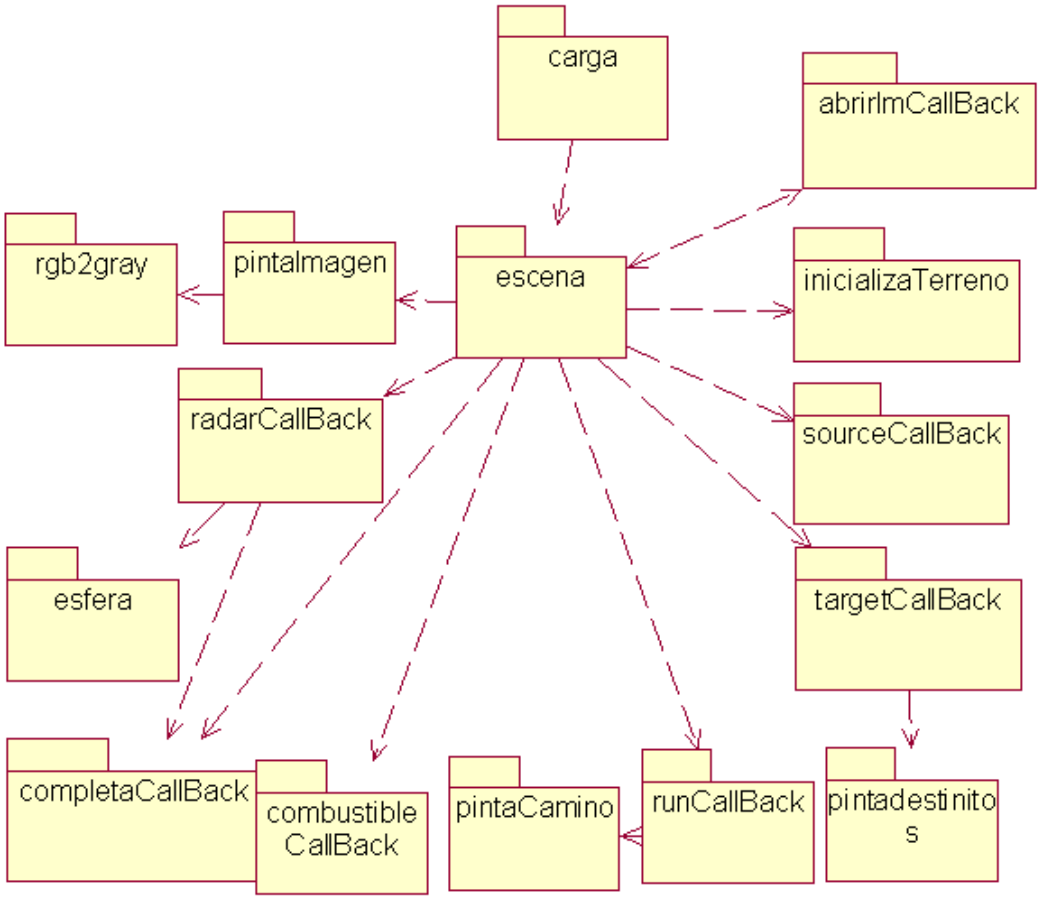
La primera de ellas nos muestra un diagrama como el siguiente:



Podemos ver que desde la pantalla principal tenemos acceso a una serie de opciones.
Alcanzamos las representaciones de las imágenes tanto en dos dimensiones como en tres dimensiones. Tenemos acceso al botón que nos permite una visión en toda la pantalla de la imagen en tres dimensiones.
También tendremos acceso a la información sobre el camino.

Finalmente podremos acceder a los diferentes menús que se tienen en la interfaz y que son “archivo” que permite abrir una nueva imagen,”map” que nos permite introducir un origen, un destino u obstáculos, “variables”, que nos deja introducir la cantidad de combustible disponible y su consumo promedio y “run” que llama a ejecutar al algoritmo y dibuja el camino.

