UNIVERSIDAD COMPLUTENSE DE MADRID

Sistemas Informáticos
Curso 2005-2006

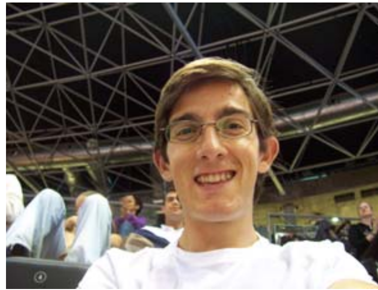
SIMULACIÓN Y MODELO DE UN SISTEMA INTELIGENTE

Esther Desviat Ponce
Miguel Gómez Cuesta
David Ramos Blanco

Dirigido por:

Matilde Santos Peñas
Departamento de Arquitectura de Computadores y Automática

Agradecimientos



Agradecérselo a los profesores que nos han guiado Jesús Manuel de la Cruz (parte modelado), Segundo Esteban (parte gráfica) y en especial a Matilde Santos directora del proyecto, por habernos ayudado a sacar el proyecto adelante. Destacar a Matilde, por el trato tan agradable que nos ha dispensado a lo largo de éste año, ha sido un placer, esperamos haber correspondido.

Siempre hay gente desinteresada que hace de la vida algo más bonito y fácil, con los que se disfruta más y se sufre menos, ahí están mis amigos: Luis Miguel, José Manuel, Alejandro, Ignacio, Mateo, Víctor, Fernando y Miguel, y mi novia María, por aguantarme tantas horas y hacerme disfrutar más de la vida.

Volver a agradecérselo también aquí a los más cercanos, mi familia. Tíos, abuelas y padres, Juan y Alicia, que siempre han estado donde tenían que estar. Una mención especial se merece Juan, mi padre, por ser quien es, por enseñarme los valores de la vida y por estar siempre, absolutamente siempre, cuando se le ha necesitado

David Ramos Blanco

En primer lugar quería agradezco a mis padres Antonio y Ascensión por haberme animado a estudiar una carrera y enseñarme que con constancia y esmero se puede conseguir casi cualquier cosa. A mis hermanos Antonio y Mari que son los que más cerca he tenido junto con mis padres y más han tenido que aguantarme durante este periodo. A mis tíos, a mis primos y mis abuelos

Agradecer a mis compañeros la ayuda que me han prestado durante estos 6 años de estudio para poder superar los obstáculos que me han ido apareciendo, en especial quería darles las gracias a Javier, José Luis, Nacho, Fernando y por supuesto David.

Agradezco a mis amigos que me han apoyado cuando he tenido cualquier problema y que gracias al tiempo de ocio del que he disfrutado con ellos, han hecho que el camino parezca más corto. Y por esas horas de estudio que hemos pasado juntos. Gracias a Míguel, Paloma, Miguel Angel, Lourdes, David y por supuesto a mi novia Laura porque juntos vamos a conseguir Licenciarnos y porque la vida con ella es más fácil.

Agradezco también a los profesores que he tenido durante toda la carrera, porque todos ellos han estado dispuestos a atenderme en todo momento y me han ayudado en cualquier problema o duda que tenía. En especial quería agradecer a los profesores que nos han ayudado a sacar adelante este proyecto.

Matilde Santos Peñas: Por guiarnos durante todo el proyecto y aconsejarnos ante cualquier reto que se nos planteaba. Gracias por tu amabilidad y tus ánimos.

Jesús Manuel de la Cruz: Por ayudarnos con el modelo de ecuaciones dinámicas que rige el comportamiento del Avión y forma la parte central del proyecto.

Segundo Esteban: Por ayudarnos a encauzar la parte gráfica del proyecto, que ha hecho que el proyecto sea más vistoso gracias a tus consejos.

Miguel Gómez Cuesta

Por fin ha llegado ese momento que tantas veces veía tan lejos. Y ahora al echar la vista atrás, es cuando uno se da cuenta de todas esas personas que han estado a tu lado en este camino.

En primer lugar quería agradecérselo a Matilde Santos, por ofrecernos un proyecto interesante y ayudarnos a sacarlo adelante. A Jesús Manuel de Cruz, por tantas charlas sobre aviones y su movimiento, por prestarnos algo de su tiempo, sabiendo lo ocupado que está. A Segundo Esteban por *echarnos un cable*, en un tema que desconocíamos por completo, como es la Simulación en Tres dimensiones.

A mis compañeros de clase, por saber entender mi situación y ayudarme siempre que ha estado en su mano. Sin vosotros la carrera no hubiese sido lo mismo.

Quería agradecérselo a mi familia. En especial a mis padres, por conseguir que haya acabado la carrera sin sacrificar mi deporte. Por tantas horas de trabajo a mi lado haga frío o calor, tanto viaje acompañándome para ganar unas horas de tiempo para así poder estudiar un poco más. Gracias por vuestro apoyo y comprensión. A mi hermana Laura, capaz de decir siempre esa frase, en el momento apropiado. A mi abuela y mis tías que, mientras yo estudiaba, rezaban a todo los santos para que tuviese suerte en mis exámenes.

A mi novio Daniel, por estar siempre a mi lado y saber qué hacer o decir para hacerme sentir bien. Gracias por *aguantarme*.

Esther Desviat Ponce

Dedicado a mis abuelos, que aunque no estén aquí, hubiesen disfrutado más que nadie, viendo a su nieta licenciada.

Autorización de uso

Nosotros los autores del proyecto: *Simulación y modelo de un Sistema Inteligente,* de la asignatura de *Sistemas Informáticos*:

Esther Desviat Ponce	**DNI: 50881281-Z**
Miguel Gómez Cuesta	**DNI: 02665338-Y**
David Ramos Blanco	**DNI: 05205509-B**

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 2006.

Esther Desviat Ponce **Miguel Gómez Cuesta** **David Ramos Blanco**

Índice

ÍNDICE **6**

1. LA IDEA INICIAL **8**

2. OBJETIVO FINAL **9**

2.1. ESPAÑOL 9
2.2. ENGLISH 9

3. INTRODUCCIÓN **10**

3.1. MODELADO 10
3.1.1. Necesidad del modelado *10*
3.1.2. Modelo *10*
3.1.2.1. Características 10
3.1.2.2. Tipos de modelo 11
3.1.3. Obtención de modelos *11*
3.1.3.1. Fases de la modelización 12
3.1.4. Verificacion del modelo *12*
3.2. SIMULACIÓN 12
3.2.1. Ventajas y desventajas de la simulación *13*
3.2.2. Tipos de Simulación *13*
3.2.3. Fases de la simulación *13*

4. DOCUMENTACIÓN Y HERRAMIENTAS DE TRABAJO **15**

4.1. LENGUAJE DE PROGRAMACIÓN Y ENTORNO DE DESARROLLO 15
4.1.1. Patrones de diseño *16*
4.2. ALMACENAMIENTO DE LA INFORMACIÓN 17
4.3. TIPO DE CONOCIMIENTO 17
4.4. REUNIONES Y TUTORÍAS 18

5. EVOLUCIÓN **20**

5.1. PRIMERA FASE: 20
5.2. SEGUNDA FASE: 21
5.2.1. Sistemas de referencia *21*
5.2.2. Ecuaciones del movimiento *23*
5.2.3. Ángulos de ataque, deslizamiento y de vuelo *24*
5.2.4. Ejes de estabilidad *25*
5.2.5. Modelo RCAM *26*
5.2.5.1. Modelo Longitudinal 26
5.2.5.2. Modelo Lateral 28
5.2.5.3. Diagrama modelo RCAM 30
5.2.5.4. Señales de referencia 36
5.2.5.5. Valores Iniciales 37

5.2.6. Superficies de mando y control ... 37
5.2.6.1. Ejes del avión ... 38
5.2.6.2. Superficies primarias ... 39
5.2.6.3. Representación con java ... 43
5.3. TERCERA FASE ... 47
5.3.1.1. Representación con MATLAB ... 47
5.3.2. Representación con Java 3D ... 52
5.3.2.1. ¿Qué es el "API 3D" de JAVA? ... 52
5.3.2.2. El "API 3D" de JAVA ... 52
5.3.2.3. Construir un escenario gráfico ... 52
5.3.2.4. Construir un escenario gráfico ... 56
5.3.2.5. Cargadores ... 58
5.3.2.6. Conclusiones ... 62
5.3.2.7. Representación en JAVA 3D del modeloRCAM ... 63
5.4. CUARTA FASE: ... 64
5.5. QUINTA FASE: ... 72
5.6. SEXTA FASE: ... 75
5.6.1. Dinámica del juego ... 76
5.6.2. Implementación de los agentes ... 77
5.6.2.1. Agentes ... 78
5.6.2.2. Equipos ... 81
5.6.2.3. Reglas ... 81

6. MANUAL DE USUARIO ... 86

6.1. PANTALLA INICIAL ... 86
6.2. MODO AVIÓN ... 87
6.3. MODO BATALLA ... 92

7. CONCLUSIONES Y FUTURAS AMPLIACIONES ... 95

7.1. CONCLUSIONES ... 95
7.2. FUTURAS AMPLIACIONES ... 95

BIBLIOGRAFÍA ... 97

1. La idea inicial

El proyecto que proponemos consiste en "*la comunicación y el aprendizaje de sistemas multiagentes*". Con agentes nos referimos a una entidad abstracta, dotada de un comportamiento definido dado por las especificaciones de acciones que al agente le está permitido hacer, que le está prohibido hacer, que está obligado a hacer y por último que es lo que hace dado los estados posibles del entorno.

Representan algo genérico (robots, aviones, jugadores...) a los que otorgaremos la capacidad de aprender dependiendo de diversos factores, como las circunstancias que les rodean, y la posibilidad de que se comuniquen entre si. Es decir, los agentes se comunican entre ellos y tomas decisiones en base a su información propia y a la que tienen de los otros agentes. Con ello pretendemos obtener un resultado y un comportamiento inteligente lo más cercano a la realidad y lo más óptimo posible.

Se hará una aplicación a la simulación de sistemas interactivos en pseudo-tiempo real.

El proyecto tiene el aliciente de poder colaborar junto con el área de *Ingeniería de Sistemas y Automática de la Universidad Complutense* en un proyecto a largo plazo de gran envergadura, con la posibilidad de poder acudir a algún congreso sobre el tema, para presentar los resultados si son satisfactorios.

Inicialmente estudiaremos la mejor manera de implementar nuestro sistema (software, arquitectura, estrategias de comportamiento, aprendizaje, etc.), evaluando los pros y los contras de cada una de ellas; finalmente elegiremos una que trataremos de defender y en base a ella construiremos nuestro sistema.

Una posible aplicación podría ser un juego de aviones, en el cual hay equipos de aviones que se enfrentan entre sí. Cada miembro del equipo aplicará una estrategia en función de la posición de sus compañeros, la posición de sus enemigos, las estrategias de sus compañeros y de su experiencia anterior (aprendizaje), y actuará en consecuencia. La evolución de esta aplicación nos hará observar un comportamiento inteligente y un aprendizaje.

O por ejemplo un sistema de agentes robots que recorren un laberinto con distintos focos de energía, obstáculos, etc., y que tienen que conseguir un objetivo. Se pueden introducir distintos grados de dificultad en el recorrido.

Para llevar a cabo este sistema nos basaremos en un proyecto realizado el curso pasado, daremos especial importancia a las partes de aprendizaje y comunicación entre agentes.

2. Objetivo Final

2.1. Español

El proyecto consiste en la simulación de un sistema multiagente con comunicación y aprendizaje, guiado mediante un modelo que reproduce las ecuaciones dinámicas reales de un avión. El aprendizaje está basado en reglas y emplea un sistema de puntuación que es el que nos permitirá en caso de duda decidirnos por ejecutar una acción u otra. Los agentes van a ser aviones, y el entorno una batalla.

A lo largo de la memoria se explica la evolución, dificultades, decisiones, áreas de trabajo e investigación, fases del proyecto...

Finalmente para comprobar el correcto funcionamiento de nuestro sistema se ha realizado una representación gráfica en tres dimensiones, utilizando para ello Java3D, con varios interfaces de usuario para facilitar el manejo del programa, la captura de datos, la representación de resultados y la visualización del propio sistema, ya sea un avión individual o un combate entre dos equipos.

2.2. English

This proyect consists in a multiagent simulation with comunication and learnig systems, leaded by a model which represents an actual aircraft dinamic equations. Learning is based on rules and uses a puntuation system that allows the user pick and execute an appropiate action in case of hesitating. These agents are aircrafts, and the set a battle.

Thoughout this text it is explained the evolution, difficulties, decisions, research and work areas, proyect phases...

Finally we have made a three dimensions graphic representation using Java3D, to check the correct running of the program out, with several user interfaces for being able to use the system easily, saving data, showing results and screening the system working just as an unique aircraft or a combat between two teams.

3. Introducción

3.1. Modelado

Conjunto de procedimientos y medios empleados para la realización de los modelos.

