Seagull

COORDINACIÓN DE SISTEMAS MULTIAGENTES PARA TAREAS DE LOCALIZACIÓN DE OBJETOS

Alejo Jesús Nevado Holgado.
Victor Martín Gómez.
Jesús Lozano Robles.

Profesora: Matilde Santos Peñas.

Autorización de uso

Nosotros los autores del proyecto: **COORDINACIÓN DE SISTEMAS MULTIAGENTES PARA TAREAS DE LOCALIZACIÓN DE OBJETOS,** de la asignatura de *Sistemas Informáticos*:

Alejo Jesús Nevado Holgado

Jesús Lozano Robles

Víctor Martín Gómez

Dirigidos por:

Matilde Santos Peñas

Autorizamos a la Universidad Complutense de Madrid a utilizar y difundir, con fines académicos, el contenido de este documento de texto, así como del contenido del CD complementario que adjuntamos con él mismo.

Madrid, 1 de Julio de 2007.

Palabras clave para búsqueda bibliográfica

1) Agentes inteligentes
2) Red neuronal
3) Algoritmos evolutivos
4) The Traveler Salesman Problem
5) Redes de convolución
6) Touring Machine
7) Java 3D
8) JADE
9) Control de vuelo en formación
10) Sistemas multiagentes

Abstract (Español):

Tal como su nombre indica, “sistema multiagente para la localización de objetos”, el presente proyecto desarrolla un sistema multiagente diseñado para la localización de un tipo determinado de objeto, en nuestro caso para la localización de naufragios y barcos. El sistema podría usarse con mínimas modificaciones para la detección aérea de otro tipo de objetos, como focos de incendios, alpinistas perdidos... pero, en este caso, el objetivo de la detección que hemos decidido como demostración de sus capacidades ha sido el mencionado.

Descripción general.

¿Y como es capaz este “sistema” de detectar náufragos en alta mar?, ¿En que consiste este “sistema”?. Se trata de un conjunto de aviones inteligentes que sobrevuelan en formación una zona seleccionada, tomando fotografías de lo que van sobrevolando e identificando autónomamente e *'in situ'* si en alguna de ellas aparecen restos de naufragio. *Pero para hacer todo esto, el sistema necesita ser pilotado a distancia ¿no?.* No. El sistema (es decir, los aviones) son completamente autónomos. Una vez que desde la base de despegue se les dice que área tienen que explorar, ellos deciden como explorarla, se van, la exploran, y vuelven al cabo de un tiempo con los resultados. *Entonces esto significa que vosotros tenéis unos aviones y que los podéis usar ya mismo para realizar esta tarea.* No. Lo que nosotros tenemos es un simulador tridimensional del escenario en donde se mueven los aviones. El sistema está pensado para ser instalado en unos aviones robóticos reales, pero actualmente sobre lo que funciona es sobre un simulador de vuelo. Nosotros nos hemos encargados de diseñar la inteligencia artificial de estos aviones, pero su instalación en robots reales tendría que ser realizada por otros como continuación de este proyecto. Un ejemplo de un candidato para esto, son los aviones del proyecto SIVA del INTA.

Capacidades y límites actuales .

Respecto a lo que son capaces de hacer estos aviones inteligentes, o agentes, diremos que pueden: calcular la ruta que deben seguir para explorar una determinada zona con forma de polígono irregular utilizando poco tiempo y combustible; comunicarse y coordinarse; volar en formación para aumentar el área que son capaces de fotografiar juntos; identificar si en las fotografías que toman aparece el objeto que buscan; no perderse si se interrumpe la comunicación con el jefe de formación o la base. Respecto a que no son capaces de hacer, diremos: no pueden despegar o aterrizar en tierra firme, en el simulador los aviones comienzan y terminan el vuelo en el aire; no pueden rescatar a los náufragos ellos mismos, tan sólo avisar a base de su posición; aún no están programados para enfrentarse exitosamente a cualquier eventualidad imprevista posible, aunque el sistema está diseñado para poder incluir capacidades como esta fácilmente.

Abstract (English):

As its name tells, “multiagent system for objects localization”, this project develops a multiagent system designed to be used to search and find a specified kind of object, in our case to find ships and wrecks. The system could be used with very few modifications to recognise other kind of objects, such as fires, lost alpinists... but, in the case presented in this memo, the detection objective that we have selected to show its capabilities, is a ship.

General description.

How is this system able of detecting wrecks in deep sea? How does this system work?. The system is a set of various planes (by default three) that fly in formation over a selected zone, making aerial photographs of the extension below them and recognising 'on the spot' if in one of them a ship appears. *To do all this, is the system piloted remotely?*. No. The system is completely autonomous. The people who use this system only have to say them what is the area the planes must explore, then the planes go there, explore this area, and after that they came back to base with the exploration results. *Then, have you a group of planes that do all that and you can use them just now to manage this task?*. No. The thing we have is a three-dimensional simulator of the scenario in which the planes work, and we run our intelligent agents there. The system is designed to be installed in real robotic planes, but presently they run in a simulator we have build for them. The artificial intelligences we have designed are intended to be installed ultimately in real planes, but its installation in those real planes would have to be made by other people as a continuation to this project. An example of a robotic candidate for that, are the planes of the SIVA project in INTA.

Present capabilities and limits.

About the capacities those intelligent planes have, they can: calculate the shortest route they must follow to search all the area described by a irregular polygon; communicate and coordinate each other; fly in formation to increase the area they are able to spot in flight; identify in which photographs, between those that they make, a ship appears; do not become lost if one of them lose the communication with the boss of the formation. About the things they can not do: They can't take off or land, in the simulator they began and end the flight in the air; they can't rescue the shipwrecked persons themselves, they only notify either if there is some wreck or not; they are not designed to confront whatever possible eventuality, but the agent architecture is devised to include more capabilities easily.

Índice

TEMA 1

Introducción y Estado del Arte

1.1 Introducción.

Nuestro proyecto de fin de carrera oficialmente recibe el nombre de "Coordinación de Sistemás Multiagente para Tareas de Localización de Objetos", pero por razones obvias nosotros hemos preferido darle un nombre algo más corto y le hemos llamado Seagull (Gaviota). La razón de que hayamos elegido este apelativo es porque, en el cine, este animal es el primero que suelen ver los náufragos cuando su deriva los acerca a tierra firme. Lo normal, llegados a este punto, es que el lector se pregunte: ¿Que tienen que ver las gaviotas y los náufragos con un sistema multiagente para la localización de objetos?. Pues muy sencillo, nuestro "sistema multiagente para la localización de objetos" lo hemos orientado a la localización de náufragos, barcos extraviados y objetos en alta mar en general. El sistema podría usarse con mínimás modificaciones para la detección aérea de otro tipo de objetos, como focos de incendios, alpinistas perdidos... pero, en este caso, el objetivo de la detección que hemos decidido como demostración de sus capacidades ha sido el mencionado.

Descripción general.

¿Y como es capaz este "sistema" de detectar náufragos en alta mar?, ¿En que consiste este "sistema"?. Se trata de un conjunto de aviones inteligentes que sobrevuelan en formación una zona seleccionada, tomando fotografías de lo que van sobrevolando e identificando autónomamente e *'in situ'* si en alguna de ellas aparecen restos de naufragio. *Pero para hacer todo esto, el sistema necesita ser pilotado a distancia ¿no?.* No. El sistema (es decir, los aviones) son completamente autónomos. Una vez que desde la base de despegue se les dice que área tienen que explorar, ellos deciden como explorarla, se van, la exploran, y vuelven al cabo de un tiempo con los resultados. *Entonces esto significa que vosotros tenéis unos aviones y que los podéis usar ya mismo para realizar esta tarea.* No. Lo que nosotros tenemos es un simulador tridimensional del escenario en donde se mueven los aviones. El sistema está pensado para ser instalado en unos aviones robóticos reales, pero actualmente sobre lo que funciona es sobre un simulador de vuelo. Nosotros nos hemos encargados de diseñar la inteligencia artificial de estos aviones, pero su instalación en robots reales tendría que ser realizada por otros como continuación de este proyecto. Sería muy interesante que un grupo de alumnos se encargara, como proyecto fin de carrera, de pasar las inteligencias artificiales que hemos creado a robots físicos, sobre todo por su relación con proyectos como el SIVA del INTA.

Capacidades y límites actuales .

Respecto a lo que son capaces de hacer estos aviones inteligentes, o agentes, diremos que pueden: calcular la ruta que deben seguir para explorar una determinada zona con forma de polígono irregular utilizando poco tiempo y combustible; comunicarse y coordinarse; volar en formación para aumentar el área que son capaces de fotografiar juntos; identificar si en las fotografías que toman aparece el objeto que buscan; no perderse si se interrumpe la comunicación con el jefe de formación o la base. Respecto a que no son capaces de hacer, diremos: no pueden despegar o aterrizar en tierra firme, en el simulador los aviones comienzan y terminan el vuelo en el aire; no pueden rescatar a los náufragos ellos mismos, tan sólo avisar a base de su posición; aún no están programados para enfrentarse exitosamente a cualquier eventualidad imprevista posible, aunque el sistema está diseñado para poder incluir capacidades como esta fácilmente.

A lo largo de esta memoria se expondrá completamente el funcionamiento de la

versión actual de Seagull. Se incluye la información necesaria para poder entender y usar el código contenido en el CD-ROM del proyecto, o incluso para poder reproducir el sistema completamente. Comenzaremos por una descripción del estado actual de la investigación en el campo de los agentes inteligentes (Tema 1). Después pasaremos a una descripción detallada de cada una de las capacidades del sistema (Tema 2), su diseño y su funcionamiento. Tras esto expondremos las conclusiones que hemos sacado tras realizar este trabajo, comentando al mismo tiempo las posibles ampliaciones y mejoras que pueden hacerse sobre cada una de las capacidades de Seagull (Tema 3). Por último dejaremos aquí la información técnica necesaria para seguir trabajando en el futuro en este sistema: diagramás UML, Javadoc, fragmentos de código importante, y algunos ejemplos de ejecución (Tema 4).

1.2 Estado del arte: Metodologías orientadas a agentes

A la hora de plantear cualquier proyecto, se hace necesario el seguimiento de una metodología clara y útil, que nos sirva para reducir el tiempo empleado en especificar el agente y cómo implementarlo.

Un poco de historia de las metodologías

La metodologías orientadas a agentes tienen muy poco tiempo. Las primeras creada data de 1996, la conocida como más-CommonKADS, [SIN01]; otro ejemplo es másE [DEL1999]; con un recorrido de 7 años. Con esto queremos indicar que todavía es un ámbito que está por desarrollar y unificar, ya que a día de hoy existen unicamente unas 10 metodologías diferentes que están siendo desarrolladas por distintos grupos científicos.

Cada metodología está planteada para un conjunto de aplicaciones y éstas están planteadas para un conjunto de usuarios y de agentes determinado. No es comparable la metodología Tropos [KOL2001] orientada al E-bussiness que una Prometheus [PAD2002] muy básicos y prácticamente orientado a definir un agente sencillo.

De entre todas las metodologías estudiadas, un analisis en profundidad hace que nos quedemos con 3 de ellas: INGENIAS [PAV2005], másE [DEL1999] Y PASSI[BUF2002].

Análisis de las metodologías

Cuales son los puntos fuertes de las metodologías.

Cuando se analiza una Metodología es necesario estudiar todo sus puntos: Si llega a nivel de implementación o no, si sigue un análisis ascendente o descendente, si tiene una herramienta de ayuda al desarrollo y desde cuántos puntos de vista está analizado el agente, etc.

Metodología PASSI

Descripcion iterativa de un agente: PASSI

Esta metodología iterativa y detallada, sigue las pautas de FIPA, que es la arquitectura básica que se sigue como estandar en agentes software y robóticos. Para ello divide el tiempo de vida en 3 partes: requisitos del sistema, modelo de sociedad e implementación del agente.

En lo referente a los requisitos del sistema, esta metodología analiza los requisitos de tres formás: primero identificar numero de agentes posibles, después los roles de los mismos, terminando con las tareas asociadas a cada rol.

Después pasa a analizar el sistema como una sociedad de agentes, para ello tiene un apartado específico (que ninguna otra metodología incorpora) que nos sirve para crear un lenguaje común usado por todos los agentes, luego está su tarea como parte de la comunidad y para terminar el protocolo de comunicaciones específico que usarán los

agentes.

Una vez descrito los agentes sólo queda ver los aspectos de su implementación, que serán analizados en función de su estrucura y el comportamiento general esperado del sistema.

La arquitectura final es una arquitectura FIPA y como programa de apoyo existe un add-on para el Rational Rose llamado PTK.

Metodología INGENIAS

La metodología INGENIAS [PAV2005] creada por profesores de la Universidad Complutense de Madrid, es derivada de otra llamada MESSAGE. La característica principal de MESSAGE que hereda INGENIAS es que crea Meta-Modelos de los agentes para que puedan aplicarse a distintos ámbitos y arquitecturas. Entre ellas la herramienta de ayuda al desarrollo de INGENIAS, el IDK, está preparada para usar el estandard FIPA.

Uno de los puntos fuertes, y por el que al final fue la escogida a seguir, es que analiza el sistema desde 5 puntos de vista distintos, véase: agente, entorno, organizacion, tareas y objetivos e interacciones. Además de ser la que analiza el sistema desde más puntos de vista y de forma más organizada y sistemática, la fases son las siguientes: diseño-concepción, diseño-elaboración, implementación y validación, todas soportadas por el IDK, lo cual le hace la herramienta más completa aunque todavía no esté desarrollada del todo.

Metodología másSE

Simplicidad y proyecto parecido: másE

La metodología másE [DEL1999] es de las que tiene más tiempo y está más probada. Está pensada para sistemás cerrados, es decir, que toda interacción posible con el sistema está o puede estar planificada. Al contrario que PASSI no tiene soporte para ontologías, aunque se está planificando su inclusión. Para ayudar al desarrollo tenemos la herramienta Agent Tool, que nos ayuda al seguimiento de la herramienta.

A la hora de desglosar en las fases, este modelo no pasa de la fase de diseño, dejando la fase de implementación a nuestra suerte.

La fase de análisis se divide en 3 partes: captura de requisitos, aplicación de casos de Uso y Refinamiento de roles. Con estos conseguimos modelos, los casos de uso, los diagramás de secuencia y los diagramás de concurrencia, que nos sirven para detectar cualquier problema en la comunicación entre agentes. Este diagrama soporta el peso de uno de los puntos fuertes de esta metodología: la comunicación.

En la fase de diseño tenemos los siguientes puntos: creación de las clases de los agentes, contruccion de las conversaciones, ensamblaje de las clases de agentes con el resto de modelos generados en cada una de las fases y el diseño final del sistema. En esta fase, que no llega más allá del diseño, no especifica que se tenga que usar el estandard FIPA, si no que deja todas las decisiones de arquitectura al Ingeniero.

Esta metodología ha sido exitosamente aplicada en [DEL2003]. En este caso se uso la metodología másE para aplicarlo en robots que simulan el rescate de otros robots.

La similitud con nuestro caso hizo que le tomáramos muy en cuenta.

Elección de metodología

La elección final.

Una vez expuestas todas las metodologías no teníamos muy claro cual seguir, por un lado másE había implementado algo muy parecido a nuestro sistema, pero sin lugar a dudas la herramienta de desarrollo no estaba muy lograda, además de que no nos permitía un análisis muy amplio de lo que es un agente.

Con todo esto al final nos quedamos con PASSI e INGENIAS; ambas llegaban hasta el nivel de implementación y aunque INGENIAS solo genera meta-modelos, PASSI ayudaba más a la hora de implementar, y los puntos de vista de análisis de un agente eran bastante completos. Al final nos decidimos por INGENIAS ya que el IDK era una herramienta que aunque no terminada, era bastante completa y además el creador de la misma es profesor de la facultad de Informática, lo cual nos otorgaba ayuda de primera mano, que PASSI no nos podría brindar.

¿Era realmente necesaria una metodología?

Cuando nos pusimos manos a la obra con la herramienta vimos que más que ser una ayuda estábamos gastando más tiempo en aprender a manejarla que en avanzar en lo que eran las partes del proyecto. Además, a la hora de implementar código, ya teníamos como base el proyecto del año pasado y nos era más fácil hacer nosotros toda la especificación e implementación que un análisis pormenorizado que no nos ayudaba a avanzar debido a nuestra inexperiencia en proyectos similares.

1.3 Estado del arte: Simulación tridimensional

Tecnologías de representación

En nuestro sistema se hace total y absolutamente necesario una representación del entorno en 3 dimensiones, ya que los aviones pueden volar a distintas alturas y todo eso ha de quedar registrado en el sistema. Por ello es muy habitual que haya representaciones en 3 dimensiones en los proyectos de simulacion de agentes robóticos.

Tecnologías de representación

En el mercado existen muchas librerías y lenguajes que nos permiten hacer representaciones tridimensionales. Las más famosas son DirectX y OpenGl. Estás librerías, basadas en lenguace C/C++ son las herramientas más potentes del mercado y la que nos permite un mayor uso a bajo nivel de todo lo que implica la creación de un entorno tridimensional. Son herramientas de uso profesional, no son interoperables entre plataformás.

A la hora de considerar un entorno de simulación es importantísimo considerar la interoperabilidad. Queremos que nuestra herramienta de simulación pueda correr en cualquier ordenador, sea cual sea su configuración. Es por ello que Sun nos da la respuesta con Java3D.

Qué es Java3D

Java3D es un conjunto de librerías en java para representar escenarios tridimensionales. Bajo la filosofía de "escribe una vez, ejecuta donde sea" Java3D permite interoperabilidad entre plataformás además de un nivel de abstracción mayor que nos permita ser más productivos y aprender más rápido a reflejar escenarios tridimensionales.

Una representación de una escena se basa en un grafo. En este grafo podemos definir objetos, su comportamiento y respecto a qué depende su escala de representación. Con esto Sun nos brinda la posibilidad de escribir código de una manera rápida, ya que oculta la mayor parte de detalles matemáticos, lo que facilita la escritura de código. El único problema que tiene Java3D es que al ser un lenguaje compilado, sus rendimientos no llegan a ser tan espectaculares como pueden ser los de OpenGl o DirectX, pero nuestras necesidades las satisface perfectamente.

Proyectos de simulación sobre Java.

Proyecto en Java3D

La simulación de este proyecto está basado en [EST2006]. Este proyecto incorpora una representación gráfica de un modelo de la dinámica de un avión (RCAM), para comprobar su funcionamiento, y tener una referencia más visual de su

comportamiento.

Este proyecto tiene la particularidad de que primero creaba una secuencia de órdenes para que ejecute un avión único que estuviera en el cielo. Luego esta secuencia de órdenes se traducía a una posición y una rotación del avión respecto de sus propios ejes. No era en si misma una representación en tiempo real, si no que calculaba todos sus movimientos para que luego el motor gráfico interpolara todas las posiciones intermedias y creaba una animación que se representaba cada intervalo de tiempo.

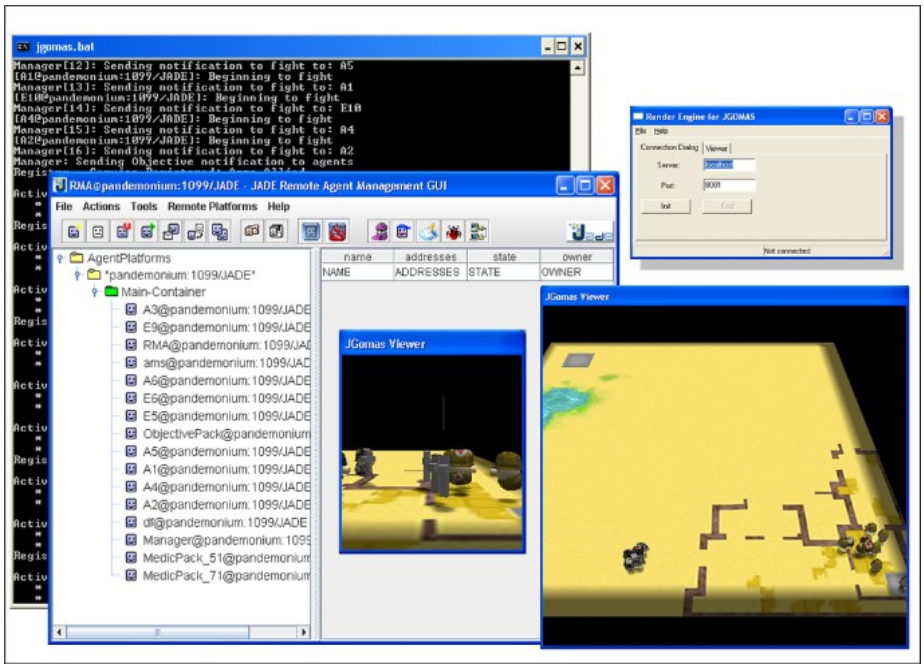


Ilustración 1: Proyecto JGomás

Problemás de proyectos anteriores

Pero este proyecto, del cual partimos, tenía los siguientes problemás:

- Precálculo de la simulación: Nuestra aplicación tiene que ser en tiempo real, y la de [EST2006] usa una función de interpolación para generar la animación.
- Lentitud del sistema: al usar una función de interpolación para generar la animación el rendimiento del sistema caía en picado y no podíamos asumir un sistema tan lento.
- Mala división en clases: el diseño de paquete, clases y métodos no estaban bien entrelazados; el código era difícilmente extensible, además de que había problemás con ciertos métodos que estaban dentro de la clase, pero que hubiera sido mucho mejor haberlos sacado de la misma.

Por ello nuestro proyecto tenía que mejorar esos aspectos e incorporarle un sistema de Agentes Jade. Para ello comprobamos que existe una aplicación llamada Jgomás (*Ilustración 1*) [JGO2007] que mediante agentes Jade y un entorno en simulación

Proyecto similares basado en OpenGl, crea un framework para la creación de sistemás Multiagente. Este framework, a pesar de estar hecho con gráficos tridimensionales, no estaba pensado para la creación de aviones, ya que solo nos brinda dos grados de libertad.

1.4 Estado del arte: Arquitectura de agentes

En este tema se explicará el estado actual de la programación con agentes inteligentes.

Definición de agente inteligente.

Definición de agente inteligente

"Agente inteligente" es un termino algo ambiguo cuya definición varía según el ámbito de la informática para el que se quiera usar el sustantivo. En inteligencia artificial, el término es usado para referirse a actores inteligentes que tienen la capacidad de observar y actuar en un medio, siendo esta la principal carasterística que los diferencia del resto de sistemás inteligentes aislados, como pueden ser los sistemás expertos clásicos. Un agente es una entidad con cierto nivel de autonomía, cierta capacidad de percepción y de acción. Ejemplos de agentes pueden ser un robot o un programa que puede moverse por internet y realizar operaciones en nombre de un usuarios. A un agente se le considera inteligente si interactúa con el medio en una forma que normalmente sería considerada como intelignete si esta interacción fuera llevada a cabo por un ser humano.

Definiciones similares a esta pueden encontrarse en [BIGUS01], pág. 3:

> *"Recientemente, el crecimiento explosivo de Internet y de la computación distribuida, nos ha conducido a crear la idea de agentes que pueden moverse por la red, interactuar unos con otros y realizar tareas para sus usuarios. Los Agentes Inteligentes usan las últimás técnicas de Inteligencia Artificial para proveer agentes software autónomos, inteligentes y móviles, que permitan extender la capacidad de alcance de los usuarios a través de Internet.[...]*
>
> *En el contexto de Agentes Inteligentes, un evento es cualquier cosa que al suceder cambia el medio o cualquier cosa que el agente debería saber.[...]*
>
> *Cuando un evento ocurre, el agente tiene que reconocer y evaluar que significa dicho evento y responder a dicho suceso.[...]*
>
> *Si queremos que nuestros agentes inteligentes nos hagan la vida más fácil, deben tener la capacidad de realizar acciones [en el medio], de hacer cosas por nosotros.[...]*
>
> *Algunos investigadores sienten que un agente debe ser también proactivo. El agente no solo debe reaccionar a eventos, si no que también debe ser capaz de planear e iniciar acciones autónomamente."*

También pueden encontrarse definiciones del mismo estilo en [ANA01], pág. 3:

"El concepto de agente caracteriza a una entidad software con una arquitectura robusta y adaptable que puede funcionar en distintos entornos o plataformás computacionales y es capaz de realizar de forma inteligente y autónoma distintos objetivos intercambiando información con el entorno, o con otros agentes humanos o computacionles. Las características deseables de un agente son:

- Funcionamiento continuo y autónomo.
- Comunicación con el entorno y con otros agentes [...].
- Robustez.
- Adaptabilidad [a distintos entornos] [...]."

Tipos de agentes

Tipos de agentes.

Una clasificación de agentes bastante aceptada, y que a nosotros nos ha parecido la más adecuada, es la que los divide en deliberativos, reactivos e híbridos, como puede encontrarse en [ANA01]. Las definiciones de cada tipo son:

- Deliberativos: "Son aquellas arquitecturas que utilizan modelos de representación simbólica del conocimiento. Suelen estar basadas en la teoría clásica de planificación. Estos agentes parten de un estado inicial y son capaces de generar planes para alcanzar sus objetivos [MAES01]. En estos sistemás parece aceptada la idea de que es necesario dotar a los agentes de un sistema de planificación que se encargue de determinar qué pasos se deben llevar a cabo para alcanzar sus objetivos. Por tanto, un agente deliberativo (o con una arquitectura deliberativa) es aquel que contiene un modelo simbólico del mundo, explícitamente representado, en donde las decisiones se toman utilizando mecanismos de razonamiento lógico basado en la correspondencia de patrones y la manipulación simbólica, con el propósito de alzar los objetivos del agente." El principal representante sería el conjunto de sistemás BDI (Belief, Desire, Intention).
- Reactivos: "Las arquitecturas reactivas se caracterizan por no tener como elemento central de razonamiento un modelo simbólico y por no utilizar razonamiento simbólico complejo. Un ejemplo típico de estas arquitecturas en la propuesta de Roodney Brooks, conocida como arquitectura de subsunción [BROOKS01]."
- Híbridos: "tanto las arquitecturas reactivas como las deliberativas presentan ciertas limitaciones. Por ello, se han propuesto sistemás híbridos que pretenden combinar aspectos de ambos modelos. Una primera propuesta puede ser construir un agente compuesto de dos subsistemás: uno deliberativo, que utilice un modelo simbólico y que genere planes en el sentido expuesto anteriormente, y otro reactivo, centrado en reaccionar ante los eventos que tengan lugar en el entorno, que no requiera un mecanismo de razonamiento complejo."

A su vez los sistemás híbridos, al estar compuestos por subsistemás, se dividirían

en dos tipos: los que tienen una arquitectura vertical y los que tienen una horizontal. En los del primer tipo solo una capa de la arquitectura tendría acceso a los sensores y actuadores. En el segundo tipo, todas las capas tendrían acceso a los sensores y actuadores.

Otra clasificación bastante interesante puede encontrarse en [BIGUS01]. más que una clasificación se trata de un análisis de las posibles capacidades de un agente. Dicho análisis se basa en la creación coceptual de un sistema de 3 coordenadas que miden las capacidades del agente. Estas coordenadas son respectivamente 'autonomía', 'inteligencia' y 'movilidad' [IBM01]. El primero de ellos se refiere al nivel de autonomía que el agente tiene para representar al usuario ante otros agentes, aplicaciones o sistemás software. Con 'inteligencia' nos referimos a la capacidad del agente para aplicar resolver los problemás que se le presentan de forma “inteligente”. La última de las coordenadas, 'movilidad', expresa la capacidad que el agente tiene para moverse en la red o en su entorno cualquiera que sea este.

Agente TouringMachine

Arquitectura TouringMachine

La arquitectura TouringMachine fue propuesta por Ferguson en el artículo [FERGUSON01], y ha recibido bastante aceptación desde entonces.

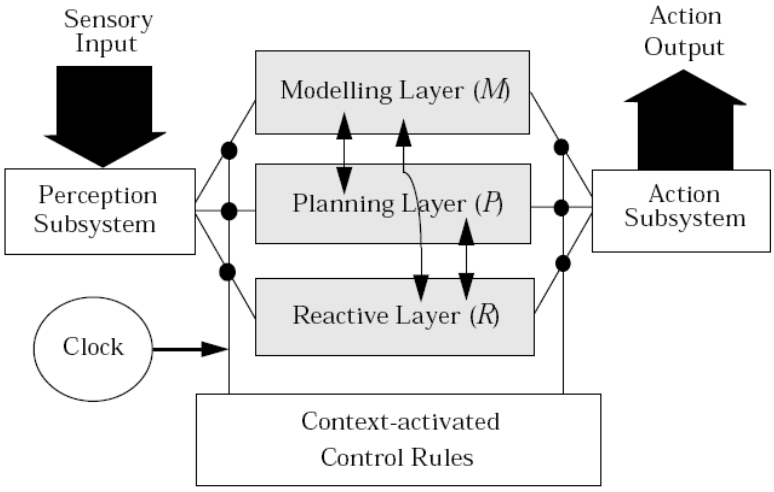


Ilustración 2: Arquitectura TouringMachine.

La arquitectura TouringMachine (*Ilustración 2*) estructura el agente en varias capas, cada una encargada de resolver problemás a distinto nivel de abstracción. De mayor nivel de abstracción a menos a estas se les llama M, P y R, dando la ultima al sistema características reactivas y convirtiendo al agente, por lo tanto, en híbrido. Estas capas no funcionan en paralelo, si no que en cada momento sólo una tiene el control sobre el agente, encargándose el controlador de decidir qué capa obtiene dicho control en cada momento. Dicho controlador es un sencillo motor de reglas que ejecuta dichas reglas con la información que obtiene de la entrada y la salida de datos de cada una de las capas. Por último, para comunicarse con el entorno, la arquitectura cuenta con otros

dos módulos, uno para percibir dicho entorno y otro para actuar sobre él.

Resumiendo: TouringMachine es una arquitectura híbrida concebida para combinar las características de los agentes Reactivos con las de los agentes Deliverativos. El agente está divido en capas de forma horizontal y no organizadas jerarquicamente. Todas las capas tienen acceso al fundo exterior y al resto de capas, tan solo definiéndose una prioridad para resolver conflictos entre capas.

Agente Interrap

Arquitectura InteRRaP

A continuación se ofrece la defición de agente InteRRaP tal y como está aparece en el artículo originario [MULLER01].

El modelo de agente InteRRap, es una extensión del modelo RATMAN ([BUCKERT01]). Interrap fué desarrollado para cumplir los requerimientos necesarior para modelar sociedades dinámicas, tal como robots interactivos. Su principal característica es que combina patrones de comportamiento con capacidades explícitas de planificación. Por una parte, la presencia de patrones de comportaqmiento permite al agente reaccionar rápidamente y de forma flexible ante cambios en el medio en el que el agente se mueve. Por otro lado, la habilidad para diseñar planes de la que dispone es generalmente reconocida como necesaria para resolver tareas más sofisticadas (*Ilustración 3*).

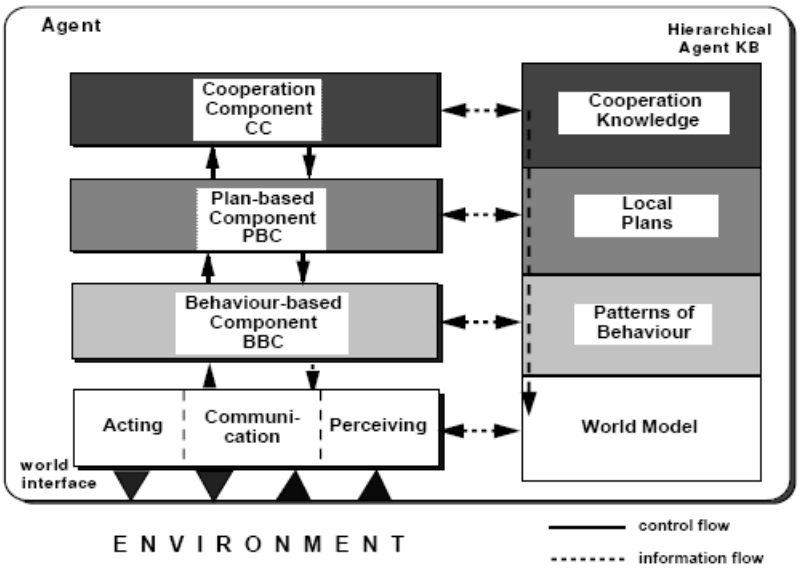


Ilustración 3: Arquitectura Interrap.

Mientra la característica más novedosa de RATMAN (la idea de estructural la base de conocimiento de acuerdo a la complejidad del conocimiento contenido) fue ámpliamente aceptada, hay un punto del sistema que es criticado, y este es la carencia de separación entre el conocimiento usado y la funcionalidad mostrada en el modelo: Los niveles de conocimiento estructurados jerarquicamente no eran construidos usando solo los conceptos de los niveles menores, si no que también eran usados para

disparar actividades en dichos niveles menores.

Interrap claramente marca la diferencia entre la pura base de conocimiento y la parte funcional del agente, mientras preserva la estructura jerarquica del modelo. Por lo tanto, las dos partes del modelo Interrap son:

- La base de conocimientos jerárquica del agente.
- La unidad de control multi-estado.

La figura 21 muestra el modelo Interrap en mayor detalle.

Resumiendo: Interrap es un modelo de agente híbrido que intenta combinar características de agentes Deliberativos con las de agentes Reactivos. Para ello dividen al agente en dos: una unidad que controla como actúa el agente y una base de conocimiento. Ambas partes están jerárquicamente organizadas en capas mediante división vertical, refiriéndose las capas superiores a decisiones y conceptos más abstractos que las inferiores.

Análisis de las distintas arquitecturas para nuestro sistema.

Análisis de posibles arquitecturas para Seagull

A continuación se muestra el análisis que seguimos para seleccionar el tipo de arquitectura de agente que sería necesario para nuestro proyecto.

Analizamos las características de los distintos tipos de agentes y contrastamos estas con los requisitos y objetivos de nuestro sistema. De este análisis concluimos que las arquitecturas deliberativa no podrían ser implementadas debido a que supone una gran carga para el hardware, pues el sistema que va en los robots se supone de poca potencia. Añadido a esto, los sistemás BDI es menos trivial hacerlos resistentes a los cambios bruscos en el entorno típicos de la navegación aérea. Los sistemás reactivos sí cuentan con esta capacidad pero, sin embargo, son demásiado sencillos como para desarrollar un comportamiento tan complejo como el requerido. Además, se hace difícil pensar en un agente reactivo con capacidad para diseñar planes a largo plazo y en coordinación con el resto del equipo de robots. Aún así, se dedujo que la arquitectura de Brooks ([BROOKS01]) probablemente podría cumplir con los requerimientos, pero se abandonó la idea debido al riesgo de que tras comenzar a implementarla nos diéramos cuenta de que esto no era así. Por todo esto se recurrió a los diseños híbridos, que pueden contar con complejas capacidades típicas de los BDI pero con la rápida reacción a eventualidades típica de sistemás reactivos.

De las arquitecturas híbridas, las más prometedoras eran la Interrap [MULLER01] y la TouringMachine [FERGUSON01]. La Interrap finalmente fue descartada debido a que, con sus tres bases de conocimiento funcionando en paralelo, parecía requerir bastante potencia de cálculo. La TouringMachine, por contra, puede dar como resultado un agente mucho más ligero que, sin embargo, no carece de la funcionalidad demandada por la especificación. Este último modelo ofrece capacidad para reaccionar a eventos rápidos e inesperados, como los agentes reactivos. Además, permite elaborar planes complejos y en coordinación con otros agentes cooperativos, como los agentes deliberativos. Concluimos por tanto que las TouringMachine tienen las siguientes características: no necesitan un hardware potente; permiten planificación; y admiten

reacción rápida. Y estos mismos eran los requisitos de nuestro sistema.

1.5 Estado del arte: Agentes

Concepto de agente

Pese a que numerosos agentes y sistemás multi-agente han sido desarrollados, se sigue debatiendo sobre el concepto de agente. En casi toda la documentación que se puede encontrar sobre el tema, se comienza diciendo que el concepto de agente es confuso ya que no existe una única definición sobre agente software.

El término agente proviene del latín "agere"que significa hacer. Agente deriva del participio "agens", término que expresa la capacidad de acción o actuación de una entidad. Por otro lado, en el diccionario de la R.A.E se dan varios significados: "persona o cosa que produce un efecto"; "persona que obra con poder de otra".

En la discusión de qué es un agente, diferentes opiniones se pueden escuchar en relación a la necesidad de la inclusión de varias características. Básicamente, un agente es una entidad de software que exhibe un comportamiento autónomo y un sistema multi-agente es un conjunto de agentes que tienen la capacidad de interactuar en un entorno común. Así, agentes en un entorno con otros agentes poseen capacidades como la comunicación, negociación, y coordinación. Características que podemos considerar opcionales se encuentran en varios tipos de agentes, como la movilidad y la necesidad de interacción con usuarios y el consiguiente aprendizaje de su comportamiento.