Si queremos verlo desde el punto de vista de los archivos implementados quedaría un esquema como éste:



Inicialmente se llama a escena desde la consola de Matlab llamando a la función carga, que creará la interfaz a partir de una imagen que se elija. Por tanto depende de "pintaImagen" e "inicializaTerreno".

Las demás relaciones son debidas a funciones de llamada en el menú (salvo "completaCallBack" a la que se llama cuando se pulsa el botón de pantalla completa).

Cuando estamos en "runCallBack" se llama a "pintaCamino"(que es el que resuelve el camino mediante el algoritmo de búsqueda A*, para representarlo en la pantalla).

La función "radarCallBack" necesita de esfera (dibujamos esferas donde se coloca un radar)

Para poder transformar nuestra imagen a una matriz que represente las coordenadas de cada punto del mapa, llamamos a rgb2gray desde "pintaImagen".

Por último vemos que "targetCallBack" depende de "pintadestinitos" al que llama para pintar los destinos parciales.

Descripción de sus archivos principales

La parte gráfica en tres dimensiones se ha implementado en Matlab. Las funciones que se han creado han sido las siguientes:

carga

Es la función que se llama al comienzo de la ejecución. Su única misión es inicializar el terreno y llamar a otra función "escena" que será la que cree la interfaz para la iteración con el usuario (se selecciona la imagen que se quiere y se visualiza con todos los menús para tratarla).

escena

Es la función que crea la interfaz. Inicialmente se muestra una lista de imágenes disponibles de las que el usuario toma una. Entonces se visualiza una ventana con la imagen en dos y tres dimensiones y aparece un menú en el que se puede seleccionar las distintas opciones para tratar la imagen y un botón que permite visualizar la imagen en tres dimensiones en la pantalla completa.

abrirImCallBack

Permite abrir de nuevo otra imagen diferente a la actual.

inicializaTerreno

Devuelve los valores originales que tenía terreno (en terreno se guarda información sobre el origen, destino, los radares y toda la información referente a nuestro mapa).

combustibleCallBack

Se pide al usuario la información necesaria en torno al combustible, esto es, cuánto combustible queda en el depósito y cuánto consume en vuelo. Actualiza terreno con estos valores.

esfera

Calcula la matriz de la superficie de una elipse dadas las coordenadas de su epicentro y el radio en los tres ejes (x,y,z) que deseamos tener.

pintaImagen

Lee la imagen seleccionada, la dibuja en tres dimensiones en la parte izquierda de la ventana y en dos dimensiones en la parte derecha de la ventana, realmente como hemos dicho anteriormente, es una imagen en tres dimensiones vista desde arriba.

sourceCallback

Si el algoritmo no está calculando el camino selecciona un nuevo origen. Se selecciona en la imagen de dos dimensiones y se le pide al usuario que introduzca la altura adecuada. Posteriormente se actualiza terreno con el nuevo origen y se dibuja por pantalla.

targetCallBack

Selecciona un nuevo destino. En este punto, a diferencia de en el caso del origen, no importa que se esté calculando el camino, ya que los destinos son múltiples, es decir, hay una lista de destinos a los que se quiere llegar y al añadir uno nuevo, simplemente se añade al final de la lista. También se debe especificar la altura a la que se encuentra. Se actualiza terreno añadiendo el destino a la lista de destinos y se dibuja el nuevo destino en las imágenes de dos y tres dimensiones diferenciándolo de los otros destinos no finales.

radarCallBack

Introduce una serie de radares en el mapa. El número de radares lo decide el usuario. Posteriormente se podrá añadir más radares, pero se le ofrece al usuario la posibilidad de introducir varios para su comodidad. La altura de los radares es la altura de esa zona en el terreno, es necesario actualizan en terreno. A diferencia de un origen o un destino, el radar no afecta a una posición en el terreno, sino que tiene un radio de acción en el que actúa con mayor o menor intensidad.