3.1.1. Necesidad del modelado

Muchas veces, es interesante saber responder a preguntas sobre un sistema sin recurrir a la experimentación, ya que ésta puede resultar cara, peligrosa o simplemente no se pueda realizar.

Por este motivo es necesaria la utilización de modelos.

3.1.2. Modelo

Modelo es una representación de un objeto.

El modelo de un sistema es cualquier tipo de descripción abstracta que refleja sus características más relevantes. Por lo general el modelo nos ayuda a entender y mejorar un sistema.

Un modelo se basa siempre en aproximaciones e hipótesis del sistema a modelar. Y se construye para un fin específico.

Un modelo es siempre un compromiso entre la sencillez y la necesidad de recoger todos los aspectos esenciales del sistema.

3.1.2.1. Características

1. Coherente: Debe dar cuenta de todas las observaciones anteriores y permitir prever el comportamiento futuro del sistema representado.

2. Dúctil: Para así poder ser mejorado al confrontarlo con la realidad.

3. General: Represente al sistema de la forma más general posible, dentro de los limites establecidos.

4. Eficaz: Refleje con exactitud el sistema.

3.1.2.2.Tipos de modelo

Existen varios tipos de modelos, distinguimos entre los siguientes grupos:

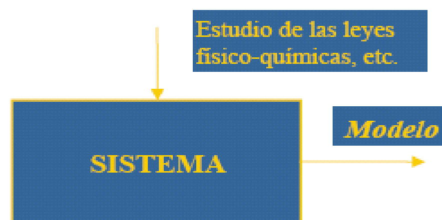
1. Modelo Físico.
2. Modelo Mental
3. Modelo Verbal
4. Modelo Matemático: Conjunto de relaciones matemáticas entre las variables del sistema. Se diferencian varios grupos entre los modelos matemáticos:

    a. Deteminístico: Ni las variables endógenas ni exógenas se pueden tomar como datos al azar. Se permite que las relaciones entre estas variables sean exactas, es decir que no entren en ellas funciones de probabilidad. Quitan poco tiempo de cómputo.
    b. Estocásticos: Por lo menos una variable es tomada como dato al azar. Las relaciones entre variables se toman por medio de funciones probabilísticas. Sirven por lo general para realizar grandes series de muestreos. Quitan mucho tiempo en el computador son muy utilizados en investigaciones científicas.
    c. Estáticos: En ellos no se tiene en cuenta el tiempo dentro del proceso. Hay una relación directa entre las variables.
    d. Dinámicos: Se tiene en cuenta las variaciones en ele tiempo. Las variaciones dependen de las señales aplicadas anteriormente.
    e. Continuos: Dependen del valor de una variable continua.
    f. Discretos: Toman valores en instantes determinados.

3.1.3. Obtención de modelos

Para obtener un modelo fiable de un sistema debemos seguir los siguientes pasos:

- Buscar información sobre el sistema: Mediante observación y el conocimiento teórico de ciertos aspectos del medio.
- Descripción del modelo
- Validación.



Esta fase debe hacerse con cuidado y supone una larga tarea.

3.1.3.1.Fases de la modelización

1. Estructurar el sistema: Dividirlo en subsistemas y determinar causas- efecto. En esta fase se identifican las señales más representativas del sistema. Es conveniente hacer un diagrama de bloques.
2. Formular las ecuaciones básicas: Relacionar variables y constantes en cada bloque.
3. Representar el modelo en el espacio de estados: Elegir un conjunto de variables de estado. Expresar las derivadas en el tiempo de cada variable sólo en función de las variables de estado y las entradas. Expresar las salidas cómo funciones del estado y de las entradas.

$$X'(t) = f(x(t), u(t))$$
$$Y(t) = h(x(t), u(t))$$

3.1.4. Verificacion del modelo

El valor del análisis depende de la calidad del modelo del sistema. Un modelo nunca es la descripción exacta de un sistema. Se desarrolla para resolver ciertos problemas relacionados con aspectos específicos del sistema; por lo que sólo es valido respecto a una finalidad, aun propósito especifico.

Para validar un modelo, debemos comparar el comportamiento del mismo con el sistema y evaluar las diferencias, de manera que algunas partes del modelo puedan ser mejoradas o eliminadas. Hay que ser críticos ante el modelo y estar dispuestos a modificarlo.

La validación esta unida al modelado y es independiente del problema concreto al que se aplique.

3.2. Simulación

Técnica de imitar el comportamiento de alguna situación o sistema mediante un modelo para obtener información sobre el sistema.

Descripción codificada de un experimento que hace referencia al modelo al que se aplica.

Un intento de repetir la realidad, concretándolo a un caso determinado en unas condiciones especificas.

3.2.1. Ventajas y desventajas de la simulación

Ventajas

- Estudiar el comportamiento de un sistema bajo diversas condiciones.
- Estudiar un sistema que no existe para ver su viabilidad, rentabilidad,..
- Analizar globalmente el funcionamiento de un sistema complejo.

Desventajas

- No es una solución óptima.
- Es difícil de probar su correctitud.
- No da respuestas exactas ni precisas.
- Requiere bastante tiempo
- Continua actualización de los datos.

3.2.2. Tipos de Simulación

En toda simulación es conveniente analizar el coste, el tiempo de ejecución y los conocimientos del sistema a simular

Existen tres tipos de simulación diferentes:

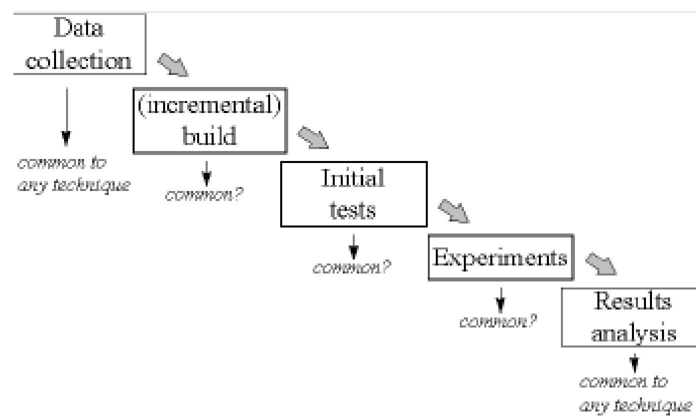
- Discreta: Evalúa en instantes de tiempo discretos.
- Continua: Variables continuas expresadas como ecuaciones diferenciales.
- Mixtas: Unos sucesos son discretos y otras variables son continuas.

3.2.3. Fases de la simulación

La simulación de un modelo se divide en las siguientes fases:

1. Recogida de Datos: Cuanto mejor sean los datos, mejor son los resultados.
2. Construir el modelo: Hay que construir un modelo, pero no se debe dar a esta parte la mayor importancia, pues lo verdaderamente relevante es la resolución del problema.
3. Verificación del modelo: Comprobar que el modelo se ejecuta correctamente y según las especificaciones.

4. Validación del modelo: Comprobar que las teorías, hipótesis de trabajo y suposiciones son correctas. ES interesante contrastar resultados con expertos de la materia.
5. Experimentación: Se divide en:
   a. Inicialización: Se eligen los parámetros iniciales.
   b. Dinámica o simulación.
   c. Terminación: Periodo de tiempo de ejecución o condiciones finales
6. Análisis de resultados: Se experimenta con el modelo con el objetivo de efectuar inferencias que permitan tomar decisiones con mayor seguridad.



4. Documentación y herramientas de trabajo

Para comenzar el proyecto estuvimos varias semanas tomando información sobre las principales herramientas y sistemas que podíamos utilizar para su desarrollo.

4.1. Lenguaje de programación y entorno de desarrollo

El planteamiento inicial del trabajo era, como hemos comentado en líneas anteriores, crear un sistema dotado de cierta inteligencia y capaz de comunicarse con el resto de agentes que trabajen a su lado de manera conjunta, por esto comenzamos a plantearnos utilizar:

- Una base de conocimiento
- Un motor de inferencias
- Un conjunto de reglas,

Aplicando, de este modo, los conocimientos aprendidos en las asignaturas de "*Inteligencia Artificial*" e "*Inteligencia Artificial Aplicada al Control*".

La que más protagonismo obtuvo fue la utilización de CLIPS. El cual es un sistema de reglas que para problemas relativamente sencillos funciona muy bien, y además permite trabajar con Java, razón que para nosotros era muy importante, ya que al ser un lenguaje que conocemos bien, nos ahorraríamos bastante tiempo en su aprendizaje y nos permitiría investigar en otras áreas del proyecto. Al final, descartamos la utilización de CLIPS por la dificultad que presenta este lenguaje para representar estructuras con cierta complejidad.

Nos informamos también de otros lenguajes de programación (Strips, CommonKads, Protege...), pero ninguno nos convenció demasiado.

También valoramos la opción de trabajar con algoritmos evolutivos, estudiados por dos de nosotros en la asignatura de *Programación Evolutiva*. Aunque esta elección no nos convenció por la cantidad de memoria que se necesita para su implementación lo que supondría mucho retardo en la posterior visualización del proyecto.

Tras valorar varias herramientas, al final decidimos trabajar con JAVA tanto por su portabilidad como por la facilidad que ofrece para trabajar con otros programas (Clips, Matlab, VRML...).

Como entorno de desarrollo usamos ECLIPSE, utilizado el curso anterior en la asignatura de *Ingeniería del Software*.

En cuanto a la representación gráfica, se nos planteaban un gran problema, pues decidimos representar a los agentes en tres dimensiones y ninguno de los miembros del equipo sabía como hacerlo, pues ninguno ha cursado la signatura de *Informática Gráfica*.

La primera solución que tomamos fue intentar visualizar a los agentes con MATLAB, pues teníamos un modelo ya hecho, que nos podía servir de ejemplo. Pero luego debido a que el resto del proyecto esta realizado en JAVA, pensamos que lo mejor sería intentar utilizar JAVA 3D y así tener un programa compacto y más intuitivo.

Destacamos que antes de empezar a escribir código repasamos los patrones de diseño, que habíamos estudiado el curso anterior en la asignatura *Ingeniería del Software*, para conseguir un programa lo más portable y genérico posible, de modo que el hacer futuros cambios en él, no supusiese un obstáculo.

4.1.1. Patrones de diseño

Para la implementación hemos utilizado los patrones de diseño que estudiamos en la asignatura de Ingeniería del Software, con el objetivo de obtener soluciones simples y elegantes a problemas específicos del diseño de software orientado a objetos, representan soluciones que han sido desarrolladas y han ido evolucionando a través del tiempo.

Empezamos a programar de manera muy modulada, cada cosa la implementábamos en clases diferentes, para que se tuviera la libertad de modificar detalles o estructuras de una manera fácil, tanto para los creadores del programa como para cualquier usuario que trabaje con él.

Posteriormente nos permitimos alguna libertad, para no complicar en demasía el proceso de programación, siempre teniendo en cuenta la posibilidad de introducirlos más a posteriori como ampliación y mejora.

A continuación vamos a enumerar los patrones empleados en el desarrollo de la práctica:

- Patrón Experto: Expresa la idea de que los objetos hacen cosas relacionadas con la información que poseen. Los beneficios que obtenemos son la conservación del encapsulamiento y el promover clases sencillas y cohesivas que son más fáciles de mantener y comprender. En nuestro caso, cualquiera de nuestros paquetes tiene una clase experta, que es, por decirlo de alguna manera, la clase principal, en torno a la cual se construyen el resto de clases. Por ejemplo, en el paquete ***java3d*** la clase experta es *Simulación*, que utiliza *DatosBatalla*, y en el paquete ***agentes*** la clase *AgenteAvion*.

- Patrón Creador: Es muy similar al patrón anterior, se pregunta quién es el responsable, o quién debería ser el responsable de crear una nueva instancia de alguna clase. Cómo ejemplos podemos tomar los mismos que en el caso anterior, en nuestro caso el experto es el creador (algo lógico).Los beneficios obtenidos es que proporciona un bajo acoplamiento, la clase creada tiende a ser visible por la clase creador.

4.2. Almacenamiento de la información

Otro problema que se nos planteaba era la manera de almacenar datos. Inicialmente se pensó utilizar una base de datos relacional, ya que es un mecanismo fuerte y flexible para el almacenamiento de conocimiento. Sin embargo ésta no era la mejor opción, ya que a pesar de ser flexibles, no son buenas representando relaciones complejas entre conceptos u objetos del mundo real.

Pensamos también usar técnicas de *Inteligencia Artificial*, como nidos y marcos. Finalmente decidimos guardar los datos en estructuras que formen parte de los propios objetos, las cuales vamos actualizando a medida que se va ejecutando el programa y obteniendo nueva información. La estructura que decidimos que sería mejor es similar a un marco, esta, nos permite tener la información centralizada, y dado que vamos a trabajar con múltiples agentes esto es de gran ayuda.

4.3. Tipo de conocimiento

Lo que esperamos que hagan nuestros agentes tiene mucha influencia en el tipo de conocimiento que debemos utilizar:

- Si nuestros agentes tienen un número limitado de situaciones en las que tiene que actuar, puede que un programa de procedimientos específicos sea la mejor solución.
- Si nuestros agentes tienen que construir o modificar modelos sobre el dominio del problema y resolver problemas con distintos niveles de abstracción, entonces los marcos o las redes semánticas son la solución.