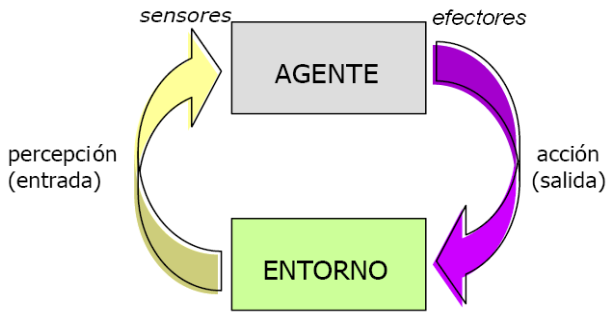


Ilustración 4: Relación del agente con el entorno

Una de las definiciones más aceptadas es la siguiente: "un agente es un sistema informático situado en un entorno y que es capaz de realizar acciones de forma autónoma para conseguir sus objetivos de diseño" (*Ilustración 4*). Según esto, los agentes están caracterizados por la autonomía, como ya se había comentado, pero no hace una distinción clara respecto a un sistema distribuido convencional.

Otras definiciones destacan los elementos característicos del comportamiento de un agente y otros requisitos de su arquitectura.

La confusión respecto al término “agente” viene del hecho de que cada investigador asocia con los agentes un juego de características algo diferente. De todas maneras, casi todas las definiciones contienen alguna combinación de características comunes:

- ***Capacidad de trabajar asincrónicamente y de manera autónoma***
  La característica esgrimida por la mayoría de los agentes presumiblemente inteligentes es que pueden funcionar con autonomía, y sin la intervención del ser humano (SHOHAM, 1993; MAES, 1990).

- ***Capacidad para cambiar su comportamiento según el conocimiento acumulado***
  La capacidad de "aprender" puede ser, generalmente, la segunda característica más asociada con los agentes inteligentes. Por ejemplo, la "ayuda adaptativa" se ha caracterizado como una característica central en arquitecturas de interfaz inteligente.
  Similarmente, un uso frecuente de los agentes está en prever las necesidades de información del usuario buscando qué bases de datos mantienen un "perfil de interés" basándose en preferencias pasadas de búsqueda. Cualquier cambio de interés del usuario deberá ser tenido en cuenta por el agente.

- ***Capacidad para tomar iniciativas***
  Un agente verdaderamente inteligente debe tener la capacidad para ejecutar tareas de acuerdo con su propia meta, separada de la del usuario. Los ambientes que condujeron a esta propuesta eran esos en los que el agente “sabe” más que el usuario sobre una materia y/o sobre las estrategias a utilizar para resolver el problema.

- ***Capacidad inferencial***
  Otra característica frecuentemente asociada a los agentes es la capacidad para realizar inferencias. Se define como la capacidad para ir más allá de las instrucciones concretas y específicas del usuario y resolver problemás utilizando alguna forma de abstracción simbólica (Shoham, 1993). En algunos casos, estas abstracciones estarán en forma de reglas (como las utilizadas en sistemás expertos). En otros casos, las inferencias pueden estar basadas en razonamientos probabilísticos.

- ***Conocimiento anterior de metas generales y métodos preferidos***
  Esta característica puede verse como una extensión de las características "aprendizaje" e "inferencia". Tiene que ver con el reconocimiento de que las soluciones a muchos problemás mundiales verdaderos requieren una comprensión de metas generales (Wilensky, 1983, Maes, 1990).
  Este nivel de comprensión es el requerido para las versiones más sofisticadas de agentes inteligentes. Un agente inteligente no solamente deber ser capaz de generar su propia meta prioritaria, sino que debe comprender también la meta del usuario para el que actúa como agente.

- ***Lenguaje natural***
  Para problemás simples, los lenguajes de scripting actuales pueden resultar suficientes para que el usuario especifique el método y el resultado deseado

al agente. Sin embargo, cuando los agentes tienen que resolver una variedad amplia de problemás cotidianos, la especificación de esos problemás llega a ser una tarea compleja. Debido a la complejidad y a causa de las preferencias naturales de un usuario, muchos investigadores asumen una interfaz de lenguaje natural (Shoham, 1993).

– ***Personalidad:***

Algunos investigadores han encontrado atractivo adjuntar características humanas a los agentes, como la personalidad. La suposición aquí parece ser que como el problema a resolver y los requerimientos interactivos de comunicación pueden llegar a ser muy grandes y complejos, únicamente un humano como entidad podría gestionar el problema. De forma similar a los humanos, la comunicación de algunos tipos de información va a estar estrechamente relacionada con la personalidad.

El concepto de agente inteligente, como podemos comprobar, va ser una complicada mezcla de características. A pesar del acuerdo sobre la sustancia del concepto y una lista final de características comunes de agente, permanece la confusión sobre el término porque el subconjunto necesario y suficiente de características no está acordado. El programa agente, para poder realizar sus actividades de forma autónoma, necesitará ser un experto en el dominio, tener conocimiento de métodos y estrategias para dividir el problema en subtareas, habilidades inferenciales y capacidades para adoptar medidas.

El que los agentes puedan tener distintos grados de parecido al ser humano contribuye a aumentar la confusión del término. Podemos imaginar agentes que trabajan bien pero que no están personificados. Sin embargo, como la delegación ha sido una actividad fundamentalmente interpersonal, la gente puede encontrar más fácil delegar a, y trabajar con, agentes que son, más o menos, como el ser humano.

Tipos de agentes

Los criterios de clasificación de agentes están inspirados en la observación de los agentes desde distintas perspectivas. Veamos los tipos de agentes desde cada una de ellas:

A) **Según sus características individuales:** Pueden distinguirse dos categorías: agentes reactivos y agentes cognitivos.

– Los ***agentes reactivos*** realizan tareas sencillas. Su comportamiento está basado en un ciclo de recepción de eventos externos/reacción. La reacción consiste en la ejecución de procedimiento o rutinas sencillas según el estado interno del agente. El comportamiento reactivo puede consistir en transitar de estado interno o ejecutar funciones internas o externas sobre el entorno.

– Los ***agentes cognitivos*** realizan tareas más complejas. Utilizan algún tipo de representación simbólica del conocimiento. Para efectuar las tareas deben llevar a cabo procesos de razonamiento y otros procesos cognitivos como la planificación o el aprendizaje.

B) **Según en el entorno en el que funcionan:** Consideramos entorno toda infraestructura computacional que rodea al agente. El entorno proporciona los medios necesarios para que el agente desarrolle su ciclo de vida, esto es que pueda ser creado, funcione y desaparezca. Desde este punto de vista podemos considerar dos tipos de agentes:

- Los agentes que ***requieren un entorno especial***, es decir, una plataforma software especifica para su correcto funcionamiento. Como ejemplo de ellos encontramos los agentes móviles cuyo entorno les proporciona todos los mecanismos para gestionar su ciclo de vida
- Los agentes que ***no requieren plataformás específicas***. Son creados por el sistema operativo y son válidos para cualquier aplicación.

C) **Según el modo de interacción: l**a interacción comprende la comunicación y el intercambio de información entre un agente y otras entidades. Se distinguen tres tipos de interacciones: agente-agente, agente-entorno, agente-usuario.

- Las comuniciaciones agente-agente se llevan a cabo mediante leguajes estandar de comunicación entre agentes, como ACL y KQML.
- Las comunicaciones agente-persona deben hacerse utilizando los medios adecuados para que las personas lo sepan interpretar.
- Los agentes especializados en la interacción con el usuario se han llamado agentes de interfaz. De esta misma forma, según la especialización del agente se distinguen agentes de bases de datos, agentes de mensajeria con HTML, agentes de generación de voz, etc.

D) **Según el modo de organización:** Siguiendo esta caracteristica, podemos distinguir dos categorías:

- ***Agentes individualistas:*** carecen de la capacidad de cooperación y realizan sus tareas solos, sin requerir la colaboración de otros agentes.
- ***Agentes cooperantes:*** pueden realizar tareas solos o colaborando con otros agentes.

E) **Según utilidad:** La perspectiva de la utilidad permite clasificar los agentes de acuerdo con la finalidad o el propósito con el que fueron construidos.

Sistemás multi-agente

El desarrollo de sistemás multi-agente es hoy en día uno de los temás de mayor discusión en el área de Agentes. Tanto la aplicación de resultados del área de Ingeniería de Software como la necesidad de generación de nuevos métodos y herramientas específicas para el desarrollo de agentes son temás que se intentan esclarecer.

Los sistema multi-agente o SMA tienen un amplio rango de aplicaciones. Uno de los campos principales lo constituyen los agentes software [SUB2000] [BIG2001] [MUR1998], los cuales se focalizan principalmente en la explotación cooperativa de recursos accesibles por internet. El otro campo de aplicaciones incluye la utilización de agentes físicos y ha dado lugar a la robótica cooperativa [KNA1998][MON2000]; campo en el cual los conceptos y las teorías de SMA encuentran un mayor reto de aplicación [STO2000]. Los SMA conjugan los modelos de la IA con los desarrollos de los sistemás concurrentes y distribuidos, tienen un gran potencial de aplicación para la solución de problemás de la vida práctica y por ende para el desarrollo de nuevos productos. Algunos de los campos de aplicación de los SMA incluyen: asistentes personales, supervisión hospitalaria, asistentes financieros, manipulación y filtrado de información aplicados a diferentes dominios, agentes bancarios, ventas y comercio electrónico, difusión de noticias y publicidad, realidad virtual y avatares, control de procesos y manufactura, telecomunicaciones, están entre otros [HUH1998] [MUR1998] [KNA1998].

Actualmente, la comunidad científica y las grandes empresas de la informática realizan trabajos en busca de normalizar los aspectos básicos de la arquitectura y formás de comunicación de los sistemás de agentes. Hasta el momento se destacan: KQML de ARPA en lo relacionado al intercambio de conocimiento, KIF en lo relativo a representación del conocimiento, ACL como lenguaje basado en el concepto de actos del lenguaje y FIPA en cuanto a la arquitectura y servicios de infraestructura [BIG2001] [MUR1998] [WEI1999]. Se han propuesto algunas plataformás y arquitecturas para el desarrollo de agentes [HUH1998] [KNA1998]. Recientemente, el estándar FIPA ha tenido gran desarrollo y en consecuencia han surgido plataformás que cumplen con este [FIP2002] [FIP2002a]. Existen otras plataformás disponibles que no cumplen con todas las especificaciones de FIPA son Agent Builder, Aglets, Voyager, CIAgent [BIG2001].

1.6 Estado del arte: Jade

La plataforma Jade

Jade es una herramienta de desarrollo de sistemás multi-agente. Está compuesta por un conjunto de paquetes en Java para la programación de agentes y está basado en el estándar FIPA (*Ilustración 5*).

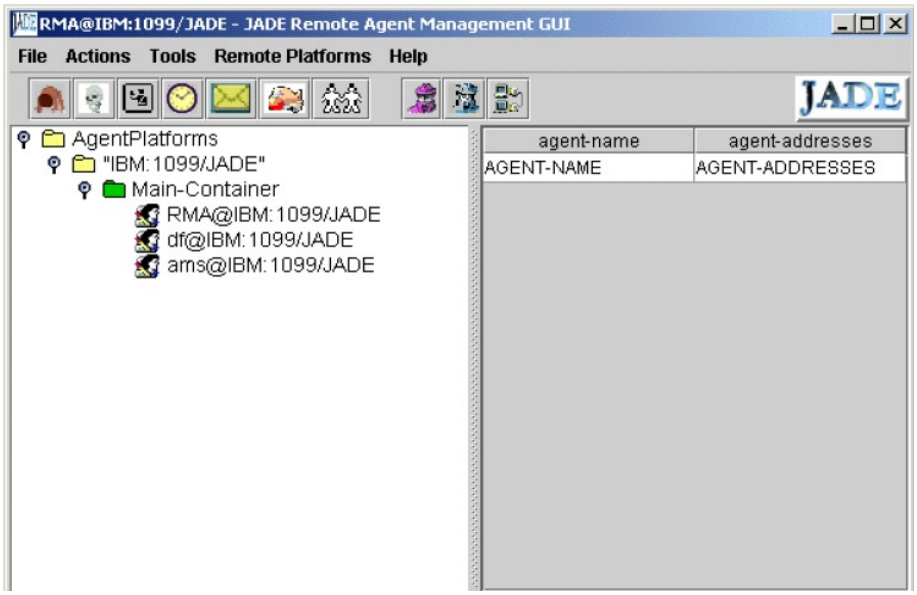


Ilustración 5: GUI de Jade

La plataforma incluye en sus librerías:

- Todo lo necesario para la creación básica de agentes.
  - La programación del comportamiento de los agentes mediante *behaviors*
- El manejo de información empleando ontologías
- La implementación de las comunicaciones entre agentes siguiendo el estándar ACL FIPA para en envío y recepción de mensajes.
- Clases útiles para la programación de protocolos basados o no en FIPA
- Agentes auxiliares:
  - RMA: Gestor de la plataforma.

- Dummy agent: Permite de forma sencilla interactuar con agentes
- Sniffer agent: Muestra las interacciones que se producen.
- DF, AMS: Es un interfaz del *Directory Facilitator*

Cabe señalar que cada agente Jade le es asignado un hilo interno Java. Además Jade puede ejecutarse en una o en varias JVM. Cada JVM es vista como un entorno donde los agentes pueden ejecutarse concurrentemente e intercambiarse mensajes.

Lanzar un agente desde el GUI de Jade

Para lanzar un agente desde la GUI sólo hay que ejecutar desde una consola del SO el siguiente comando:

```
 > export CLASSPATH=/ruta/jade/lib/jade.jar:/ruta/jade/lib/iiop.-
jar:/ruta/jade/lib/http.jar
 java jade.Boot -gui
```

El resultado tras la ejecución es la ventana de la ilustración 6. En ella se distinguen dos partes principales: plataforma y contenedores, donde se almacenarán los agentes, y las características de los agentes.

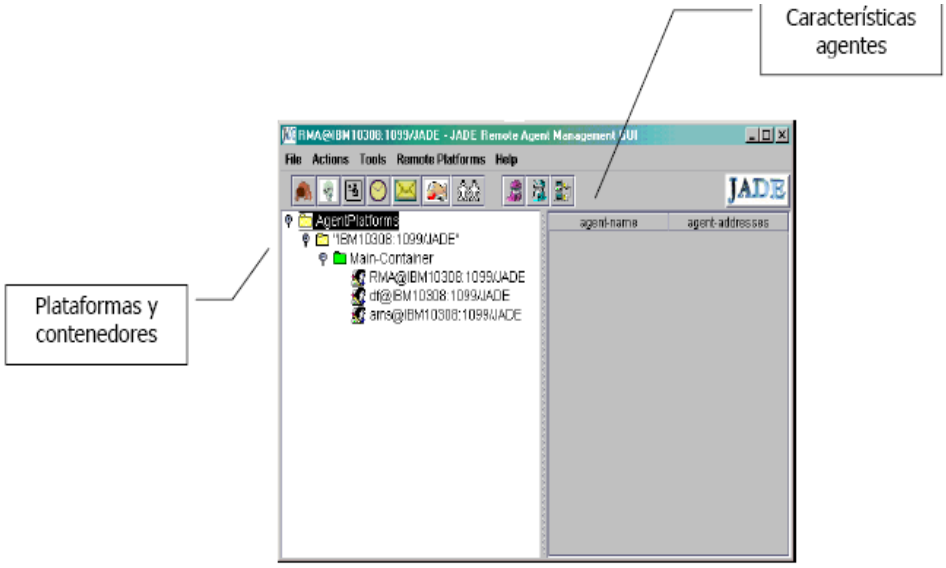


Ilustración 6: Partes de la GUI de Jade

A continuación, para crear un nuevo agente tenemos que seleccionar el botón *New Agent* estando posicionados sobre el contenedor deseado:

Como ya sabemos de la norma FIPA, cada agente debe tener un nombre, que forma parte de la descripción del agente que, a su vez, contiene más datos relativos a cómo mandarse mensajes(*Figura 7*).

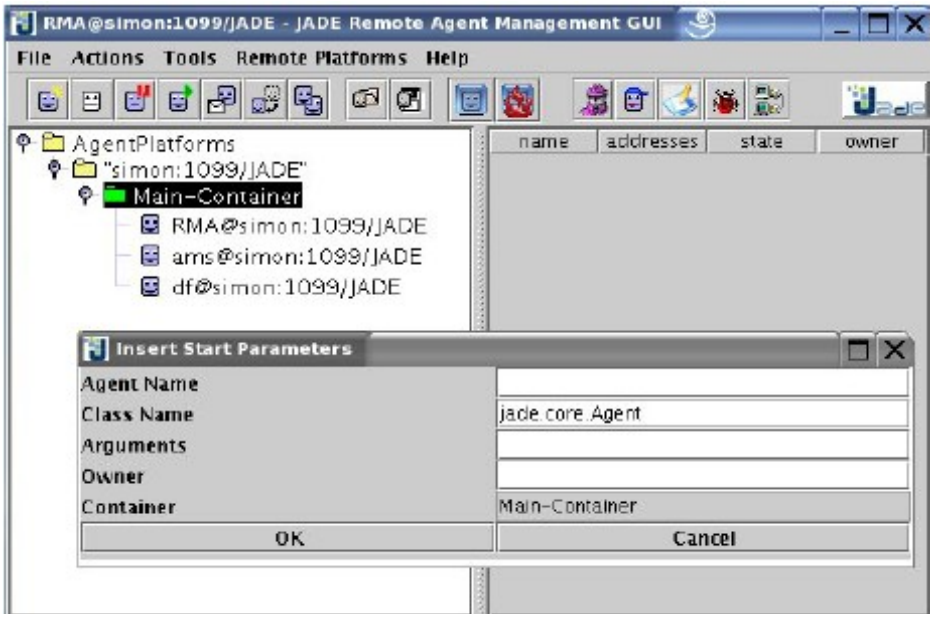


Ilustración 7: Creación de un nuevo agente

En JADE, la descripción del agente, el AID (Agent Identifier) tiene su correspondiente clase java:

```
jade.core.AID
```

a partir de la cual se pueden manejar los nombres y la demás información con facilidad. La clase Agent incorpora el método getAID() que permite recuperar el nombre del agente. El nombre del agente, un identificador único global, va a tener la estructura

```
<nickname>@nombre-plataforma>
```

Por lo tanto, el agente Agent1, localizado en la plataforma P1 va a tener el nombre Agent1@P1. Hablando de las direcciones que forman parte del AID, solamente tendremos que preocuparnos de ellas cuando el agente con el que queramos contactar esté en otra plataforma.

Instalación de Jade

La versión 3.4.1 de JADE (Java Agents Development Environment) puede descargarse en el siguiente enlace:

```
http://jade.tilab.com
```

aunque para poder hacerlo es necesario rellenar un formulario y aceptar las condiciones de uso. Una vez registrado y descargada la distribución tendremos los siguientes archivos comprimidos:

– **jadeAll.zip***(8.9MB):* un zip que, a su vez, alberga los siguientes cuatro ficheros que aparecen a continuación.

- **jadeBin.zip***(1.6MB)*: este archivo contiene JADE ya compilado y listo para ser utilizado.
- **jadeDoc.zip***(5.5MB)*: este archivo contiene toda la documentación JADE incluida la Guia del Administrador(Administrator's Guide) y la Guia del Programador(Programmer's Guide)
- **jadeSrc.zip***(1.8MB)*: este archivo contiene todo el código fuente de JADE.
- **jadeExamples.zip***(270KB)*: este archivo contiene código fuente de ejemplos y demostraciones de programás JADE. Todos los ejemplos y demostraciones deben ser compiladas.

Lo _unico que necesita JADE para funcionar es una versión correcta del Java Run Time Environment, la 1.5. Una vez hemos obtenido en nuestra cuenta el fichero *jade-All.zip*, creamos un directorio aparte para jade, colocamos allí la distribución y, desde una consola, procedemos como sigue:

```
>unzip jadeAll.zip
>unzip jadeBin.zip
>unzip jadeDoc.zip
>unzip jadeSrc.zip
>unzip jadeExamples.zip
```

Como podrá comprobarse se ha creado un directorio jade y bajo este directorio un directorio lib que es en donde están todos los jars necesarios para ejecutar la plataforma.

El agente básico en JADE

La manera más sencilla e inmediata de construir un agente JADE es heredando de una clase básica Java denominada jade.core.Agent. Esta clase incluye los métodos

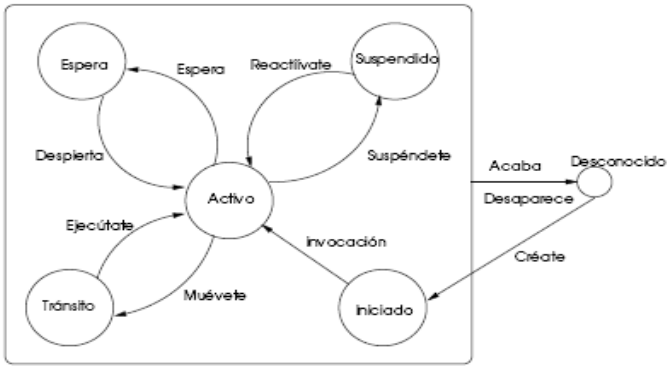
```
public void setup()

public void action()
```

que para el programador son los más importantes inicialmente. El primero ha de realizar todas las tareas necesarias para que el agente pueda comenzar su ejecución con todo lo necesario. El segundo es realmente el cuerpo de ejecución del agente. Hasta aqu, podríamos pensar que el modelo de agente que existe detrás de la plataforma JADE es simplista, pero no es as. El comportamiento del agente esta determinado por

el AFD que aparece en la ilustración 8 y que aparece también en el documento FIPA [10].

En la figura podemos ver que cuando el agente se crea pasa a un estado de iniciado. Posteriormente, cuando la plataforma invoca su método action() pasa a estar activo y de ahí puede pasar a suspendido, esperando un determinado evento o en transito, moviéndose de un entorno de ejecución JADE a otro.



AFD que define el comportamiento de un agente JADE

Ilustración 8: Comportamiento de un agente Jade

Un agente puede estar en disferentes estados tal y como se requiere en las especificaciones FIPA. Los estados son los siguientes:

- **Iniciado**: el agente está construido, pero no esta registrado todavía con AMS.
- **Activo**: el agente está registrado con AMS, tiene un nombre y una dirección y puede acceder a todas las características de Jade.
- **Suspendido**: el agente está actualmente parado. El hilo interno esta suspendido y ninguna de los comportamiento del agente están siendo ejecutados.
- **Espera**: el agente está bloqueado a la espera de algo. Internamente el hilo está dormido en el monitor de Java y será despertado cuando alguna condición se cumpla.
- **Transito**: un agente móvil entra en este estado mientras está migrando a otra nueva localización.
- **Eliminado:** el agente ha muerto definitivamente. El hilo interno del agente ha finalizado su ejecución y el agente no será registrado más con AMS.

Un ejemplo de código para crear un agente mínimo en JADE es el que podemos ver a continuación:

```
import jade.core.*;
public class HelloWorldAgent extends Agent{
public HelloWorldAgent(){}
public void setup(){}
public void action(){
```

```
System.out.println(``Hello World'');}
}
```

Para finalizar, el siguiente organigrama (*Ilsutración 9*)representa el ciclo de vida de un agente Jade:

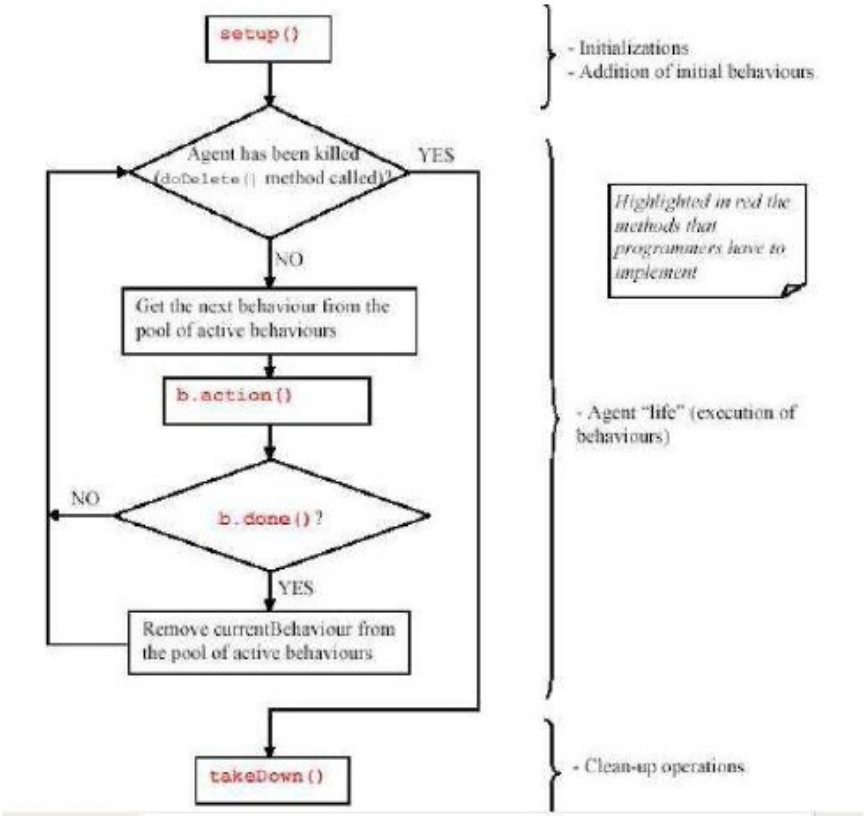


Ilustración 9: ciclo de vida de un agente Jade

Comunicación entre agentes: Estándares

En un sistema multiagente, los agentes han de comunicarse unos con otros. Se hace, pues, necesaria la intervención de un lenguaje de Comunicación entre Agentes cuya finalidad sea hacer transparente el intercambio de datos entre agentes distribuidos. Un requisito básico es la estandarización de los mensajes utilizados durante la comunicación. Los dos lenguajes de comunicación más utilizados actualmente son: KQML (Knowledge Query and Manipulation Languaje) y FIPA-ACL (Foundation for Intelligent Physical Agents – Agent Comunication Languaje):

– El estándar FIPA

FIPA ("Foundation for Intelligent Physical Agents") [Url_FIPA] Fue creada en 1996 para producir software estándar para sistemás basados en agentes, heterogéneos e interactuando entre ellos. FIPA ha elaborado una serie de especificaciones que pueden

ser usadas para el desarrollo de sistemás basados en agentes.

El objetivo de las especificaciones es facilitar la interacción entre agentes y sistemás de agentes a través de diferentes plataformás de agentes de diferentes fabricantes.

Las distintas especificaciones FIPA van progresando a traves de un ciclo de vida, desde un estado preliminar, experimental hasta el estado estándar. Cualquier especificacion puede ser reprobada, previo estado obsoleto, cuando se considera innecesaria.

Las especificaciones FIPA se pueden organizar en cinco materias, según el dominio de agentes al que va dirigido:

- Aplicaciones
- Arquitecturas Abstractas
- Comunicación entre Agentes
- Gestión de Agentes
- Transporte de Mensajes de los Agentes

– El estándar KQML

El grupo de Knowledge Sharing del DARPA (Defence Advance Research Projects Agency) obtuvo como principal resultado un lenguaje del que FIPA ACL tomó su filosofía, y que precede a éste, el KQML(Knowledge Query and Manipulation Language). Fue concebido tanto como un leguaje de formato de mensajes como un protocolo de tratamiento de mensajes para soportar compartición de conocimiento entre agentes.

KQML usa un conjunto distinto, aunque parecido, de actos comunicativos o performativas. Concretamente, KQML define 34 performativas, aunque deja abierta la posibilidad de definir otras adicionales, clasificadas en tres categorias : de discurso, de intervencion y mecánica de conversación y de comunicación en red y facilitación.

Conclusiones

Se ha optado por las especificaciones FIPA, debido principalmente a dos razones. Por un lado, dichas especificaciones se han convertido en un estándar reconocido en el desarrollo de sistemás multi-agentes. Por otro lado, dichas especificaciones están referidas no únicamente al lenguaje de comunicación entre los agentes (como es el caso de otros estándares como el KQML) sino que abarcan aspectos como la gestión de los agentes en una plataforma o el flujo lógico en una conversación entre agentes.

TEMA 2

Capacidades de Seagull

2.1 Descripción del sistema completo:

Un sistema software suele describirse usando diagramás en los que aparecen cajas y flechas conectando dichas cajas. Estas cajas suelen representar componentes estructurales del sistema y las flechas suelen simbolizar relaciones y acciones entre ellas. Seagull es un sistema bastante complejo, por lo que esta colección de módulos y flechas representaría un jeroglífico no muy útil de no ser dividido en subsistemás manejables. Exactamente este mismo problema, que surge a la hora de explicar un proyecto, surge a la hora de ponerse a diseñarlo. Por esta razón la estructura del sistema fue dividida en varias secciones, cada una encargada de resolver un conjunto de requisitos relacionados. Para realizar una explicación genérica del sistema Seagull, expondremos la misma división en subsistemás, con las relaciones entre sus requisitos, que se estableció durante su etapa de diseño.

Comenzaremos primero describiendo los requisitos que debe cumplir el sistema. Después pasaremos a una explicación de Seagull desde un punto de vista estrutural. Por último pasaremos a otra descripción del mismo sistema pero desde el punto de vista de su funcionamiento y comportamiento.

Requisitos del sistema.

Requisitos del sistema

Los mencionados conjuntos de requisitos principales del proyecto son tres:

A) Agentes inteligentes: Implementar un conjunto de agentes inteligentes que puedan ser instalados en aviones robóticos. Cada agente ha de manejar un avión y debe ser capaz de comunicarse con el resto de agentes-aviones para volar en formación coordinadamente. El conjunto de agentes-aviones han de ser capaces de organizar una búsqueda sobre un territorio de forma eficiente, así como ser capaces de actuar ante imprevistos de forma dinámica. El conjunto de aviones y su organización ha de ser completamente autónomo en su tarea de búsqueda y rescate.

B) Simulación virtual: Ya que los agentes no van a ser probados en aviones reales, debe ofrecerse un entorno tridimensional completamente simulado. El entorno debe simular el terreno, la dinámica de los aviones y su vuelo, las posibles interferencias y los eventos de ocurrencia aleatoria ante los que deben actuar los aviones.

C) Interfaz de observación y depuración: Lo anterior (los agentes y el entorno simulado en el que se mueven) debe ser presentado al usuario a través de una interfaz. Dicha interfaz debe ofrecer toda la información relevante acerca de los agentes y del entorno tridimensional, así como recursos básicos para poder observar detalladamente y depurar el funcionamiento interno de los agentes.

Descripción estructural del sistema.

Estructura general del sistema

La división de la arquitectura de Seagull en tres componentes obedece a este agrupamiento de los requisitos en tres conjuntos principales, encargándose cada componente estructural de uno de ellos(*Ilustración 10*). Estos componente son los siguientes:

A) Paquetes 'agent': Son los encargados de implementar los agentes inteligentes. Todos estos paquetes están pensados para poder ser instalados sin modificación en aviones robóticos. Se usa el framework JADE para esta sección.

B) Paquetes 'environment_simulator': Implementan el simulador del mundo tridimensional en el que se mueven los agentes-aviones. Esto se realiza usando Java3D.

C) Paquetes 'system_interface': Implementan la interfaz del sistema. Se usan las librerías de java-swing.

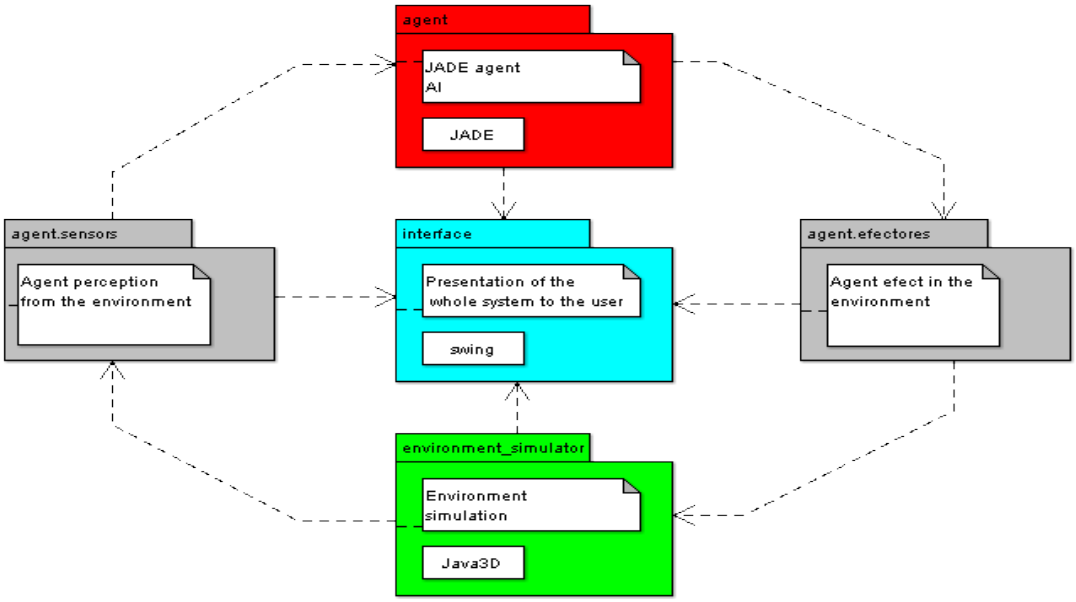


Ilustración 10: Arquitectura general del sistema: El sistema puede dividirse en tres conjuntos de paquetes, cada uno especializado en una función. En el gráfico se muestran estos tres conjuntos (rojo, azul y verde) más dos paquetes del conjunto 'agent' (grises), usados en la comunicación entre 'agent' y 'environmento_simulator'. Las flechas discontinuas muestran el flujo de la información ente paquetes.

Para la comunicación entre la inteligencia artificial de cada agente (paquete 'agent') y el simulador virtual del entorno (paquete 'environment_simulator'), se usan dos subpaquetes de 'agent' cuya mención es importante:

A1. 'agent.sensors': Simula los sensores hardware del avión robótico. El paquete 'environment_simulator' genera en él todos los cambios que van produciéndose en el entorno simulado, para que el paquete 'agent' pueda detectar dichos cambios leyendo este subpaquete.

A2. 'agent.efectors': Simula los efectores del avión robótico, por lo que realiza la comunicación en sentido contrario. El agente inteligente modifica determinadas variables en él que luego son leídas por el simulador para saber que quiere hacer el agente y simular lo que ocurre.

Esta división de la estructura en tres conjuntos de paquetes puede representarse de la siguiente manera(*Ilustración 11*). En el diagrama aparecen los subpaquetes 'sensors' y 'efectors' fuera de 'agent' por motivos de comprensión.

Descripción funcional del sistema (modelo vista-controlador)

La implementación del sistema ha sido distribuida en distintos paquetes con el fin de albergar una estructura coherente y agrupar aquellas clases relacionadas entre si en los mismos paquetes. Se ha seguido un modelo vista controlador:

- **Modelo**: En nuestro caso el modelo corresponde a las clases del paquete agent donde se especifica la representación de la información de los agentes con la cual el sistema opera.
- **Vista**: El formato empleado para interactuar con el usuario es la interfaz implementada en el paquete system_interface.
- **Controlador**: El paquete environment_simulation cumple la labor de controlador. Este responde a los eventos, usualmente acciones del usuario e invoca cambios en el modelo y en la vista.

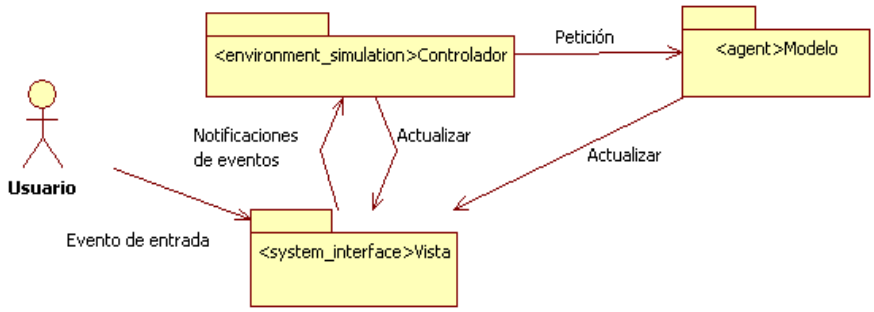


Ilustración 11: Modelo vista controlador

Contenido de los paquetes del sistema.