runCallBack

Llama al algoritmo a través de la función "pintacamino" y obtiene el camino óptimo. Posteriormente lo dibuja en las imágenes de dos y tres dimensiones.
A la hora de dibujar, para que se vea el trayecto mejor representado, se ha dividido el camino entre cada dos puntos recorridos en mil unidades para que tarde más en dibujar el camino y el usuario pueda observar la trayectoria dinámicamente mientras se dibuja.
Este procedimiento lo realiza mientras haya destinos no alcanzados, tomando como origen en cada caso el destino anterior (salvo el primero, que tendrá como origen el especificado).

pintaCamino

Desde aquí se llama al algoritmo de búsqueda implementado en Java. Tenemos que transformar el cubo que describe cada casilla del terreno como su objeto java correspondiente, para más tarde llamar al método de Java que resuelve el algoritmo.
Una vez que tenemos el camino, lo recorremos para dividirlo en tres listas diferentes: la que contiene las filas, la que contiene las columnas y la que contiene las alturas.

pintadestinitos

Dibuja en las imágenes los destinos parciales de forma diferente al destino final para diferenciarlo.

completaCallBack

Actualiza la pantalla completa para que dibuje todo lo que es dibujado en las pantallas pequeñas (radares, destinos, camino, y origen). Cada vez que se pulse sobre el botón de pantalla completa, se llama a esta función.

Descripción de las principales funciones predefinidas de Matlab utilizadas:

Principales funciones predefinidas en Matlab, serán de utilidad para cualquier desarrollador que reutilice el código implementado.

Vemos una breve descripción de las funciones predefinidas utilizadas en este proyecto.

- inputdlg: Crea una caja de diálogo de entrada. Devuelve la respuesta del usuario.
- ginput: Entrada de datos utilizando el ratón. Devuelve las coordenadas del punto donde se pincha con el ratón. Se puede añadir como argumento de entrada el número de posiciones que se quiere leer.
- round: Redondea al entero más cercano.
- axis: Distinto tipo de opciones que permiten la escala de los ejes y su apariencia.
- num2str: Convierte el número de entrada en una cadena.
- findobj: Localiza los objetos gráficos que cumplan las propiedades que se le pasen como argumento y devuelve sus manejadores.
- delete: Elimina archivos o ficheros gráficos.
- size: Devuelve las distintas dimensiones de un array.
- ellipsoid: Genera un elipsoide con las coordenadas especificadas.
- currentAxes: Establece cuales son las coordenadas destino para las coordenadas de los hijos.
- subplot: Creación y control de múltiples coordenadas.
- hold: Mantiene la gráfica actual en la figura, nos permite añadir superficies y otras representaciones 3D al cubo actual
- surf: Dibuja una superficie en 3 dimensiones sombreada.

- alpha: Establece propiedades de transparencia en los objetos de las coordenadas actuales.

- comet3: Dibuja un gráfico animado en el que un círculo traza los puntos que se le pasan como argumento en la pantalla.

- imread: Lee la imagen de un fichero gráfico

- rgb2gray: Convierte de formato RGB a formato de intensidad.

- close: Elimina la figura especificada.

- meshgrid: Genera dos matrices X e Y para representación gráfica.

- double: Convierte a precisión doble

- view: Especifica el punto de vista.

- min: Calcula el mínimo elemento de un array.

- max: Busca el máximo elemento de un array.

- CurrentFigure: Es la ventana designada para recibir las salidas gráficas.

- figure: Crea una figura de objeto gráfico.

- javaaddpath: Añade una entrada a una ruta dinámica de Java.

- javaObject: Construye un objeto de Java

- floor: Redondea al entero más próximo por defecto.

- dir: Muestra la lista del directorio

- listdlg: Crea una caja de diálogo con una lista de selección.

- strcat: Concatena las cadenas que se le pasan como argumento.

- uicontrol: Crea un objeto de control en la interfaz con el usuario. Se puede asignar distintas propiedades que lo caracterizan como las coordenadas o si es visible. Hay distintos tipos de elementos que se pueden crear:

    1. text: Crea un texto.
    2. pushbutton: Crea un botón
    3. etc.