- Sin embargo si los agentes tienen que responder a preguntas o generar hechos sobre algo existente, entonces debemos utilizar predicados lógicos del tipo if-then-else. En el caso de que el problema presente incertidumbre deberíamos añadir a las reglas *if-then-else* redes Bayesianas y si necesitamos una respuesta óptima los algoritmos genéticos son la solución.

En nuestro caso con estructuras *if-then-else* simples, tenemos suficiente.

La cantidad de inteligencia requerida por nuestros agentes, en términos de dominio de conocimiento y potencia de los algoritmos de razonamiento, está relacionado con el grado de autonomía y el grado de movilidad de los mismos:

- Si nuestros agentes tienen que tratar con un amplio rango de situaciones, entonces necesitan una base de conocimiento grande y un motor de inferencia flexible.
- Si es móvil, debe haber una base de conocimiento pequeña y compacta y un razonamiento "ligero", también se hacen menos recomendables las reglas if-then-else, en este caso, una red neuronal podría ser la solución para éstos agentes.

Otro tema de discusión fue decidirnos sobre si el conocimiento debería estar centralizado o distribuido (recordemos que vamos a trabajar con un sistema multiagente, si sólo tuviéramos un agente no tendríamos tal problema):

- centralizado en algún agente o en algún objeto que desempeñara ésta función,
- distribuido: cada agente posee su propio conocimiento independientemente del que tenga el resto de sus compañeros o enemigos.

Una buena solución sería tener algo intermedio, en la que los agentes tienen información suficiente como para ser autónomos, pero tienen cierta dependencia con el resto de agentes, como para conocer su posición, su estrategia u otros parámetros

4.4. Reuniones y tutorías

Después de estar las primeras semanas recopilando información por separado, decidimos mantener varias reuniones los tres miembros del grupo juntos. En estas reuniones cada componente expuso sus ideas sobre las primeras tareas a realizar, el reparto de estas, las herramientas que deberíamos utilizar y cuales podían ser los principales objetivos que deberíamos de cubrir con nuestro proyecto.

En paralelo a estas reuniones internas del grupo, mantuvimos sesiones tutorizadas para concretar algunos puntos del proyecto con los profesores Matilde Santos Peñas y Jesús Manuel de la Cruz García. Estas charlas nos sirvieron para concretar los objetivos a corto plazo que deberíamos conseguir así como para fijar las herramientas que íbamos a utilizar para desarrollar del proyecto.

Estas reuniones se han ido realizando durante todo el curso, a razón de una al mes aproximadamente e intensificándose en aquellos momentos en los que estábamos mas atascados, y nos han servido para evaluar el estado del desarrollo del proyecto y para planificar cuales serían los siguientes pasar a seguir. Además nos han servido para resolver dudas sobre la parte del proyecto que estábamos implementando en cada momento.

Así mismo para que tener una idea de la envergadura del proyecto y de cuales eran las partes de él que más problemas nos iban a ocasionar, consultamos las memorias de los proyectos de los dos últimos años:

- [Mestre04]
- [Gonz05]

5. Evolución

5.1. Primera Fase:

En la primera fase del proyecto decidimos crear la estructura básica de los agentes. Nos basamos en las ideas que aparecen en uno de los libros de la bibliografía, mas concretamente en [Bigus01]. Con esto pretendíamos que los agentes se comunicaran, actualizamos las variables de estado de forma sincronizada y creamos un sistema de reglas que constituían la inteligencia de los agentes.

Una vez hecho esto, Matilde Santos nos propuso tener una reunión con Jesús Manuel de la Cruz, el cual nos modificó la idea inicial del proyecto. Seguimos trabajando con agentes, pero ahora el objetivo principal no es la comunicación entre ellos.

Nuestro nuevo proyecto consiste en hacer un modelo de un avión real, imitando sus movimientos, sus velocidades, sus ángulos de giro,…Este nuevo trabajo parte de un proyecto antiguo [CruzAr]. Nuestro avión seguirá los movimientos dados por las ecuaciones dinámicas del modelo RCAM. Para ello se nos facilitó un documento .doc explicativo y un archivo en MATLAB.

Tras esta reunión decidimos dejar apartado, lo hecho hasta ese momento sobre agentes, ya que sería muy complicado unir las nuevas ecuaciones de movimiento con lo explicado por el [Bigus01] sobre agentes.

No obstante, pensamos que sería interesante, para comprobar que el avión se moviera coherentemente, hacer la representación en tres dimensiones.

También decidimos que sería atractivo ver a varios aviones moviéndose a la vez e interactuando entre sí. Por lo que optamos por representar una batalla entre dos equipos de aviones.

5.2. Segunda Fase:

Como ya dijimos anteriormente, Jesús Manuel nos facilitó un archivo en MATLAB, que contenía las ecuaciones del modelo RCAM. Nuestro principal objetivo fue traducir estas ecuaciones de MATLAB a código JAVA. Con el fin de que cada avión pueda tener una dinámica de vuelo independiente del resto.

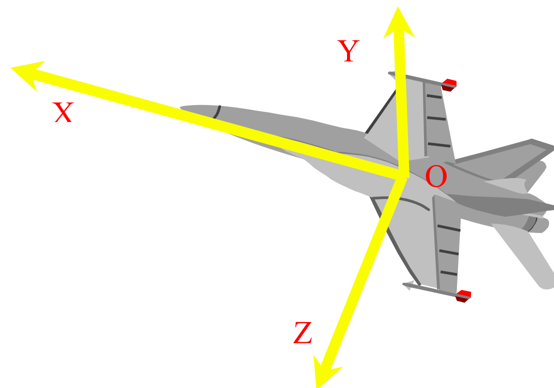
A continuación describimos el comportamiento típico de un avión, lo que puede ayudar a comprender físicamente el comportamiento del modelo de RCAM, que es el que hemos utilizado y que explicaremos más detalladamente en páginas posteriores.

5.2.1. Sistemas de referencia

Normalmente las ecuaciones que describen la dinámica de un avión se obtienen considerando a este como un cuerpo rígido de masa constante y sujeto a fuerzas gravitacionales, aerodinámicas y de propulsión. Sus movimientos se pueden describir dando la posición y velocidad de su centro de gravedad (CoG) y la orientación y velocidad angular de unos ejes ortogonales fijos al cuerpo y que giran con el, con respecto a unos ejes inerciales. Los ejes fijos al avión y que giran con él se denominan *ejes cuerpo*, F_B. Estos ejes los suponemos situados con su origen en el centro de gravedad del avión (CoG) y orientados de manera que el piloto los ve de la siguiente forma (figura 1):

- OX hacia adelanta
- OY hacia fuera del ala derecha
- OZ hacia abajo

Figura 1. Ejes cuerpo



Las medidas en ejes cuerpo llevan el subíndice B, sin embargo en estas notas y para simplificar, los ejes de referencia y las medidas en ejes cuerpo se dan sin subíndices.

Los ejes inerciales de referencia, F_E, se suelen considerar fijos con respecto a la tierra y que giran con ella. Aunque estos ejes no son en realidad inerciales, hacemos esta suposición para simplificar las ecuaciones. En la figura 2 se muestran estos ejes con el subíndice E. Normalmente los ejes se toman de modo que el eje OX_E apunta hacia el norte, el eje OY_E apunta hacia el Este y el eje OZ_E apunta hacia centro de la tierra, si bien cualquier otra orientación sería válida. En la figura 2 también se muestra el sistema de referencia F_V respecto del cual se indica la orientación de los ejes cuerpo. Estos ejes son paralelos a los ejes fijos en tierra pero se mueven con el vehículo y el sistema de referencia al que dan lugar se denomina sistema vertical ligado al vehículo (vehicle-carried vertical frame) o simplemente *sistema de referencia*.

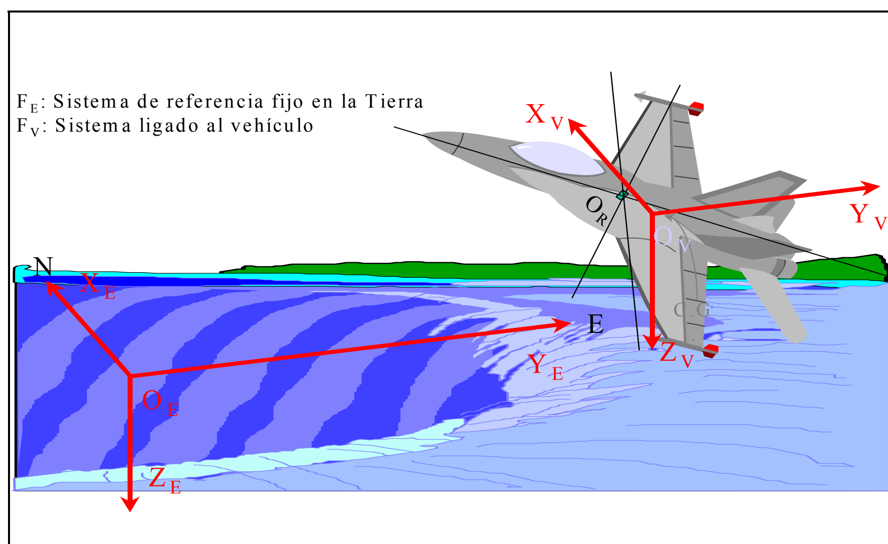


Figura 2. Sistemas de referencias inercial y ligado al vehículo

La velocidad inercial instantánea del CoG del vehículo suponemos que tiene como componentes [U, V, W] en el sistema de referencia F_B. Suponemos también que la velocidad angular instantánea del vehículo, con respecto al sistema de referencia F_V, expresada en ejes cuerpo tiene como componentes [P, Q, R]. Estas son las velocidades que se tienen también con respecto al sistema inercial y por tanto son las medidas por los giróscopos fijos a los ejes cuerpo.

5.2.2. Ecuaciones del movimiento

Para describir el movimiento del avión con respecto a la tierra basta con describir la orientación de los ejes cuerpo con respecto al sistema de referencia F_V. Para ello se utilizan los ángulos de Euler que se definen mediante tres giros que permiten pasar de un sistema de referencia a otro. Más concretamente, los giros para pasar de F_V a F_B se consigue con:

- Rotar un ángulo ψ el eje OX_V para que coincida con la proyección de OX en el plano X_VY_V (eje k_1). $\dot{\Psi}$ es un vector según OZ_V.
- Rotar un ángulo Θ el eje k_1 para que coincida con el eje OX. $\dot{\Theta}$ es un vector según el eje k_2.
- Rotar un ángulo Φ el eje k_3 para que coincida con el eje OZ. $\dot{\Phi}$ es un vector según el eje OX.

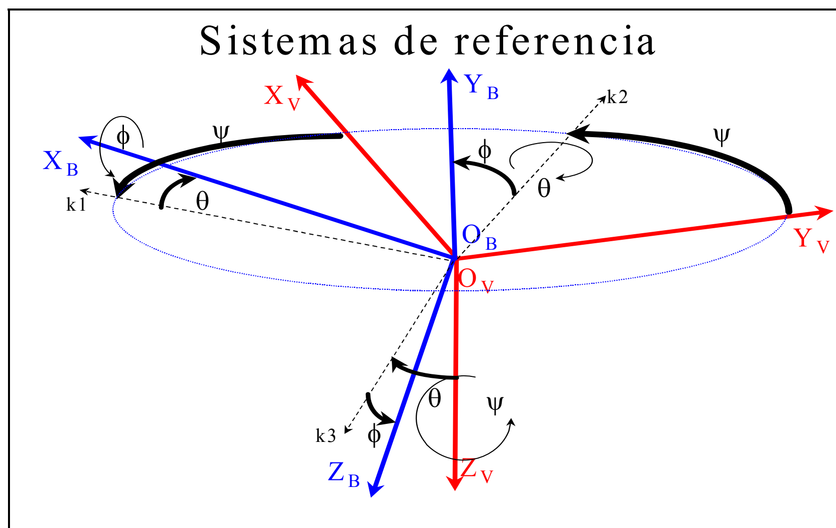


Figura 3. Transformación de ejes ligados a ejes cuerpo

La matriz de transformación que pasa de F_V a F_B es

$$R_{BV} = T_\phi T_\theta T_\psi = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\Phi & s\Phi \\ 0 & -s\Phi & c\Phi \end{bmatrix} \begin{bmatrix} c\Theta & 0 & -s\Theta \\ 0 & 1 & 0 \\ s\Theta & 0 & c\Theta \end{bmatrix} \begin{bmatrix} c\Psi & s\Psi & 0 \\ -s\Psi & c\Psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

donde c = cos, s = sen. Se verifica $R_{VB} = R_{BV}{}^T$

De este modo podemos transformas las velocidades de un sistema a otro:

$$\mathbf{V}_{\mathrm{B}} = \begin{bmatrix} U \\ V \\ W \end{bmatrix} = R_{\mathrm{BV}}\mathbf{V}_{\mathrm{V}} = R_{\mathrm{BV}} \begin{bmatrix} U_V \\ V_V \\ W_V \end{bmatrix}$$