Vamos a entrar más detalladamente en la funcionalidad de cada uno de los paquetes del sistema. Esencialmente el código de la aplicación tiene la siguiente división:

A) ***Paquete agent:*** En este paquete han sido implementadas las clases relacionadas

con los agentes Jade. A su vez, se encuentra dividido en diferentes subpaquetes cuya finalidad es la de distribuir la funcionalidad de las clases. Así distinguimos los siguientes subpaquetes:

- *agent*: Paquete principal en lo referente a los agentes. En el se implementa el pilar básico de una aplicación Jade, el agente básico. Además se definen aquellas clases que servirán de patrón a la hora de definir los contoladores, las variables estados y los *beliefs states* de los agentes *boss* y *searcher*.
- *agent.boss*: En él se almacenan las clases referentes al agente boss.
- *agent.searcher*: De manera análoga al anterior, implementa las clases relacionadas con los agentes *searcher*.
- *agent.sensors*: Cada agente dispone de objetos que representan sensores. En este paquete son implementados.
- *agent.efectors*: Del mismo modo, este paquete almacena las clases referentes a efectores, los cuales serán instanciados por los agentes.
- *agent.task*:Paquete de gran trascendencia. Aquí están implementadas las tareas que serán permutadas por el controlador de cada agente definiendo de esta forma el comportamiento del agente.
- *agent.task.commonFSMStates*: Aquí únicamente se define un estado común para algunas tareas. Así se ha evitado la repetición de código de manera innecesaria, además de una mayor claridad en el mismo.

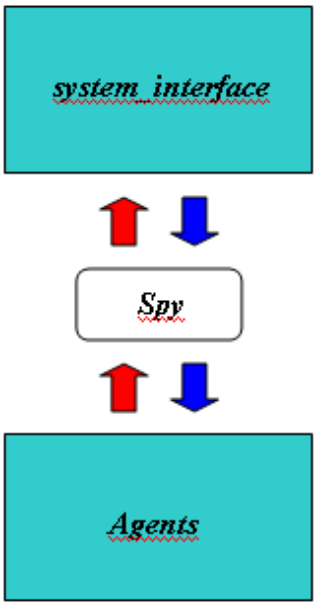


Ilustración 12: Relación de la interfaz con los agentes mediante un objeto espía

B) ***Paquete system_interface:*** Incluye aquellas clases relacionadas con la GUI del sistema. El objetivo del proyecto ha sido desde un primer momento que el sistema se asemejara lo más posible al mundo real manteniendo la independencia de los agentes. Debido a que la interfaz representa el entorno

donde se desenvuelven los agentes, no sería correcto permitir que la interfaz pudiera contenerlos como simples instancias y así poder acceder a sus atributos ya que esto les restaría independencia. La interfaz debería tener acceso a todo aquello que en el mundo real estaría al alcance del entorno (por ejemplo: altura de vuelo o posición) pero que a la hora de la simulación no era posible. Así surgió la idea del espía para el interfaz representado en mediante la clase SpyForInterface de este paquete. Con ello se solventaron todos los problemás de independencia. Como se aprecia en la Ilustración 12, el espía actúa de intermediario entre la interfaz y los agentes. El resto de clases de este paquete representan las distintas ventanas y paneles que conforman la interfaz del sistema.

C) ***Paquete environment_simulation:*** Es el encargado de representar y controlar el entorno 3D de la aplicación. El paquete está subordinado a la clase *simulationWindow* perteneciente a *system_interface* ya que todas sus acciones se aplican sobre o son debidas a la citada clase:cualquier acción que sea seleccionable por el usuario, ya sea tomar foto, parar la simulación o cambiar la cámara de posicion es necesario que pase por *simulationWindow* y de esta al environment_simulation. Todo esto se corresponde con el comportamiento habitual del controlador de un sistema que sigue un diseño modelo vista controlador. De esta forma podemos implementar mecanismos de control por parte del usuario, además de darnos una escalabilidad mayor al programa.

D) ***Paquete util***: En este paquete se definen todas aquellas clases auxiliares que nos ayudan a la hora de implementar el sistema. A continuación se hace una breve descripción sobre todos ellos:

- Clase *Constants*: define algunas constantes útiles.
- Clase *Coordinates*: los objetos de esta clase implementan las coordenadas de los agentes.
- Clase *Environment*: esta clase representa el entorno donde se desenvuelven los agentes. Lo más destacado es el buffer interno donde se almacenan los mensajes de radio enviados por los agentes. De allí, a su vez, los agentes tomarán los mensajes del entorno.
- Clase *MathExtended*: estensión de la clase predefinida Math con el fin de cumplir algunos objetivos necesarios.
- Clase *Photograph*: La clase representa una fotografía tomada por el sensor PhotographicCamera.
- *Clase RadioMessage*: Implementa un mensaje de radio.
- Route: en esta clase se definen los puntos que formarán una ruta.
- *Clase SearchArea*: extiende de la clase *polygon* y es utilizada para representar el área en el que los agentes realizarán la búsqueda.

2.2 El agente: Una nueva arquitectura.

Para cumplir los requisitos del paquete "agent" se ha diseñado una arquitectura de agente nueva procedente del diseño TouringMachine [FER1996]. Esta arquitectura podría clasificarse como híbrida, pues contiene capacidades reactivas al mismo tiempo que usa elementos propios de los sistemás BDI. Aparte de las posibilidades típicas de un agente, nuestro diseño también cuenta con paralelismo, pues realiza varias tareas al mismo tiempo de forma concurrente. En la presente memoria nos referiremos a esta arquitectura como ParallelMachine.

Arquitectura TouringMachine

La arquitectura TouringMachine (*Ilustración 2*)estructura el agente en varias capas, cada una encargada de resolver problemás a distinto nivel de abstracción. De mayor nivel de abstracción a menos a estas se les llama M, P y R, dando la ultima al sistema características reactivas y convirtiendo al agente, por lo tanto, en híbrido. Estas capas no funcionan en paralelo, si no que en cada momento solo una tiene el control sobre el agente, encargándose el controlador de decidir que capa obtiene dicho control en cada momento. Dicho controlador es un sencillo motor de reglas que ejecuta dichas reglas con la información que obtiene de la entrada y la salida de datos de cada una de las capas. Por último, para comunicarse con el entorno, la arquitectura cuenta con otros dos módulos, uno para percibir dicho entorno y otro para actuar sobre él.

Arquitectura ParallelMachine

Nuestro diseño se ha basado en esta idea de Ferguson de varias capas manejadas por un controlador y con acceso a dos sub-sistemás de percepción y de acción. Dos importantes diferencias son las que hacen de nuestro sistema una ampliación de este clásico diseño TouringMachine. La primera es que en nuestra arquitectura en lugar de las capas M, P y R tenemos lo que hemos llamado *Tasks*, que pueden funcionar en paralelo y que no están organizadas por nivel de abstracción. La segunda diferencia es que contamos con una representación común del mundo exterior usada por las tareas (*Tasks*) y el controlador. Dicha representación se almacena en dos niveles de abstracción, uno más cercano al mundo exterior (llamado, como no, *beliefs*) y otro más cercano al controlador (llamado *variables*).

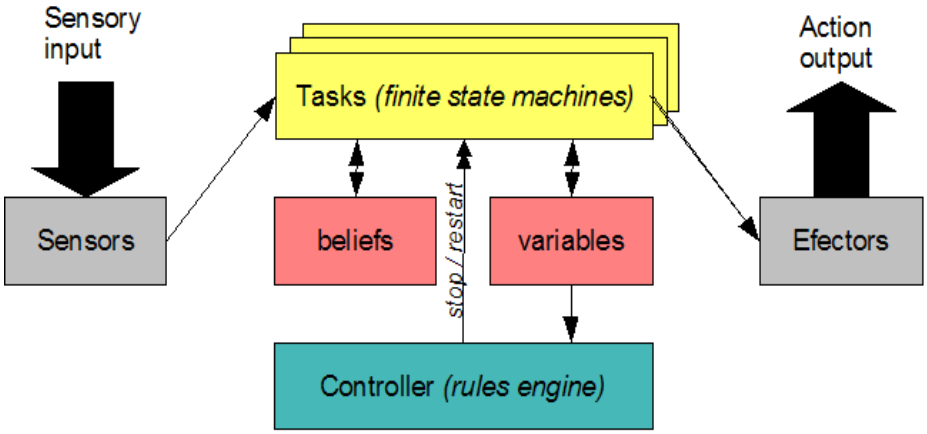


Ilustración 13: Arquitectura ParallelMachine. El agente es controlado por un conjunto de tareas (amarillo) funcionando en paralelo. Que tareas se activan y cuales se desactivan es decidido por el controlador (azul). El sistema almacena una doble representación del mundo exterior (rosa).Las flechas sencillas representan el flujo de la información, la flecha doble representa la capacidad del controlador para activar y desactivar tareas.

Recapitulando, en la arquitectura ParallelMachine el agente es controlador por un conjunto de tareas que funcionan en paralelo. Cada tarea es una máquina de estados que realiza un conjunto de operaciones en cada estado y pasa de un estado a otro según los que lea en *variables*, *beliefs* y *sensors*. Cada tarea tiene implícito uno o varios objetivos, siendo diseñada la mencionada máquina de estados para conseguir dicho objetivo en un amplio espectro de casos e imprevistos. No todas las tareas están funcionando al mismo tiempo, si no que solamente están activas las tareas relacionadas con los objetivos presentes del agente. Las tareas no tienen definido un nivel de abstracción determinado, pudiendo combinar acciones de gran abstracción con operaciones reactivas para conseguir su objetivo. Recordemos que esta arquitectura no tiene una representación explícita de los objetivos y las intenciones, pero sin embargo aquí vemos que si que existe una representación implícita de estos, pues cada tarea activa significa determinados objetivos pendientes. Para resolver conflictos entre tareas a la hora de usar los efectores, cada una tiene asociada una prioridad, tomando el control del efector la de mayor prioridad. Qué tareas son activadas, cuáles permanecen paradas y cuales son interrumpidas es decidido por el controlador. El controlador es el que define el comportamiento del agente a más alto nivel de programación (que no de abstracción), pues maneja tareas enteras, mientras que las tareas maneja operaciones básicas *variables*, *beliefs* y *efectors*. La representación del mundo exterior se realiza en dos módulos separados: *beliefs* y *variables*. La diferencia raíz entre ambas es que *variables* puede disparar tareas, mientras que *beliefs* no (aclaramos que no es el módulo *variables* el que dispara tareas, si no el controlador tras leer el módulo *variables*). Esto da lugar a la diferencia "macroscópica" más evidente entre ambos: que *beliefs* mantiene una representación del mundo más cercana a éste, mientras que *variables* almacena una representación "precodificada" más cercana al controlador donde, entre otras cosas, abundan los tipos boolean. Esta división en dos se realiza para aumentar el rendimiento del motor de reglas y para facilitar la depuración del sistema al aislar la fuente de disparo de tareas. Por ultimo están los módulos *efectors* y *sensors*, que simulan el comportamiento de los sensores y efectores del robot avión en el que iría instalado el agente inteligente.

Es fácil darse cuenta que la programación del agente puede dividirse en dos niveles: el nivel del controlador y el nivel de las tareas. Con programar a nivel de controlador nos referimos a decidir las reglas del controlador y, por lo tanto, como este maneja a las taréas. Con nivel de tareas nos referimos a implementar las máquinas de estado que son las tareas. La gran ventaja de este método y esta división en dos niveles es que puedes trabajar en un nivel con gran independencia de como está hecho internamente el otro. Así pues, cuando estás construyendo la máquina de estados de una tarea que tiene que cumplir un objetivo, tan sólo tienes que tener en cuenta como se puede conseguir el propio objetivo y como está almacenado el conocimiento en *variables* y *beliefs*. No tienes que fijarte directamente en como va a usarla el controlador, tan solo tienes que definir bien el conjunto de situaciones para las que está diseñada tu máquina de estados. Por otro lado, cuando programás las reglas del controlador, no tienes que preocuparte por como son internamente las tareas, tan solo debes comprobar que la tarea soporta el conjunto de situaciones a las que se enfrentará tal y como la quieres usar. El programador de tareas debe informar al programador del controlador del conjunto de situaciones que soporta la tarea, por lo que se recomienda que este primero describa dicho conjunto en su javadoc.

Implementación.

Implementación ParallelMachine

Hasta ahora la descripción de la arquitectura PerallelMachine se ha centrado en su funcionalidad, sin detallar su implementación por tratarse de una descripción general. A continuación comentaremos como se ha llevado a cabo dicha implementación.

Para construir esta arquitectura se ha usado el framework JADE que, aunque está principalmente pensado para agentes Web, contiene la funcionalidad necesaria para hacer el sistema más seguro y algo más sencillo de realizar. El diagrama UML de clases siguiente muestra la descripción general de los paquetes '*agent*'.

Un agente JADE es, en el caso más básico, una clase que extienda a jade.core.Agent. Como funciona dicho agente se determina añadiéndole un jade.core.Behaviour o alguna extensión de dicha clase. La familia de clases Behaviours está dividida en dos tipos: sencillos y compuestos. Los sencillos son los que desencadenan un comportamiento básico en el agente: lineal, cíclico... . Los compuestos son comportamientos que contienen sub-comportamientos (un jade.core.Behaviour que contiene a su vez más jade.core.Behaviours) que pueden ser a su vez compuestos: sub-comportamientos ejecutándose en paralelo, sub-comportamientos encadenados en forma de máquina de estados... . Este segundo tipo (los compuestos) puede mostrar comportamientos bastante complejos sin necesidad de escribir demásiado código. Nosotros hemos usado uno de estos comportamientos compuestos para nuestros agentes, más concretamente se trata de 3 comportamientos anidados, dos compuestos y el último sencillo.

La clase AgentPlane, que es la que extiende a jade.core.Agent, la usamos para crear e inicializar el resto de clases del agente inteligente. La clase ExtendedParallelBehaviour es el comportamiento compuesto de nuestro agente que, al extender a jade.core.behaviour.ParallelBehaviour, desarrolla un comportamiento paralelo: ejecuta de forma parela todos los jade.core.Behaviours que le sean añadidos. De esta última tarea se encarga la clase Controller, que va añadiéndole y quitándole

sub-comportamientos según le indique el motor de reglas que posee. Estos sub-comportamientos son extensiones de la clase Task, la cual es un jade.core.behaviour.FSMBehaviour, es decir, un comportamiento compuesto que funciona como una máquina de estados. Cada estado de esta máquina es un comportamiento sencillo jade.core.behaviour.OneShotBehaviour, que ejecuta un segmento de código java y termina. VariablesState y BeliefsState implementan los dos módulos de representación del mundo exterior. Por último las clases Sensors y Efectors implementan los módulos de comunicación con el entorno. Todas las tareas son las subclases de la clase Task, e instancian las anteriores cuatro clases para poder comunicarse con ellas. En el diagrama de la ilustración 14 se ha representado la tarea TaskSentinel a parte de las otras debido a su importancia. Esta tarea, que siempre está activa, se encarga de mantener a VariablesState y a BeliefsState actualizados a lo que la clase Sensors lee del mudo exterior.

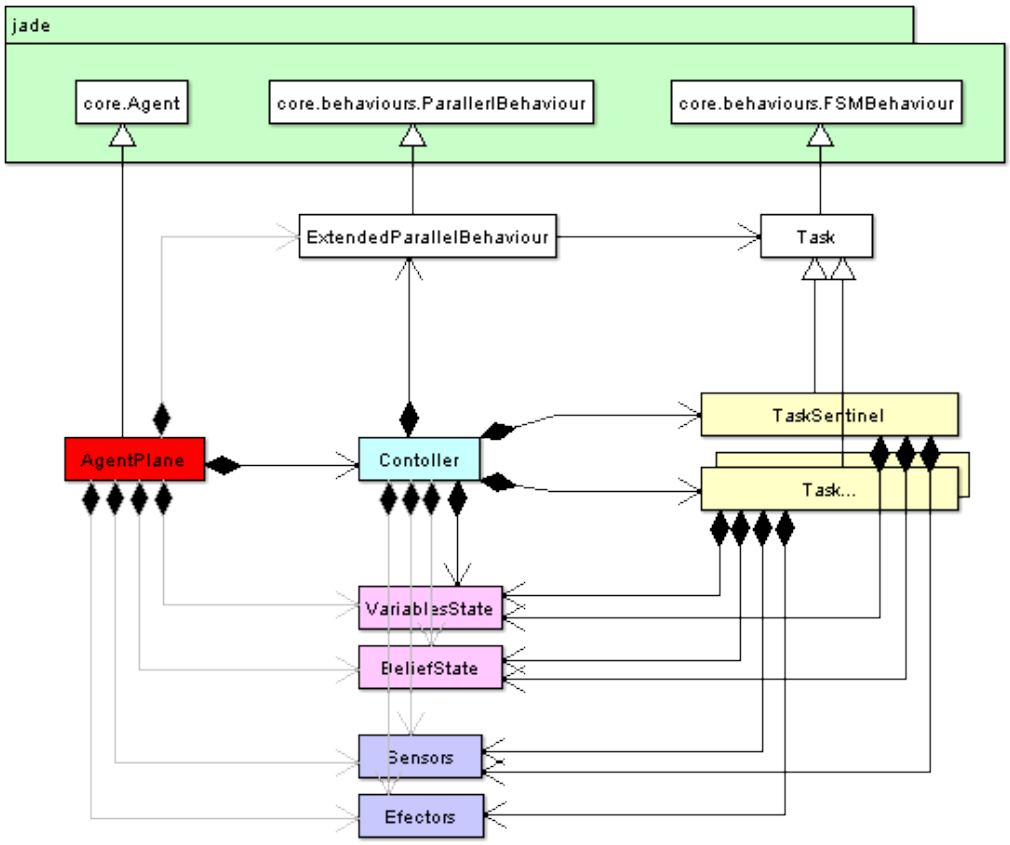


Ilustración 14: Implementación de la arquitectura ParallelMachine usando JADE. Las flechas en gris significan que dicho objeto solo es manejado por la clase de que es instancia para ser pasado a otras clases que si lo usan.

El controlador

El controlador conforma una de las partes más transcendentes del agente. Se trata de un sistema basado en reglas cuya misión es la de permutar las tareas del agente La configuración de estas reglas permite que coexistan varias tareas activas a la vez, esto es, la concurrencia de tareas. Como se describirá en la sección siguiente, las tareas han sido diseñadas para ser desempeñadas de manera exclusiva o por el agente *boss* o por los agentes *searchers*. Al existir distintas tareas según el papel desempeñado por el agente, han sido implementados dos controladores, uno para los agentes *searchers: ControllerSearcher* y otro para el agente *boss ControllerBoss.*

En cuanto a la implementación, ambos controladores heredan de la clase *Controller* que se localiza dentro del paquete *agent*. Dentro de el se instancia a los *sensores* y efectores. Los primeros le permiten verificar el cumplimiento de las reglas y por tanto, la activación y desactivación de las tareas. Los segundos son requeridos por algunas tareas para llevar a cabo su misión, por ejemplo, de nada serviría la tarea de envío de *checkpoint* si no pudiera acceder a la radio para efectuar el envío. El único método de la clase, *shootRules*(), es el encargado de verificar las reglas para permutar las tareas activas. Dentro de este método, el orden de comprobación de las reglas no es aleatorio sino basado en la prioridad de las mismás. Así pues, la determinación de la ruta mediante la tarea *TaskGenerateRoute* es prioritaria respecto a la tarea *TaskFlyToObjective*, ya que no se puede volar a un objetivo sin haberse definido previamente.

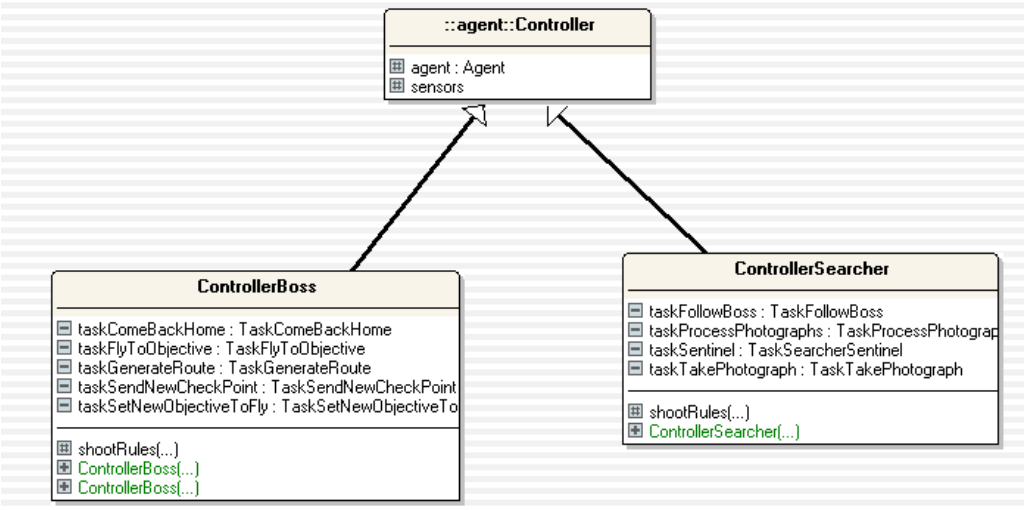


Ilustración 15: relación de herencia entre los controladores

Las tareas.

Fundamento de las tareas.

Como se explicó antes, las tareas son máquinas de estados implementadas usando la funcionalidad ofrecida por JADE. Mostrado en un diagrama de clases, su relación directa con otras clases del sistema quedaría como aparece en la Ilustración 16.

Las tareas son las unidades mínimás de procesamiento del agente, es decir, el agente hace todos los cálculos necesarios para desempeñar su labor a través de tareas. Si el agente necesita ir de un punto a otro, entonces lanzará una tarea que realizará todos los calculos y hará todas las comprobaciones necesarias para cumplir dicha

misión. Si el agente debe evitar chocarse con obstáculos cuando está volando, tendrá una tarea específica para evitar obstáculos una vez son detectados, y otra que estará siempre alerta para detectar si hay un obstáculo en el camino.

Entrada y salida de las tareas.

Este funcionamiento podría compararse con el de un microprocesador CISC, donde cada instrucción realiza un cálculo complejo usando los distintos elementos del hardware. Si las instrucciones de un microprocesador deben tener un sitio de donde coger los datos iniciales y otro (o el mismo) donde dejar los resultados de sus cálculos, las tareas también necesitan algún sitio de donde leer el estado actual del agente y donde dejar los resultados de su trabajo. Este lugar son las variables (VariablesState) y a las creencias (BeliefsState) del agente. Las tareas almacenan los resultados de su ejecución en estas clases y, a su vez, recogen los resultados de otras tareas desde estas mismás clases.

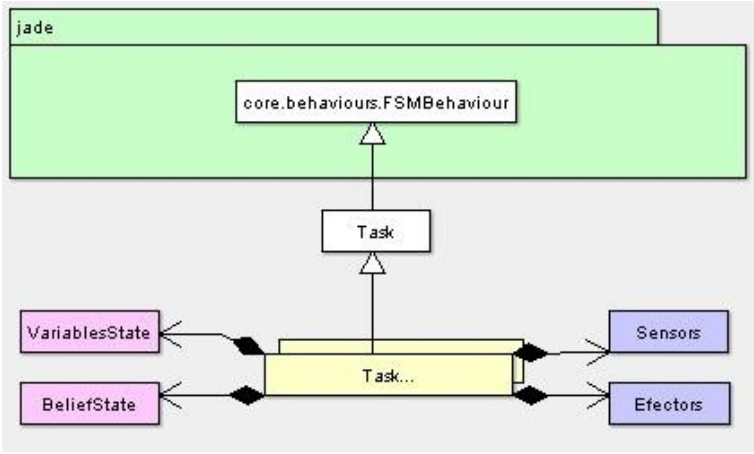


Ilustración 16: Relación de las tareas con el resto de clases del sistema.

Las tareas no solo necesitan los datos procedentes de otras tareas que puedan encontrar en VariablesState y en BeliefsState, si no que en muchas ocasiones necesitan datos de primera mano procedentes del exterior todavía no procesados por nadie. Estos datos son los recogidos por los sensores en tiempo real, por lo que las tareas también tienen acceso a la clase Sensors. Por lo tanto los sensores del agente no funcionan por interrupción, si no que tienen almacenado el dato de la ultima lectura que realizaron, siendo este dato leído por las tareas cuando lo necesiten. Para emular el funcionamiento por interrupción en algunos casos en los que sería deseable (como para detectar eventos urgentes como la proximidad de un obstaculo con el que se va a colisionar), existe la tarea Centinela (clase TaskSentinel). Esta tarea lee continuamente los sensores en busca de algún evento que deba ser tratado urgentemente, si cualquiera de estos eventos es detectado, el Centinela actualizará el estado del agente convenientemente (VariablesState y BeliefsState), esta modificación hará que el controlador comprueba todas las reglas de disparo de nuevo y active las tareas adecuadas para tratar dicho evento.

Los resultados de ejecución de las tareas también necesitan tener una salida al exterior, es decir, las tareas deben tener la capacidad de actuar sobre el comportamiento del agente físico (del robot avión) aparte de sobre su estado software.

Para ello las tareas tienen también acceso a los efectores del avión (clase Efectors). Si varias tareas piden acceso al mismo tiempo a un Efector, obtendrá el control del mismo la que tenga mayor nivel de prioridad.

Aparte de estas clases usadas como entrada y salida de datos de las tareas, estas también suelen contar con variables internas solo utilizadas por ellas. Dichas variables no tienen salida al exterior (es decir, a otras tareas o a los efectores), pero son usadas por la tarea como datos intermedios de los cálculos que realice.

Funcionamiento interno de las tareas.

Respecto a como las tareas realizan sus cálculos, ya se dijo antes que estas se comportaban como máquinas finitas de estados. Efectivamente cada tarea cuenta con un conjunto de estados determinados, donde cada estado realiza un conjunto de operaciones más o menos sencillo. Con este comportamiento, las tareas pueden modificar su funcionamiento según la situación presente, y realizar trabajos aparentemente complejos combinando en orden conjuntos sencillos de operaciones. Un ejemplo sencillo de la máquina de estados de una tarea es el mostrado en la Ilustración 17.

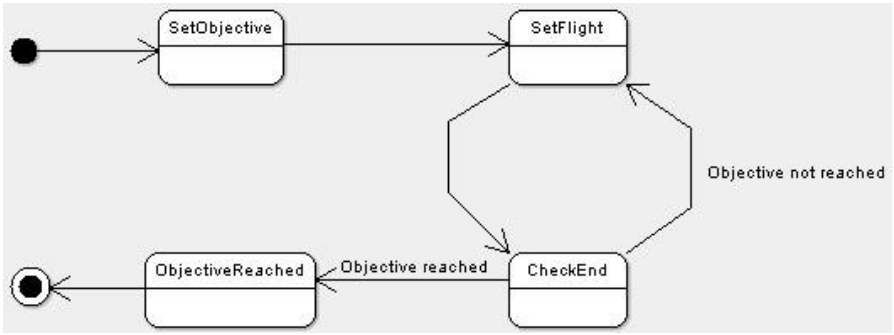


Ilustración 17: Un ejemplo de la máquina de estados de una tarea.

Las variables internas antes comentadas, suelen usarse para comunicar unos estados con otros. Los resultados de las operaciones realizadas por un estado son almacenados en variables internas de la tarea para ser luego usados por otros estados o para decidir que estado será el siguiente en ejecutarse.

Lista de tareas.

A continuación se ofrece una lista de las tareas usadas por los agentes. Como se ha dicho antes, cada tarea está diseñada para desempeñar una función determinada necesitada por el agente, se podría decir que cada tarea persigue un objetivo. Como los objetivos de los agentes buscadores suelen ser distintos que los del agente jefe, entonces habrá tareas que serán usadas solo por uno o por otro, siendo el resto usadas por ambos. En la descripción de cada tarea se indicará si es una tarea usada por el jefe, por los buscadores o por ambos.

TaskFlyToObjective (Boss)

Esta tarea del jefe se encarga de dirigir al avión hacia un determinado objetivo.

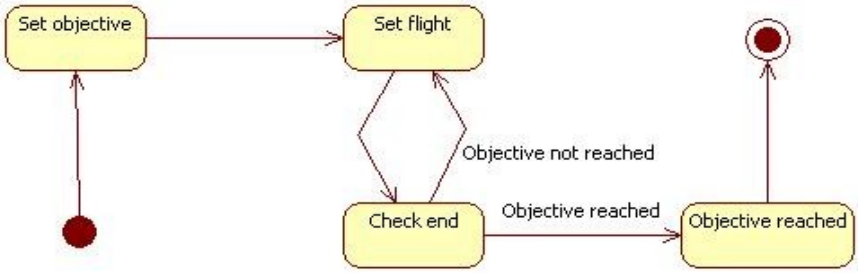


Illustration 18: TaskFlyToObjective

Cuando esta tarea es activada por el controlador, el avión comenzará a modificar su vuelo para alcanzar la coordenada objetivo definida en los Beliefs del agente como *'coordinatesNewObjectiveToFly'*.

Esta taréa usa el estado *'SetFlight'*. Este es un estado común, usado por varias tareas, especializado en controlar al avión para dirigir a este hacia un determinado punto en el espacio. En el estado *'setObjective'*, la tarea fija un par de campos de la clase *'SetFlight'* para luego poder usarla como estado de su máquina de estados. En *'checkEnd'* se compreuba si se ha llegado a una cierta distancia prudente del objetivo. Si esto es cierto se pasará al estado *'objectiveReached'*, donde se modificarán las *'variables'* necesarias para que el controlador y el resto de tareas sepan que se ha alcanzado el objetivo (*Ilustración 18*).

TaskFollowBoss (Searcher)

Esta tarea de buscador se encarga de mantener a este siguiendo al jefe dentro de la formación. más específicamente, una vez activada esta tarea hace que el agente siga a la posición que le ha tocado dentro de la formación.

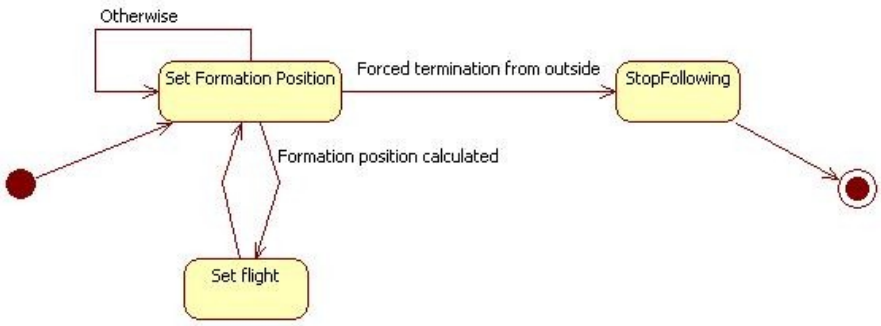


Illustration 19: TaskFollowBoss

En el estado *'setFormationPosition',* el agente comprueba la posición que tiene asignada dentro de la formación y calcula la coordenada en la que deberá encontrarse acordemente. Una vez calculada esta coordenada, pasa al estado *'setflight'*, en donde se maneja al avión para que vaya hacia dicho punto en el espacio. Cualquier tarea o el controlador pueden hacer que esta tarea deje de ejecutarse modificando la variable *'intoFormation'*. Hecho esto, la tarea pasará al estado *'stopFollowing'*, en donde modificará los Beliefs y Variables necesarios para informar que ha terminado a quien pueda interesarle.

TaskGenerateRoute (Boss)

Esta tarea del jefe se encarga de general la ruta que los agentes seguirán para recorer el area de búsqueda asignada. Para ello se usa un algoritmo evolutivo cuyos pormenores son detallados en la sección "2.3 - Calculo de la ruta óptima"(*Ilustración 20*).

En el estado *'generateGraphNodes'*, la tarea descompone el área de búsqueda en subregiones, asignándole a cada una un nodo de un grafo representativo de dicho area. En el estado *'generateRoutesPopulation'*, se genera una población de rutas, cada una definida como una forma de recorrer los nodos de dicho grafo. En *'evaluateAndSelectPopulation'* se evalua lo buena que es cada ruta de acuerdo a una expresión a optimizar. Las que mejor optimicen dicha expresión serán usadas para generar la población de la siguiente generación. En *'crossoverPopulation'* se aplica el operador Cruce a todos los individuos de la población. Si bien en la versión presentada en el proyecto no se realiza dicha operación, se ha dejado dicho estado en blanco en la implementación para permitir que futuros programadores que vallan a dedicarse a continar este proyecto puedan incluirlo. En el estado *'mutatePopulation'* se muta a todos los individuos de la población. El ciclo de estos tres estados se repite hasta que se consigue un individuo con una ruta suficientemente buena, o hasta que pasa un tiempo límite. Entonces se seleccionará dicha mejor ruta y se pasará al estado *'end'*, en donde se hace constar en *'Beliefs'* y *'Variables'* que ya hay disponible una ruta de vuelo.

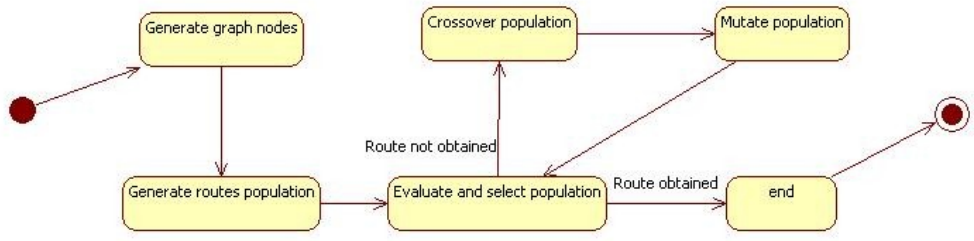


Illustration 20: TaskGenerateRoute

TaskProcessPhotographs (Searcher)

Esta tarea se encarga de procesar las fotografias que se han dicho tomando de la superficie que hay bajo el avión. Esta tarea se activará cada vez que haya fotos en el buffer de la cámara fotográfica. Una vez activada procesará todas las fotografías que encuentre en el buffer para averiguar si hay barcos en ella o no. En el momento de escritura de este borrador de memoria no se ha conseguido implementar un método que diferencie entre fotografías con barcos y sin ellas. Ahora mismo se está trabajando en posibles redes neuronales de convolución para poder realizar este procesado. No obstante se explica aqui esta tarea, aparte de estar incluida en el código, por si en el futuro alguien está interesado en dotar a Seagull de esta capacidad.

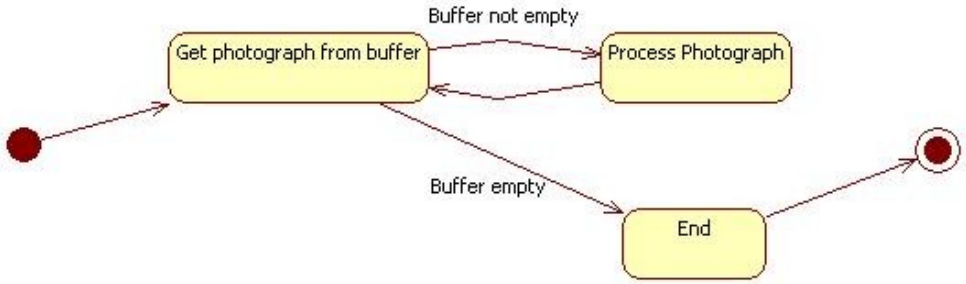


Illustration 21: TaskProcessPhotographs

El estado 'getPhotographFromBuffer' comprueba si hay fotografías en el buffer de la cámara fotográfica, de ser esto cierto, pasará a 'processPhotograph'. En este segundo estado se procesará la fotografía usando el método de reconocimiento de imágenes idones. Una vez el buffer esté vacio, se pasará al estado 'end', en donde, como en las otras tareas, se harán constar los cambios producidos (*Ilustración 21*).

TaskSearcherSentinel (Searcher)

Esta tarea estña siempre activa y es la encargada de estar pendiente de los sensores a la espera de posible sucesos que deban saber el resto de tareas, pero para los cuales no hay ninguna otra tarea a la espera de los mismos. En la versión actual de Seagull, solo ha sido necesarioque esta guarde constancia del sensor *'reloj'*, pues el resto de

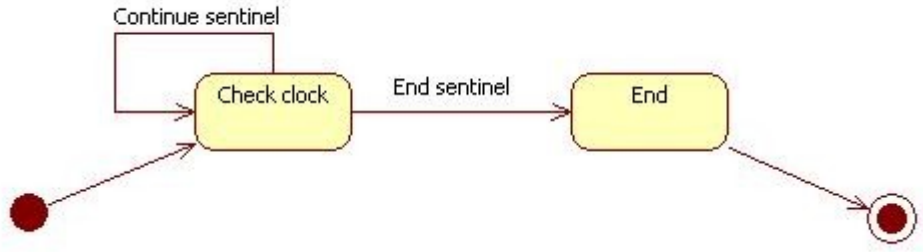


Illustration 22: TarkSearcherSentinel

eventos importantes son siempre esperados por alguna otra tarea que ésta puede reaccionar cuando ocurran. Con esto, la tarea simula un evento que ocurre cada cierto

tiempo y que indica a la cámara fotográfica del avión que tome una fotografía del área que está sobrevolando en estos momentos. Si en futuras ampliaciones de Seagull es necesario simular la ocurrencia de otros eventos provenientes de los sensores, esta tarea está diseñada para este fin. Solo sería necesario incluir un estado *'ChekCollision'*, por ejemplo, para conseguir que el agente fuera consciente de colisiones inminentes con objetos del entorno. Dicho estado chequearía si cualquier objeto detectado en el radar está a menos de X distancia. De ser cierto cambiaría las *'variables'* necesarias para que los mecanismos de evasión se pusieran en marcha.