- uimenu: Crea un menú en la ventana de la figura actual. Se le puede asignar distintas propiedades como el tipo de objeto que se crea o una etiqueta para diferenciarlo de otros objetos similares.

Integración Matlab y Java

El código que resuelve el algoritmo de búsqueda se ha realizado en Java y la representación gráfica de la búsqueda se ha realizado en Matlab, por lo que se hacía necesaria una integración entre ambas tecnologías.

La compatibilidad de Java con Matlab es total, de tal forma que se puede llamar a cualquier clase de las librerías de Java importándola. Cualquier carpeta que contenga clases Java se puede utilizar creando un lazo como veremos más adelante. Todo esto hace que la iteración entre ambas tecnologías sea sencilla e idónea para nuestro proyecto salvo algunos problemas de eficiencia.

Cómo llamar a librerías Java:

La función que hace las llamadas a Java es "pintacamino" y necesita utilizar objetos de Java del tipo ArrayList, por lo que se debe importar mediante la siguiente sentencia "import java.util.*".

Con esta instrucción podemos crear objetos de cualquier clase que se encuentre en la librería java.util.

Cómo llamar a las clases creadas por nosotros en Java:

Para llamar al código necesitamos establecer primero un lazo con las clases Java. Hay dos tipos de lazos: dinámicos y estáticos.

Se ha utilizado lazo de tipo dinámico, pues sólo queremos establecer uno, parece más conveniente que cuando abandonemos la aplicación se rompa el enlace a que permanezca, ya que sólo es necesario cuando se utiliza este algoritmo y al establecer un único enlace la eficiencia de la aplicación no se ve resentida.

Para realizar el lazo dinámico necesitamos una instrucción como ésta: "javaaddpath('ruta donde están las clases');". Hay que tener en cuenta que los únicos archivos que reconoce de Java son los *.class (no son válidos los *.java).

Cuando se quiere crear un objeto java hay que diferenciar entre los objetos de una clase de una librería y los objetos de una clase de las implementadas no pertenecientes a las librerías.

En el primer caso sirve con asignar a una variable el nombre de la clase (con los parámetros entre paréntesis y separados por comas si los tiene o sin paréntesis en caso de no tener parámetros).

En el segundo caso debemos especificar que se trata de un objeto de Java mediante la siguiente sentencia: "javaObject(parámetros)". En parámetros se encuentra en primer lugar la clase de la que se crea el objeto junto con los argumentos, los cuales deben estar en el mismo orden en el que se declaró en el constructor del objeto de la clase correspondiente.

La información del mapa del terreno que se almacena en el proyecto de Matlab es necesaria transformarla en un objeto, en este caso un ArrayList que representa el cubo, para poder introducirla como parámetro de entrada del algoritmo de Java.

Para ello es necesario recorrer la matriz, estructura de Matlab y extraer posición a posición su información para poder introducirla en un nuevo objeto de la clase ArrayList vacío inicialmente.
Por motivos de eficiencia en lugar de en el recorrido añadir un nuevo elemento al ArrayList con todos los puntos de mapa, se ha introducido sólo los significativos, es decir, donde había algo, creando un objeto de la clase EspacioNode y añadiéndolo al ArrayList creado previamente.

Posteriormente, se utiliza un método, perteneciente a la clase Espacio, en el que se le pasa el ArrayList y las dimensiones de la matriz y devuelve otro ArrayList en el que se ha completado al anterior con la información más irrelevante, es decir, donde no se encontrada nada (origen, destino u obstáculo). El motivo es que crear un objeto en Matlab de una clase Java es más lento que crearlo desde el propio Java, por lo que se buscaba crear tan pocos objetos como fuera posible.

Para poder llamar a completa cubo necesitamos crear previamente un objeto de tipo Espacio al que se le pasará el ArrayList completo como atributo. También será necesario establecer cual es el origen y el destino.