5.2.3. Ángulos de ataque, deslizamiento y de vuelo

Para analizar el movimiento del avión a veces se utilizan coordenadas polares (V_T, α, β) en lugar de las ortogonales. Donde V_T es la magnitud de la velocidad, α es el ángulo de ataque y β el ángulo de deslizamiento (sideslip). Donde la definición de los ángulos son las siguientes:

$$U = \mathrm{V_T}\cos\alpha\cos\beta$$
$$V = \mathrm{V_T}\cos\alpha\ \sin\beta$$
$$W = \mathrm{V_T}\sin\alpha$$

lo que implica

$$\mathrm{V_T} = \sqrt{U^2 + V^2 + W^2}$$
$$\alpha = \tan^{-1}\left(W/\sqrt{U^2 + V^2}\right)$$
$$\beta = \tan^{-1}(V/U)$$

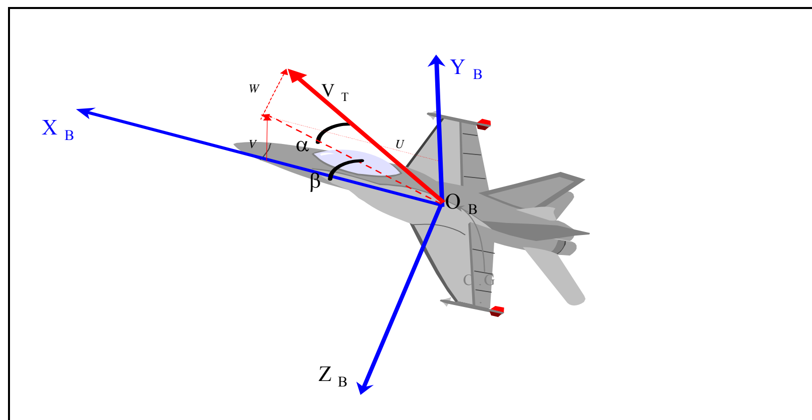


Figura 4. Ángulos de ataque y de deslizamiento

A veces, en lugar de utilizar $W_V \equiv -\dot{h}$, e V_V para analizar los movimientos del avión con respecto a tierra, se utilizan los ángulos de vuelo vertical y horizontal (vertical and horizontal flight path angles) γ, χ. A γ también se le denomina "*inertial flight path angle*", y a χ "*inertial track angle*". Su definición es como sigue:

$$U_V = V_T \cos\gamma \cos\chi$$
$$V_V = V_T \cos\gamma \sin\chi$$
$$-W_V \equiv \dot{h} = V_T \sin\gamma$$

lo que implica que

$$V_T = \sqrt{U_V^2 + V_V^2 + W_V^2}$$
$$\gamma = \tan^{-1}\left(\dot{h}/\sqrt{U_V^2 + V_V^2}\right)$$
$$\chi = \tan^{-1}\left(V_V/U_V\right)$$

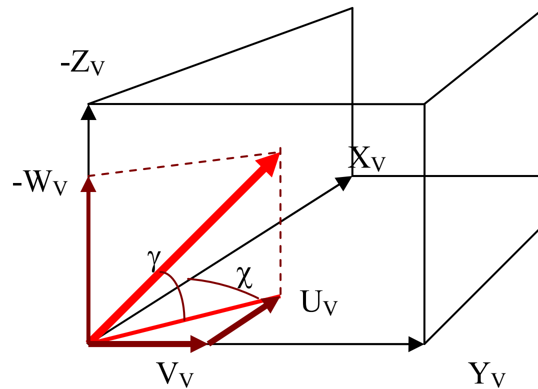


Figura 5. Ángulos de vuelo

5.2.4. Ejes de estabilidad

El avión dispone de cuatro elementos de control:

- el empuje de los motores (δ_{TH1}, δ_{TH2} que consideramos como una única acción $\delta_{TH} = \delta_{TH1} + \delta_{TH2}$)
- Tres elementos aerodinámicos diseñados para producir un momento en torno a cada uno de los tres ejes del avión:
  1. La desviación δ_A de los alerones produce un momento según el eje OX.
  2. La desviación δ_T del timón de profundidad (tailplane) produce un momento según el eje OY, y
  3. La desviación δ_R del timón de dirección produce un momento según el eje OZ.

En cada caso una desviación positiva de la superficie de control produce un momento negativo en el eje apropiado.

5.2.5. Modelo RCAM

5.2.5.1.Modelo Longitudinal

Variables de estado

$X(1) = q$ velocidad de cabeceo en ejes cuerpo rad/seg
$X(2) = \theta$ ángulo de cabeceo (ángulo de Euler) rad
$X(3) = u_B$ componente x de la velocidad inercial en ejes cuerpo m/seg
$X(4) = w_B$ componente z de la velocidad inercial en ejes cuerpo m/seg

Señales de medida

$Y(1) = q$ velocidad de cabeceo en ejes cuerpo rad/seg
$Y(2) = n_x$ factor de carga horizontal en ejes cuerpo = F_x/mg -
$Y(3) = n_z$ factor de carga vertical en ejes cuerpo = F_z / mg -
$Y(4) = w_V$ componente z de la velocidad inercial en ejes F_V m/seg
$Y(5) = V = Va$ velocidad inercial total o velocidad con respecto al aire m/seg

Señales de control

$U(1) = \delta_T$ deflexión del alerón trasero rad
$U(2) = \delta_{TH}$ posición de aceleración de los motores rad

Las señales de control actúan sobre unos actuadotes que se modelan como sistemas de primer orden. Incluimos la dinámica de los actuadotes en el modelo, para ello introducimos dos nuevos estados, uno X_T para el modelo de primer orden del alerón trasero, y otro X_{TH} para el modelo de primer orden del acelerador.

El modelo resultante es:

$$X' = AX + BU \qquad Y = CX + DU$$

Dónde:

```
Alon = [
    -0.9817        0    -0.0007   -0.0153   -2.4360    0.6131
     1.0000        0         0         0         0         0
    -2.2343   -9.7754   -0.0325    0.0744    0.1871   19.6200
    77.3559   -0.7727   -0.2261   -0.6685   -6.4784         0
          0         0         0         0   -6.6667         0
          0         0         0         0         0   -0.6667 ]
```

```
Blon = [
          0        0
          0        0
          0        0
          0        0
     6.6667        0
          0   0.6667 ]
```

```
Clon = [
    1.0000         0          0         0         0         0
    0.0077   -0.0000    -0.0033    0.0076    0.0191    2.0000
   -0.2661         0    -0.0230   -0.0681   -0.6604         0
         0  -79.8667    -0.0289    0.9996         0         0
         0         0     0.9996    0.0295         0         0 ]
```

```
Dlon = [
    0    0
    0    0
    0    0
    0    0
    0    0]
```

De ahí las siguiente ecuaciones. La primera corresponde a la X’ y la segunda a la Y:

$$\begin{bmatrix} \dot{q} \\ \dot{\theta} \\ \dot{u}_B \\ \dot{w}_B \\ \dot{X}_T \\ \dot{X}_{TH} \end{bmatrix} = \begin{bmatrix} -0.9817 & 0 & -0.0007 & -0.0153 & -2.4360 & 0.6131 \\ 1.0000 & 0 & 0 & 0 & 0 & 0 \\ -2.2343 & -9.7754 & -0.0325 & 0.0744 & 0.1871 & 19.6200 \\ 77.3559 & -0.7727 & -0.2261 & -0.6685 & -6.4784 & 0 \\ 0 & 0 & 0 & 0 & -6.6667 & 0 \\ 0 & 0 & 0 & 0 & 0 & -0.6667 \end{bmatrix} \begin{bmatrix} q \\ \theta \\ u_B \\ w_B \\ X_T \\ X_{TH} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 6.67 & 0 \\ 0 & 0.67 \end{bmatrix} \begin{bmatrix} \delta_T \\ \delta_{TH} \end{bmatrix}$$

$$\begin{bmatrix} q \\ n_x \\ n_z \\ w_V \\ V_A \end{bmatrix} = \begin{bmatrix} 1.0000 & 0 & 0 & 0 & 0 & 0 \\ 0.0077 & 0 & -0.0033 & 0.0076 & 0.0191 & 2.0000 \\ -0.2661 & 0 & -0.0230 & -0.0681 & -0.6604 & 0 \\ 0 & -79.8667 & -0.0289 & 0.9996 & 0 & 0 \\ 0 & 0 & 0.9996 & 0.0295 & 0 & 0 \end{bmatrix} \begin{bmatrix} q \\ \theta \\ u_B \\ w_B \\ X_T \\ X_{TH} \end{bmatrix} \quad (5.1)$$

5.2.5.2. Modelo Lateral

Variables de estado

$X(1) = p$	velocidad de alabeo en ejes cuerpo	rad/seg
$X(2) = r$	velocidad de guiñada en ejes cuerpo	rad/seg
$X(3) = \phi$	ángulo de alabeo (ángulo de Euler), PHI	rad
$X(4) = \psi$	ángulo de guiñada (ángulo de Euler), PSI	rad/seg
$X(5) = v_B$	componente y de la velocidad inercial en ejes cuerpo	m/seg

Señales de medida

$Y(1) = \beta$	ángulo de deslizamiento lateral	rad
$Y(2) = p$	velocidad de alabeo en ejes cuerpo	rad/seg
$Y(3) = r$	velocidad de guiñada en ejes cuerpo	rad/seg
$Y(4) = \phi$	ángulo de alabeo (ángulo de Euler)	rad
$Y(5) = \chi$	ángulo de vuelo horizontal, CHI	rad

Señales de control

$U(1) = \delta_A$	deflexión del alerón	rad
$U(2) = \delta_R$	deflexión del timón	rad

Las señales de control actúan sobre unos actuadotes que se modelan como sistemas de primer orden. Incluimos la dinámica de los actuadotes en el modelo, para

ello introducimos dos nuevos estados, uno X_A para el modelo de primer orden del alerón, y otro X_R para el modelo de primer orden del timón.

El modelo resultante es:

X'= AX + BU	**Y= CX + DU**

Donde:

```
Alat= [
   -1.2667    0.5498    0      0    -0.0214   -0.8402   0.2568
    0.0522   -0.5207    0      0     0.0046   -0.0176  -0.3332
    1.0000    0.0289    0      0     0         0        0
    0         1.0004    0      0     0         0        0
    2.30977  -79.9666   9.7896 0    -0.1699    0        2.0384
    0         0         0      0     0        -6.6667   0
    0         0         0      0     0         0       -3.3333]

Blat = [
    0      0
    0      0
    0      0
    0      0
    0      0
    6.67   0
    0      3.33]

Clat = [
    0        0        0        0        0.0125   0   0
    1.0000   0        0        0        0        0   0
    0        1.0000   0        0        0        0   0
    0        0        1.0000   0        0        0   0
    0        0       -0.0288   1.0000   0.0125   0   0 ]

Dlat = zeros(5,2)
```

De aquí se deducen estas ecuaciones, que como en el modelo longitudinal, corresponden la primera a la X' y la segunda a la Y:

$$\begin{bmatrix} \dot{p} \\ \dot{r} \\ \dot{\phi} \\ \dot{\psi} \\ \dot{v}_B \\ \dot{X}_A \\ \dot{X}_R \end{bmatrix} = \begin{bmatrix} -1.2667 & 0.5498 & 0 & 0 & -0.0214 & -0.8402 & 0.2568 \\ 0.0522 & -0.5207 & 0 & 0 & 0.0046 & -0.0176 & -0.3332 \\ 1.0000 & 0.0289 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.0004 & 0 & 0 & 0 & 0 & 0 \\ 2.3097 & -79.9666 & 9.7896 & 0 & -0.1699 & 0 & 2.0384 \\ 0 & 0 & 0 & 0 & 0 & -6.6667 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -3.3333 \end{bmatrix} \begin{bmatrix} p \\ r \\ \phi \\ \psi \\ v_B \\ X_A \\ X_R \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 6.67 & 0 \\ 0 & 3.33 \end{bmatrix} \begin{bmatrix} \delta_A \\ \delta_R \end{bmatrix}$$

$$\begin{bmatrix} \beta \\ p \\ r \\ \phi \\ \chi \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0.0125 & 0 & 0 \\ 1.0000 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.0000 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.0000 & 0 & 0 & 0 & 0 \\ 0 & 0 & -0.0288 & 1.0000 & 0.0125 & 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ r \\ \phi \\ \psi \\ V_B \\ X_a \\ X_r \end{bmatrix} \qquad (5.2)$$

Otras señales intermedias que es importante conocer y que se usan en el cálculo de las anteriores son:

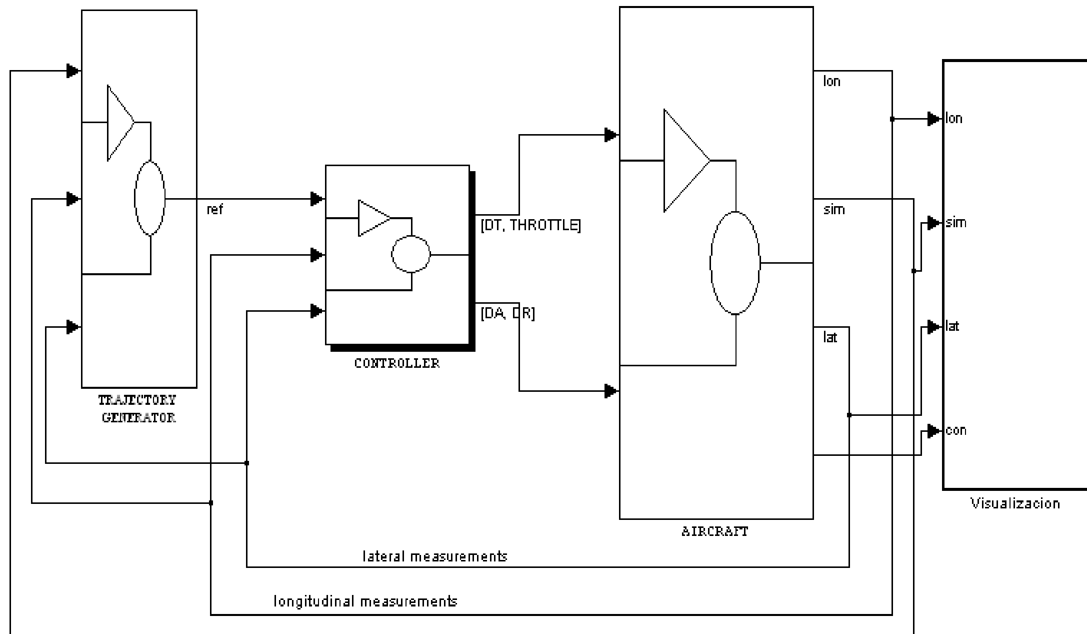
u_V	componente x de la velocidad inercial en ejes F_V	m/seg
v_v	componente y de la velocidad inercial en ejes F_V	m/seg
x	posición x del CoG en F_E	m
y	posición y del CoG en F_E	m
z	posición z del CoG en F_E	m
α	ángulo de ataque	rad
γ	ángulo de vuelo vertical	rad

5.2.5.3.Diagrama modelo RCAM

A continuación mostramos un diagrama de cajas del funcionamiento de la dinámica del avión, como se observa, para conseguirle necesitamos utilizar:

1. Un módulo *Controller*, que aporta al avión los elementos de control (DT, TH, DA, DR) que este utiliza para modificar su posición.
2. Un módulo *Trajectory_Generator,* que calcula las referencias que se le pasarán al controlador. Se ven detalladamente a continuación.
3. Un módulo *AirCraft*, que calcula tanto el estado actual del avión X, como la salida Y, (vistos anteriormente).
4. Un módulo *Visualización*. Este modulo sólo es necesario para visualizar los valores que hemos ido obteniendo. En nuestro caso carece de sentido.

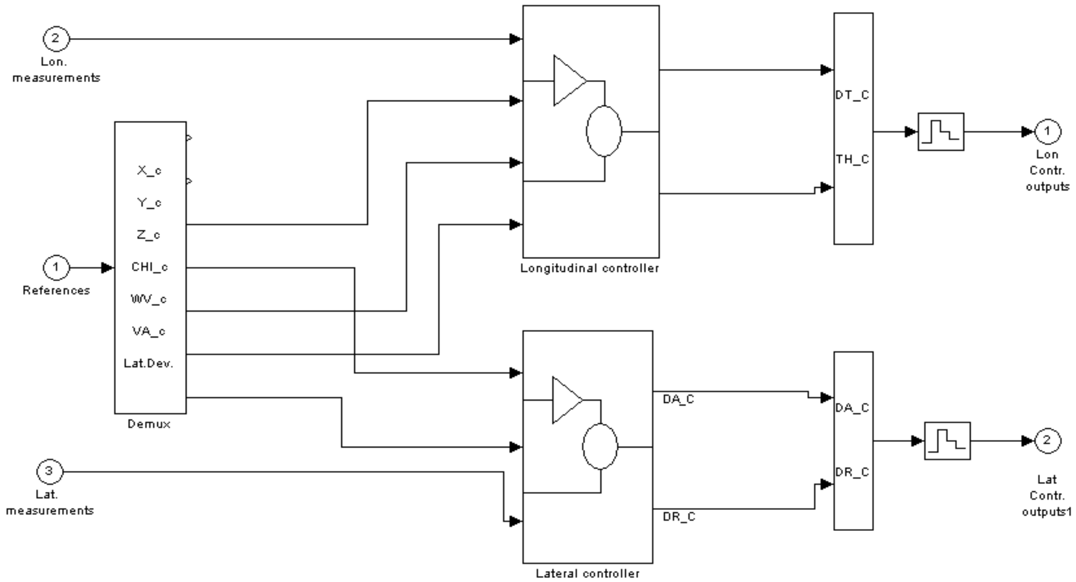
A continuación mostramos un diagrama de cajas del funcionamiento de la dinámica del avión, como se observa, para conseguirle necesitamos utilizar un controlador que aporta al avión los elementos de control (DT, TH, DA, DR) que este utiliza para modificar su posición:



Controlador

Como se puede ver en el esquema, se divide en dos:

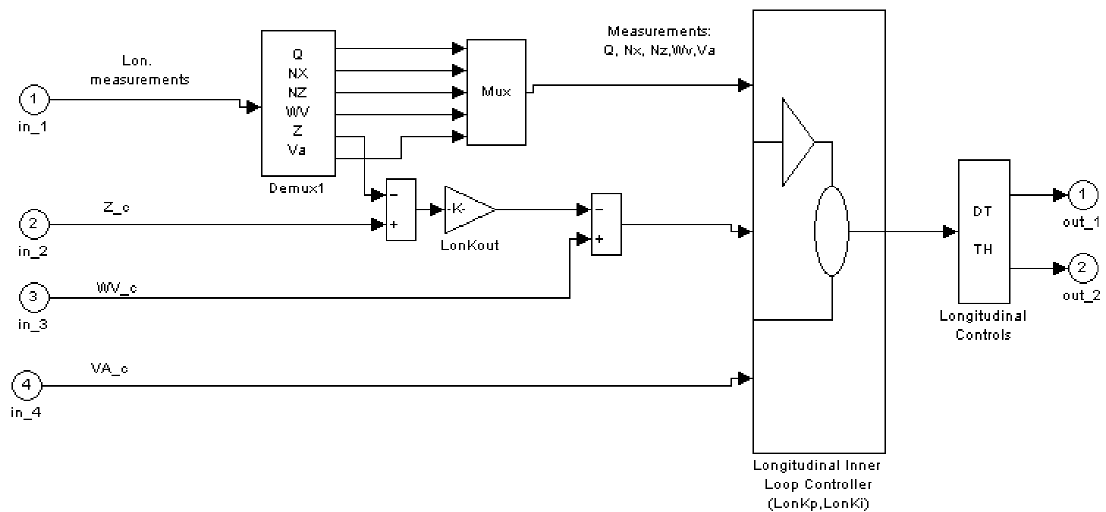
1. Controlador longitudinal
2. Controlador lateral



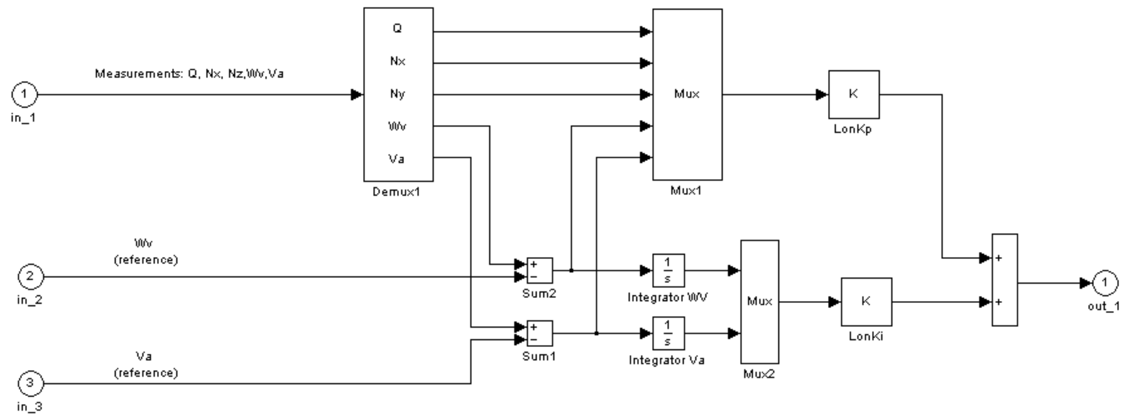
1. Controlador Longitudinal

La ley de control tiene un lazo externo para el control de la altura y un lazo interno para estabilización y control de las velocidades total y vertical en ejes F_V. Las figuras siguientes muestran los lazos de control externo e interno.

Lazo externo:



Lazo interno:



Se toman como constantes los siguientes vectores de ganancia:

LonKp = [
0.4755 0.0532 -0.0838 -0.0169 -0.0055
0.0455 -1.3063 -0.3047 -0.0152 -0.1221]

LonKi = [
-0.0033 -0.0014
0.0004 -0.0227]

LonKout = - 0.1027

2. Controlador Lateral

La ley de control tiene un lazo externo para el control de la desviación lateral, que no usamos, y un lazo externo para estabilización y control de la desviación lateral β y del ángulo de trayectoria horizontal χ.. Las figuras siguientes muestran los lazos de control externo e interno

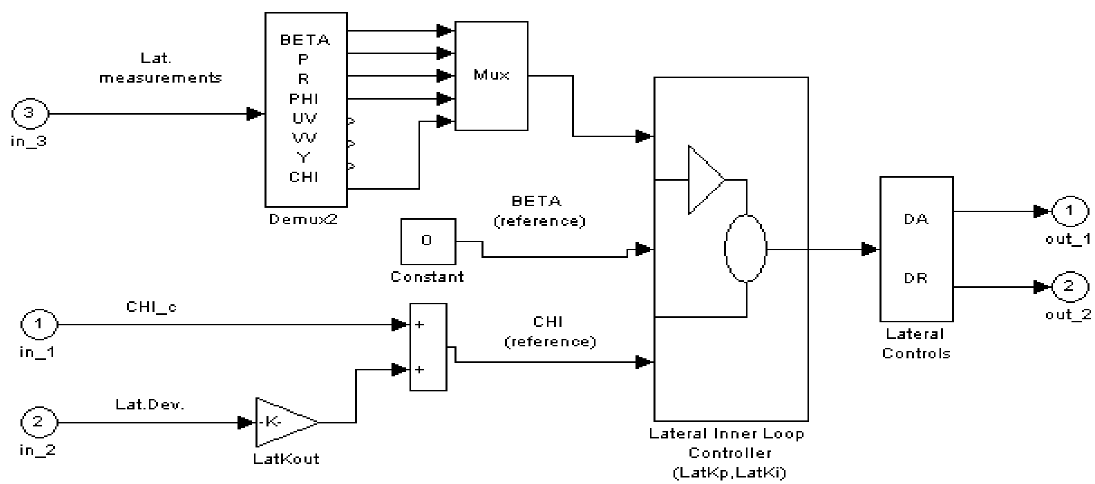
Los valores de los vectores de ganancia son:

LatKp = [
-3.6246 1.7016 2.9057 3.0480 13.1933
-1.5216 -0.0782 2.4251 -0.2268 1.0320]

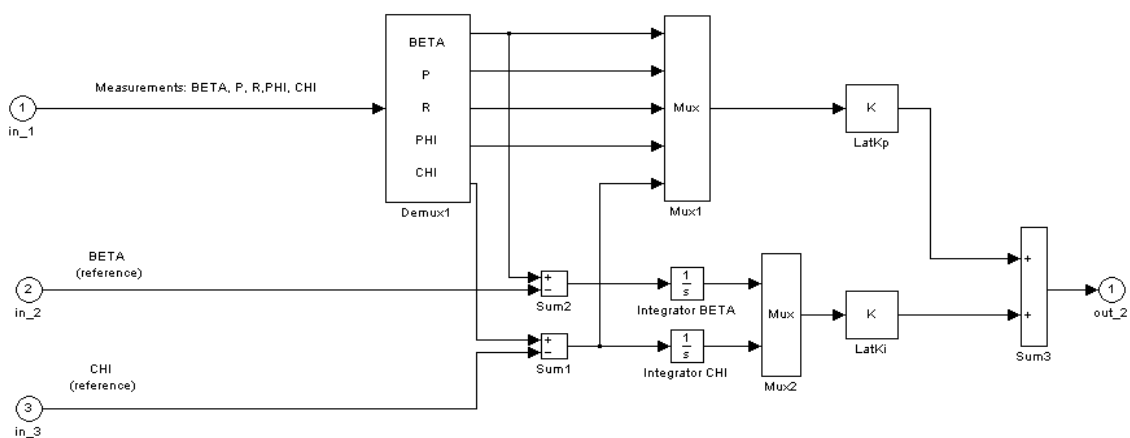
LatKi = [
0.6869 2.2288
-0.7237 0.1820]

LatKout = 0.001;

Lazo externo.