La tarea en su forma actual es sencilla. En el estado *'CheckClock'* se comprueba si ha pasado el tiempo necesario para hacer una nueva fotografía. Si es cierto se modifica la variable *'haveToDoPhotographs'* que indica que es necesario tomar una nueva fotografía. Puede forzarse la terminación de esta tarea para que pase al estado *'end'* y termine, de todas formás la versión actual de Seagull no necesita esto y se mantiene a la tarea activa en todo momento.

TaskTakePhotographs (searcher)

Esta tarea es la encargada de realizar fotografías con la cámara fotográfica y colocarlas en el buffer de ésta. Cada vez que el avión ha recorrido una cierta distancia o cada vez que le es indicado por la variable *'haveToDoPhotographs'*, la tarea toma una fotografía.

El primer estado es *'checkPosition'*, en el cual la tarea comprueba si se ha recorrido suficiente espacio desde que se tomó la última fotografía como para hacer otra. También chequea si en la variable *'haveToDoPhotographs'* se le indica que tome una imagen. Si cualquiera de estas dos condiciones se cumple, entonces se pasa al estado *'takePhotograph'*, en donde se toma la fotografía y se la almacena en el buffer. Se puede forzar externamente que esta tarea termine, pasando entonces al estado *'end'* y finalizando (*Ilustración 23*).

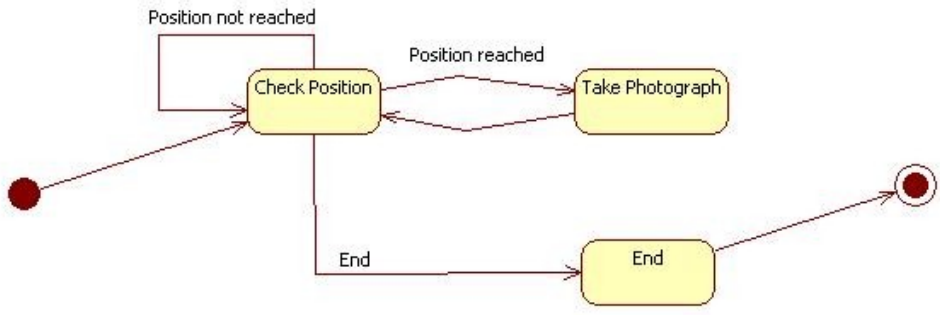


Illustration 23: TaskTakePhotograph

TaskComeBackHome(Boss)

Esta tarea es la encargada de modelar el comportamiento del agente Boss una vez se

ha alcanzado el último punto de la ruta. Un agente que tenga esta tarea activa, retornará al punto de inicio de la misión. Los demás agentes, si los hubiera, al tener como tarea activa *taskFollowBoss*, le seguirán hasta el punto de inicio. En esta tarea se pueden distinguir cuatro estados, uno de ellos nuevamente es el estado 'SetFlight', por lo que se omite:

1. ***Estado stateSetHome:*** En este estado el agente, una vez llegado al último punto de la ruta, determina cuales son las coordenadas del punto de destino o base. Una vez finalizado, se transita al estado *StateCheckEnd.*
2. ***Estado StateCheckEnd:*** En este estado, el agente verifica si se ha alcanzado el punto de destino o base. Mientras no se alcanza las coordenadas del punto de destino, el agente seguirá en este estado. Una vez alcanzado el punto de destino, se alcanzará el estado *StateHomeReached.*
3. ***Estado StateHomeReached:*** Estado final. Informa que el punto de destino ya ha sido alcanzado.

En el siguiente diagrama de estados se muestra gráficamente las transiciones de estado que conforman la tarea:

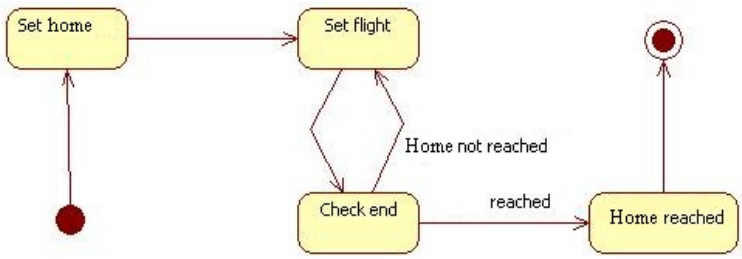


Ilustración 24: taskComeBackHome

TaskSendNewCheckPoint(Boss)

Mediante esta tarea, el agente jefe o *Boss* puede enviar por radio cual es el nuevo punto de encuentro. De esta forma, los demás agentes pueden actualizar las coordenadas del *checkPoint*. Sólo dos estados implementan esta tarea:

1. ***Estado StateSetNewCheckPoint:*** En este estado, conocida la ruta, el agente calcula cual es el siguiente punto de encuentro. Una vez calculado, se transita al estado StateSendNewCheckPoint.
2. ***Estado StateSendNewCheckPoint:*** Alcanzado este estado, el agente ya ha determinado cual es el nuevo punto de encuentro. Por tanto, en este estado se envía este nuevo checkPoint mediante la radio. El mensaje con las nuevas coordenadas tiene una frecuencia acordada previamente y conocida por los demás agentes, de tal forma que puedan comprobar de manera cíclica los mensajes a esta frecuencia para conocer los cambios del punto de encuentro.

En el siguiente diagrama de estados se muestra gráficamente las transiciones de estado que conforman la tarea:

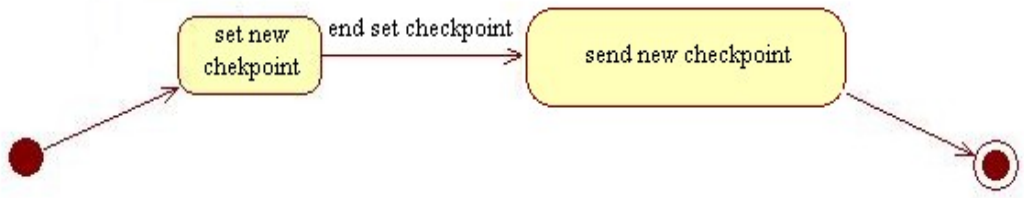


Ilustración 25: taskSendNewCheckPoint

2.3 Calculo de la ruta óptima.

En este tema se explicará como el agente usa un algoritmo evolutivo para resolver uno de los principales requisitos del sistema. Estamos hablando de que el agente jefe, antes de iniciar la exploración de la zona que el usuario le ordena investigar, debe decidir de que manera va a recorrerla entera. Este algoritmo evolutivo está implementado en la tarea *TaskGenerateRoute*, que ejecuta dicho algoritmo a través de una máquina de estados.

A continuación definiremos en detalle el problema a resolver y después mostraremos como funciona el algoritmo evolutivo que lo resuelve.

El problema.

Definición del problema

Como se ha debido explicar con anterioridad en esta memoria, el objetivo primordial del sistema Seagull es detectar posibles naufragios en una zona e informar de los posibles hallazgos al equipo de rescate humano que está usando a los agentes. Cada vez que el usuario (el mencionado equipo de rescate) utiliza a los agentes para explorar una zona de alta mar, debe decirle a éstos exactamente que zona es la que quiere que exploren. Para ello se les indican las coordenadas de los vértices de dicho área, como se muestra en la figura

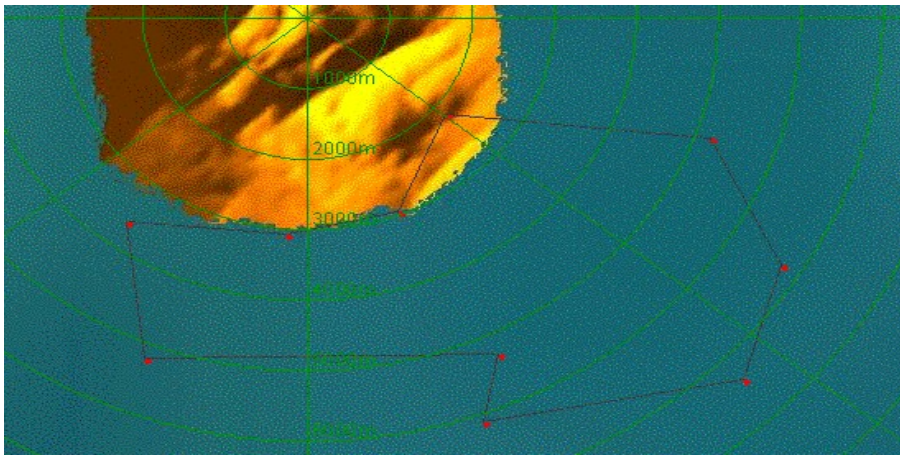


Illustration 26: Antes de comenzar el vuelo, a los agentes les es indicada el área que deben explorar.

Una vez los agentes conocen la región en la que deben buscar, tienen que decidir autónomamente de que forma la van a explorar. Obviamente sería deseable que los agentes realizaran dicha exploración de manera óptima o sub-óptima, gastando el menor combustible posible, realizándola en el menor tiempo posible, etc. Al mismo

tiempo deben supervisar la zona por completo, es decir, no debe quedar ninguna parte de dicho área sin haber sido fotografiado y comprobado que haya o no restos de naufragio u otros objetos de interés.

Equivalencia del problema

Este problema de explorar el área de forma completa y óptima (o sub-óptima), es equivalente a encontrar una determinada ruta sobre dicho área. Muchos se preguntarán ¿Que tiene que ver un área de dos dimensiones con una ruta de vuelo?. Esta reducción del problema se explica a continuación.

Si uno de los agentes-avión se encuentra en la posición (x_A, y_A, z_A) , hay una región en la superficie

$$R=\{(x, y, z) : z=0 \wedge \sqrt{(x_A-x)^2+(y_A-y)^2} \leq D\}$$

que puede ser fotografiada por la cámara del agente, considerando *D* como el alcance horizontal de la cámara fotográfica a esa altura.

Si un agente pude fotografiar esa zona circular y, al mismo tiempo, se mueve siguiendo una linea recta, entonces puede cubrirse todo el espacio que dista a menos de D de la linea que sigue el avión en su vuelo, como se muestra en la Ilustración 27.

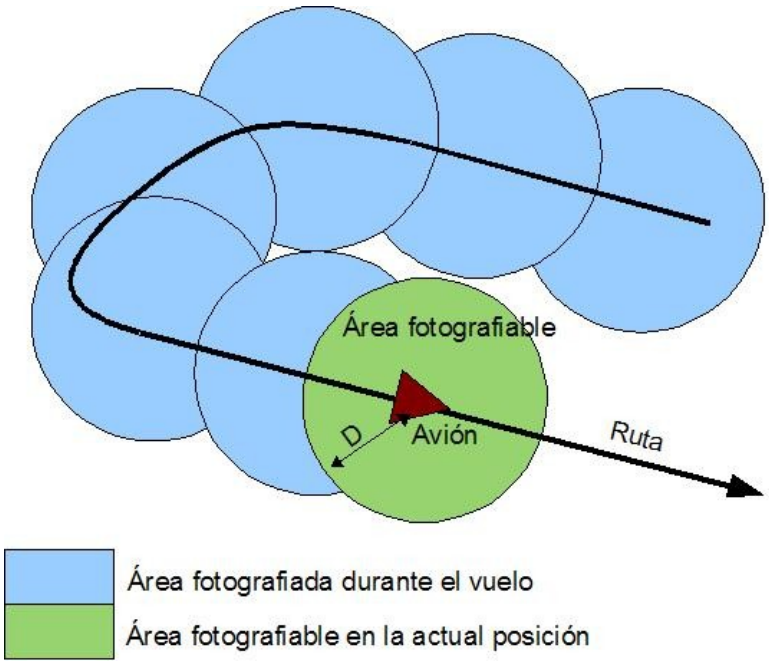


Illustration 27: A lo largo de su vuelo, el avión cubre todo el espacio que dista a menos de D de algún punto de la ruta de vuelo.

Si los agentes deben cubrir todo un polígono siguiendo esta técnica, entonces deben calcular la línea plegada sobre dicho polígono de menor longitud que pueda cubrirlo entero. En la figura 28 pueden verse dos soluciones a este problema. Ambas rutas cubren el polígono entero, pero la primera de ellas de manera muy poco eficiente, y la segunda de forma óptima o sub-óptima. En el siguiente apartado explicaremos el algoritmo que se ha seguido para conseguir soluciones del segundo tipo mostrado en dicha figura.

Hasta ahora se ha explicado el problema para un solo avión, pero no se ha hablado del caso de que exista más de un avión volando a lo largo de la ruta en formación, que es el caso con el que se enfrenta Seagull. Ahora se mostrará como el problema es generalizable para el caso de varios aviones volando en formación.

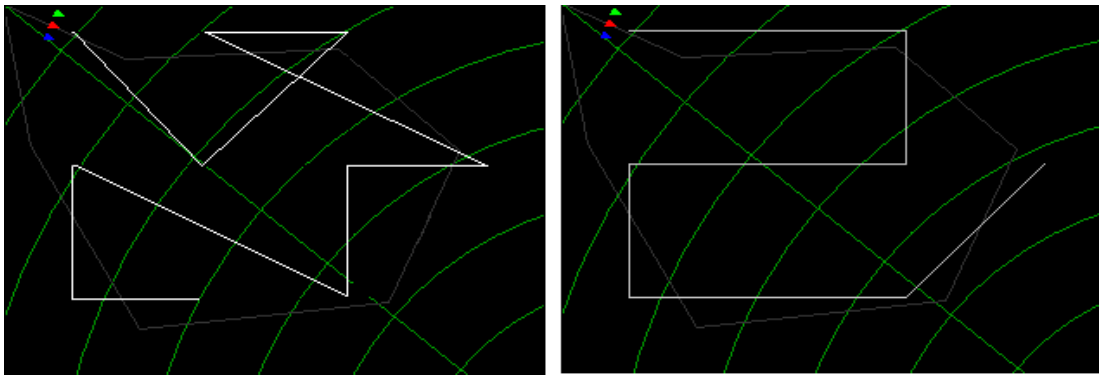


Illustration 28: Dos posibles rutas que cubren el polígono entero, la primera de ellas de forma no eficiente y la segunda de forma óptima o sub-óptima.

En el caso de que tengamos dos aviones volando en formación paralela a lo largo de la ruta, como se muestra en la figura 29, el área fotografiable por ambos deja de ser circular. Sin embargo, si se mantiene que los aviones vuelen todos siguiendo la misma dirección indicada por la ruta, el resultado final es idéntico al obtenido para un solo

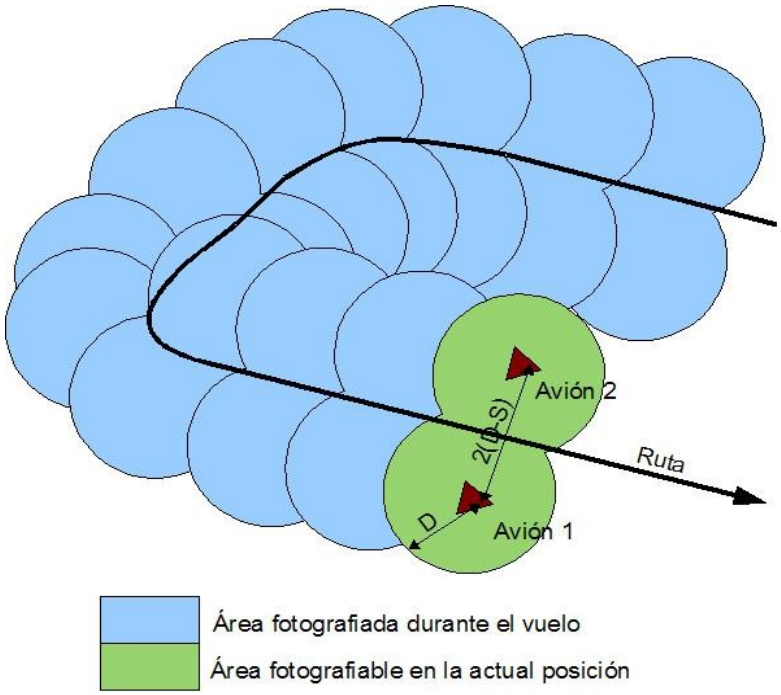


Illustration 29: A pesar de que el área fotografiable en cada instante en el caso de más de un avión volando en formación deja de ser un circulo, el área cubierta por los aviones a lo largo del vuelo es de la misma naturaleza que para el caso de un solo avión. Todos los puntos a menos de una determinada distancia de la ruta son cubiertos.

avión, al menos cuando la ruta es una línea recta. El área fotografiable a lo largo del vuelo es la que se encuentra a menos de 'AD-(A-1)S' de la ruta, donde S es una distancia de seguridad para asegurar que no quedan resquicios no fotografiables entre

los dos aviones, y 'A' es el número de agentes que hay en la formación. Los aviones vuelan en paralelo a una distancia 'D-S' los unos de los otros.

Como decíamos, para el caso de una ruta en forma de línea recta es obvio que el problema sigue teniendo la misma naturaleza: todos los puntos a menos de una determinada distancia de la ruta son fotografiados, el resto no. Sin embargo en el caso de una línea con curvas, como la mostrada en la *Ilustración 28*, no es tan obvio que el problema sigua siendo el mismo, de hecho si no se toman ciertas precauciones deja de serlo. La precaución que hay que tomar es que hay que aumentar el número de fotografías que se hacen por unidad de tiempo en relación con el ángulo del giro pues, de no hacerse, quedarían zonas sin cubrir en la parte exterior de la curva, como se muestra en la figura Ilustración 30. Este aumento de fotografías es directamente proporcional al número de aviones de la formación e inversamente proporcional al ángulo de giro de la curva.

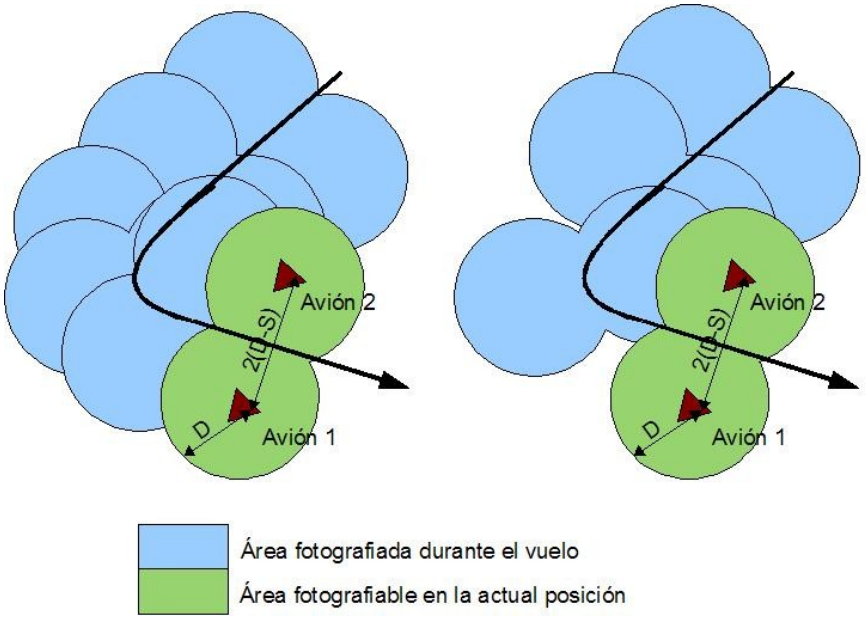


Illustration 30: El número de fotografías tomadas por unidad de tiempo, debe ser aumentado al tomar una curva, para asegurar que no quedan áreas sin fotografías en la parte externa de la curva. Así es como se ha procedido en la ilustración de la izquierda. Sin embargo en la ilustración de la izquierda, el número de fotografías tomada por unidad de tiempo no ha sido aumentada al tomar la curva.

La solución.

Técnica seguida

Para calcular una solución sub-óptima a este problema, se ha optado por un algoritmo evolutivo, ya que al ser un problema NP-completo cualquier solución exacta sería imposible de realizar por el limitado procesador de un avión robótico. El algoritmo evolutivo funciona de la manera usual. En este caso crea un población de rutas y las hace evolucionar seleccionando en cada generación a las mejores y realizando mutaciones sobre ellas para crear la siguiente población. El algoritmo no

usa el operador cruce, tan solo el operador mutación. A continuación se explica como consigue reducirse este problema a un algoritmo evolutivo.

Preprocesamiento del problema

Como primer paso, el algoritmo de solución preprocesa el problema para convertirlo en el "Travelling Salesman Problem". Después de esto el algoritmo soluciona el problema siguiendo una de las técnicas comúnmente usadas para resolver este típico problema, y algoritmo evolutivo.

Para transformarlo en el "Travelling Salesman Problem", primero convertimos el polígono que hay que explorar en un grafo. Debido a que cuando el avión se encuentra en una posición determinada este puede fotografiar todo el área que se encuentre a una distancia menor que D, entonces podemos crear una distribución uniforme de puntos sobre el polígono. Si dichos puntos están dispuestos formando una cuadrícula y distan unos de otros menos que 2D, entonces, con que el avión pase por todos los puntos, nos habremos asegurado que ha fotografiado todo el área. Por lo tanto, dichos puntos dispuestos en forma de cuadrícula pueden considerarse como los nodos de un grafo completamente conectado. Cada arista del grafo tendría asociado un peso igual a la distancia que separa los dos nodos (puntos en el espacio) que conecta. Encontrar el recorrido del grafo de menor peso total es el "Travelling Salesman Problem". Y dicho recorrido es equivalente a la ruta óptima que recorre todo el polígono fotografiándolo entero.

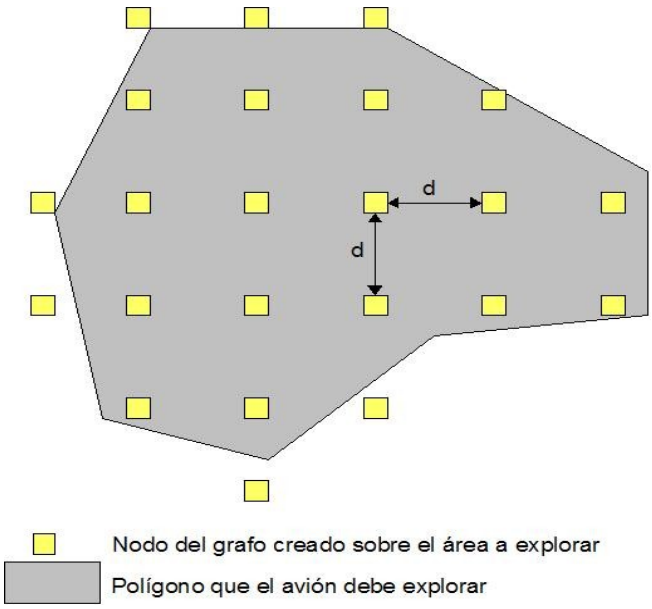


Illustration 31: Explorar el área es equivalente a pasar por un conjunto de puntos distribuidos uniformemente sobre el polígono que define dicho área, manteniendo d≤2D.

Para hacer este cálculo de manera manera rápida, el algoritmo primero engrosa el polígono a explorar una distancia D. Después comprueba qué vértices de una cuadrícula de lado 'd' caen dentro de dicho polígono engrosado. Así nos aseguramos que los bordes del polígono también sean fotografiados, pues si solo nos quedáramos con los vértices que caen dentro del polígono sin engrosar, las secciones cerca de los bordes del polígono que estén a una distancia mayor que D del nodo más cercano quedarían fuera del alcance de la cámara fotográfica. En la figura 32 se muestra la diferencia entre los nodos que caen dentro del polígono engrosado y los que tan solo están dentro del polígono no engrosado.

Comunicación con el algoritmo evolutivo

Una vez tenemos los nodos del grafo, debemos crear la primera población para pasársela al algoritmo evolutivo. Eso se hace sencillamente recorriendo N veces todos los nodos de manera aleatoria, donde N es el número de individuos de la población que evolucionará. Cada vez que se recorre el grafo, se crea un individuo, que es un simple array indicando el orden en el que se recorren los nodos. En los mencionados recorridos aleatorios del grafo, siempre se empieza por el nodo que se encuentre más cercano a la posición de despegue de los aviones. Es decir, el primer nodo de todos los recorridos es el más cercano a la posición inicial de los aviones. Eso se hace así porque, obviamente, si los aviones tuvieran que ir primero a algún nodo que no fuera el más cercano a su posición inicial, éste no sería un recorrido óptimo y muy probablemente tampoco sub-óptimo. Después, cuando se evolucione a la población, el algoritmo mantendrá también que todos los individuos de la población tengan como primer nodo a recorrer el más cercano a la base de despegue de los aviones.

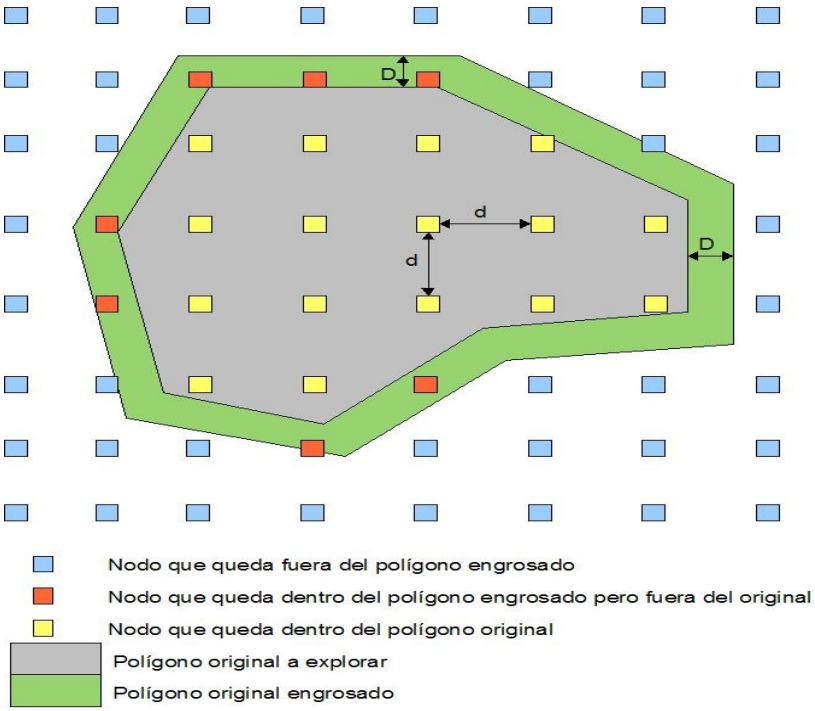


Illustration 32: Deben considerarse todos los nodos que caen dentro del polígono engrosado, pues si no pueden quedar áreas cercanas a los bordes del polígono sin fotografiar.

Algoritmo evolutivo

Respecto al algoritmo evolutivo, este funciona de la manera usual, usando el operador mutación y haciendo uso de la técnica "élite". Inicialmente el algoritmo tiene la población de N individuos aleatorios generadas por el algoritmo explicado anteriormente. En cada generación a partir de esta población, el algoritmo evalúa todos los individuos que la componen. La función de evaluación de un individuo es la suma total de los pesos de las aristas que forman su recorrido. Una vez evaluados todos los individuos, estos son colocados descendentemente de acuerdo con el valor devuelto por la función de evaluación. Los primeros E individuos, son pasados sin mutar a la siguiente generación. Los primeros P individuos (incluyendo a los de la élite) son usados para generar los N-E individuos restantes de la siguiente generación. Esta se crea aplicando el operador mutación sobre los P individuos, empezando por los individuos de menor valor en su mencionada función de evaluación (es decir, los

mejores). Estos individuos padres no son destruidos al aplicar el operador mutación, si no que se crea una copia de los mismos y sobre ésta se aplica dicho operador. Esto es necesario, a parte de para conservar los individuos mejores, para que cuando hayamos usado todos los individuos P para generar parte de la nueva población pero todavía queden individuos que generar (pues N-E suele ser mayor que P), puedan volverse a recorrer todos los padres de manera ascendente para seguir generando la nueva población (*Ilustración 33*).

Respecto al operador mutación, este es de ejecución muy sencilla para hacer el algoritmo rápido. El operador tan solo intercambia posiciones del recorrido de manera aleatoria. Cada posición del recorrido tiene probabilidad 2M de ser intercambiado por otra posición del recorrido. Por ejemplo, en el recorrido { 2, 5, 3, 4, 1 } este operador podría intercambiar las posiciones 1 con la 3 y la 2 con la 5, obteniendo el recorrido { 3, 1, 2, 4, 5 } (*Ilustración 34*).

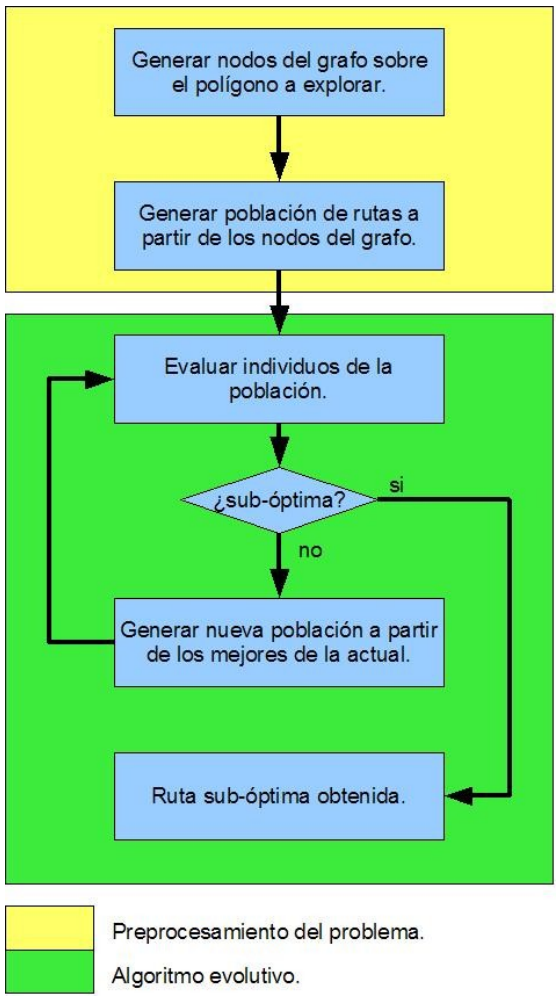


Illustration 33: Algoritmo de resolución completo.

Durante la explicación del algoritmo evolutivo, se han introducido determinadas constantes sin detenerse a explicar su significado o valor. Estas constantes son:

- N = Son el número de individuos total de la población.
- P = El número de padres que se selecciona en cada generación para producir la

siguiente población con el operador mutación.

- E = El número de individuos élite que pasará sin modificación a la siguiente población.
- M = El doble de este valor es la probabilidad de que una arista sea afectada por el operador mutación.

Acerca de su valor, después de varias pruebas se llegó a la conclusión de que una buena combinación de valores para el tamaño de los problemás con los que normalmente se enfrenta en Seagull, es:

- N = 300
- P = 100
- E = 5
- M = 0.03

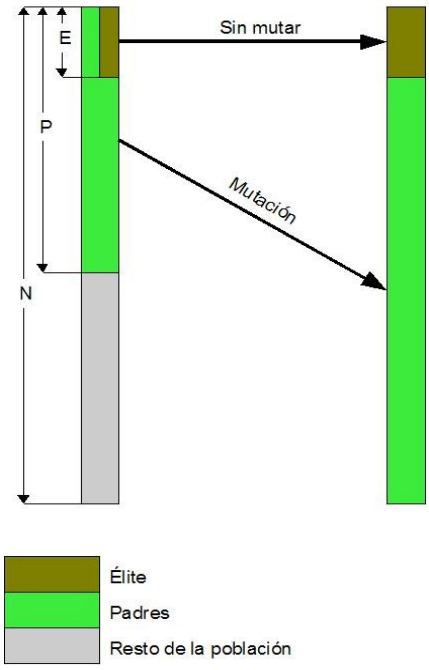


Illustration 34: En cada generación, la población total de rutas se divide en Élite, Padres y el resto. La élite es pasada sin mutar a la siguiente generación Los padres (incluyendo a la élite) son usados para generar el resto de la nueva población con el operador Mutación.

2.4 Reconocimiento de imágenes.

En este tema se explicará cual es la técnica que usan los aviones para reconocer las fotos que toma la cámara fotográfica del avión. El reconocimiento que deben hacer es distinguir entre fotos aéreas en las que aparece un barco y entre fotos en las que no aparece ningún barco. La tarea que realiza esta identificación es *TaskProcessPhotograph*, la cual recoge las fotografías tomadas almacenadas en un buffer y decide si en dicha foto aparece un barco o no.

A continuación se hará primero una descripción más detallada del problema de identificación a resolver. Después se explicará la primera técnica con la que se intentó resolver el problema, la cual no funcionó. Por último se detallará la técnica que finalmente se usó, no solo mostrando su funcionamiento, si no varios ejemplos para demostrar su eficacia.

El problema.

Definición del problema

Como se decía antes, el problema que hay que resolver aquí es un problema de reconocimiento de patrones en imágenes. La misión principal de los agentes-avión es detectar si dentro de la zona de búsqueda hay algún resto de naufragio y, en caso afirmativo, informar de su posición. Para conseguir esto, el agente, según va volando sobre el área de búsqueda, va tomando fotografías de la superficie con una cámara instalada en el suelo del avión. Estas fotografías serán fotos aéreas de la zona que se encuentra justo por debajo del mismo, ya sea en color, en blanco y negro, o en infrarrojo. Por lo tanto, si el agente está sobrevolando en cualquier momento un resto de naufragio, este aparecerá en la fotografía que tome en ese momento. Si el agente

Illustration 35: El velero que debe ser identificado por los agentes. No obstante, en las fotografías que los aviones toman, este velero aparecerá visto desde arriba.

cuenta con algún mecanismo de identificación de imágenes, entonces podrá averiguar

autónomamente en que posiciones del área de búsqueda hay naufragios y en cuales no, informando a la base en el primer caso. Básicamente este es el problema de Reconocimiento Automático de Objetivos aplicado a fines civiles, y para resolverlo se han usado extensivamente las investigaciones publicadas al respecto.

Vamos a concretar un poco el problema al sistema Seagull. Nuestro sistema está pensado para ser instalado en agentes robóticos reales, pero actualmente es tan solo una simulación por ordenador con el fin de poderlo programar y testear. En nuestra simulación nosotros hemos usado veleros como el de la imagen 63 para hacer de resto de naufragio. Aunque dicho velero no aparente demásiado estar al borde de hundirse, el método de identificación usado puede entrenarse para identificar cualquier tipo de objeto, por lo que es válido usarlo en su lugar. Respecto al tipo de fotografía, se han usado imágenes en color pasadas por un filtro rojo para aparentar que han sido tomadas por una cámara de infrarrojos (Ilustración 2). Por último el tamaño de la imagen es de 507x630 pixels.

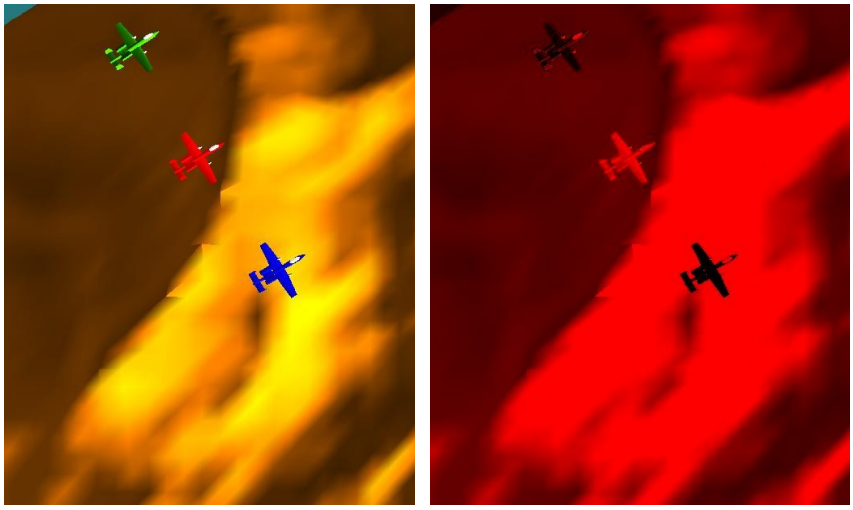


Illustration 36: Una fotografía del escenario en la que se mueven los agenetes y es misma fotografía tras aplicarle el filtro rojo. Los agentes analizan imágenes del segundo tipo para simular fotografía tomadas por una cámara infraroja.

Planteamientos de resolución

Para solucionar este problema desde un principio se pensó en usar redes neuronales, pues tienen la capacidad de reconocer patrones [ROJ1996] [KEV1997]. Además uno de los integrantes del grupo tiene pensado comenzar al año que viene una tesis doctoral muy relacionada con este paradigma de la Inteligencia Artificial, por lo que estudiar el problema mediante este planteamiento le serviría como entrenamiento en este área. Por ser las estudiadas más comúnmente, la primera idea fue usar un Perceptrón Multicapa y entrenarlo mediante Backpropagation sobre una colección de fotos. Como se explicará más adelante, enseguida nos encontramos con numerosos problemás al seguir este planteamiento. Debido a esto Matilde Santos, nuestra profesora de proyecto, nos puso en contacto con Gonzalo Pajares, pues él ha trabajado ya en visión y tiene publicados varios libros sobre este tema. Quisiéramos agradecerle a Gonzalo la orientación que nos prestó en este respecto, pues, entre otras cosas, nos avisó de las dificultades que un Perceptrón Multicapa tendría al enfrentarse a este problema. Gracias a esta orientación pasamos a investigar otros tipos de redes

neuronales más aptas para tal tarea y dejamos a un lado la idea del Perceptrón Multicapa, en la cual con toda seguiridad nos habríamos quedado estancados sin conseguir nada.