Con toda esta información ya se puede crear un objeto de la clase Algorithm que es donde se encuentra el método runAlgorithm que implementa la búsqueda.
A este método se le llama con tres argumentos: el objeto de Espacio creado, el combustible disponible y el consumo promedio.

El algoritmo nos devuelve un ArrayList de EspacioNode que establece el camino mínimo calculado. Para convertirlo de nuevo a un formato de Matlab, se recorre el ArrayList y se extrae en cada iteración una fila, una columna y una altura (las del mismo elemento) mediante métodos de acceso a los EspacioNode(getFila(), getColumna() y getAltura()) y se almacenan en listas de Matlab.

Coordinación de grupos de proyecto.

Cuando enfocamos por primera vez la posibilidad de realizar este proyecto sabíamos que había otro grupo interesado en el mismo, la aproximación inicial era que cada uno de los grupos se encargaría de hacer un desarrollo por separado, es decir aunque se tratara del mismo planteamiento, realizarlo como dos proyectos diferenciados, para este propósito se pensó que cada grupo podía usar un algoritmo de búsqueda diferente, pero no era una buena opción, porque de entre los algoritmos conocidos para calcular la distancia mínima el algoritmo A* es muy superior al resto, ya que es el algoritmo más rápido de entre los que te garantizan encontrar solución si la hay. Surgió entonces la posibilidad de coordinar ambos grupos para poder trabajar sobre el mismo proyecto, de esta manera se podía avanzar más en el desarrollo del mismo, y se evitaba el problema anteriormente comentado. A lo largo del año dicha coordinación se puede separar en tres fases.

Primera fase: Trabajo conjunto.

Una vez asignado el proyecto a ambos grupos comenzamos la coordinación, como era necesario primero establecer claramente los objetivos que queríamos alcanzar y los medios para ello, la primera fase del proyecto fue totalmente conjunta.

Aparece la necesidad de utilizar distintas herramientas, que comienza en esta fase y se extiende hasta el final del desarrollo, aunque los detallamos aquí por coincidir en el tiempo.

Para una correcta comunicación entre los seis componentes fue necesario habilitar un grupo de correo vía e-mail. El contacto continuo ha sido imprescindible, ya que tomar decisiones se convierte en una tarea dificultosa.

No solo es fundamental la comunicación, a la hora de implementar todos tenemos que tener la misma versión y a ser posible la más actualizada, no sólo del código del programa sino también de los archivos y especificaciones. Para este fin hemos utilizado una herramienta CVS (Concurrent Versions System), alojada en freepository, en la cual los seis miembros figurábamos como administradores, una muestra más del trabajo conjunto a lo largo del proyecto.

Para una planificación detallada de cada iteración del desarrollo creamos documentos para reflejar:

- Los puntos a tratar en las sucesivas reuniones.
- Las funcionalidades que iban a ser implementadas en dicha iteración se incluían en el documento de casos de uso, el cual se iba incrementando a medida que los requisitos crecían.
- En otro documento se concretaba el reparto de tareas entre los distintos componentes, y el tiempo estimado para esa iteración.

Segunda fase: Reparto de tareas.