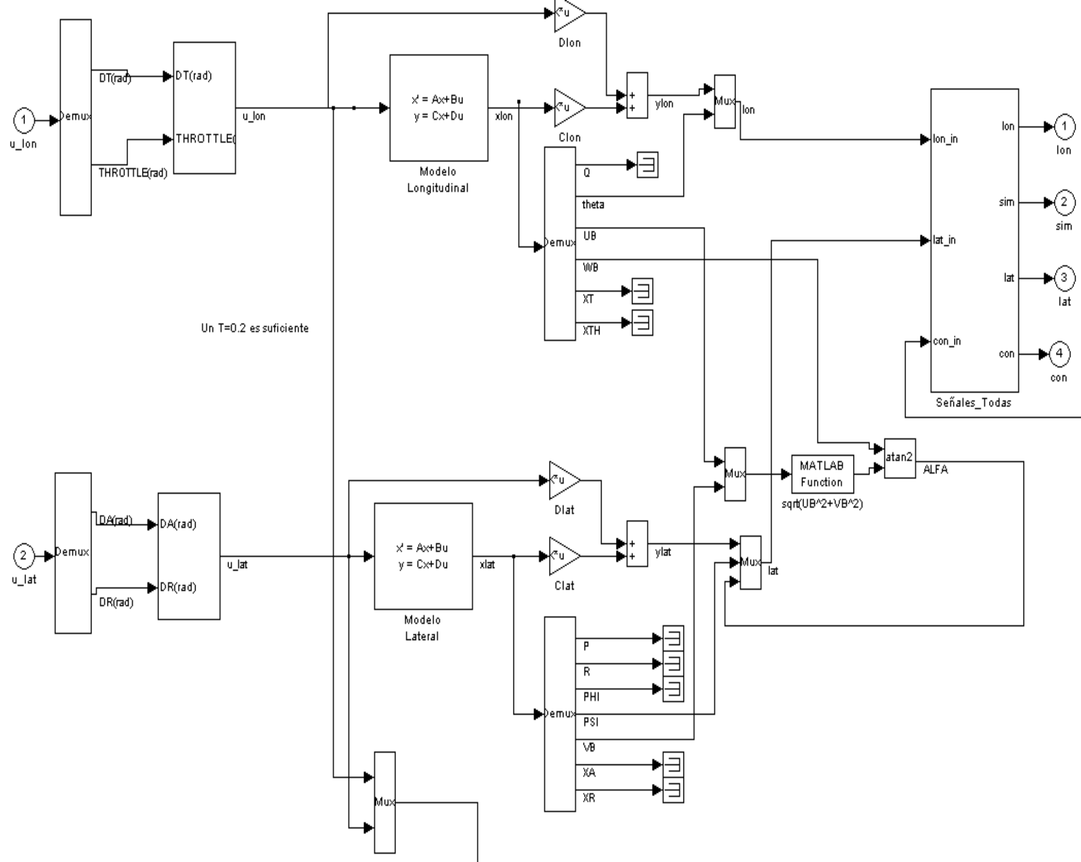


Lazo interno:



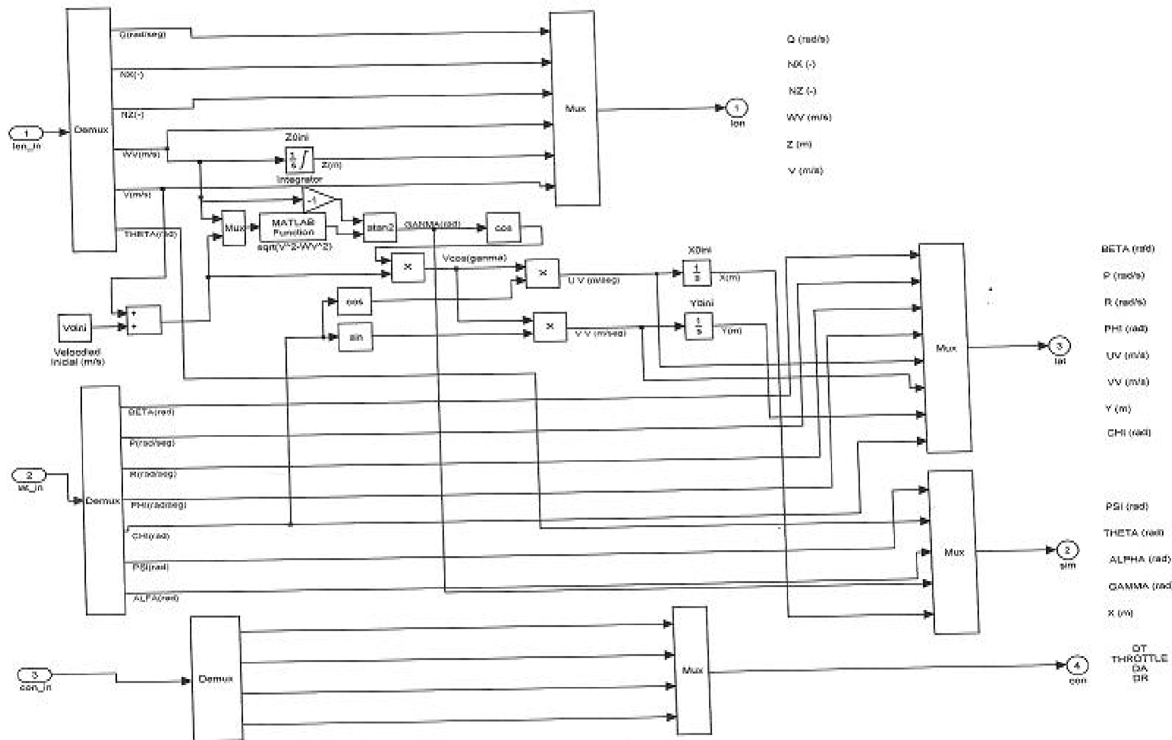
AirCraft

Aunque, ya está explicado anteriormente, mostramos ahora los diagramas detallados de cada parte del módulo principal del modelo RCAM.



Ambos modelos, el longitudinal y el lateral, tienen separada la ecuación de estado de la salida de modo que las señales del vector de estado se usan para generar otras señales. En el bloque Señales_Todas se utilizan las medidas y otras señales de estado para generar todas las señales ligadas a la evolución del avión. Las señales son: las velocidades U_V, V_v, en ejes F_V, las posiciones X, Y y Z del CoG en ejes F_E y el ángulo vertical de vuelo γ. También se da el valor inicial a la velocidad total.

Esquema del bloque Señales_Todas:



5.2.5.4.Señales de referencia

Las señales de referencia se dan el bloque TRAJECTORY GENERATOR que se usan en el modelo son:

1. Modelo longitudinal:

   - las variaciones respecto a la velocidad inicial total Va_c
   - la velocidad según la dirección Z en ejes F_v , WV_c
   - la altura deseada Z_c

Es importante darse cuenta que la integración de la señal de velocidad proporciona la referencia de la señal de altura. No obstante, un cambio en la referencia de altura se puede dar sin dar una referencia de velocidad WV.

2. Modelo lateral:

   - el rumbo se da con la señal CHI_c.

No se da referencia de desviación lateral. Esta señal no es necesaria para el control del avión.

5.2.5.5. Valores Iniciales

El valor inicial del vector de estado del modelo longitudinal debe ser cero. Ya que los valores corresponden a variaciones respecto de los valores de trimado, que podemos considerar también nulos.

El valor inicial del vector de estado del modelo lateral debe ser también cero, salvo la componente del ángulo de guiñada que indica el rumbo inicial del avión y que puede ser el ángulo que se desee. Fijarse que el vector columna de la matriz de estado A correspondiente a esta componente es todo cero. Para evitar saltos bruscos al iniciar la simulación la señal de referencia CHI_c debe estar al mismo valor del ángulo de guiñada inicial.

Otros valores iniciales se dan en el bloque *AIRCRAFT/Señales_Todas*. La velocidad inicial está puesta a un valor constante de 80 m/seg. En este bloque también se deben iniciar los integradores que proporcionan los valores iniciales para las posiciones del CoG del avión en ejes F_E

Es importante darse cuenta que la velocidad total que se propaga al control y a la visualización es la variación respecto al valor inicial, pero la real es la que se utiliza para calcular el ángulo vertical de vuelo, y también las posiciones y velocidades en ejes F_v.

Valores Iniciales:

V0ini = 80;   % Velocidad Total inicial en m/seg

X0ini = 0;   % Posicion inicial del CoG en eje X (m)
Y0ini = 0;   % Posicion inicial del CoG en eje Y (m)
Z0ini = 0;   % Posicion inicial del CoG en eje Z (m)

T = 0.2;   % Periodo de integración

5.2.6. Superficies de mando y control

Además de que un avión vuele, es necesario que este vuelo se efectúe bajo control del piloto; que el avión se mueva respondiendo a sus órdenes. Los primeros pioneros de la aviación estaban tan preocupados por elevar sus artilugios que no prestaban mucha atención a este hecho; por suerte para ellos nunca estuvieron lo suficientemente alto y rápido como para provocar o provocarse males mayores.

Una de las contribuciones de los hermanos Wright fue el sistema de control del avión sobre sus tres ejes; su Flyer disponía de timón de profundidad, timón de

dirección, y de un sistema de torsión de las alas que producía el alabeo.

Por otro lado, es de gran interés contar con dispositivos que, a voluntad del piloto, aporten sustentación adicional (o no-sustentación) facilitando la realización de ciertas maniobras.

Para lograr una u otra funcionalidad se emplean superficies aerodinámicas, denominándose primarias a las que proporcionan control y secundarias a las que modifican la sustentación.

Las superficies de mando y control modifican la aerodinámica del avión provocando un desequilibrio de fuerzas, una o más de ellas cambian de magnitud. Este desequilibrio, es lo que hace que el avión se mueva sobre uno o más de sus ejes, incremente la sustentación, o aumente la resistencia.

5.2.6.1.Ejes del avión

Se trata de rectas imaginarias e ideales trazadas sobre el avión. Su denominación y los movimientos que se realizan alrededor de ellos son los siguientes:

Eje longitudinal: Es el eje imaginario que va desde el morro hasta la cola del avión. El movimiento alrededor de este eje (levantar un ala bajando la otra) se denomina alabeo (en ingles "roll"). También se le denomina eje de alabeo, nombre que parece más lógico pues cuando se hace referencia a la estabilidad sobre este eje, es menos confuso hablar de estabilidad de alabeo que de estabilidad "transversal".

Eje transversal o lateral: Eje imaginario que va desde el extremo de un ala al extremo de la otra. El movimiento alrededor de este eje (morro arriba o morro abajo) se denomina cabeceo ("pitch" en ingles). También denominado eje de cabeceo, por las mismas razones que en el caso anterior.

Eje vertical: Eje imaginario que atraviesa el centro del avión. El movimiento en torno a este eje (morro virando a la izquierda o la derecha) se llama guiñada ("yaw" en ingles). Denominado igualmente eje de guiñada.

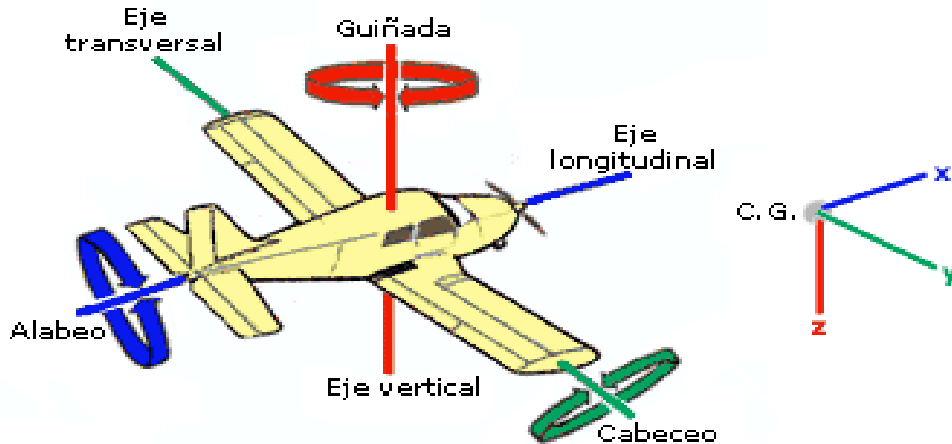


Fig.1.5.1 - Ejes del avión y movimientos sobre ellos.

En un sistema de coordenadas cartesianas, el eje longitudinal o de alabeo sería el eje "x"; el eje transversal o eje de cabeceo sería el eje "y", y el eje vertical o eje de guiñada sería el eje "z". El origen de coordenadas de este sistema de ejes es el centro de gravedad del avión.

5.2.6.2. Superficies primarias

Son superficies aerodinámicas movibles que, accionadas por el piloto a través de los mandos de la cabina, modifican la aerodinámica del avión provocando el desplazamiento de este sobre sus ejes y de esta manera el seguimiento de la trayectoria de vuelo deseada.