Primer intento: el framework JOONE (fallido).

El framework JOONE

Para implementar redes neuronales, Java cuenta con un framework con licencia GPL muy prometedor (Ilustración 37). Este framework permite realizar casi todo tipo de redes neuronales: multicapa, Kohonen, Hopfield, asociativas, estocasticas, y cualquier tipo de combinación de estas en módulos que se pueden controlar de forma diferente. Si esto no era suficiente, el framework también te permite usar distintos tipos de funciones de transferencia: sigmoide, lineal, tangente hiperbólica, "winner takes all"... y ofrece distintos tipos de conexión entre las capas: conexión completa, conexión directa, retroalimentación... . Por último, para poder analizar la red neuronal, JOONE ofrece gráficas estilo MatLab para observar distintos parámetros de la red que se esté construyendo. Esta herramienta puede usarse de dos maneras, mediante un Interfaz Gráfica o directamente llamando a sus objetos desde código fuente Java. La primera forma puede usarla cualquiera que sepa ya algo de redes neuronales. La segunda, aparte de tener que saber Java, es bastante aparatosa de usar, aunque ofrece mayores posibilidades que la interfaz. El framework con la interfaz puede ser descargada de su página oficial ([JOO2007]), donde también puede encontrarse información al respecto. También existe una comunidad de foros sobre esta herramienta ([JOO2007a]) donde programadores preguntan dudas de su uso y comentan como resuelven problemás con ella.

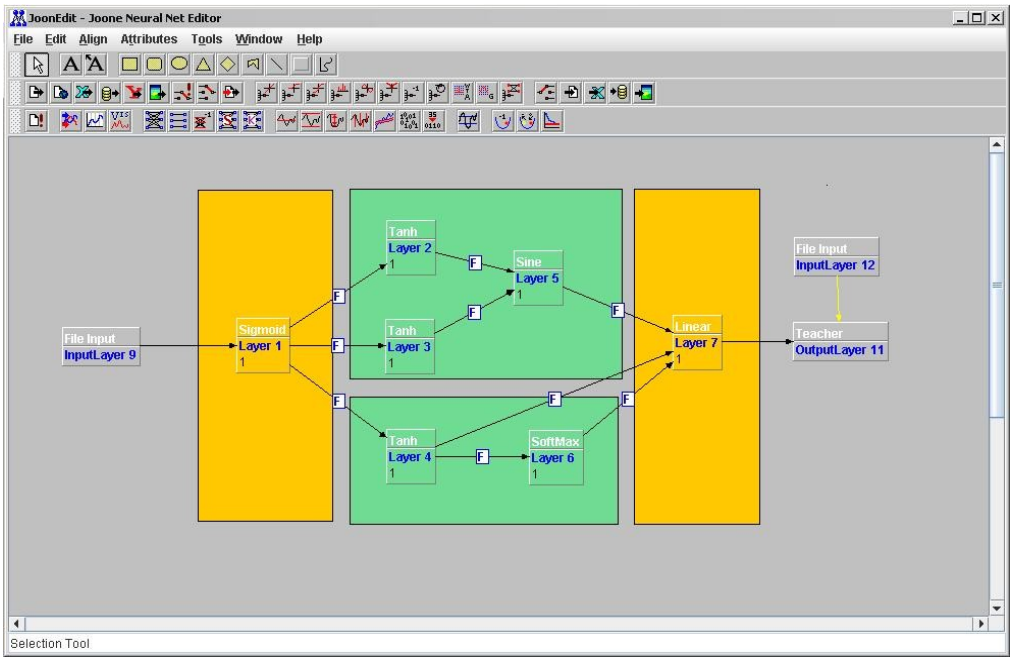


Illustration 37: Screenshot de la interfaz de JOONE. En este ejemplo se ve una red neuronal de varias capas creada con este framework.

Descrito así quedan pocas dudas de porqué se escogió este framework para implementar la red neuronal, además era la única librería de Java con este fin gratuita. El encargado de realizar esta parte del proyecto ya había trabajado con JOONE y conocía sus características, por lo que no dudó en volver a usar esta herramienta. No obstante, mientras esta librería le había dado muy buen servicio en otras ocasiones, para este problema no hubo tanta suerte, como se explicará en un momento.

Dificultades del problema

Entre los primeros problemás con los que se encontró está el gran tamaño de las fotografías. Si la imagen a identificar es de 507x630 pixels, entonces, solo en la primera capa de la red neuronal, tendría que haber 319.410 pesos (conexiones) por perceptrón. Si este número no es ya suficientemente grande, no olvidemos que la red debe ejecutarse en un pequeño ordenador instalado en un avión robótico que seguramente se colapse al usar una red tan grande. Para solucionar esto se optó por comprimir la imagen usando Wavelets y, después, partirla en trozos de 63x63 pixels: la red neuronal tendría que identificar varias imágenes de 63x63 pixels por cada fotografía tomada (Ilustración 38). Aún así el número de pesos por perceptrón en la primera capa seguía siendo relativamente alto (3.969).

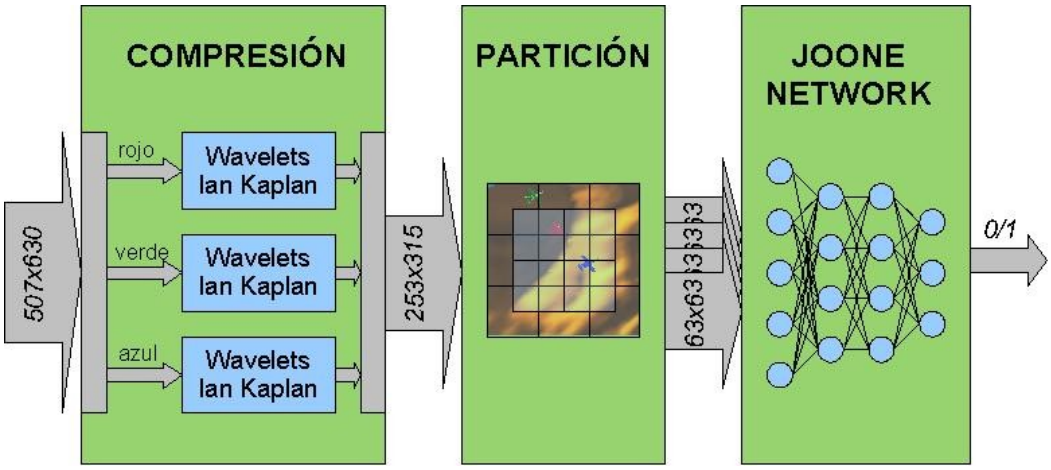


Illustration 38: Este diagrama muestra el proceso de análisis completo que se utilizaba en este primer intento de abordar el problema con redes JOONE.

Los límites de JOONE

Aparte de esto, el tratamiento de imágenes con JOONE todavía no está bien conseguir, aunque sus autores lo anuncian como una de las próximás mejoras. Las redes neuronales de JOONE por defecto leen los datos de un archivo de texto con el formato:

0.0 ; 1.0 ; 0.0 ; 0.0 ; 0.0 ; 1.0 ;
0.0 ; 0.0 ; 1.0 ; 1.0 ; 0.0 ; 1.0 ;
1.0 ; 1.0 ; 0.0 ; 0.0 ; 0.0 ; 0.0 ;
1.0 ; 0.0 ; 0.0 ; 0.0 ; 0.0 ; 0.0 ;

Por lo que cada fotografía habría que pasarla previamente a un archivo con este formato. Usando JOONE directamente desde código fuente puede conseguirse que la red al menos lea los datos de entrada y devuelva la salida de la red en matrices de

double. Sin embargo, para entrenarla, estás obligado a crear un archivo donde se especifiquen los patrones de entrenamiento (las relaciones entrada - salida deseadas) con la sintaxis mostrada. Este último requisito no solo hace más tedioso el problema, si no que puede llegar a aumentar la dificultad del mismo hasta hacerlo irresoluble. Esto es así porque una simple colección de 200 fotografías de 63x63 pixels, por ejemplo, si la codificas con unos y ceros en ASCII, puede llegar a ocuparte decenas de megabytes, con lo dificultoso que es manejar tal archivo en un simple bloc de notas.

El último problema que ofrecía JOONE era la forma de conectar la primera capa de la red neuronal a los pixeles de la imagen, o la forma de conectar unas capas con otras en general. En muchas ocasiones interesa crear una red neuronal con pesos compartidos, es decir, que distintas conexiones usen el mismo peso. Por ejemplo, en redes neuronales de convolución, muy usadas para reconocimiento de imágenes, se usan pesos compartidos. Sin embargo JOONE no te permite relacionar conexiones de tal forma, por lo que cada neurona artificial debe tener sus conexiones con pesos únicos. Otro gran problema de JOONE es como trata los datos que fluyen por la red neuronal. Para JOONE tanto los datos de entrada como los almacenados en las capas de la red tienen forma lineal de vector, es decir, una sola dimensión. Dado que las imágenes son en dos dimensiones, el tener que enfinarlas en vectores de una dimensión puede convertir algunas operaciones triviales en casi imposibles, como es el aplicar un pequeño filtro cuadrado a toda la imagen.

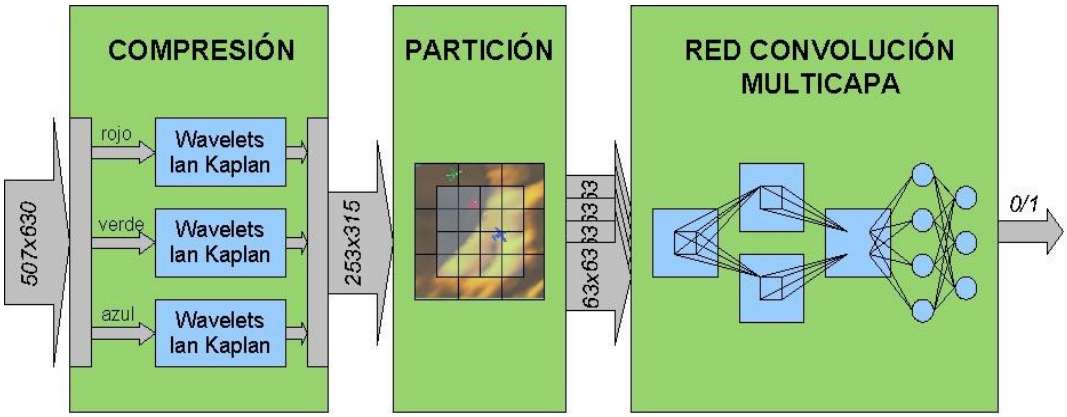


Illustration 39: Este diagrama muestra el proceso de análisis completo que se utilizó finalmente y con el que se consigue realizar la identificación de las fotografías correctamente.

Además, JOONE requiere que la red neuronal construida tenga en la primera capa una neurona por entrada, y en la última otra neurona por salida. Si tenemos que crear una neurona por pixel en la imagen, entonces el número de perceptrones de la red aumenta enormemente, y con él el número de operaciones a realizar durante el aprendizaje. Si estas neuronas usan una función de transferencia lineal no saturada, entonces las operaciones a realizar durante el entrenamiento son como si tales neuronas no existieran. Pero esto no soluciona el problema, pues JOONE parece no entrenar si la red tiene más de 600 neuronas, aunque estas usen una función lineal. Partir en trozos más pequeños las fotografías para reducir el número de neuronas en la primera capa tampoco valía, pues el framework también se cuelga si intentas realizar un entrenamiento con más de 1000 patrones.

En definitiva, JOONE actualmente es un framework muy útil sólo para problemás relativamente pequeños. Su facilidad de manejo a través de la interfaz y la enorme

Conclusión.

variedad de herramientas ofrecidas son dignas de elogio. De todas formás esta biblioteca está todavía activa, es decir, hay gente trabajando en ella para ampliarla y parece que surgen nuevas versiones cada varios meses. Probablemente dentro de un par de años todos los problemás aquí comentados habrán desaparecido, y JOONE se halla convertido en una buena herramienta para construir redes neuronales con Java.

Sengundo intento: Redes de Convolución (exitoso)

Investigación actual al respecto.

Un tipo de red neuronal ya propuesto para la identificación de imágenes es la llamada "Red Neuronal de Convolución". Estas redes neuronales básicamente analizan imágenes componiendo filtros. Por otro lado, como se dijo antes, el problema que tratamos de resolver aquí es básicamente el problema "Detección Automática de Objetivos", sobre el que ya se ha realizado abundante investigación debido a sus connotaciones militares [PAU1995] [TER1995] [DAV1995] [HEG1995] [STE1995]. Las redes neuronales de convolución y otros tipos de redes han sido ya usadas para resolver este problema u otros similares de reconocimiento de imágenes [DAI1999] [SHO1996] [RAV1995]. Para resolver el problema aquí planteado básicamente hemos aplicado una de las redes neuronales de convolución propuestas en estos artículos para identificación de objetivos [PAU1995]. Como se aprecia, para este segundo intento de resolución del problema, me informé mucho más acerca del tema. Un libro que he usado mucho, y que me ha sido de la más inestimable ayuda, ha sido [ROJ1996].

Descripción general de la resolución.

Para concretar como ha sido la técnica usada para reconocer las imágenes, diremos que esta se puede descomponer en dos etapas, una de preprocesamiento de la imagen y otra de análisis con la red neuronal (Ilustración 39). En la primera, al igual que se hizo con las redes JOONE, las imágenes de 507x630 se reducen usando wavelets y después se cortan en pequeñas imágenes de 63x63 pixels. En la segunda etapa, estas pequeñas imágenes son analizadas por la red neuronal. Estas dos fases se explicarán a continuación con más detalle.

Preprocesamiento de las imágenes

El problema de la segmentación.

Las imágenes que hay que analizar son muy grandes en comparación con el objeto que hay que identificar en ellas. Si el tamaño total de esta es de 507x630, el velero usado como objetivo de la identificación aparece con un tamaño de alrededor de 30x15 pixels, careciendo el resto de la fotografía de interés por ser usualmente solo agua (*Ilustración 40*). En "Detección Automática de Objetivos", al problema de distinguir entre las secciones de la fotografía con algo de interés del resto, se le suele llamar "segmentación" [STE1995]. Nosotros esto lo hemos resuelto reduciendo primero la imagen a cuatro veces su tamaño mediante wavelets, y después partiendo esta imagen en secciones aún más pequeñas con cuidado de que no pueda quedar ningún velero partido entre dos de estas secciones y que por lo tanto no fuera identificado al analizar ninguna de ellas.

Las wavelets usados para reducir la imagen, han sido creadas por Ian Kaplan, y la clase Java que los aplica puede descargarse desde [KAP2001]. El usado es

"Daubechies dB4", el cual usa una función wavelet g_i y otra de escalado h_i (también llamados filtro paso alta y filtro paso baja) para realizar la compresión. Desde el punto de vista matemático, podemos considerar la señal que analizan los wavelets como un vector infinito de reales, donde cada elemento expresa el valor de la señal en el momento $t_i = i \cdot \Delta t$ Siguiendo este planteamiento, la transformada usada por Daubechies D4 puede expresarse como una matriz infinita de la forma:

$$
\begin{aligned}
a_i &= \ldots h0, h1, h2, h3, 0, 0, 0, 0, 0, 0, 0, \ldots \quad s_i \\
c_i &= \ldots g0, g1, g2, g3, 0, 0, 0, 0, 0, 0, 0, \ldots \quad s_{i+1} \\
a_{i+1} &= \ldots 0, 0, h0, h1, h2, h3, 0, 0, 0, 0, 0, \ldots \quad s_{i+2} \\
c_{i+1} &= \ldots 0, 0, g0, g1, g2, g3, 0, 0, 0, 0, 0, \ldots \quad s_{i+3} \\
a_{i+2} &= \ldots 0, 0, 0, 0, h0, h1, h2, h3, 0, 0, 0, \ldots \quad s_{i+4} \\
c_{i+2} &= \ldots 0, 0, 0, 0, g0, g1, g2, g3, 0, 0, 0, \ldots \quad s_{i+5} \\
a_{i+3} &= \ldots 0, 0, 0, 0, 0, 0, h0, h1, h2, h3, 0, \ldots \quad s_{i+6} \\
c_{i+3} &= \ldots 0, 0, 0, 0, 0, 0, g0, g1, g2, g3, 0, \ldots \quad s_{i+7}
\end{aligned}
$$

donde g_i y h_i son los coeficientes de la transformación. La multiplicación de esta matriz por el vector columna de la señal, produce el nuevo vector columna de elementos a_i y c_i , donde el primero es una versión más suave de la sección *'i'* de

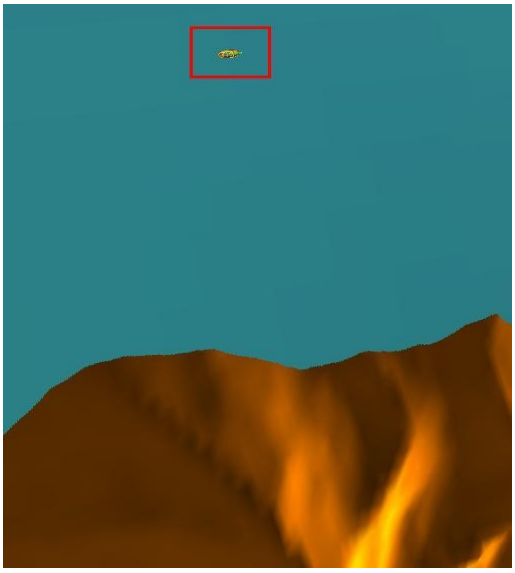


Illustration 40: En esta imagen se observa una típica fotografía que debe ser analizada por el agente. En ella la parte de interés es la señalada con un rectángulo rojo. El resto no contiene objetos que puedan confundirse con un velero.

la señal de origen y el segundo un coeficiente wavelet. Los valores obtenidos a_i pasarían a almacenarse en la primera mitad del nuevo vector, y los c_i en la segunda mitad. Esta primera mitad de la nueva señal podría considerarse como la señal anterior comprimida, e incluso podría aplicarse de nuevo el algoritmo sobre ella para conseguir versiones aún más comprimidas.

La señal codificada obtenida mediante esta transformación, puede devolverse a su estado anterior mediante la transformación inversa. Planteada igualmente como una matriz infinita que se multiplica a un vector-señal columna, esta transformada quedaría:

$$
\begin{aligned}
s_i &= \ldots h2, g2, h0, g0, 0, 0, 0, 0, 0, 0, 0, \ldots & a_i \\
s_{i+1} &= \ldots h3, g3, h1, g1, 0, 0, 0, 0, 0, 0, 0, \ldots & c_i \\
s_{i+2} &= \ldots 0, 0, h2, g2, h0, g0, 0, 0, 0, 0, 0, \ldots & a_{i+1} \\
s_{i+3} &= \ldots 0, 0, h3, g3, h1, g1, 0, 0, 0, 0, 0, \ldots & c_{i+1} \\
s_{i+4} &= \ldots 0, 0, 0, 0, h2, g2, h0, g0, 0, 0, 0, \ldots & a_{i+2} \\
s_{i+5} &= \ldots 0, 0, 0, 0, h3, g3, h1, g1, 0, 0, 0, \ldots & c_{i+2} \\
s_{i+6} &= \ldots 0, 0, 0, 0, 0, 0, h2, g2, h0, g0, 0, \ldots & a_{i+3} \\
s_{i+7} &= \ldots 0, 0, 0, 0, 0, 0, h3, g3, h1, g1, 0, \ldots & c_{i+3}
\end{aligned}
$$

donde s_i serían los distintos valores de la señal anterior recuperados.

Las transformaciones explicadas aquí están planteadas para señales infinitas. No obstante, en la realidad nunca nos encontramos con tales señales de longitud infinita, por lo que el algoritmo se encuentra con varios problemás al analizar el comienzo y el

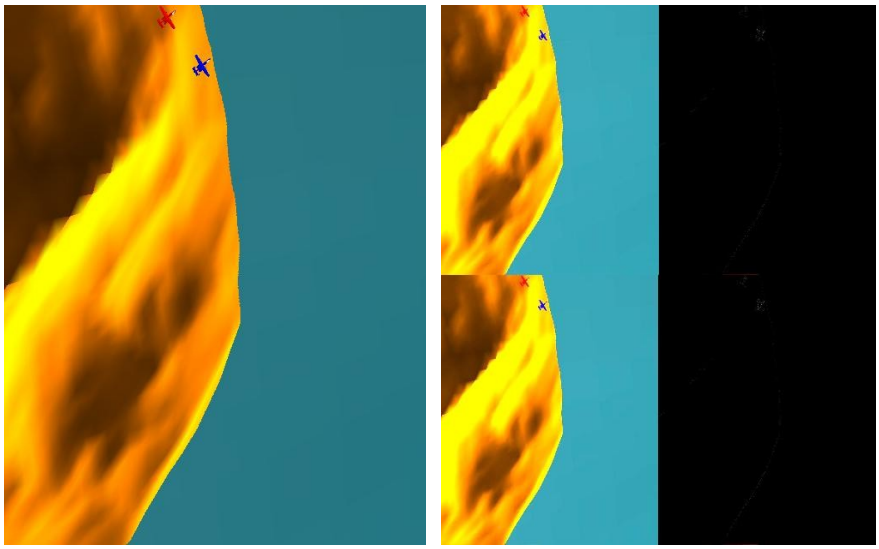


Illustration 41: En esta imagen se muestra una fotografía típica y al lado el resultado de aplicar waveletes a esta de la forma descrita.

final de la señal. Existen varias técnicas para solucionar este problema, el que usa el algoritmo de Ian Kaplan es considerar a la señal periódica, por lo que el final del vector considerado continúa por el principio del mismo.

Imágenes con wavelets.

Este algoritmo se ha explicado para el caso de señales lineales, pero sin embargo nuestras imágenes no son tal tipo de señal. Por un lado las fotografías son de dos dimensiones, donde cada pixel guarda relación con los cuatro pixeles de alrededor. Por otro lado, si la imagen es en color, cada pixel es un solo valor donde está codificada la información de la cantidad de rojo, verde y azul de dicho punto.

La primera de estas dos diferencias con respecto a una señal lineal puede ser ignorada. Por ejemplo podemos analizar la imagen por columnas, creando un nuevo vector de una dimensión donde están almacenados los pixels de la imagen leídos por columnas, es decir, los valores de una columna seguidos por los de la siguiente. Aplicando el algoritmo descrito a este nuevo vector se obtiene la señal codificada por wavelets. Se pude recomponer una imagen en dos dimensiones recogiendo los valores de este último vector exactamente de la misma manera con la que colocamos ahí los

valores de la imagen inicial antes de aplicar la transformada. El resultado obtenido será el mostrado en la imagen (*Ilustración 41*). En la señal que se obtiene leyendo la imagen de esta manera, el escalón encontrado al pasar de una columna a la siguiente es analizado perfectamente por los wavelets. No obstante, si se concatenan varias transformaciones en un solo sentido sobre dicha señal, no se ha comprobado si aparecen defectos en la parte superior e inferior de la imagen. En nuestro caso solo aplicamos la transformada una vez y no se observa ningún problema aparente.

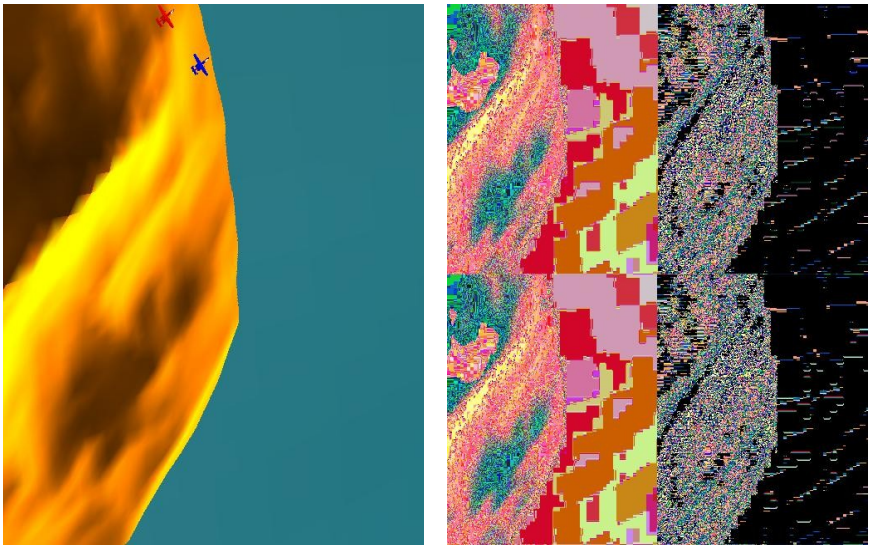


Illustration 42: En esta imagen se muestra una fotografía típica y al lado el resultado de aplicar waveletes a esta sin haberla descompuesto previamente en sus componentes rojo, verde y azul.

La primera de estas dos diferencias es más complicada, ya que la forma en que cada pixel almacena la información sobre cada color hace que el valor absoluto de cada pixel varíe de forma aparentemente estocástica. Si, por ejemplo, cada pixel es de 3 bytes, uno para cada color, el valor almacenado en cada punto de la imagen en binario contendría 24 bits. Los 8 bits más significativos serían para el rojo, los 8 bits del medio para verde, y los 8 menos significativos para azul. Así almacenados, pequeños cambios en la tonalidad de rojo de un pixel a otro, hacen que el valor representado por estos 24 bits varíe enormemente. Sin embargo cambios similares en azul apenas cambian dicho valor. Por lo tanto, si creamos una señal leyendo estos valores directamente por columnas, el resultado obtenido sería una secuencia que describiría el nivel de rojo de los pixels más ruido de gran amplitud procedente del verde y ruido de pequeñal amplitud procedente del azul. Si se realiza este análisis así el resultado será el visto en la *Ilustración 42*.

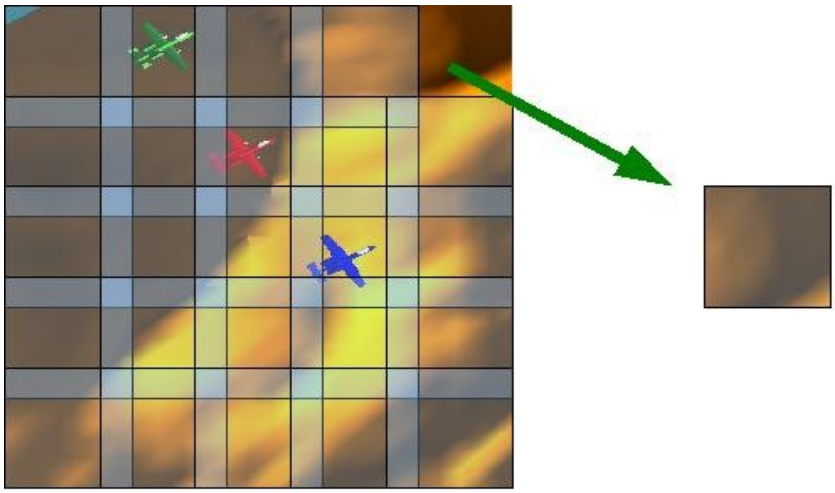


Illustration 43: Descomposición de una imagen en trozos más pequeños. Las distintas subdivisiones de superponen unas a otras para evitar que un velero no quede representado totalmente en ninguna de las sub-imágenes pequeñas.

Para evitar este problema hay que descomponer la imagen en sus tres colores, rojo, verde y azul, y analizar cada color por separado. Después pueden volver a unirse los tres colores para volver a formar una imagen, aunque esta última operación hay que realizarla con algunas reservas. Si tu tienes una señal cuyos valores están comprendidos entre, digamos, 10 y 0, y aplicas una de las mencionadas transformadas directa e inversa, los valores de la señal resultante no estarán totalmente comprendidos entre dichos límites. Los valores más altos de la señal que rozaban el límite superior, podrán haberlo superado un poco, e igualmente con los valores cercanos a 0 con respecto al límite inferior. Esto es así porque, si aplicas la transformada directa y después inversa a una señal con los wavelets aquí explicados, la señal recuperada no es exactamente la inicial, si no que sus valores pueden haber variado minimamente. Cuando nosotros separamos la imagen en sus componentes rojo, verde y azul, los valores de cada componente se encuentran entre 0 y 255. Por lo tanto, al aplicar wavelets, algunos valores de la señal obtenida al leer la imagen por columnas pueden haberse salido de dichos límites. Dado que estamos tratando con cadenas de bits, si tenemos 8 bits para un color, el que un valor de un color supere a 255 quiere decir que dichos 8 bits se desbordan y en vez de tener 256 (100000000) tenemos 0 (00000000). Algo similar ocurre si de un valor próximo a 0 obtenemos, tras aplicar las transformadas directa e inversa, un valor negativo. Por lo tanto puede perderse gran cantidad de datos si no se realiza el análisis con wavelets de las componentes con cuidado. El cuidado que hay que tener es simplemente forzar que todos los valores que superen a 255 tras realizar la transformada valgan 255 exactamente, y lo mismo con los valores que superen 0 (los cuales habrá que forzar que valgan exactamente 0).

El método descrito para analizar una imagen con wavelets puede verse en el diagrama (Ilustración 3). No obstante, en Seagull, la fotografías tomadas contienen solo el color rojo para simular el que sean imágenes infrarrojas, por lo que la descomposición de las imágenes en sus tres colores para aplicar wavelets no sería necesaria en nuestro caso. Sin embargo hemos considerado adecuado describir aquí lo que hay que hacer si se quieren tratar imágenes en color, pues podría ser muy interesante en ampliaciones futuras de nuestro sistema el realizar el análisis de las imágenes en color. Los algoritmos descritos para aplicar wavelets a imágenes en color ya están implementados en el código y funciona perfectamente, por lo que futuros alumnos interesados en esta ampliación no tendrían que preocuparse por ello.

Division de las fotografías.

Por último en la segmentación de las imágenes tomadas por la cámara fotográfica, está la división de esta en imágenes más pequeñas, tal como se muestra en (21). Una vez se ha aplicado la técnica de wavelets a la fotografía para reducir su tamaño a una cuarta parte, el resultado es dividido en pequeñas imágenes de 63x63 pixels. La única peculiaridad que hay que comentar aquí es que la división no se hace de forma "directa", si no que se superponen unas divisiones a otras tal como se muestra en (*Ilustración 44*). Es decir, hay secciones de la fotografía original que aparecen en dos o incluso cuatro sub-imágenes distintas. Esto se hace así para evitar que ningún objeto identificable quede partido entre dos sub-imágenes y su reconocimiento por parte de la red neuronal quede anulado. La cantidad de espacio que se superponen unas divisiones a otras queda determinada por la longitud máxima del objeto a identificar. Si esto se hace así, dicho objeto nunca quedará partido en todas las sub-imágenes, siempre habrá alguna que lo contenga entero.

La arquitectura de la red neuronal

La última fase del análisis de las fotografías es el presentar las sub-imágenes de 63x63 pixels a una red neuronal que identifica si en ellas aparece algún barco. La red neuronal usada para realizar esta tarea es la propuesta en [PAU1995]. Aquí se describirá la arquitectura de la red que hemos implementado nosotros.

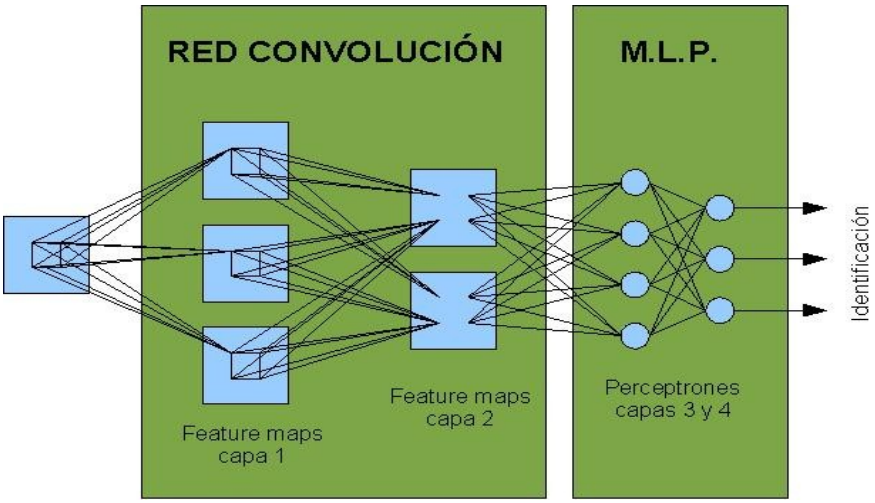


Illustration 44: Arquitectura de una red híbrida formada por una subred de convolución y una subred perceptrón multicapa.

Arquitectura de la red de convolución.

El tipo de red neuronal usada está compuesta de dos partes: una primera red de convolución para extracción de características de la imagen, y una segunda red multicapa para clasificación de patrones. La red de convolución puede consistir de una o varias capas con uno o varios "mapas de características" (feature maps) cada una. Un feature map es una distribución bidimensional de neuronas del mismo tamaño que la imagen de entrada. Cada feature map aplica un filtro distinto sobre cada una de las imágenes almacenadas en los feature maps de la capa anterior (o de la imagen de entrada si se trata de la primera capa). Como se muestra en la figura (Illustration 44) el uso de un filtro consiste en aplicar una pequeña matriz de pesos a cada región de una

imagen almacenada en un *feature map* o presente en la entrada. Los resultados de aplicar estos pesos son sumados y almacenados en el *feature map* correspondiente de la capa siguiente, que puede realizar esta misma operación con varios filtros distintos a feature maps distintos y sumar sus resultados. Es decir, un feature map es una capa bidimensional de neuronas donde todas las neuronas utilizan la misma colección de pesos, que son aplicados a las neuronas de los *feature map* anteriores que ocupan posiciones alrededor de la posición de la neurona del feature map actual. La segunda parte de la red neuronal, la subred multicapa, es una típica red con capas de neuronas unidimensionales y sinapsis completas con las capas anteriores. La entrada de esta red neuronal es la salida de la última capa de la red de convolución.

Concretando a la red neuronal usada en nuestro sistema, su subred de convolución tiene una sola capa con dos feature maps y la subred multicapa cuenta con dos capas de dos neuronas cada una. Los dos *feature map* aplican cada uno un filtro distinto a la imagen de entrada. El tamaño de estos filtros es de 5x5 pixels cada uno. Las dos neuronas de entrada de la subred multicapa están conectadas totalmente a los dos feature maps de la subred de convolución. Las otras dos neuronas de esta subred, las neuronas de salida, están conectadas totalmente a la capa de entrada de la subred multicapa. Esta descripción puede verse más claramente en la *Ilusitración 45*.

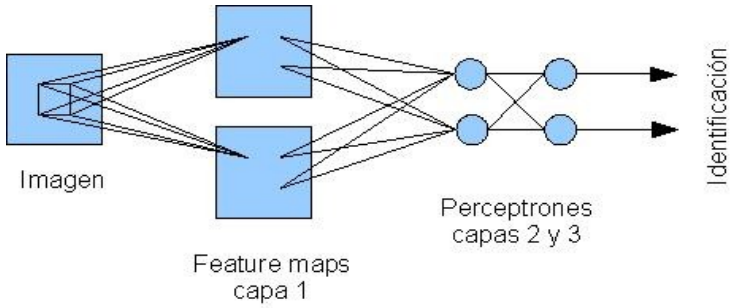


Illustration 45: Arquitectura de la red neuronal usada por nosotros para resolver el problema de identificación de fotografías.

Comportamiento deseado.

Nosotros queremos que esta red neuronal, ante una imagen que se pone como entada, nos diga si en ella aparece un velero como el mostrado en la *Ilustración 35* o no (en las fotografías a tratar el velero se ve desde arriba, como en *Ilustración 40*). La salida de la red será *(1,0)* si en la foto no hay velero, y *(0,1)* si si hay velero. Debido a que estamos usando la función sigmoide en las neuronas, los datos tratados por la red no son ceros y unos, si no valores reales. Esto se traduce en que salidas que cumplan *(>0,5 , <0,5)* se considerarán como que se identifica que en la foto no hay velero, y salidas que cumplan *(<0,5 , >0,5)* se interpretarán como que la red está detectando que hay un barco en la fotografía mostrada. Otras salidas se considerarán no concluyentes.

A continuación se explicará como se ha entrenado a la red y que resultados se han obtenido.

Entreamiento de la red neuronal.

Pasos del entrenamiento.

El entrenamiento de una red neuronal consiste en tres pasos básicos:

1. Creación del conjunto de patrones que es usado para entrenar la red.
2. Selección del tipo de algoritmo de aprendizaje y de sus patrones de entrenamiento.
3. Presentar los patrones de forma iterativa y modificar los pesos de acuerdo al algoritmo de algoritmo de aprendizaje seleccionado.