Tras finalizar los objetivos propuestos para la primera aproximación en dos dimensiones, como se ha descrito en el punto de explicación del desarrollo, teníamos que afrontar ahora un punto de vista en tres dimensiones, la complicación del algoritmo crecía, y teníamos que buscar un medio factible para poder mostrar dichos resultados, es decir, aparecía una clara división en el proyecto por una parte el algoritmo (Java) y por otra la interfaz (Matlab). Fue necesarias varias reuniones para esclarecer los puntos claves que debían tratar ambas partes, además aparecieron nuevos requerimientos para ambos casos, y un nuevo problema, la integración de ambas partes, que se incluyó principalmente en la parte de la interfaz. Además para una buena integración, fue necesario detallar con mucho cuidado las funciones y objetos necesarios para la comunicación entre ambas partes.
Una vez teníamos la planificación terminada era necesario hacer el reparto de tareas, como hemos dicho teníamos dos partes muy diferentes, que se podían implementar de forma independiente gracias a que la planificación comentada era muy precisa en cuanto a las funcionalidades que debían tener. Fueron estas las razones que nos llevaron a separar el grupo en dos, y así poder encargarse tres personas del algoritmo y el resto la interfaz, tras varias propuestas concluimos en que lo mejor era separarnos según habíamos creado los grupos iniciales del proyecto, y así conseguir una mejor coordinación, porque como es natural, al escoger un proyecto con otros dos compañeros es necesario saber de ante mano que el trabajo, tanto en horas como en posibilidades, es compatible para todos los componentes del grupo, lo que facilita el desarrollo del proyecto.

Tercera fase: Integración.

Tras finalizar cada grupo por separado su tarea asignada llegamos a la integración, momento en el que ambos grupos nos volvimos a reunir para conectar la interfaz y el algoritmo, gracias a una buena planificación inicial, conectar ambos programas fue rápido y no muy dificultoso, aunque fue necesario solventar nuevos problemas que aparecieron, principalmente de eficiencia, como ya detallamos en la explicación del desarrollo e implementación, los cuales se resolvieron en la medida de lo posible, por nuestro grupo, el encargado de la interfaz.

Conclusiones de la coordinación:

La coordinación que hemos llevado a cabo ha sido eficaz, ha cumplido satisfactoriamente las necesidades de nuestro proyecto, así mismo ha sido eficiente, no hemos tenido retrasos debido a una mala comunicación y no ha habido problemas significativos a la hora de repetir trabajo innecesario.

Además ambos grupos hemos conseguido tener pleno conocimiento de los detalles principales del proyecto, aunque hayamos profundizado más en aspectos de implementación. Dado que en la primera fase el trabajo fue conjunto, y que toda la

planificación de requisitos y restricciones a lo largo del año, así como las decisiones principales, han sido tomadas en conjunto.

En consecuencia, la coordinación ha sido en general muy satisfactoria, en cuanto a nuestro trabajo se refiere.

Perspectivas futuras

De cara al futuro se pueden plantear que elementos se pueden añadir al proyecto, y cuales se podrían mejorar.

Por un lado hay que comentar la elección de utilizar Matlab para la representación gráfica en tres dimensiones. Presentaba la ventaja de la compatibilidad entre Matlab y Java que es completa, la cual permitía crear un objeto de Java de forma inmediata. Sin embargo, de cara al futuro sería aconsejable pensar en utilizar otro recurso que sea más eficiente, ya que a medida que aumenta el tamaño del mapa los problemas de eficiencia que presenta aumentan de manera elevada, y puede ser interesante incluir zonas que no aparezcan visibles hasta que se alcancen, tareas que implementan programas de simulación más avanzados.

Posibles incorporaciones:

- Para optimizar el realismo de las representaciones se podría utilizar, como herramienta de desarrollo, Mapping Toolbox que está incluida en MATLAB.

- De cara a la implementación, se podría pensar en añadir una serie de obstáculos que fueran objetos móviles, lo cual se acercaría más a un sistema real, como por ejemplo la existencia de otros aviones o de misiles que pudieran alcanzarle. Para lo cual sería necesario tener en cuenta la trayectoria, ya que un avión que se está alejando, por ejemplo, no resultaría un problema, mientras que si su recorrido puede interceptar el nuestro habría que considerarlo como obstáculo.

- Añadir información que no esté visible en el mapa, sino que se le administre, por ejemplo, la posibilidad de que otro avión o una torre de control pueda avisar de un misil que acaba de ser lanzado en su dirección, o un avión no detectado. Así mismo, un avión podría ser capaz de avisar a otros aviones de la existencia de cualquier contratiempo.