Las superficies de control son tres: alerones, timón de profundidad y timón de dirección. El movimiento en torno a cada eje se controla mediante una de estas tres superficies. La diferencia entre un piloto y un conductor de aviones es el uso adecuado de los controles para lograr un movimiento coordinado. Veamos cuales son las superficies de control, como funcionan, y como las acciona el piloto.

Alerones: Palabra de origen latino que significa "ala pequeña", son unas superficies móviles, situadas en la parte posterior del extremo de cada ala, cuyo accionamiento provoca el movimiento de alabeo del avión sobre su eje longitudinal. Su ubicación en el extremo del ala se debe a que en esta parte es mayor el par de fuerza ejercido.

El piloto acciona los alerones girando el volante de control ("cuernos") a la izquierda o la derecha, o en algunos aviones moviendo la palanca de mando a la izquierda o la derecha.

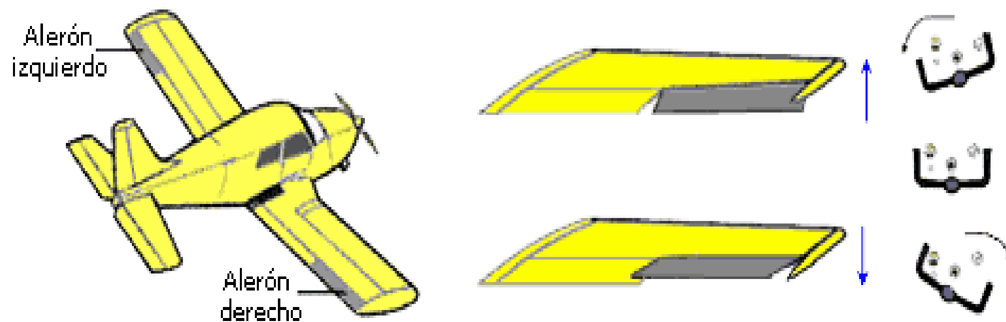


Fig.1.5.2 - Alerones y mando de control.

Funcionamiento: Los alerones tienen un movimiento asimétrico. Al girar el volante hacia un lado, el alerón del ala de ese lado sube y el del ala contraria baja, ambos en un ángulo de deflexión proporcional a la cantidad de giro dado al volante. El alerón arriba en el ala hacia donde se mueve el volante implica menor curvatura en esa parte del ala y por tanto menor sustentación, lo cual provoca que esa ala baje; el alerón abajo del ala contraria supone mayor curvatura y sustentación lo que hace que esa ala suba. Esta combinación de efectos contrarios es lo que produce el movimiento de alabeo hacia el ala que desciende.

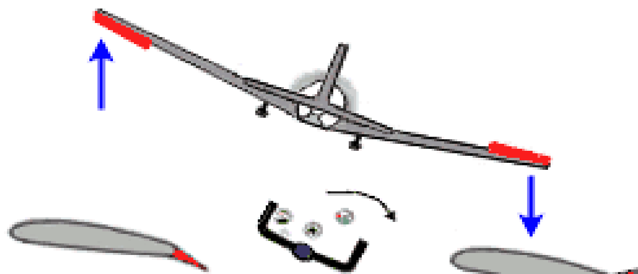


Fig.1.5.3 - Funcionamiento de los alerones.

Supongamos por ejemplo que queremos realizar un movimiento de alabeo a la derecha: giramos el volante a la derecha; el alerón del ala derecha sube y al haber menos sustentación esa ala desciende; por el contrario, el alerón abajo del ala izquierda provoca mayor sustentación en esa ala y que esta ascienda.

Timón de profundidad: Es la superficie o superficies móviles situadas en la parte posterior del empenaje horizontal de la cola del avión. Aunque su nombre podría sugerir que se encarga de hacer elevarse o descender al avión, en realidad su accionamiento provoca el movimiento de cabeceo del avión (morro arriba o morro abajo) sobre su eje transversal. Obviamente, el movimiento de cabeceo del avión provoca la modificación del ángulo de ataque; es decir que el mando de control del timón de profundidad controla el ángulo de ataque.

En algunos aviones, el empenaje horizontal de cola es de una pieza haciendo las funciones de estabilizador horizontal y de timón de profundidad. El timón de profundidad es accionado por el piloto empujando o tirando del volante o la palanca de control, y suele tener una deflexión máxima de 40º hacia arriba y 20º hacia abajo.

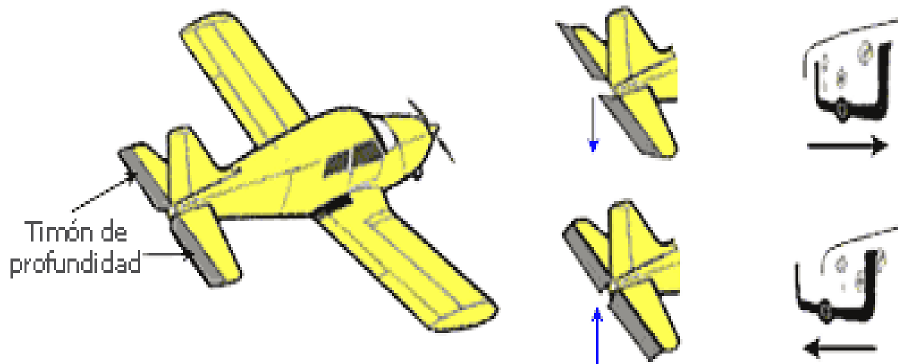


Fig.1.5.4 - Timón de profundidad y mando de control.

Funcionamiento: Al tirar del volante de control, esta superficie sube mientras que al empujarlo baja, en algunos aviones se mueve la totalidad del empenaje horizontal. El timón arriba produce menor sustentación en la cola, con lo cual esta baja y por tanto el morro sube (mayor ángulo de ataque). El timón abajo aumenta la sustentación en la cola, esta sube y por tanto el morro baja (menor ángulo de ataque). De esta manera se produce el movimiento de cabeceo del avión y por extensión la modificación del ángulo de ataque.

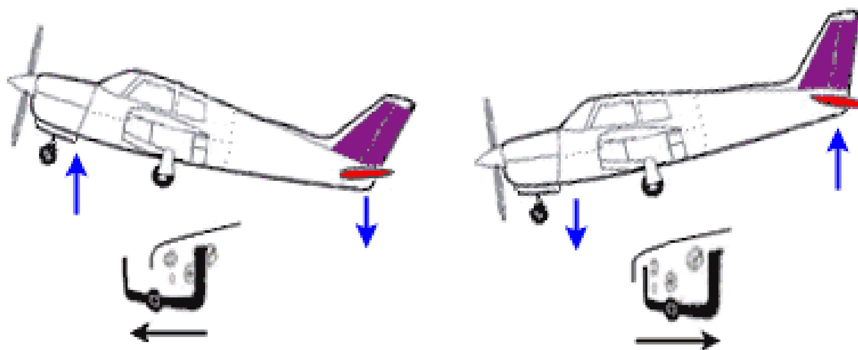


Fig.1.5.5 - Funcionamiento del timón de profundidad.

Timón de dirección: Es la superficie móvil montada en la parte posterior del empenaje vertical de la cola del avión. Su movimiento provoca el movimiento de guiñada del avión sobre su eje vertical, sin embargo ello no hace virar el aparato, sino que se suele utilizar para equilibrar las fuerzas en los virajes o para centrar el avión en la trayectoria deseada. Suele tener una deflexión máxima de 30º a cada lado.

Esta superficie se maneja mediante unos pedales situados en el suelo de la cabina.

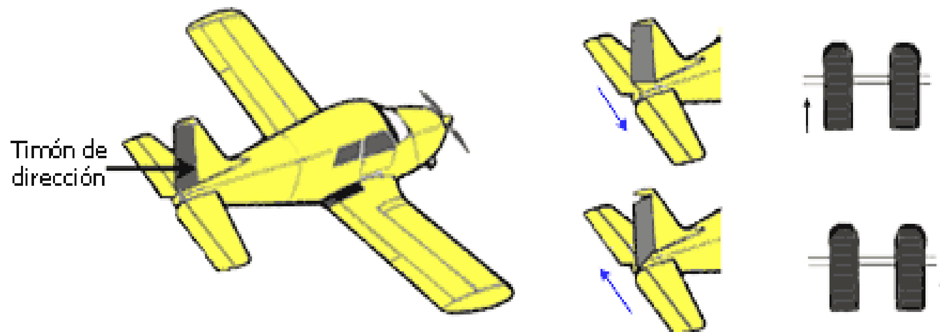


Fig.1.5.6 - Timón de dirección y pedales de control.

Funcionamiento: Al pisar el pedal derecho, el timón de dirección gira hacia la derecha, provocando una reacción aerodinámica en la cola que hace que esta gire a la izquierda, y por tanto el morro del avión gire (guiñada) hacia la derecha. Al pisar el pedal izquierdo, sucede lo contrario: timón a la izquierda, cola a la derecha y morro a la izquierda.

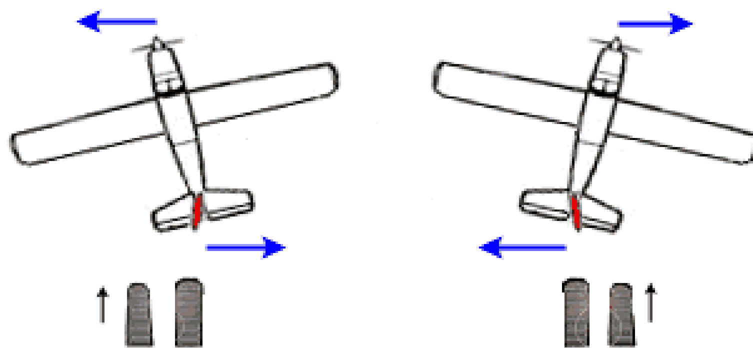


Fig.1.5.7 - Funcionamiento del timón de dirección.

El manejo de los mandos de control, según se ha visto es bastante intuitivo:

- Alabeo a la derecha → volante a la derecha.
- Alabeo a la izquierda → volante a la izquierda.
- Morro abajo (menor ángulo de ataque) → empujar el volante.
- Morro arriba (mayor ángulo de ataque) → tirar del volante.
- Guiñada a la derecha → pedal derecho.
- Guiñada a la izquierda → pedal izquierdo.

Al basarse los mandos de control en principios aerodinámicos, es obvio que su efectividad será menor a bajas velocidades que a altas velocidades. Es conveniente tener esto en cuenta en maniobras efectuadas con baja velocidad.
El que las superficies de control estén lo más alejadas posible del Centro de Gravedad del avión no es casualidad, sino que debido a esta disposición su funcionamiento es más efectivo con menor movimiento de la superficie y menos esfuerzo.

5.2.6.3.Representación con java

Como ya hemos comentado anteriormente, en esta fase nos limitamos a pasar este modelo creado en MATLAB a lenguaje de programación JAVA. Esta parte nos costó mucho trabajo pues nos encontramos con varias dificultades:

- Ningún miembro del equipo conocía MATLAB a fondo, pues aunque habíamos trabajado con este programa en algunas asignaturas, no sabíamos usar, ni entendíamos correctamente *Simulink*. Esto supuso un periodo de aprendizaje.
- Algunas funciones definidas por *Simulink*, no están establecidas en JAVA, por lo que tuvimos que crear métodos que las simulasen.
- Otras funciones de MATLAB, variaban un poco a las definidas en JAVA. Por lo que encontrar el error en estos casos fue complicado.
- Los valores de referencia en MATLAB se cambian en un determinado instante de tiempo. Nosotros en JAVA, no trabajamos en tiempo sino en pasos, por lo que tuvimos que hacer una conversión.
- Nos costó bastante trabajar con los valores iniciales, pues MATLAB actualiza en el instante de tiempo 0 todos sus módulos, mientras que JAVA lo hace secuencialmente.

El modelo RCAM, lo tenemos implementado como una clase aparte dentro del paquete ***modelo.*** Distinguimos dos clases de modelo, una *modeloRCAM* y otra *modeloRCAMBatalla*, pues aunque las dos simulen el comportamiento de un avión. Varían un poco, según los datos que nos interese guardar para una u otra representación.

Definimos brevemente, la representación que hemos hecho en JAVA del modelo RCAM. Indicamos sólo los métodos y atributos más relevantes.

Atributos:

- time : Variable que nos marca cada cuanto tiempo se actualizan los datos. En nuestro caso, como en el modeloRCAM es 0.2.

- referencias: Vector donde guardamos las señales de referencia que le entran al modelo del avión y que harán que modifique su rumbo, su altura o su

velocidad. Estas señales de referencia son: X_c, Z_c, CHI_c, WV_c, VA_c y LatDes.

- longControl: Vector que guarda las señales de control que le entran al controlador longitudinal. Estas señales son: DT y TH.

- latControl: Vector que guarda las señales de control que le entran al controlador lateral. Éstas son: DA y DR.