Creación del conjunto de entrenamiento.

La selección del conjunto de imágenes entrenamiento es un paso muy importante que influye directamente en si la red convergerá o no. Un mal conjunto de patrones puede hacer que una buena red neuronal no aprenda nunca y viceversa, por lo que hay que tener mucho cuidado en este paso. Nosotros hemos usado el conjunto de imágenes mostrado en la tabla (), que son un total de 13 fotografías con barco y 36 sin barco. En la selección de este conjunto se ha tenido gran interés en que representen lo mejor posible el tipo de fotografías que simbolizan. En las fotos con barco se han incluido ejemplares en donde el barco aparece con distintos niveles de rotación y en distintas posiciones de la imagen. En la otra colección de fotos, se han incluido todo tipo de objetos que no deben ser confundidos con el velero, como secciones de tierra, aviones desde distintas perspectivas y veleros que aparecen seccionados en el borde de la fotografía. En donde se ha hecho más hincapié con este conjunto de imágenes, ha sido en que la red no confunda aviones, que están volando por debajo de la cámara, con el barco que se busca.

Backpropagation usado y sus parametros.

Respecto al tipo de algoritmo usado para entrenar la red, se han probado numerosas mejoras y variaciones de backpropagation hasta dar con la que producía mejores resultados. El método que finalmente se usó fue backpropagation puro con momento de inercia, todas las demás modificaciones a este algoritmo probadas empeoraban notablemente la velocidad de convergencia o incluso hacía que la red no convergiera nunca. La razón de esto es que las velocidades a las que aprenden cada una de las partes de la red neuronal usada deben guardar unas relaciones muy concretas las unas con las otras, si esto no se cumple, la red no convergerá. Los parámetros usados para conseguir esta exacta relación han sido: “learning rate” de 5 y “momentum rate” de 0,2 para los filtros aplicados en los *feature maps*; $5 \cdot 10^{-4}$ y $2 \cdot 10^{-5}$ respectivamente para los pesos que conectan la red de convolución con la red multicapa; y 0,05 y 0,002 para los pesos entre las dos capas de la red multicapa. Pequeñas modificaciones a estos valores empeoran en la mayoría de los casos el comportamiento del aprendizaje. Por último, la selección de los pesos iniciales se hizo como es aconsejada en [ROJ1996]. Los pesos se eligen aleatoriamente con una función de probabilidad constante dentro de un margen *(-a,+a)*, dependiendo de la red en cada caso. En el nuestro el entrenamiento mostraba un mejor comportamiento con valores seleccionados de el segmento *(-1,+1)*.

Entre los otros algoritmos que se probaron y se descartados están el Rprop, que es una mejora de backpropagation explicada en [ROJ1996]. Se comprobó que este método producía resultados notablemente mejores para el caso de redes multicapa, pero sin embargo, en nuestra red muestra resultados catastróficos, saturando en apenas un par de épocas las neuronas de los feature maps. Esto seguramente es posible a que este algoritmo varía de forma dinámica la velocidad aprendizaje de cada peso individualmente, por lo que el necesario y frágil equilibrio de velocidades descrito en el párrafo anterior queda roto. Otra mejora a backpropagation usada a menudo, es poner un valor mínimo a la derivada de la función sigmoide que se calcula en la fase

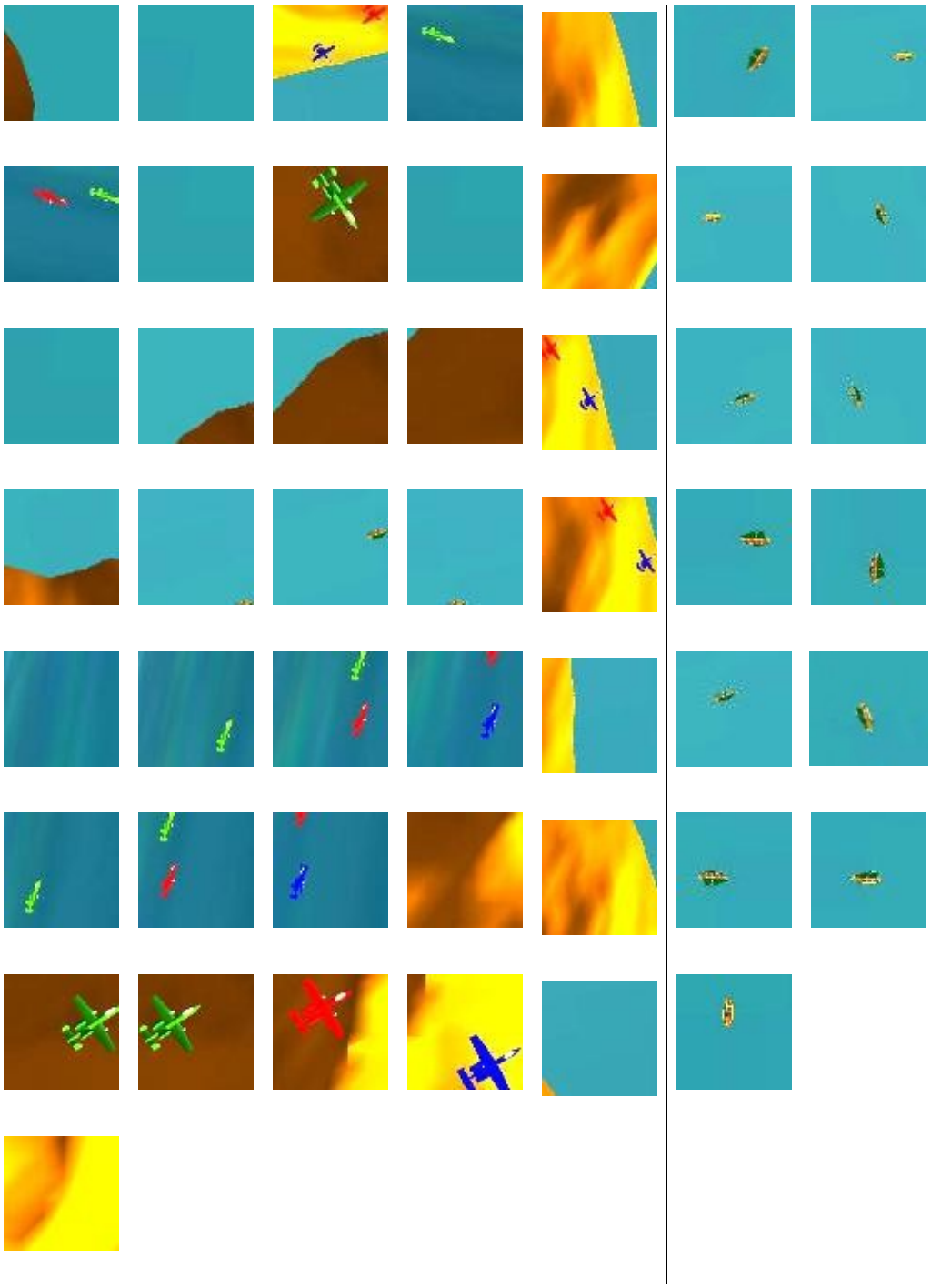


Table 1: En esta tabla se muestra la totalidad de imágenes presentes en el conjunto de entrenamiento usado.

"feed backwards" (por ejemplo un mínimo de 0,01), con la intención de sacar rápidamente a la red de las zonas planas de la función de error durante el entrenamiento. Sin embargo, probablemente por la misma razón que la supuesta para el Rprop, este método impedía que la red convergiera. Por último también se probó el sumarle una pequeña constante al cálculo de la derivada de la sigmoide, pero el aprendizaje incluso iba más lento con esta mejora que normalmente funciona en otras redes neuronales.

Por último, la presentación de los patrones de entrenamiento a la red es realizada siguiendo el siguiente algoritmo:

Presentación de los patrones.

```
C = conjunto de fotografías con barco;
S = conjunto de fotografías sin barco;
e = número de épocas;
FOR i < ( e * (tamaño de C + tamaño de S) )
     Entrenar la red con un patrón seleccionado aleatoriamente de C;
     Entrenar la red con un patrón seleccionado aleatoriamente de S;
     i++
END FOR
```

La razón de que se presente en cada iteración un patrón positivo (con barco) y otro negativo (sin barco), es que la red no tienda a aprender a decir siempre la misma respuesta. Además, así el entrenamiento se vuelve independiente de la diferencia en número entre las fotos con barco y las sin barco presentes en el conjunto de entrenamiento. Si no se hiciera así y hubiera muchos más patrones negativos, por ejemplo, seguramente la red aprendería a decir siempre que no hay ningún barco en la foto, le muestres la imagen que le muestres.

Resultados de la red.

Convergencia del entrenamiento.

El comportamiento de esta red neuronal se mostrará exponiendo primero como ha convergido el entrenamiento realizado, después indicando cuantas imágenes han identificado correctamente tras el entrenamiento, y por ultimo viendo como respondió a un conjunto de imágenes de test que nunca se le habian presentado con anterioridad.

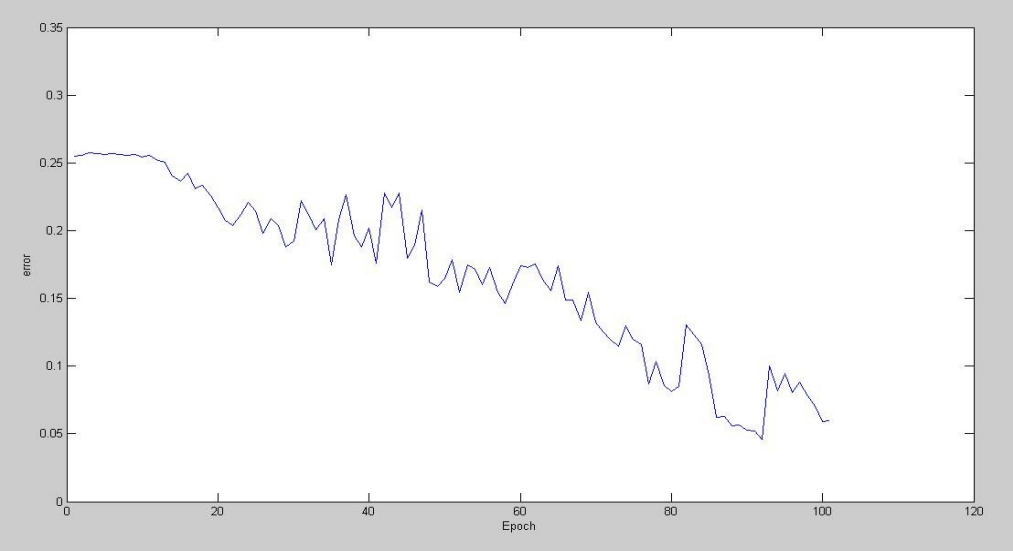


Illustration 46: Evolución del error cuadrático medio durante el entrenamiento de la red.

Nuestra red neuronal, con el conjunto de patrones de aprendizaje mostrados en () y con los parámetros indicados con anterioridad, suele comenzar el proceso de aprendizaje con un error cuadrático medio sobre el conjunto de entrenamiento de entre *0,20* y *0,40*. El entrenamiento lo prolongamos durante 100 épocas, es decir, a la red neuronal se le presentaron fotografías unas 9.800 veces, y este proceso tardó alrededor de 20 minutos con un AMD Athlon 3200+. La convergencia del error cuadrático

puede observarse en *Ilustración 46*. Aunque en la imagen puede apreciarse que el error podría haber seguido disminuyendo, no se continuó con el entrenamiento para evitar la posibilidad de que la red acabara memorizando las fotografías, cosa que puede ocurrir perfectamente en los pesos que conectan la red de convolución con la red multicapa. En otros entrenamientos realizados con esta misma red neuronal pero realizados con más épocas, la red acabó reconociendo todas las fotografías del conjunto de entrenamiento, con un error del orden 10^{-6}, pero después no conseguía reconocer casi ninguna imagen nueva que se le presentase y que no estuviera en dicho conjunto.

Resultados con el conjunto de entenamiento.

Tras el entrenamiento, la red mostraba un error de 0,059. De las 36 fotografías del conjunto de entrenamiento sin barco, la red identificaba las 36 como negativas (es decir, que efectivamente no aparecía el velero en ellas). De las 13 imágenes con barco de este mismo conjunto, todas ellas daban un resultado positivo (es decir, la red identificaba que en ellas aparecía el velero). En resumidas cuentas, la red identificada 36 de 36 y 13 de 13 fotografías sin barco y con barco correctamente.

Acertadas / Total	***Fotos sin barco***	***Fotos con barco***
Conjunto de entrenamiento.	*36 / 36*	*13 / 13*
Conjunto test.	*20 / 20*	*10 / 10*

Table 2: Número de identificaciones correctas de la red neuronal sobre los conjuntos de fotografías usados.

Resultados con un conjunto test.

Como es usual cuando se trabaja con redes neuronales, tras el entrenamiento también se le presenta a la red un conjunto de fotografías que no halla visto nunca. El conjunto de test que nosotros usamos es el mostrado en *Ilustración 47*, que está compuesto por 10 fotografías con barco y 20 sin barco. De este conjunto también identificó correctamente todas las imágenes. En la tabla 2 se muestran estos resultados.

Y esto, ¿Porque funciona?

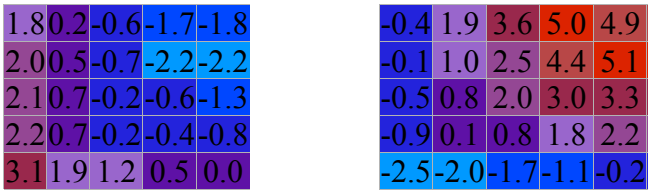


Illustration 47: Valores de los filtros usados por los dos feature maps tras el entrenamiento.

Una pregunta interesante que suele surgir al trabajar con redes neuronales, y más cuando consigues que una tal red finalmente funciones es, ¿Que es lo que está haciendo realmente la red neuronal?, ¿Como es capaz de realizar tan bien esta operación para la que la he entrenado?. En nuestra red neuronal, puede que ver como son los filtros que la red de convolución ha hallado vierta un poco de luz sobre este asunto. Dichos filtros son los mostrados en *Ilustración 47*. Parece que ambos filtros se han especializado en la tarea de detectar esquinas, el primero de ellos en la parte inferior izquierda, y el otro en la parte superior derecha del filtro. El filtro de la derecha, además, probablemente también es capaz de detectar bordes rectos en la parte inferior del mismo y en la parte izquierda. También ambos filtros parecen ser complementarios el uno con respecto al otro, pues donde uno tiene pesos de valor más alto, el otro tiene los pesos de valor menor.

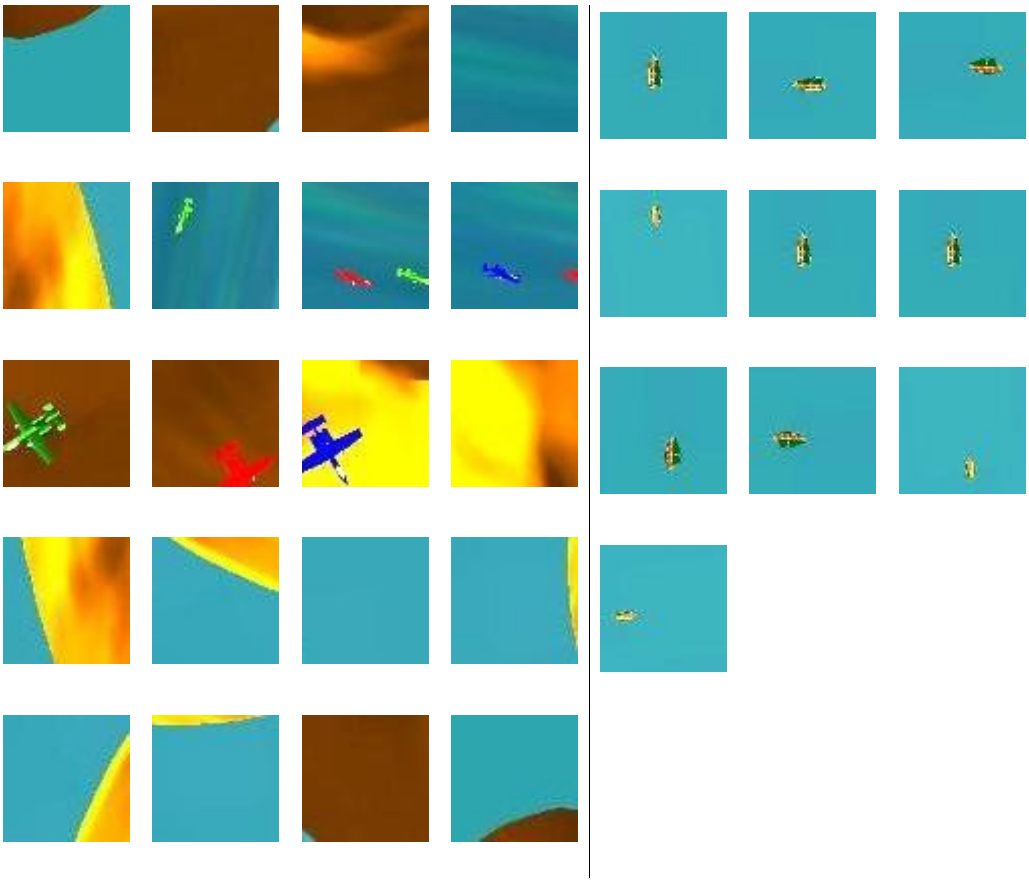


Table 3: Colección de fotografías usadas para testear si la red neuronal había aprendido. Estas imágenes no fueron usadas en el proceso de entenamiento de la red.

2.5 Control de vuelo (vuelo en formación)

En este tema se explicará como se ha implementado el control de vuelo de los agentes-avión. En el estado común de la tarea *StateSetFlight*. Este es un estado que puede ser usado por cualquier tarea que lo invoque, pasando a formar parte de la máquina de estados que define dicha tarea. Las tareas que usan este estado son *TaskFlyToObjective* y *TaskFollowBoss*.

Cada una de las tareas mencionadas resuelve un problema distinto dentro del ámbito del control de vuelo. La primera de ellas maneja al avión para hacer que este alcance un objetivo en el espacio, normalmente lejano y estático. La otra tarea maneja al avión para que este siga al jefe de la formación, considerando su posición dentro de la formación como un objetivo que se mueve con el tiempo. A continuación se enunciarán con mayor detalle estos dos problemás.

El problema.

Definición del problema

Como decíamos antes, el control del avión puede dividirse en dos problemás distintos. El primero de ellos es como conseguir posicionar al avión en un puto estático en el espacio: el avión se encuentra inicialmente en un punto P_1 y queremos que vaya a otro punto P_2 que no se mueve en el espacio. De aquí en adelante llamaremos a esto problema *Problema de Alcance*. El segundo problema puede definirse de forma idéntica al anterior, con la diferencia de que ahora el punto P_2 también se mueve al igual que el avión. En este segundo caso nuestro avión debe procurar situarse en todo momento lo más cerca posible de dicho punto objetivo, por lo que debe tener también en cuenta su velocidad para predecir su posición futura y adelantarse a sus movimientos. A este último caso lo denominaremos *Problema de Seguimiento*. Puede observarse que el problema de seguimiento puede considerarse una generalización de el problema del alcance, siendo este último el caso particular para el que la velocidad del objetivo móvil es cero.

Vuelo en formación

Una de las contribuciones más importantes del proyecto es el control de formación. Esto es, como controlar a los aviones para que vuelen siguiendo una formación fija. Como muchos se habrán dado cuenta, el control en formación puede ser reformulado en función de el *Problema de Seguimiento*. Si contamos con un algoritmo que resuelva el Problema de Seguimiento de forma eficaz, entonces podemos usar este mismo algoritmo para general el vuelo en formación. Tan solo tenemos que pasarle como objetivo la posición que debe ocupar el avión dentro de la formación de vuelo, la cual se mueve con el tiempo junto con el resto de la formación. Si, por ejemplo, la posición del avión dentro de la formación está definida en función de la actual posición y orientación de el jefe de formación, entonces, al moverse el jefe también se mueven todas las así definidas posiciones de la formación y, con ellas, todos los aviones que estén ejecutando el algoritmo de control de vuelo.

En la formación, los aviones deben seguir sus respectivas posiciones de la formación en todo momento, adelantándose a los posibles cambios en su dirección y

velocidad. Esto podría implementarse mediante una comunicación regular con el jefe de formación: Cada vez que el jefe de formación fuera a realizar alguna maniobra o un cambio de velocidad, avisaría por radio al resto de agentes. Esto tiene numerosos

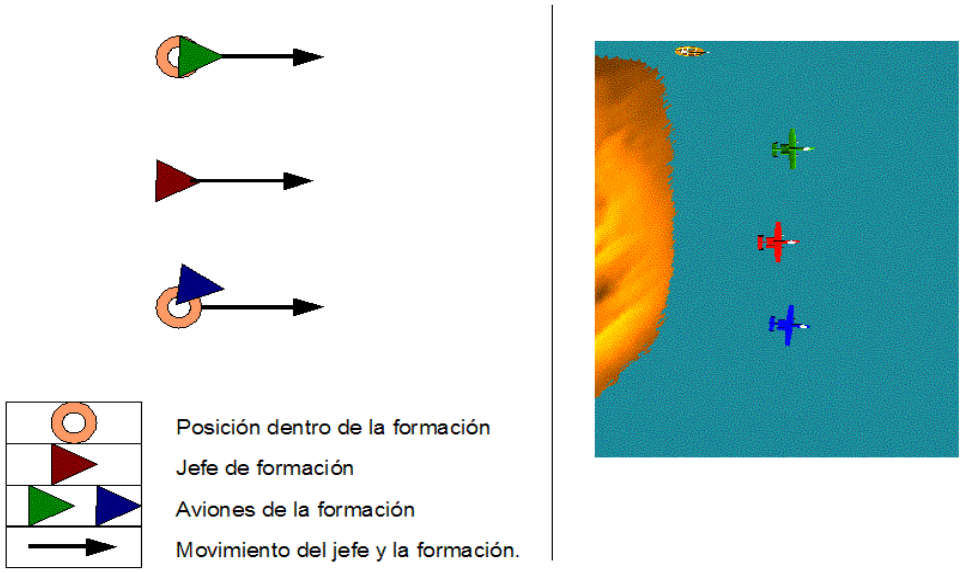


Illustration 48: Funcionamiento de la formación de vuelo. El control de vuelo en formación es reformulado con el Problema de Seguimiento. En la imagen izquierda puede verse un diagrama de dicha interpretación, y en la de la derecha el control de formación funcionando en la simulación.

inconvenientes. Por una parte el jefe tendría que generar mensajes de radio muy amenudo, con los problemás de sobrecarga que esto conlleva. Por otra parte las probables interferencias podrían deteriorar en gran medida este mecanismo. Otra posible implementación que no se apoya tanto en la comunicación directa sería una basada en radar. Los agentes podrían verse unos a otros a través del radar, realizando comunicaciones tan solo para programar las maniobras complejas. Seagull ha optado por esta segunda solución, que será explicada a continuación.

La solución.

Para solucionar el *Problema de Seguimiento* (y por tanto el *Problema de Alcance* y el vuelo en formación) se ha optado por implementar un estado que desarrolla un algoritmo de control adecuado. Dicho estado, como se ha mencionado antes, puede ser usado como parte de la máquina de estados de cualquier tarea que requiera solucionar alguno de estos problemás. El estado es denominado en el código *StateSetFlight*.

Control de vuelo

El control de vuelo está dividido en dos capas. Una de ellas cambia de estrategia de control según sean las condiciones del entorno de vuelo, permaneciendo la otra siempre igual. La capa estática usa técnicas típicas de la Teoría de Control clásica, sin embargo la capa dinámica usa técnicas algo más cercanas al campo de la Inteligencia Artificial, por lo que el resultado final es un control mixto que intenta aprovechar las ventajas de cada uno de estas disciplinas. En el diagrama Illustration 48 se describe esta estructura de control.

Capa estática

La capa estática usa un controlador denominado RCAM, que controla un modelo dinámico del comportamiento de un avión a través de un denominado control lateral y un control longitudinal. Este módulo ha sido implementado por el departamento DACyA de la Universidad Complutense de Madrid, y su descripción detallada puede encontrarse en [JES2005]. No nos detendremos a describirlo internamente, ya que tan solo su uso en el marco de otro programa, y no su implementación, ha sido una de las aportaciones de los autores de Seagull. Su comportamiento como una caja negra es que recibe la dirección y velocidad deseados y, a partir de ellos, calcula el control que debe ejercer sobre el modelo interno del comportamiento del avión para conseguir que este adquiera dicha velocidad y dirección.

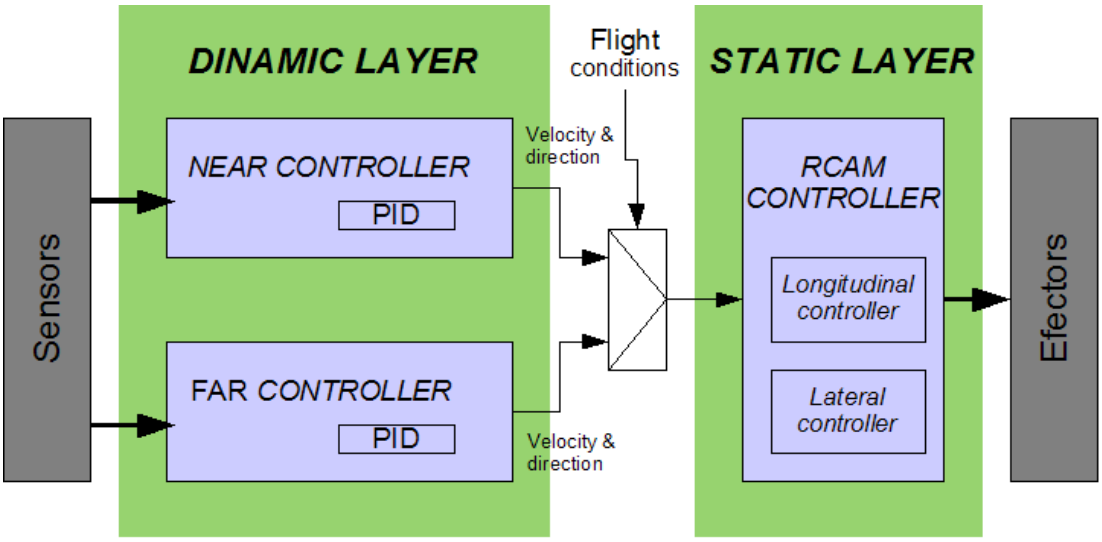


Illustration 49: El control de vuelo está dividido en dos capas, una dinámica y otra estática. La capa dinámica cambia según las condiciones de vuelo. La capa estática siempre es la misma.

Capa dinámica

La capa dinámica se ocupa de calcular la velocidad y dirección que debe pasarle al RCAM para conseguir un comportamiento más complejo que el que se conseguiría si solo se usara el RCAM. El cálculo de dichos parámetros cambia según las condiciones de vuelo. Si el avión se encuentra más cerca del objetivo que una distancia límite, entonces se usará el denominado Control Cercano. Si el avión se encuentra más lejos que esta distancia, se usará el Control Lejano. La diferencia de controlador son debidas al diferente balance de importancia entre la posición del objetivo y su movimiento según este se encuentre cerca o lejos del agente, pues según se acorta la distancia entre ambos cobra mayor relevancia el movimiento del objetivo.

El funcionamiento del control lejano es muy sencillo. Tiene almacenadas dos posiciones antiguas del avión y otras dos del objetivo que está siguiendo, llamemos P_{1A} y P_{2A} a las dos primeras y P_{1O} y P_{2O} a las otras dos respectivamente. Cada cierto tiempo, el controlador lee la actual posición del avión (P_A) y del objetivo (P_O) y calcula la distancia de cada una con P_{1A} y P_{1O} respectivamente. Si, por ejemplo, la diferencia $\|P_A - P_{1A}\|$ supera un valor humbral, entonces modificará las posiciones almacenadas con:

$$P_{2A} = P_{1A}$$
$$P_{1A} = P_A$$

y esto mismo se hace con P_{1O} y P_{2O}. El objetivo de estas operaciones es conseguir que los vectores $\vec{P}_1 - \vec{P}_2$ no tengan un valor demásiado cercano a cero. Si no se hiciera, los operandos de las operaciones que se describirán a continuación serían de ordenes de magnitud demásiado dispares, surgiendo resultados no deseados.

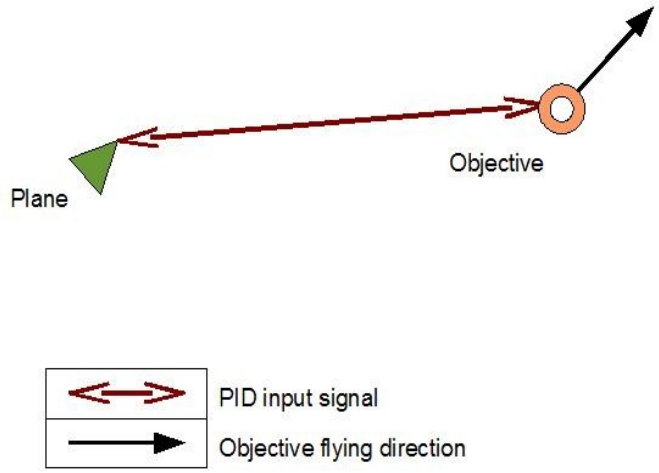


Illustration 50: La distancia entre el objetivo y el avión es introducida a un PID para calcular la velocidad que se le pasará al RCAM.

Una vez se ha calculado con los vectores $\vec{P}_1 - \vec{P}_2$, estos son considerados una aproximación de la velocidad que están siguiendo el avión y el objetivo. Dicho valor es introducido a un PID discreto que trata dicho escalar y sus anteriores valores como derivada de la señal $\vec{P}_O - \vec{P}_A$. La señal $\vec{P}_O - \vec{P}_A$ es la proporcional del PID, y la integral y la suma de todos los anteriores valores de esta "señal". La respuesta que el PID calcula de esta menara será luego usada como la señal de velocidad que se le pasará al RCAM. En el diagrama Illustration 50 se muestra el significado geométrico de los cálculos anteriores.

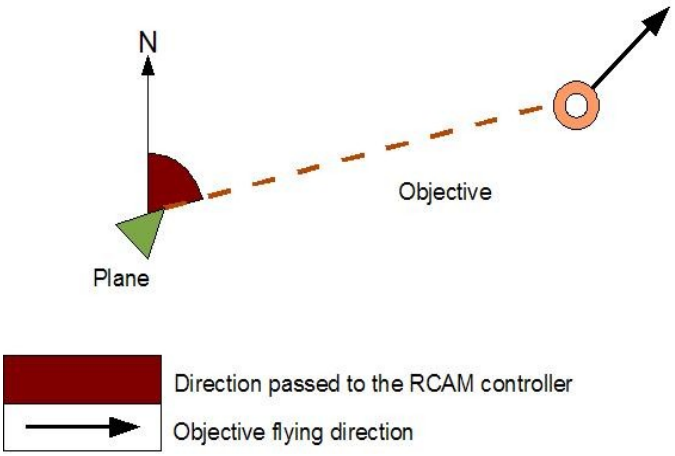


Illustration 51: Al RCAM se le pasa como dirección la dirección relativa hacia la que se encuentra el objetivo.

Respecto a la dirección que se le pasará al RCAM, esta es calculada de manera directa. Al RCAM se le pasa la dirección a la que se encuentra el objetivo en relación con la posición actual del avión. Para calcular dicha dirección, se usan los vectores P_{1A} , P_{2A} , P_{1O} y P_{2O} . En la figura *Ilustración 51* se muestra su significado geométrico.

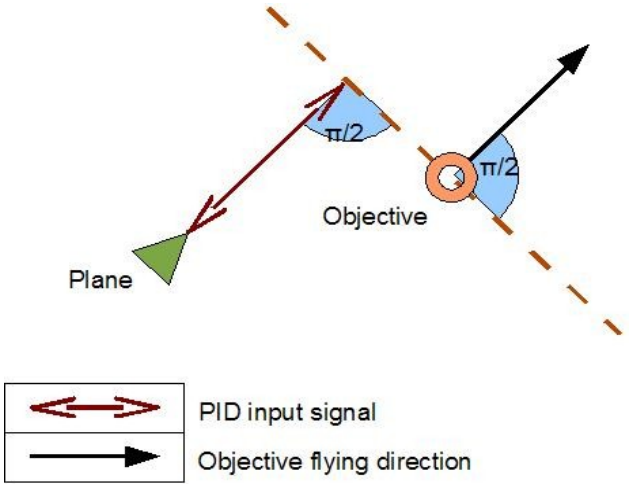


Illustration 52: En el control cercano, como entrada del PID se usa la separación entre la posición del avión y la perpendicular del objetivo respecto a su dirección de movimiento.

Respecto al control cercano, este funciona de manera similar pero más compleja que la del control lejano. Para calcular la velocidad que le pasará al RCAM, realiza los mismás operaciones con los vectores P_{1A} , P_{2A} , P_{1O} y P_{2O} , pero ahora P_{1O} y P_{2O} no son las posiciones del objetivo mismo. En su lugar, lo que toma es

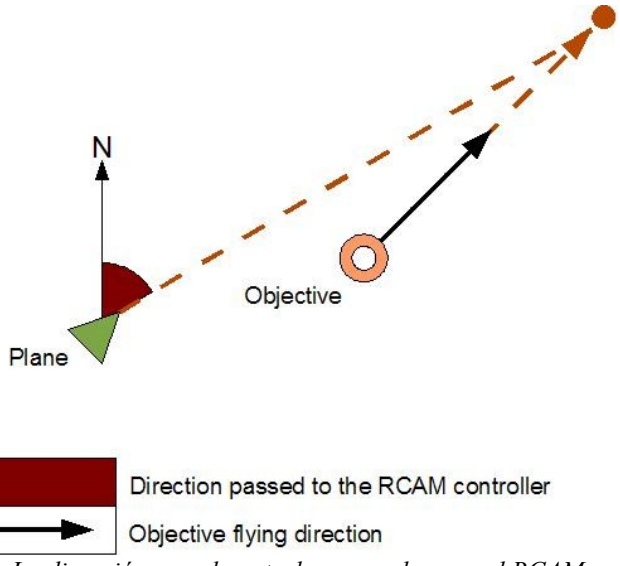


Illustration 53: La dirección que el control cercano le pasa al RCAM es la relativa a un punto bastante más avanzado que el objetivo.

el punto paralelo al objetivo que se encuentre justo enfrente de él, considerando como "enfrente" la dirección hacia la que avanza dicho objetivo. Al igual que antes $\vec{P}_1 - \vec{P}_2$ será lo que el PID considerará como la derivada de $\vec{P}_O - \vec{P}_A$, que es el

valor proporcional del PID, siendo la integral la suma de todos los anteriores valores de $\vec{P}_O - \vec{P}_A$. El significado geométrico de la resta $\vec{P}_O - \vec{P}_A$ puede observarse en 84.

Por último la dirección que el control cercano le pasa al RCAM es la dirección relativa de un punto bastante más avanzado que el objetivo. Esto se puede ver en el diagrama 82.

El PID usado en los dos controles es un PID con valor máximo de saturación en el termino integral y con un valor umbral para el cómputo de este mismo termino. La integral de la señal no podrá superar el mencionado valor de saturación. Al mismo tiempo tan solo señales de error por encima del valor umbral son procesadas y sumadas al valor actual del término integral.

2.6 Comunicación entre agentes

Pese a que JADE incluye una plataforma FIPA que nos podría facilitar la comunicación entre agentes, la idea fue desde un principio que el sistema se ajustara lo más fielmente posible a la realidad. Con este fin, se implementó sobre cada agente una radio que le permitiera comunicarse con los demás agentes de su entorno.

El proceso de comunicación es el siguiente: un agente que esta interesado en trasmitir información a los demás, crea un *RadioMessage* y lo transmite al entorno en una determinada frecuencia. El entorno queda, por tanto, definido por un buffer de *RadioMessages* a distintas frecuencias. Los posibles agentes receptores, es decir, aquellos que estén dentro del alcance de la radio del emisor tomaran el mensaje, correspondiente a la frecuencia de su propia radio. De esta forma dos agentes quedaran comunicados empleando una frecuencia común.

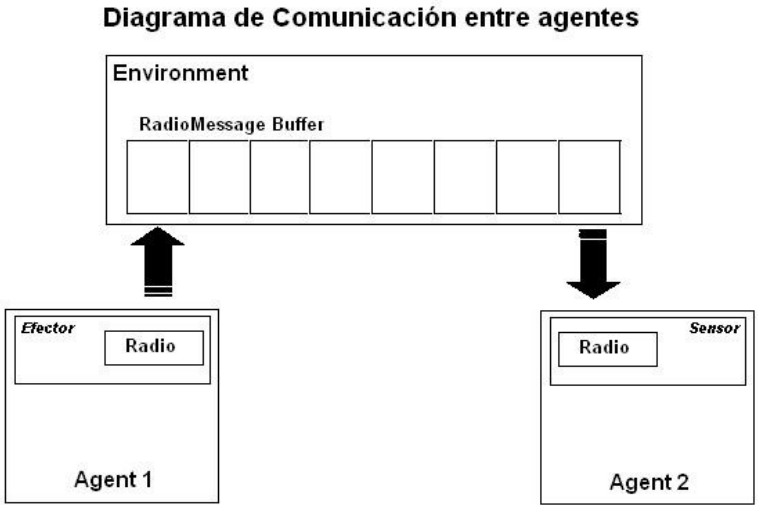


Illustration 54: Sistema de comunicación entre los agentes.