- Incluir nuevos tipos de obstáculos con distintas características como, por ejemplo, radares con diferentes alcances. Otro ejemplo sería discriminar entre los radares del enemigo, que son un obstáculo a evitar, y los propios, que no presentan problema alguno.

- Considerar varios destinos, pero no el orden en el que deben ser recorridos, se establece como siguiente destino el primero que se alcance, es decir calcular los caminos mínimos a cada destino y escoger el menor de ellos. Sería necesario un estudio más detallado para contemplar todas las posibles decisiones en cuestión del orden del recorrido comentado. Hay que tener en cuenta que en próximas evaluaciones la situación puede haber cambiado.

- Considerar el distinto consumo del avión en función de factores como la altura que alcanza, si está despegando o aterrizando, si hay turbulencias, etc.

En resumen, la toma de decisión del camino mínimo puede hacerse incrementalmente más real, si se añaden requisitos adecuados. El trabajo que hemos realizado es una base para un desarrollo que se podría aumentar con múltiples trabajos, porque en un sistema real el número de agentes que pueden intervenir es muy grande. Para la toma de decisiones se tendría que utilizar otras técnicas de inteligencia artificial, e investigar en la prioridad de dichas decisiones, por ejemplo, si solo tenemos dos opciones en uno de los camino el avión va a ser interceptado por un misil, y en el otro colisionaría con un avión enemigo, y no hay ninguna otra posibilidad, podemos pensar que es más prioritario elegir el avión para al menos haber derribado un avión enemigo, o por el contrario establecer que el daño puede ser menor si solo le alcanza en un ala (por ejemplo) para poder rescatar el artefacto.

Las perspectivas de futuro son por tanto muy amplias y dejan volar nuestra imaginación.

Conclusiones

Los simuladores de vuelo son herramientas útiles para entrenar los programas desarrollados para calcular trayectorias "inteligentes" de los aviones. En este proyecto hemos intentado desarrollar uno de estos simuladores e integrarlo con una inteligencia que calcula su trayectoria.

Para desarrollarlo hemos utilizado MATLAB que es una aplicación que contiene muchas herramientas y utilidades que permiten además diversas funcionalidades, como la presentación gráfica en 2 y 3 dimensiones. Esos útiles están agrupados en "paquetes" (*toolboxes*). A Matlab se le puede añadir paquetes especializados para algunas tareas (por ejemplo, para tratamiento de imágenes). Matlab ha resultado ser una herramienta muy potente y útil para nuestro trabajo

La cooperación entre distintos grupos es un medio muy eficaz para conseguir mejores objetivos. Para que se pueda realizar es necesaria una buena especificación, planificación y coordinación de cada iteración en el proceso de desarrollo. En nuestro caso ha sido muy satisfactoria aunque siempre mejorable.

En este proyecto, hemos aplicado conocimientos adquiridos durante la carrera, a la vez que nos ha sido necesario la investigación de nuevos campos y tecnologías.

La necesidad de integrar dos tecnologías tan distintas, como un lenguaje de alto nivel que es JAVA, con una herramienta de cálculo numérico y tratamiento de datos, que es MATLAB, hace que el proyecto abarque muy distintos campos en cuanto a conocimientos se refiere.

Bibliografía

- Matlab. The language of technical computing. Creating graphical user interfaces; www.mathworks.com

- Mapping Toolbox 2.1. Analyze and visualize geographic information; www.mathworks.com

- Aprenda Matlab 7.0 como si estuviera en primero;
  Escuela Técnica Superior Ingenieros Industriales. Universidad politécnica de Madrid

- Marchand, P.; Graphics and GUIs with Matlab; Boca Raton, Florida

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

- Rich, E. y Knight, K.; *Artificial Intelligence;* McGraw-Hill, New York, 2ª edición, 1991 (edición en español, 1994). ;

- Gonzalo Pajares y Matilde Santos; *Inteligencia Artificial e ingeniería del Conocimiento;* RA-MA, 2005 (Primera Edición en español);