Como se puede apreciar en la *Ilustración 54*, es necesario distinguir dos tipos de interpretaciones de la radio según la tratemos como efector o como sensor. Esto no solo repercute conceptualmente, sino también a la hora de la implementación. Debido a la estructuración del código del proyecto, los sensores y efectores quedan distribuidos en paquetes diferentes. Esto implica que la Radio queda definida mediante dos clases: una en el paquete sensores, implementando la funcionalidad como sensor de la misma, y otra en el paquete efectores que, análogamente, implementa su funcionalidad efectora.

La unidad mínima de comunicación*: RadioMessage*

En toda comunicación se hace necesario que tanto el emisor como el receptor conozcan cual es la estructura del mensaje. En nuestro caso, la determinación de dicha estructura se ciñó a lo que comúnmente caracteriza a un mensaje de radio: una frecuencia y un contenido. Para no limitar las posibilidades de comunicación, el contenido de un mensaje no se limita a una cadena de caracteres o *String*, sino que puede contener cualquier *object*. Así el contenido puede tratarse desde un simple mensaje de texto, hasta una imagen tomada por la cámara incorporada a los agentes. El siguiente diagrama muestra la estructura de un mensaje:

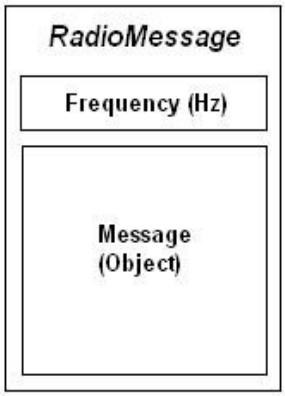


Illustration 55: Estructura de un mensaje de radio.

2.7 Simulación de vuelo

A la hora de crear la interfaz de simulación de vuelo se generaron los siguientes retos:

1. Generación de posiciones en tiempo real.
2. Creación de una cámara de seguimiento que siguiera con una perspectiva cenital al lider del grupo.
3. Captura de imágenes para reconocer patrones en una red.

Simulación en tiempo real

Mecanismos para el tiempo Real

A la hora de utilizar una aplicación que simule el tiempo real, es necesario estar provistos de un mecanismo que nos permita actualizar los valores cada vez que estos pierden su validez.

Para ello Java3D nos provee de unas herramientas llamadas "Behavours" (comportamientos). Los comportamientos son trozos de código que se ejecutan cuando un evento los ha desencadenado. Estos eventos o estímulos pueden ser desde que un elemento ha sido clickeado o que hayan pasado un cantidad de frames o de segundos.

En nuestro Proyecto, necesitábamos actualizar tanto los sensores como los actuadores del sistema cada cierto tiempo, a la par que necesitábamos nuevos valores de rotación y posicion para cada uno de los aviones. Es por ello que hemos creado la Clase AvionBehavour, que cada 20 milisegundos actualiza el sistema entero.

Preprocesamiento vs Tiempo Real

Comparándolo con la aplicación de [EST2006], aunque en un principio podría parecer que precalcular la animación y luego visualizarla tendría menos carga para el procesador, todo lo contrario. El sistema en tiempo real está mucho menos saturado y tiene una mayor capacidad de respuesta ante eventos asíncronos no esperados del sistema.

Esto es debido a que la función alfa, aquella que nos sirve para generar puntos intermedios y crear la animación, gasta muchos recursos de sistema, y por eso el tiempo real sobrecarga menos la CPU, aunque a priori parezca otra cosa.

Uso de una cámara seguimiento

Motivación cámara de seguimiento

Cuando se planteó la necesidad de la creación de una cámara que siga al jefe con una perspectiva cenital, se esgrimieron las siguientes razones:

- Comodidad de que la cámara siga a los aviones de cerca.

- Necesidad para depurar las formaciones desde un buen punto de vista
- Comprobar si los aviones van por la ruta indicada.

Aunque para este último punto, también se ha creado un radar, el resto de razones justificaban plenamente la necesidad de que la cámara siga al jefe de la escuadra.

Para ello nos seguimos apoyando en nuestra clase "AvionBehavour" para que siempre que el avión cambie de posición, la cámara esté encima de él mirando hacia abajo. Aunque pueda sonar fácil, los mecanismos de coordenadas y de generación de puntos de vista con la cámara no son baladí. Un pequeño problema en las escalas o un desajuste a la hora de multiplicar los vectores nos dan una distorsión del comportamiento esperado de la cámara; además de la necesidad de anular los efectos del ratón sobre la misma.

Capturas de imágenes.

Necesidad de Captura de imágenes

Una parte indispensable para el salvamento es el reconocimiento de los barcos. Para ello es necesario implementar en el sistema algún modo mediante el cual se puedan tomar fotos de lo que hay debajo del avión sin que el rendimiento caiga o que se note algún parpadeo. Para entender un poco las aproximaciones antes tenemos que explicar un poco como funciona el renderizado en java3D.

Explicar cómo se renderiza

Cuando diseñamos una escena, siempre vamos a tener una serie de objetos, el sistema de coordenadas que los guía y sus dependencias respecto a otros objetos y sistemás de coordenadas del sistema. Al conjunto de todos estos elementos se le llama "grafo de escena".

El grafo de escena es la representación de lo que realmente está ocurriendo en nuestro sistema.Para que sea visualizable necesitamos un objeto que nos permita saber las características de dónde queremos que se visualiza. Ese objeto es el Canvas3D.

Canvas3D existen de dos tipos: los que están en primer plano y los que están ocultos. La diferencia sustancial entre el primero y el segundo es que el primero ha de estar adjunto a un contenedor que se vea por pantalla (un JPanel, por ejemplo) y renderiza siempre y cuando ese objeto esté visible en el monitor. Sin embargo, los que están ocultos, no pueden estar adjuntados a ningún contenedor y sólo renderizan si explícitamente se les obliga a ello. el resultado de ese renderizado solo queda almacenado en un buffer de memoria intermedio.

TipoCanvas	*¿Adjunto a contenedor?*	*¿Cómo renderiza?*
Primer Plano	*Necesario, si no, no renderiza*	*Si está visible*
Segundo Plano	*Si se adjunta da error en ejecución*	*A petición del sistema*

Implementaciones probadas

Pues bien, a la hora de tomar fotos el principal problema es que la cámara ha de posicionarse debajo del avión mirando hacia abajo, guardar la imagen en algún lado (ya sea memoria o disco duro) y volver a su posición inicial. Para ello tenemos 3 posibilidades de implementación.

1. **Tener dos grafos de escena idénticos con distintos tipo de Canvas3D:** uno de ellos tendría un Canvas3D en primer plano, que es el que veríamos por pantalla y el otro estaría en segundo plano.
   Por desgracia, este método es demásiado complejo de implementar, además de que habría muchos problemás de rendimiento del sistema al duplicar el grafo de escena y tener que estar sincronizados

2. **Un único grafo de escena con un Canvas3D en segundo plano:** Estaríamos todo el rato forzando el renderizado y pasaríamos el dibujo que haya generado al JPanel que lo visualizara. Cuando se tomásen fotos, lo que haríamos no pasar esa foto al JPanel con lo cual no se vería parpadeo ninguno.
   Este sistema tenía el problema de que parpadeaban las imágenes, ya que al no incorporar un sistema de Doble Buffer de escritura como swing, la animación se veía parpadeando

3. **Un único grafo de escena con un Canvas3D en segundo plano y otro en primer plano:** Aprovechando que Java3D permite tener varios Canvas de visualización en el mismo grafo de escena, podemos tener un Canvas3D que se estuviera siempre ejecutando y que cuando tocase hacer una foto dejase de renderizar, se cambiase la cámara al punto deseado, se tomáse la foto y se volveria la cámara a su punto original y que continuara.

   En un principio parecía la solución ideal pero aun así teníamos un problema de rendimiento. Si el sistema pedía que las fotos fuera a hacerlas demásiado rápido el almacenar cada una e un archivo de disco duro suponía demásiado tiempo en entrada salida y el sistema empezaba a dar parones y tirones.

   Este problema no ha supuesto inconveniente ya que el sistema está preparado para tomar una foto cada diez segundos y que no se guarde en disco duro, si no que se pasaría directamente a la red neuronal, lo cual evita todo el problema.

2.8 La interfaz de usuario

Esta sección tiene como objeto mostrar todas las posibilidades que el sistema ofrece al usuario. Aunque el manejo de la interfaz del sistema es bastante intuitivo, es necesario realizar algunas aclaraciones de uso. La interfaz del sistema está formada por tres ventanas:

Ventana inicial o de presentación: *InitialFrame*

Como su nombre indica, esta ventana es una ventana de tránsito, cuya misión es la de presentar el proyecto al usuario. En este punto el usuario dispone de dos opciones: *Configurar la Simulación* o *Salir* del sistema.

Illustration 56: Ventana inicial de presentación

Ventana de configuración: *ConfigurationForm*

La ventana de configuración es la encargada de establecer cuales serán los parámetros de la simulación.

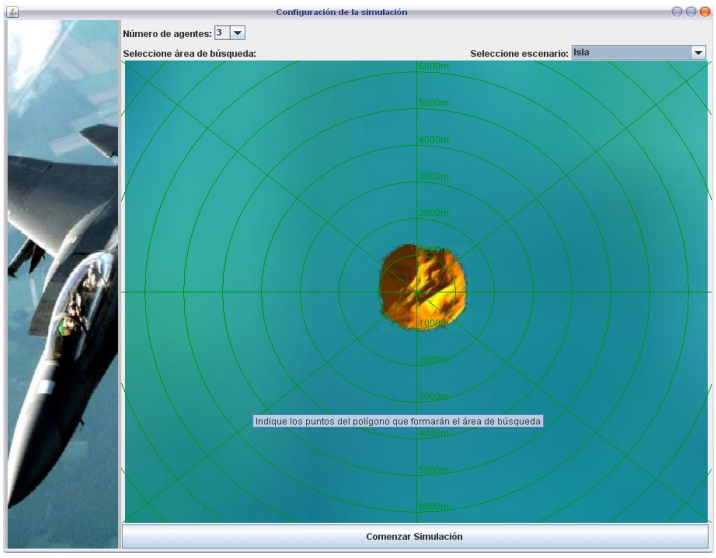


Illustration 57: Ventana de configuración

Con un simple vistazo sobre la ventana observamos que los tres aspectos fundamentales de configuración son los siguientes:

- ***Número de agentes***: Por defecto este número será tres, aunque puede hacerse variar entre 1 y 3. El usuario debe tener en cuenta que siempre uno de los agentes debe ser el que cumpla la misión de jefe(*Boss*) de escuadrón. De este modo, el primer agentes creado siempre tendrá el rol de jefe, mientras que el resto tendrán el rol de buscador.

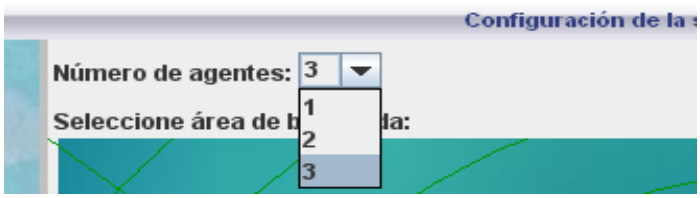


Illustration 58: Selección del número de agentes

- ***Escenario***: El usuario tiene la opción de elegir el entorno de simulación sobre el cual se desarrollará la misión. En esta primera versión únicamente está implementado el escenario de *La Isla*, pero en futuras ampliaciones el número de escenarios aumentará.

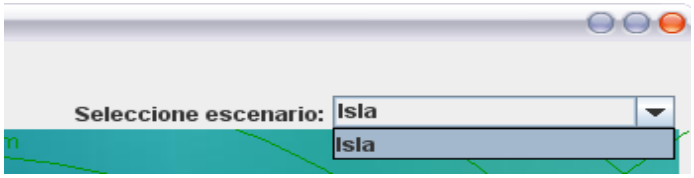


Illustration 59: Selección del escenario

– ***Mapa de selección del área de búsqueda:*** Esta es sin duda la parte más trascendente de la configuración. Sobre este mapa se debe indicar cual será el área de búsqueda dónde los agentes tendrán que trazar su ruta para localizar objetivos. Para ello, el usuario debe ir pulsando sobre el mapa aquellos puntos que formarán el polígono que compone el área de búsqueda. Para que no existan errores a la hora de introducir los puntos, estos deben ser añadidos de izquierda a derecha y de arriba abajo. Las siguientes ilustraciones muestran cómo se realizaría la introducción de los puntos para un área de búsqueda cuyo polígono está formado por cinco vértices.

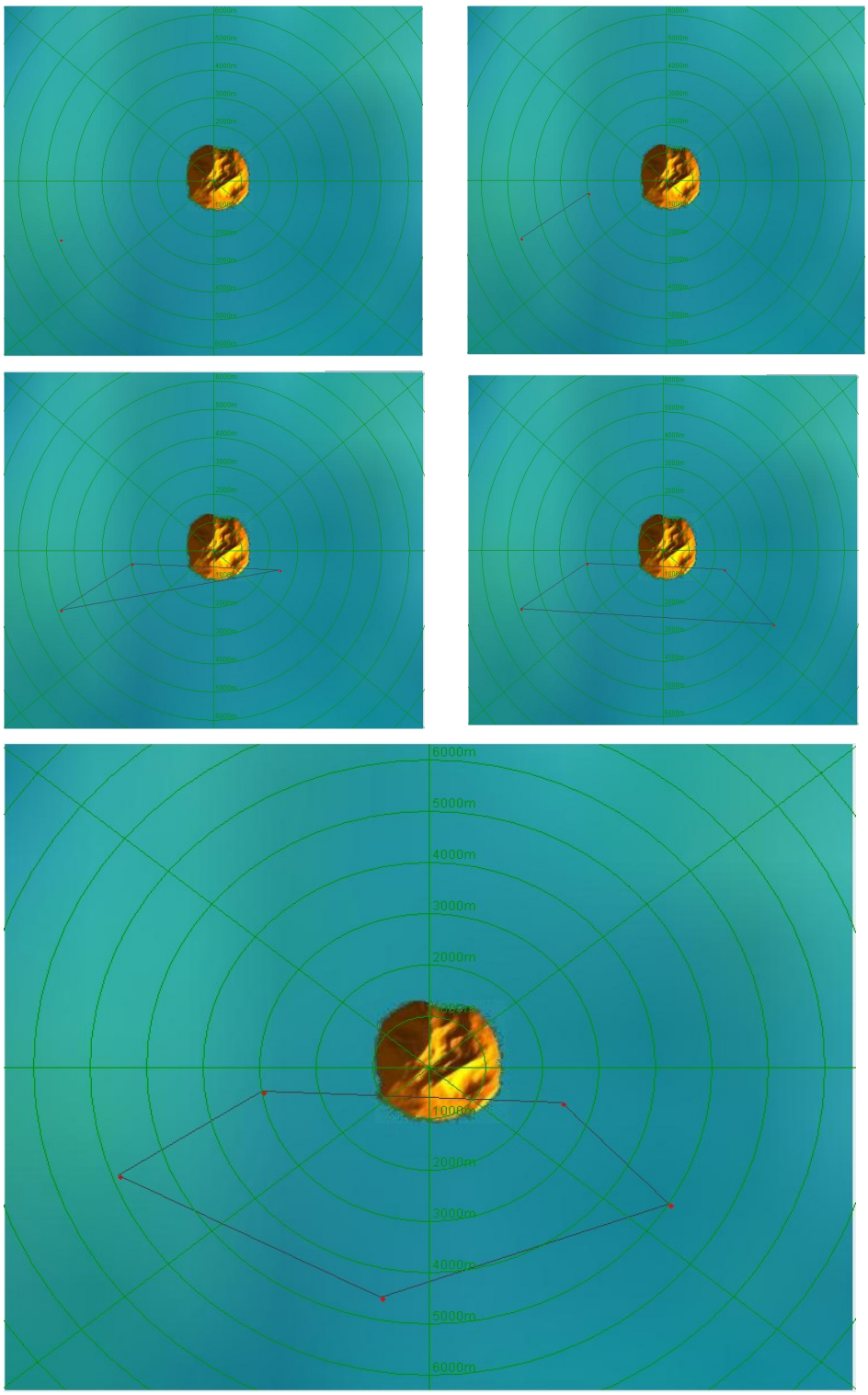


Illustration 60: Selección del area de búsqueda.

Una vez tengamos configurada la simulación, el siguiente paso será pulsar el botón de *Comenzar Simulación* situado en la parte baja de la ventana.

Illustration 61: Botón para comenzar la simulación.

Ventana de Simulación: *SimulationForm*

Nos encontramos ante la ventana principal del sistema. Sobre ella, el usuario podrá visualizar la simulación en 3D, controlar los parámetros de los agentes, verificar el comportamiento de los agentes en base a sus tareas, comprobar los mensajes del sistema mediante consola, visualizar los mensajes de radio y comprobar la posición de los agentes en función del entorno gracias al radar que el sistema incluye.

La siguiente figura muestra cual es el aspecto general de la ventana de simulación.

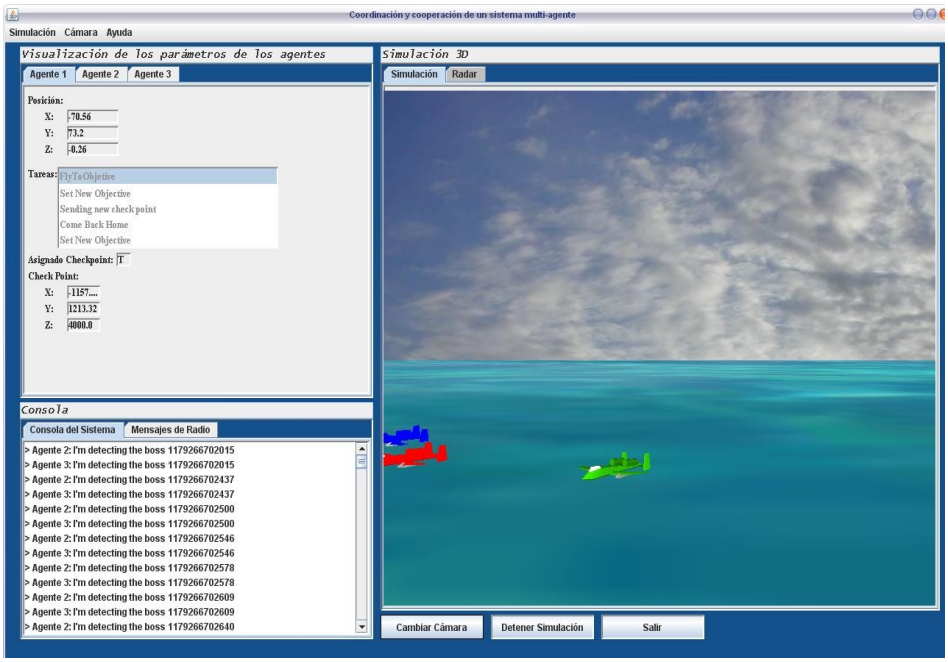


Illustration 62: Screenshot de la interfaz durante la simulación.

Antes de comenzar a explicar el contenido de los distintos paneles que conforman la interfaz del sistema, vamos a señalar la utilidad de los botones disponibles.

- Cambiar Cámara: Permite al usuario realizar un cambio de perspectiva de la simulación 3D respecto a la perspectiva actual.
- Detener/Reanudar Simulación: detiene o activa la simulación en un momento preciso determinado por el usuario.
- Salir: Sale del sistema.

Cabe señalar que todas estas opciones también están disponibles en los menús desplegables de la propia interfaz.

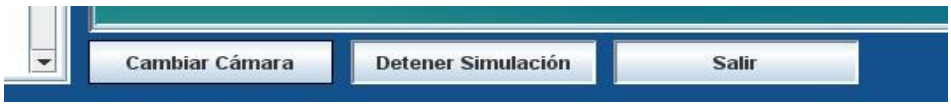


Illustration 63: Cuadro de controles para manejar la simulación.

La ventana de simulación o principal está formada, a su vez, por diferentes paneles. A continuación se tratara destalladamente cada uno de ellos:

– **Panel de Visualización de los parámetros de los agentes:** En este panel se visualizan todos los parámetros relevantes referentes a los agentes, esto es: las coordenadas de su posición, sus tareas activas y en espera y las coordenadas del punto de encuentro. Sólo cabe señalar que las tareas activas son aquellas resaltadas en azul.

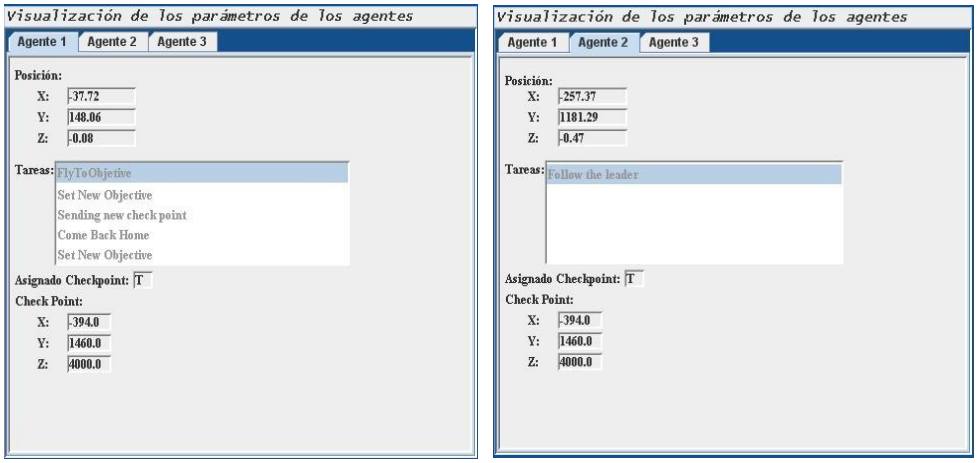


Illustration 64: Panel de visualización de los parámetros de los agentes.

– **Panel de Consola:** En el se encuentra la consola del sistema, encargada de mostrar aquellos mensajes internos de la aplicación, y la consola de Radio, la cual muestra los mensajes de radio transmitidos por los agentes.

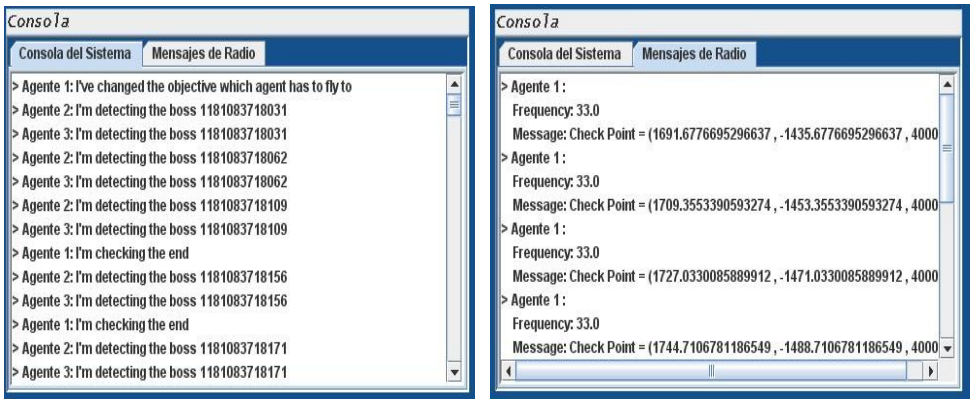


Illustration 65: Panel de visualización de los parámetros de los agentes.

– **Panel de simulación 3D:** Sobre el se realiza la simulación en 3 dimensiones, tal y como indica su nombre, pero además incluye un radar general del entorno con lo que facilita las localización de los agentes y sus objetivos, y a su vez facilita la labor del programador al permitirse depurar la posición de los agentes de forma rápida y precisa.

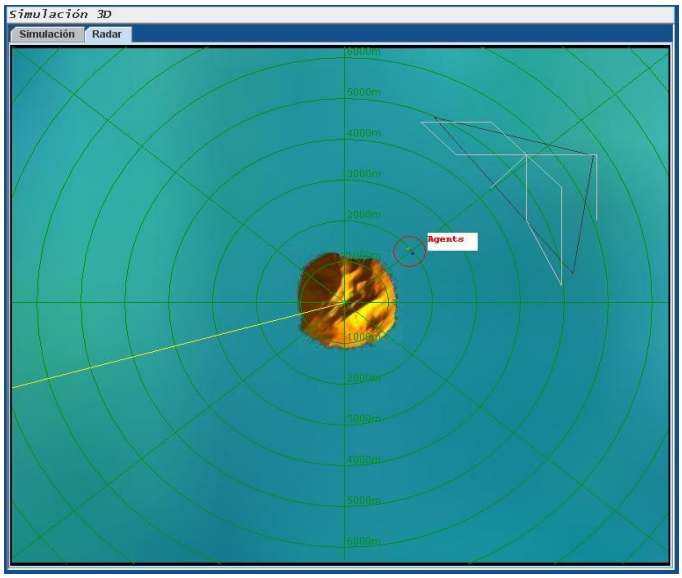


Illustration 66: Panel de simulación 3D (Radar).

Como se aprecia en la figura anterior, el radar no sólo muestra la posición actual de los agentes sino que además se permite visualizar el área de búsqueda y la ruta calculada por el sistema. Las señas métricas permiten al usuario hacerse una idea real del desplazamiento de los agentes en el entorno.

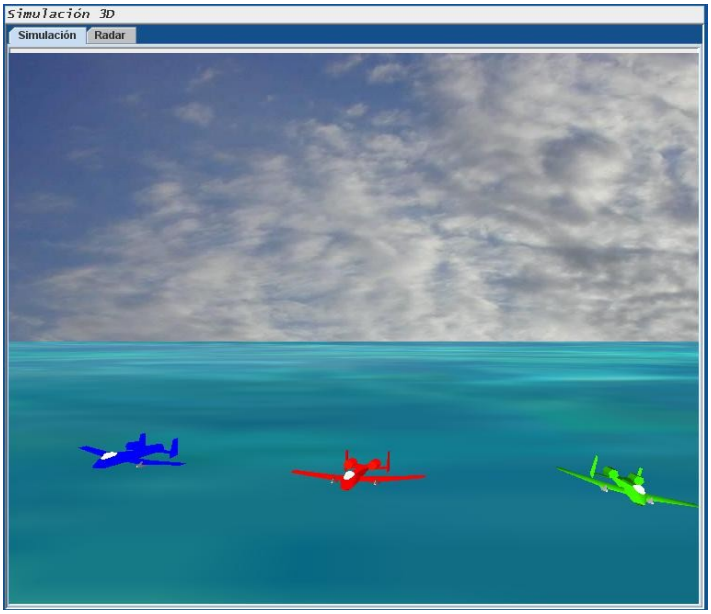


Illustration 67: Panel de simulación 3D (camara normal)

La simulación 3D ofrece la posibilidad de modificar la perspectiva de visualización. Para ello hay dos opciones: presionar el botón *cambiar cámara* disponible en la interfaz del sistema, o, mediante el ratón, desplazar la cámara arrastrando el cursor sobre el panel manteniendo presionado el botón derecho del mismo. Si además el ratón dispone de dispositivo de ruleta, se tendrá la posibilidad de realizar *zoom in* y *zoom out* de la simulación.

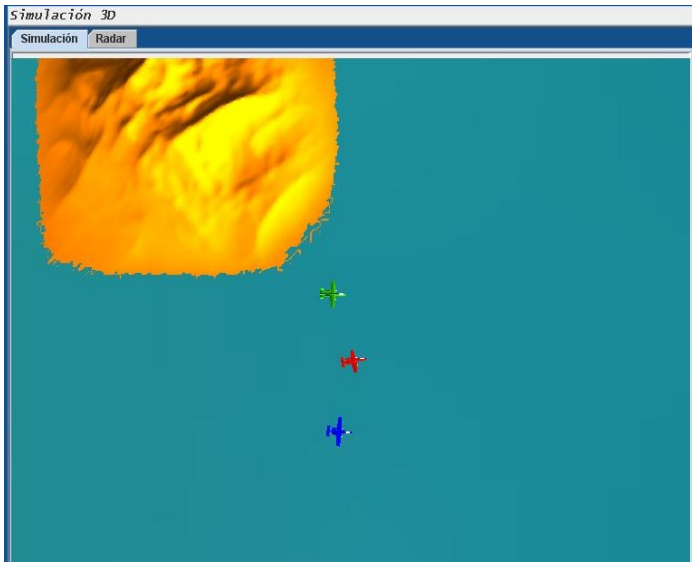


Illustration 68: Panel de simulación 3D (camara "sigue al jefe")

TEMA 3

Conclusiones y ampliaciones

3.2 El Agente: Una nueva arquitectura.

En esta sección comentaremos las conclusiones que hemos sacado tras trabajar con la arquitectura ParallelMachine que hemos creado. Diremos si su diseño nos ha parecido bueno o malo, si es fácil utilizarlo para construir agentes, que recomendaciones damos a posibles informáticos que piensen en utilizarlo y las ampliaciones y mejoras que podrían realizarle futuros alumnos e investigadores que vallan a continuar el proyecto.

Pros y contras.

Pros

Los beneficios de usar esta arquitectura han hecho posible que construyamos un sistema bastante complejo con facilidad. Terminamos este proyecto con la sensación de que todas las técnicas de I.A. y demás sistemás montados sobre la arquitectura del agente descansan sobre una base estable y robusta. Las ventajas de ParallelMachine que hemos detectado durante su uso son:

- Gran potencia: La arquitectura permite crear comportamiento muy complejos. Al basarse en la ejecución en paralelo de distintas tareas, pueden crearse agentes que realicen trabajos asombrosamente complejos que puedan dividirse en sub-trabajos concurrentes. Esto ocurre, por ejemplo, en nuestro caso de vuelo automático: el agente debe volar a lo largo de una ruta, pero al mismo tiempo debe tomar fotografías, analizar las que ya haya tomado, comunicarse con el jefe, mantener la formación de vuelo...
- Fácil trabajo en grupo: Al ser las tareas totalmente independientes unas de otras, puede dividirse su implementación muy fácilmente entre los miembros de un equipo de programadores. Incluso el controlador puede ser implementado por otra persona sin que esta tenga que estar continuamente comuicándose con los creadores de cada una de las tareas. Con que cada tarea tenga definidas unas precondiciones, postcondiciones y condiciones durante su ejecución, el que está construyendo el controlador, u otra tarea, puede tratarla como una caja negra. Tan solo en algunas ocasiones es necesario conocer la máquina de estados de la tarea, cuando esta debe trabajar en estrecha coordinación con otra. Incluso en este último caso, tan solo es necesario conocer su máquina de estados, no como esta implementado cada uno de ellos, para poder programar esta coordinación.
- Portabilidad: Las tareas creadas para un agente ParallelMachine podrían ser usadas por otro agente. Por ejemplo, en Seagull las tareas del jefe son intercambiables con las de los buscadores, necesitándose solo incluir las variables y creencias que dicha tarea use en las bases de variables y creencias del otro agente.

– Escalabilidad: El sistema es escalable, pues puede aumentarse la complejidad de su comportamiento añadiendo más tareas de forma sencilla. El comportamiento de la arquitectura a este respecto es como el de un programa concurrente típico: pueden meterse todas las tareas concurrentes que se quieran, pero habrá que tener gran cuidado en las posibles interferencias que puedan crearse unas tareas con otras. En la versión que entregamos de Seagull ni siquiera hemos tenido que introducir un valor de prioridad a las tareas para resolver conflictos generales. De todas formás, si se va a usar ParallelMachine para construir sistemás mucho más complejos, sería deseable incluir este mecanismo de resolución general de conflictos u otros semejantes.

Contras. Respecto a los inconvenientes de esta arquitectura citaremos los que nosotros hemos descubierto:

– Coordinación: Para que la coordiación de unas tareas con otras pueda programarse de forma sencilla, es muy importante seguir las instrucciones que se citan en el apartado siguiente. Nos referimos a crear una definición de cada tarea en función de sus precondiciones, postcondiciones y condiciones durante ejecución y a definir claramente el significado de cada variable de estado y de cada creencia. Si no se hace así, acabarás programando todas las tareas como si fueran una sola pero con el código distribuido en distintas clases.

– Sensores y efectores: La implementación actual del acceso a los efectores y sensores es un poco aparatosa. Todos los sensores y todos los efectores se acceden con un solo método *getSensor()*, *getEfector(),* al que tienes que pasarle un parámetro diciéndole a que sensor quieres acceder. La inicialización de los mismos se produce de una manera similar, con homologos métodos *setSensor()*, *setEfector()*. No obstante esto puede solucionarse fácilmente creando el par de métodos *get()* y *set()* para cada sensor y efector de manera individual. Nosotros no lo hemos hecho porque ya estábamos acostumbrados a la forma de acceso mencionada.

Recomendaciones de uso.

Reglas cívicas de uso. Para que todo sea tan efectivo y sencillo como se dice, es muy importante seguir ciertas reglas éticas cuando se vaya a trabajar en grupo con ParallelMachine. Describimos las que nosotros hemos considerado a continuación:

– Reunión inicial: Antes de comenzar a cacharrear con la arquitectura, es muy importante tener un par de reuniones con todos los miembros de el grupo de trabajo. En ellas ha de definirse primero como queremos que sea el comportamiento completo del agente. Después habrá que dividir ese comportamiento complejo en sub-comportamientos concurrentes, con el objeto de que cada tal sub-comportamiento sea desarrollado por una tarea a implementar. Deberá quedarse de acuerdo cual va a ser el funcionamiento “de caja negra” de cada una de estas tareas, para que cuando cada uno se

ponga a implementar sus tareas sepa como van a responder las demás (en el siguiente punto se concreta esto). Por último deberán definirse las variables de estado y las creencias del agente (más adelante también se concreta que queremos decir con esto). Recordemos que las primeras son las responsables de que las tareas sean activadas por el controlador, y los segundos se ocupan de representar el estado actual del mundo y del agente para que las tareas tengan acceso a dicha información. Si no se definen bien, cada miembro del grupo acabará teniendo una interpretación algo distinta de las mismás.

- Definición de tareas: Para que la programación del sistema sea sencilla, es bastante importante que las tareas se definan con unas precondiciones, postcondiciones y condiciones durante su ejecución. De no hacerse así, no podrá tratarse a dicha tarea como una caja negra, por lo que al final tendrás que pensar en como actúa su propia máquina de estados para crear la coordinación que buscas con ella.
- Variables de estado: También es muy importante definir claramente que significa cada variable de estado y cada creencia del agente. Si no se hace así con las primeras, las tareas acabarán activándose cuando no deben, y producirán el lanzamiento de otras tareas que no deberían ser lanzadas. Si no se hace con lo segundo, las tareas acabarán leyendo unos datos pensando que son otros y, por lo tanto, interpretando mal el estado del mundo y del agente.

Si se siguen las reglas aquí descritas, la programación del agente será bastante cómoda en relación a la complejidad de comportamiento que se consigue por parte del agente.

Futuras ampliaciones.

Respecto a las posibles futuras ampliaciones, ya se ha comentado algo acerca de por donde podrían ir estas encaminadas.

Mecanismos de coordinación.

Por un lado, para asegurarse que el sistema es escalable sin perder la coordinación entre tareas, sería deseable introducir mecanismos de coordinación entre las mismás. Un tal mecanismo muy sencillo y pero de gran capacidad, sería introducir un valor de prioridad a cada tarea. Ante un conflicto de acceso a un efector o a un sensor, se esperaría la tarea con menor prioridad. Ante un conflicto más grave, como que una tarea quiera hacer una operación contraria a la de otra tarea, puede definirse que la que tenga menor prioridad espere a que la de mayor prioridad finalice, o simplemente que la de menor prioridad sea forzada a terminar. Otros mecanismos de coordinación que se pueden introducir, son cualquiera de los usualmente empleados en concurrencia y en programación con hilos, como pueden ser: candados, semáforos, monitores, barreras, colas... ; cualquiera de ellos valdría.

La ampliación Super-ParallelMachine

Una gran ampliación de ParallelMachine, sería hacer al sistema entero “anidable”. Es decir, más o menos que pudiera anidarse una ParallelMachine dentro de otra el número de veces que se desee. Esto se podría conseguir haciendo que una tarea no estubiera limitada a ser una máquina de estados, si no que también pudiera ser un sistema completo controlador-variables-creencias-tareas. Así, las tareas que implementaran tal sistema podrían encargarse de sub-tareas muy complejas, utilizando

unas variables de estado, creencias y reglas de controlador independientes del resto de tareas. Esto podría ser útil para agentes que desarrollen trabajos realmente complejos. No obstante, para el uso de tal arquitectura Super-ParallelMachine, como mínimo sería necesario un gran número de ingenieros para implementar el gran número de tareas que haría falta. Aunque al ser las tareas portables, tal y como se ha comentado antes, posiblemente unas supertareas podrían usar tareas de otras supertareas, por lo que la programación de tal sistema puede que no requiriera tal alto número de ingenieros. Otra posible solución para trabajar con esta nueva arquitectura tan compleja, sería utilizar algún mecanismo auto organizativo, como algoritmos evolutivos. Podrían

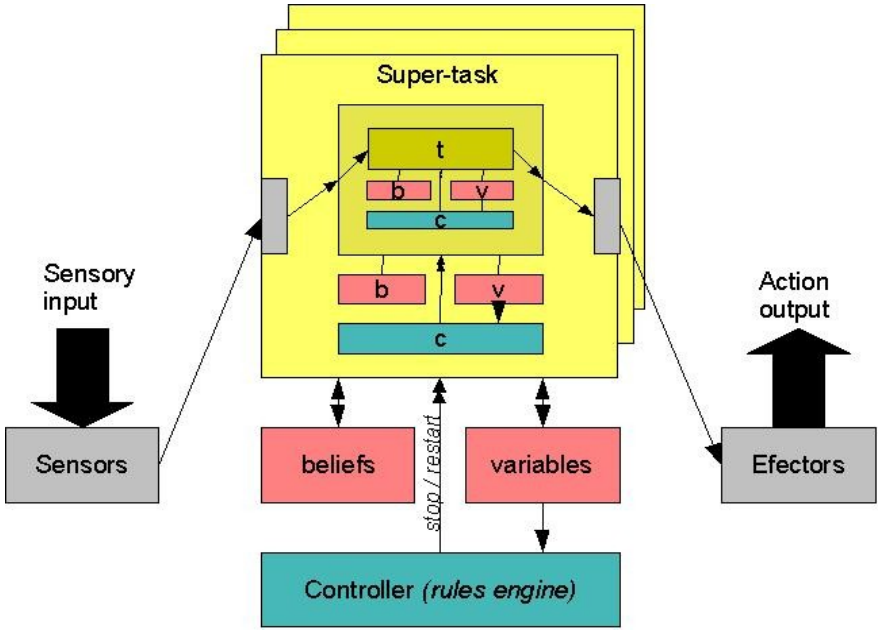


Illustration 69: Arquitectura Super-ParallelMachine, una de las ampliaciones sugeridas. En esta ampliación cada tarea puede ser un subsistema controlador-variables-creencias-tareas con la capacidad de ocuparse de sub-tareas muy complejas.

crearse agentes Super-ParallelMachine para resolver una determinada tarea, inicialmente estos agentes estarían muy poco programados (es decir, su arquitectura apenas tendría tareas). Se podría diseñar un algoritmo evolutivo con la capacidad de mutar los agentes añadiéndoles y eliminándoles tareas, así como modificándoles el resto de parámetros del mismo (reglas del controlador, la máquina de estados de cada tarea...). Tras muchas generaciones en las que los distintos agentes competirían para realizar lo mejor posible la tarea que deben resolver, podrían conseguirse agentes realmente complejos y de gran potencia. Nuestra opinión personal es que sería muy emocionante dedicar un proyecto de final de carrera totalmente a esta labor: algoritmos evolutivos aplicados sobre la arquitectura Super-ParallelMachine. Creo que podrían conseguirse resultados realmente interesantes.

3.3 Cálculo de la ruta óptima.

Las conclusiones a este respecto no son demásiadas, ya que se trata de un problema sobre el que se ha escrito mucho (El problema del viajante), y está resuelto con una técnica también bastante usual (algoritmos evolutivos con operador mutación). Lo que puede ser importante es comentar un par de mejoras que podrían realizarse sobre la técnica usada para calcular la ruta de vuelo.

Posibles mejoras.

A.E. con heurística.

Una de las primeras posibles mejoras sería incluir una heurística en el operador de mutación del algoritmo evolutivo. Tal y como está programado ahora, el operador intercambia dos posiciones de la ruta de manera totalmente aleatoria, cuando en muchas ocasiones es obvio que con el cambio se aumenta la longitud de la ruta. Una forma de incluir tal heurística sería añadir un par de datos más a cada gen (recordemos que cada gen es una posición de la ruta, la secuencia de genes define la secuencia de puntos que sigue la ruta). El primero de estos dos datos sería un entero que indicaría el número de generaciones que el gen anterior al actual en la secuencia lleva sin mutar. El otro gen sería otro entero que llevaría la misma cuenta para el gen posterior al actual. Cuando uno de estos genes es intercambiado por mutación, su contador relacionado pasa a 0. El operador mutación ahora partiría la secuencia de genes en un punto con una probabilidad inversamente proporcional al mencionado valor que indica cuanto tiempo han estado los dos genes contiguos de ese punto juntos. Así, de acuerdo con la teoría de los "building blocks", irían apareciendo en el código genético secuencias de genes que indican un recorrido mínimo de los mismos. La mejora en tiempo del algoritmo podría ser bastante grande.

A.E. con operador de cruce.

Otra posible mejora es la inclusión del operador de cruce. No se ha pensado demásiado sobre como podría ser incluido dicho operador, pero dicha tarea es al menos no trivial. De todas formás, si se consiguiera programar un operador de cruce para este problema, seguramente aumentaría la eficiencia del algoritmo al favorecer "building blocks" pequeños.

Resolución con redes neuronales.

Otra orientación interesante que se podría dar a este problema sería resolverlo con redes neuronales. Tal como se comenta en [ROJ1996], las redes neuronales de Hopfield pueden ser usadas para minimizar algunas funciones de coste. En la página 361 de dicho título, se discute como puede solucionarse el Problema del Viajante con el modelo de Hopfield.

3.4 Reconocimiento de imágenes.

Esta ha sido para mí la parte más interesante del proyecto junto con la creación de la arquitectura ParallelMachine. Durante su desarrollo he aprendido mucho sobre redes neuronales y he descubierto un modelo de red que me parece muy original y curioso (me refiero a las usuales redes neuronales de convolución).

Numerosas dificultades.

Entre las cosas que he aprendido acerca de redes neuronales, está lo difíciles que son de entrenar, las dificultades en decidir los parámetros y la estructura de la red, y lo complicado que puede llegar a ser crearse un framework de redes neuronales. Además, todas estás dificultades se multiplicaban las unas a las otras por la razón de que, cuando por una de ellas el sistema no te funcionaba, te era imposible averiguar por cual de las tres causas era el fallo: si por un mal algoritmo de entrenamiento, por una mala estructura de la red o parámetros de entrenamiento, o por un fallo de programación en el código.

El conjunto de entrenamiento.

Por un lado, la selección del conjunto de entrenamiento era bastante difícil. Con una colección de fotos grandes para que la red aprendiera bien, el algoritmo de entrenamiento no convergía. Supongo que será porque en una colección tan grande hay fotos con características repetidas (fotos en las que es todo agua, fotos en las que aparece una zona de costa con aproximadamente tal o cual forma...). Esto puede producir que, al seleccionarse durante el entrenamiento las imágnes de forma aleatoria, si durante varios pasos se presentan a la red fotos muy similares, esta empezará a memorizar ese tipo de fotos, y por lo tanto, "olvidará" lo que había aprendido con anterioridad cuando se le habían presentado fotos alternando características.

Parámetros de entrenamiento y arquitectura.

Otro problema ha sido la selección de las velocidades de aprendizaje de cada capa. Al principio todas las capas tenían el mismo *"learning rate"* y el mismo *"momentum rate"*, y la red aprendía de forma distinta con estos parámetros distintos para cada capa. A partir de ahí empecé a intuir que podía conseguirse un mejor entrenamiento con velocidades de aprendizaje distintas, y a base de prueba y error dí con los valores mostrados en esta memoria. Con la selección de la arquitectura de la red, sin embargo, no he tenido problemás. Pues ya sabía por el artículo [PAU1995] que tal red tenía que funcionar.

Mi código.

Respecto a la creación del framework de redes neuronales, encontramos problemás que surgen al programar que, al unirse con las dificultad anteriores, han hecho del asunto todo un reto. Ante el fracaso de JOONE y al haber comprobado lo lenta que era esta librería para redes relativamente pequeñas, decidí crearme mi propia mini-biblioteca de redes neuronales para asegurarme de su optimización en tiempo. La construcción de la red descrita en [PAU1995] requería que el framework permitiera crear el típico M.L.P, redes de convolución, y la red híbrida que combina ambas

arquitecturas. Por lo tanto, en definitivas cuentas, tuve que programar directamente desde Java tres tipos distintos de redes neuronales y que todas se ejecutaran y entrenaran correctamente. Finalmente esto necesitó de alrededor de 20 clases, una cantidad de código bastante grande que tuve que testear de forma incremental para asegurarme de su corrección. Aún así, fui incapaz de detectar unos cuatro o cinco errores menores que pasaron desapercibidos en dichos tests incrementales. Estos errores tuve que detectarlos en las pruebas finales con las fotografías reales de Seagull, lo cual me fue enormemente difícil al juntarse con las dificultades antes descritas. Cuando la red no entrenaba, me era imposible saber si esto era por una mala versión de backpropagation, malos parámetros de entrenamiento, un mal conjunto de fotografías, un error de programación en mi código o si sencillamente yo estaba entendiendo radicalmente mal el artículo [PAU1995] y mi red estaba abocada al fracaso.

Posibles mejoras.

Un par de mejoras que podrían aplicarse a las redes neuronales aquí usadas son las siguientes.

Afinar parametros.

Para empezar, los parámetros de entrenamiento de la red no están afinados. Simplemente se han probado distintos valores hasta que se ha conseguido una red que funciona bastante bien. Probablemente los valores actuales de *"learning rate"* y *"momentum rate"* pueden mejorarse al menos un poco. Con toda seguridad se conseguiría un sustancial mejora aplicando "enfriamiento simulado" a tales parámetros. Tal y como se observa en la gráfica de convergencia 77, una vibración probablemente producida por un *"learning rate"* demásiado alto, impide disminuir aún más el error. Aunque posiblemente si se disminuye aún más el error en este caso, se esté causando un *"overfitting"* del conjunto de entrenamiento y la red memorice las fotografías del mismo. Sin embargo, en problemás de identificación más complejos, dicha vibración probablemente impida que la red aprenda el patrón requerido. En dicho caso un enfriamiento simulado de los parámetros, o cualquier otra técnica que aumente la convergencia, podría resolver el problema.

Mejoras a backpropagation.

También podrían probarse las mencionadas técnicas (Rprop, valor mínimo de derivada de la función sigmoide, suma constante a la derivada de la sigmoide...) de forma más amplia. Pues posiblemente pueda aplicarse alguna de estas técnicas a alguna sección de la red y conseguir mejores resultados.

3.5 Control de vuelo (vuelo en formación).

Al igual que en el caso del algoritmo evolutivo que resolvía la decisión de la ruta de vuelo, aquí tampoco hay demásiado que decir como conclusiones. Acerca de control de vuelo de aviones y las técnicas relacionadas ya se ha escrito mucho, y mejorar el método de vuelo actual solo es cuestión salir "ahí fuera" y leer más documentación que nosotros al respecto. La única novedad con respecto a controles de vuelo típicos, es el hecho de que nuestros aviones van en formación, lo cual ha requerido un poco de imaginación por nuestra parte. Aún así, analizando el problema con cuidado, podría mejorarse el método que nosotros hemos diseñado.

¿Red neuronal de control?.

Una posible modificación que sería interesante, es aplicar redes neuronales al problema de volar siguiendo al jefe de la formación. Esto no tendría porque mejorar el funcionamiento del método de control y, además, seguramente sea bastante difícil hacerlo. Aún así parece interesante, pues el teorema de Kolmogorov avisa de la capacidad de las redes neuronales para aproximar cualquier tipo de función, por lo que teóricamente podría conseguirse así la función que produce el control más óptimo de vuelo en formación. Una posible forma de abordar el problema sería usar la red neuronal para minimizar el tiempo que el avión se encuentra fuera del punto en el que debería estar para considerar que está dentro de la formación. Las entradas de la red neuronal serían las posiciones actuales del jefe, del avión que sigue al jefe, y del punto en donde debería encontrarse para estar dentro de la formación. También se alimentarían a la red las velocidades a las que se mueven cada uno de dichos puntos. Las salidas de la red serían las señales de rumbo y aceleración que van a RCAM. La red luego podría entrenarse con un algoritmo evolutivo o con alguna adaptación de backpropagation.

3.6 Conclusiones Simulación de vuelo.

El uso de lenguajes y API de interfaces gráficos donde el rendimiento de las mismás no sea, un elemento vital, hace de Java3D una fantástica selección, además de su intrínseca capacidad multiplataforma.

Aunque los detalles gráficos no sean muy preciosistas, detallan perfectamente el estado del sistema, dándonos oportunidades de ver cómo funciona el sistema y de mejorarlo. Sería interesante por ejemplo ampliar el sistema con unos mensajes encima de los aviones indicándonos en qué estado se encuentra actualmente. También se puede ampliar el sistema con varios grupos de aviones volando en formación por el sistema, al igual que tener varios tipos de mapa distintos e introducir más detalles a los mapas actuales.

Además la estructura de clases está muy bien definida, lo cual hace que una posible ampliación sea más que fácil y satisfactoria.

TEMA 4

Información técnica

4.4 Ejemplo de ejecución:

Ejemplo del sistema completo.

Una vez ejecutado el simulador, aparecerá la ventana de presentación que se muestra en la siguiente figura:.

Illustration 70: Ventana inicial de presentación

Para continuar hay que pulsar el botón Configurar Simulación. Seguidamente aparecerá la siguiente ventana de configuración:

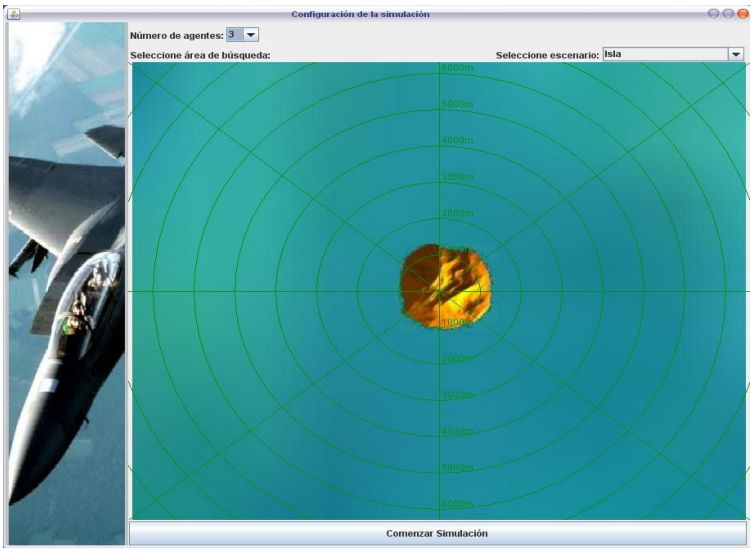


Illustration 71: Configuración de la simulación.

En esta ventana se determinaran los parámetros de la simulación: número de agentes que realizarán la búsqueda, el tipo de escenario y el área de búsqueda. Esta área se seleccionará indicando sobre el mapa los puntos que forman parte del polígono que define el área de búsqueda. Los puntos deberán ser insertados de arriba a abajo y de izquierda a derecha para que el sistema interprete correctamente el área. Las siguientes ventanas muestran como se configuraría una simulación con tres agentes y sobre un escenario formado por una isla:

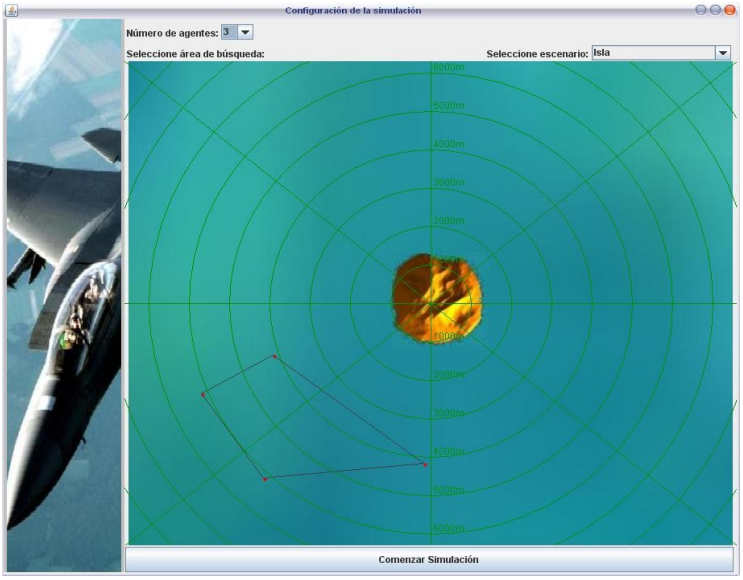


Illustration 72: Selección del área de búsqueda.

Ahora ya estamos en disposición de comenzar la simulación. Para ello pulsamos el botón Comenzar Simulador de la ventana de configuración. Una vez pulsado se abrirán dos ventanas: la primera corresponde a la GUI de Jade, la cual podemos minimizar o cerrar para tener una mejor perspectiva de la segunda ventana, la ventana de simulación.

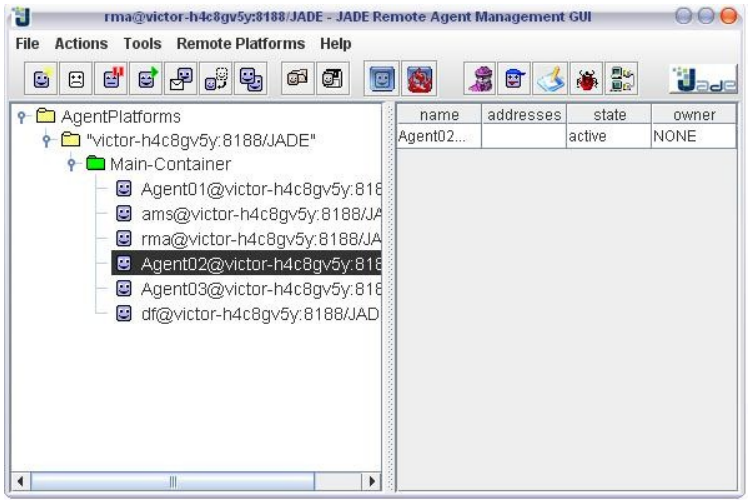


Illustration 73: Consola JADE.

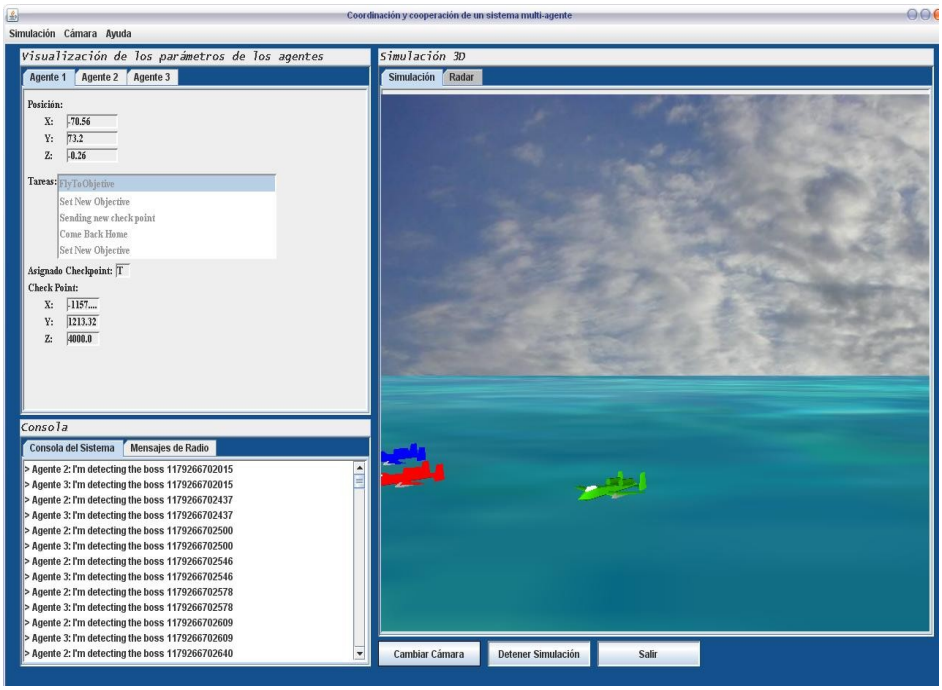


Illustration 74: Screenshot de la interfaz durante la simulación.

Como vemos, la ventana de simulación se divide en tres paneles: panel de visualización de los parámetros de los agentes, panel de consola y panel de simulación 3D. Inicialmente nos fijaremos en este último. Si pinchamos sobre la pestaña Radar veremos el radar del sistema, junto con el área de búsqueda seleccionada en la ventana de configuración y la ruta que el sistema ha calculado para dicha área.

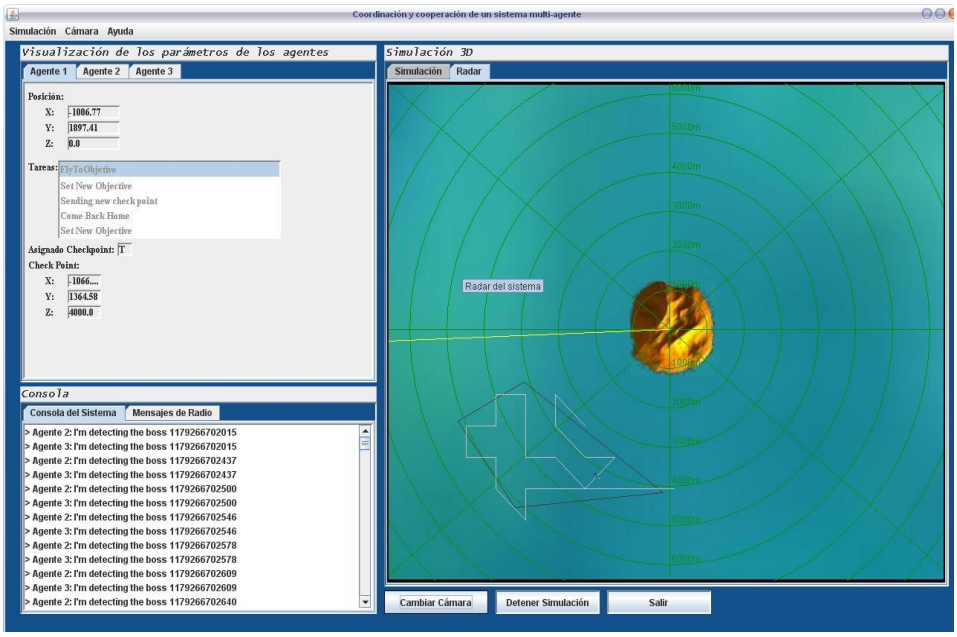


Illustration 75: Radar durante la simulación.

De este modo podemos visualizar donde se localizan los agentes en cada momento en el escenario, así como en que punto de la ruta se localizan.

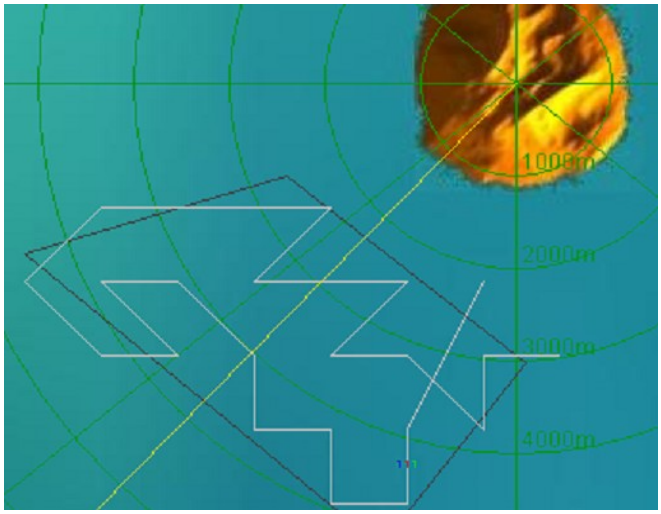


Illustration 76: Detalle de la ruta decidida por los agentes para explorar el área de búsqueda.

Una de las opciones que nos ofrece el Simulador 3D es la de cambio de perspectiva de la cámara. Para ello, sólo hay que presionar el botón *Cambiar Cámara* obteniéndose el siguiente resultado en la ventana de simulación:

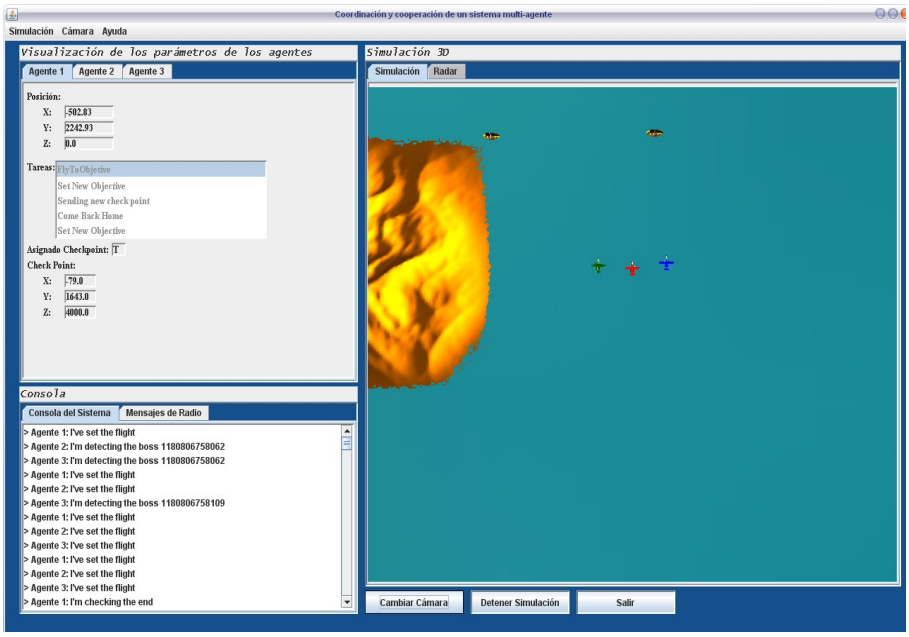


Illustration 77: Cámara superior "sigue al jefe" durante la simulación.

Ahora nos fijamos en los parámetros de los agentes. Inicialmente se muestra los parámetros del agente 1, al que denominamos agente *Boss*. Si queremos ver los parámetros de los otros agentes, podremos pasar de uno a otro mediante las pestañas. Así, por ejemplo, si queremos visualizar la posición del agente 3 tendríamos:

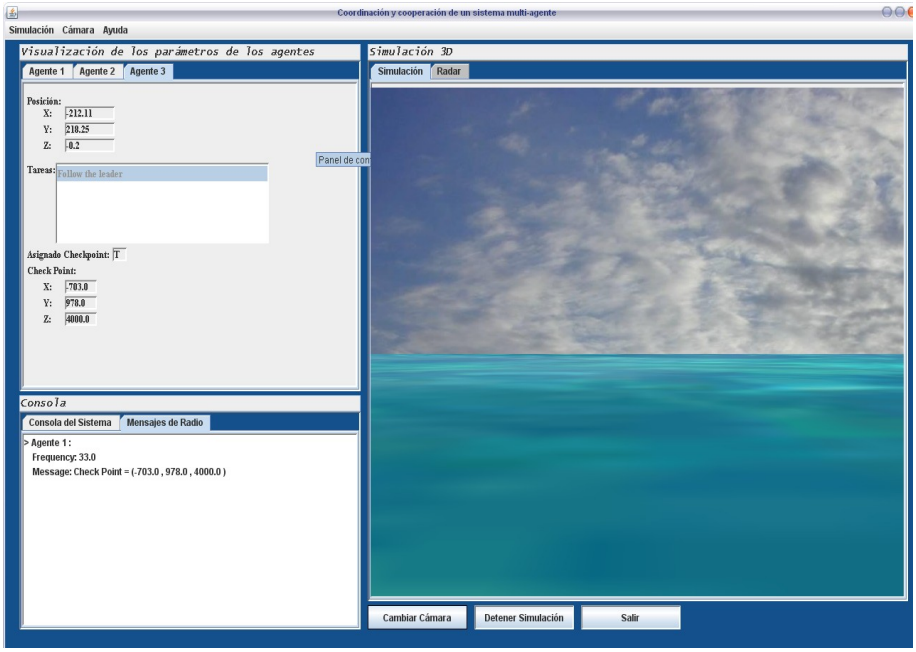


Illustration 78: Screenshot de la interfaz.

Ahora comprobamos los mensajes de radio que los agentes se han enviado entre sí. En este caso únicamente el agente *Boss* se ha puesto en contacto con los demás para comunicarles la variación del punto de encuentro o *check point*.

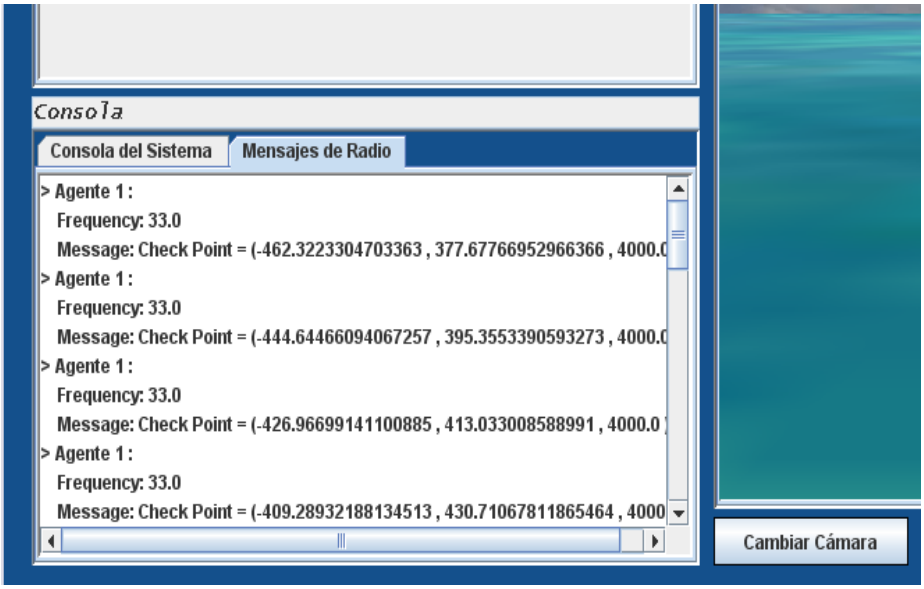


Illustration 79: Consola mostrando la comunicación entre los agentes durante la simulación.

Para finalizar, una vez que los agentes han completado la ruta calculada, regresan al punto de partida para dar por finalizada la misión.

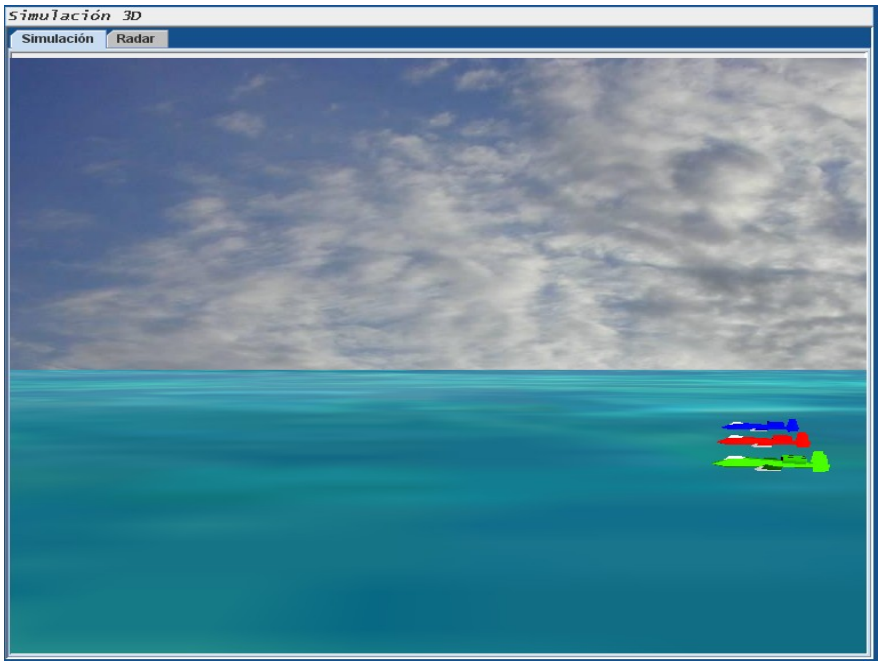


Illustration 80: Los agentes volando en formación durante la simulación.

Bibliografía

SIN01: M. P. Singh, A. Rao, M. J. Wooldridge., Intelligent agents IV: Agent theories, architectures and languages. (2001)

DEL1999: DeLoach, S.A., Multiagent systems engineering: A methodology and language for desingning.. (1999)

KOL2001: Kolp M., Apersan Castro J., UML for agent oriented software development: The Tropos proposal. (2001)

PAD2002: Padgham L., Winikoff M., Prometheus: A pragmatic methodology for engineering intelligent agents. (2002)

PAV2005: Pavon J. Gómez-Sanz J., Fuentes R. M., The Ingenias methodology and tools. (2005)

BUF2002: Burrafato P., Conssentino M., Designing a multi-agent solution for a book store with PASSI methodology (2002)

DEL2003: DeLoach S.A., Matson E.T. and Li Y., Exploiting agent oriented software engineering in cooperative robotics... (2003)

EST2006: Esther D. Ponce et al., Simulación y modelo de un sistema inteligente. (2006)

JGO2007: Departamento de Sistemas Informaticos y Programacion., Game Oriented Multiagent System. (2007)

BIGUS01: Joseph P. Bigus et al, Constructiong Intelligent Agents Using Java (2001)

ANA01: Ana Mas, Agentes Software y Sistemas Multi-Agente (2005)

MAES01: Maes P., Situated Agents Can Have Goals (1989)

BROOKS01: Brooks, R. A., Artificial Intelligence. (1991)

IBM01: , Intelligent Agent Resource Manager (1996)

FERGUSON01: Innes A. Ferguson, Integated Control and Coordinated Behaviour: A case for agent models (1995)

MULLER01: Jörg P. Müller et al, The Agent Arquitecture InteRRaP: Concept and application. ()

BUCKERT01: H. J. Bückert et al, RATMAN: Rational Agents Tesbed for Multiagent Networks (1991)

SUB2000: Subrahmanian V. S. et al, Heterogeneous agent systems (2000)

BIG2001: Joseph P. Bigus et al, Constructiong Intelligent Agents Using Java (2001)

MUR1998: Murch R., Johnson T., Intelligent Software Agents (1998)

KNA1998: Knapik M. Johnson J., Developing Intelligent Agents for Distributed Systems. (1998)

MON2000: Moreno Carlos Suárez, González E, Amirat Y, Loaiza H., Real

MagiCol 99: Team Description (2000)

STO2000: Stone, P. H., Layered learning in multiagent systems: winning approach to robotic soccer (2000)

HUH1998: Huhns M. Singh M., Readings in Agents (1998)

WEI1999: Weiss G. (Editor), Multiagent Systems (1999)

FIP2002: FIPA - Foundation for Intelligent Physical Agents., http://www.fipa.org (2002)

FIP2002a: FIPA - Foundation for Intelligent Physical Agents, Publicly Available Implementations of FIPA (2002)

FER1996: Innes A. Ferguson, Integated Control and Coordinated Behaviour: A case for agent models (1995)

ROJ1996: R. Rojas., Neural Networks: A Systematic introduction. (1996)

KEV1997: Kevin Gurney., An introduction to Neural Networks. (1997)

JOO2007: The Joone Team, Joone Homepage. (2007)

JOO2007a: The Tiki Community., The Joone Wiki Site. (2007)

PAU1995: Paul D. Gader et al., Segmentation Free Shared Weight Networks for Automatic Vehicle Detection. ()

TER1995: Terry Huntsberger, Biologically Motivated Cross-modality Sensory Fusion System for ATR (1995)

DAV1995: David P. Casasent et al., Classifier and Shift-invariant Automatic Target Recognition Neural Networks (1995)

HEG1995: Heggere S. Ranganath et al., Self Partitioning Neural Networks for Target Recognition. (1995)

STE1995: Steven K. Rogers et al., Neural Networks for Automatic Target Recognition. ()

DAI1999: M.N. Dailey et al., Organization of fase and object recognition in modular neural networks. (1999)

SHO1996: Soheil Shams, Multiple Elastic Mudules for Visual Pattern Recognition. (1996)

RAV1995: A. Ravichandran et al., Studies on Object Recognition From Degraded Images Using Neural Networks. (1995)

KAP2001: Ian Kaplan, Wavelets and signal processing. (2001)

CRUZ2005: de la Cruz G.,Jesús M. et al., Ecuaciones Dinámicas de Aviones. (2005)