- XLong: Vector que guarda las variables de estado del modelo longitudinal. Estas variables son: Q, THETA, UB, WB, XT y XTH.

- YLong: Vector que guarda las variables de salida del modelo longitudinal. Estas son: Q, NX, NZ, WV y V.

- XLat: Vector que guarda las variables de estado del modelo lateral: P, R, PHI, PSI, VB, XA y XR

- YLat: Vector que guarda las variables d salida del modelo lateral. BETA, P, R, PHI y CHI.

- V0Ini: Velocidad inicial del avión.

Definimos como constantes *LonKout, LonKp, LonKi, LatKout, LatKp y LatK*i, así como las matrices *ALAT, BLAT, CLAT, DLAT, ALON, BLON, CLON* y *DLON*.

Guardamos en otros atributos también, las señales que nos interesan representar gráficamente.

Metodos:

Destacamos aquellos que se corresponden con cada uno de los módulos del modelo RCAM:

- AirCraft.
- Controlador.
- Trayectory_Generator.
- Señales_Todas.
- ControladorLateral
- ControladorLongitudinal.

Utilizamos métodos auxiliares para calcular las variables de estado y de salida de cada uno de los modelos:

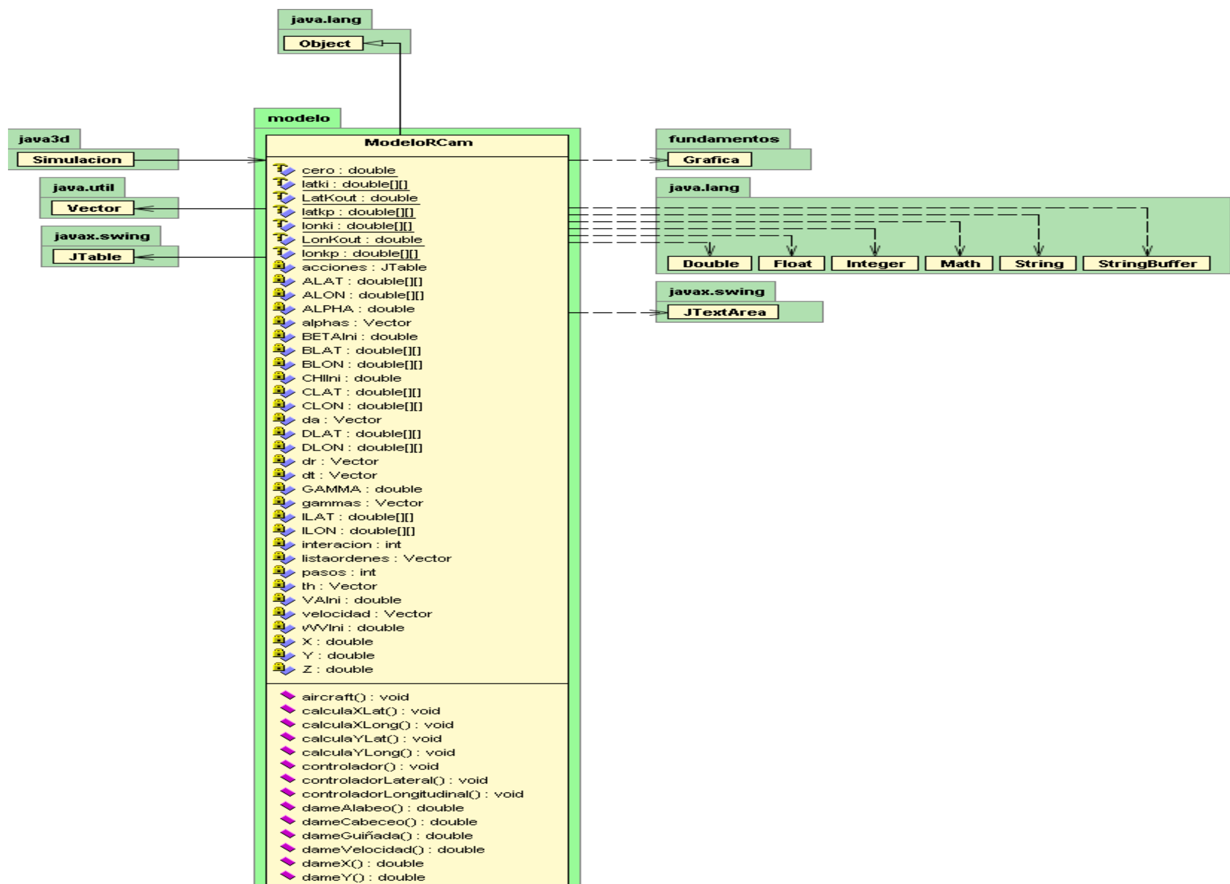
- CalculaXLong
- CalculaXLat.
- CalculaYLong.
- CalculaYLat

Para que fuese más fácil de depurar y de entender, decidimos crear accesores y mutadores para cada una de las señales que forman parte del modelo.

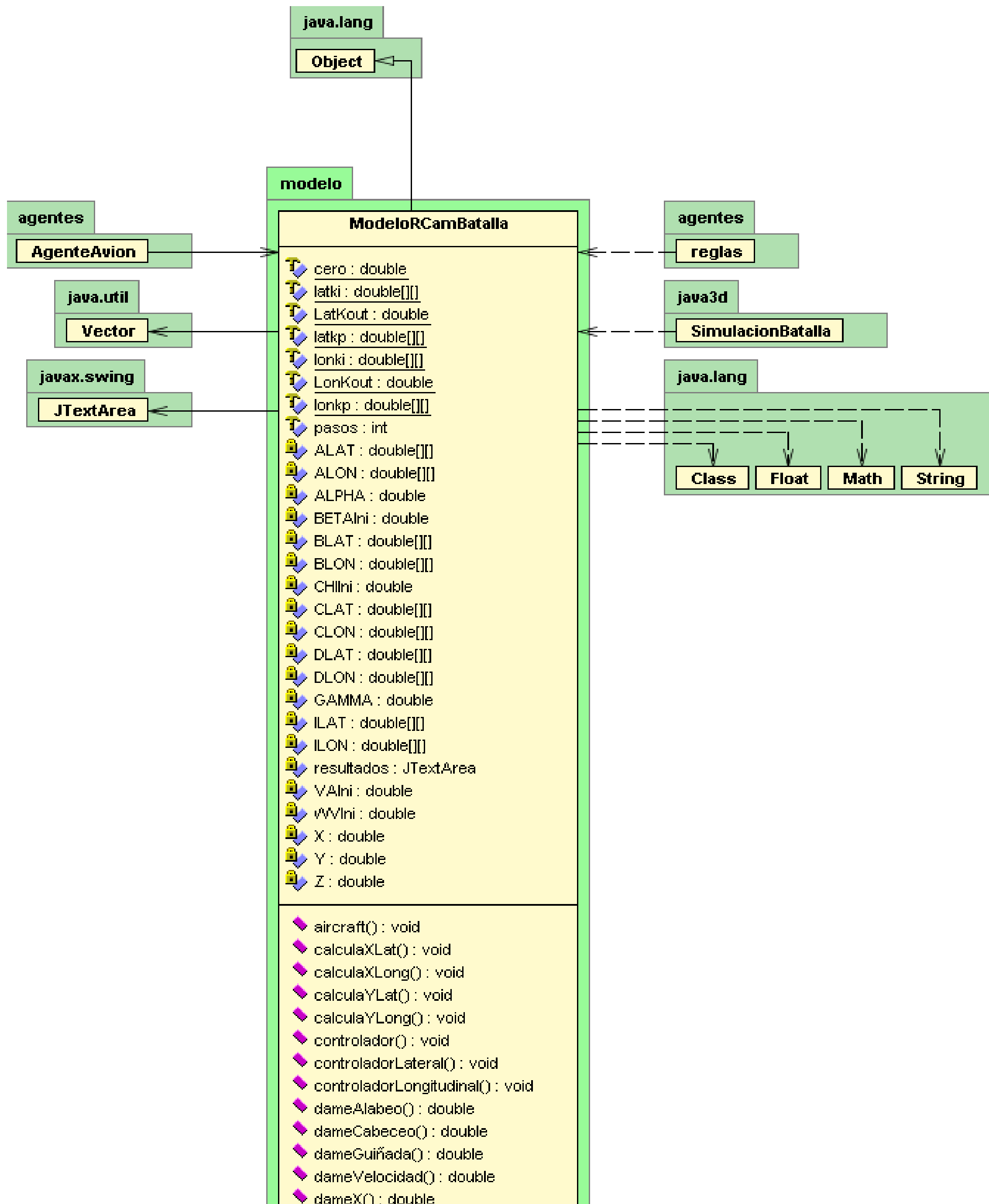
Como métodos auxiliares, que creamos en JAVA, para hacer las funciones ya predefinidas en MATLAB, donde la función que desempeña cada uno de ellos salta a la vista, destacamos:

- sumaMatrices.
- multiplicaMatrices.
- multiplicamatrizescalar
- Integra.
- Tangente.

A continuación mostramos, para que quede más claro, el diagrama de clases de la clase *ModeloRCAM*:



Mostramos también el diagrama e la clase *ModeloRCAMBatalla*



5.3. Tercera Fase

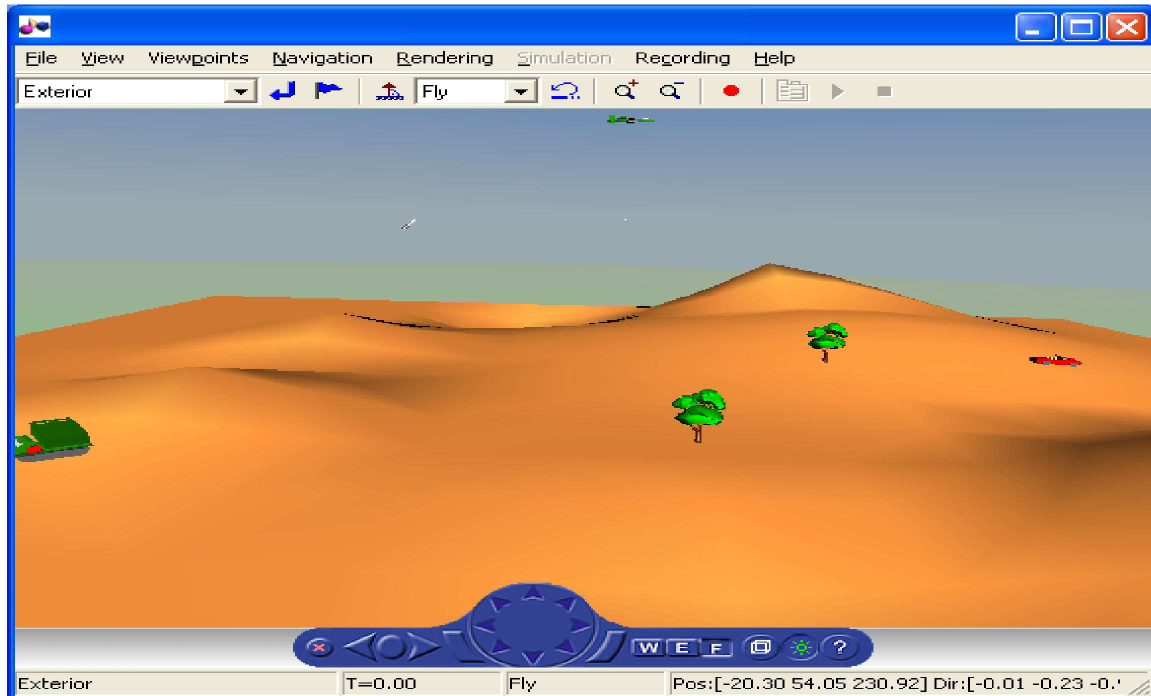
Una vez traducido el modelo RCAM a JAVA, pensamos que lo mejor para probar si funcionaba correctamente era hacer una simulación del movimiento en tres dimensiones. Puesto que ningún componente del grupo había programado antes en tres dimensiones, Matilde nos recomendó hablar con el profesor de *Informática Gráfica* Segundo Esteban San Román, el cual nos aconsejó intentar visualizar primero el avión con las herramientas que proporciona MATLAB y una vez tuviéramos eso funcionando, intentar hacerlo con JAVA 3D con cargadores de objetos definidos mediante VRML para conseguir tener un programa compacto en el que el tiempo de visualización fuese simultaneo al tiempo de ejecución.

5.3.1.1.Representación con MATLAB

Para ello hemos utilizado la herramienta MATLAB y las *ToolBox de VRL* (Virtual Reality Lenguaje).

El modelo del entorno (relieve, paisaje, aviones, etc.), nos lo proporcionó Segundo, junto con un programa que permitía cargarlo y verlo por pantalla. Tuvimos ciertos problemas a la hora de ejecutarlo ya que es necesario tener determinadas bibliotecas (toolbox) para poder visualizarlo. El programa lo modificamos con el objetivo de conseguir una interacción entre nuestro programa JAVA y el programa en MATLAB que nos permitía trabajar con los ficheros WRL (lenguaje en que está diseñado el entorno).

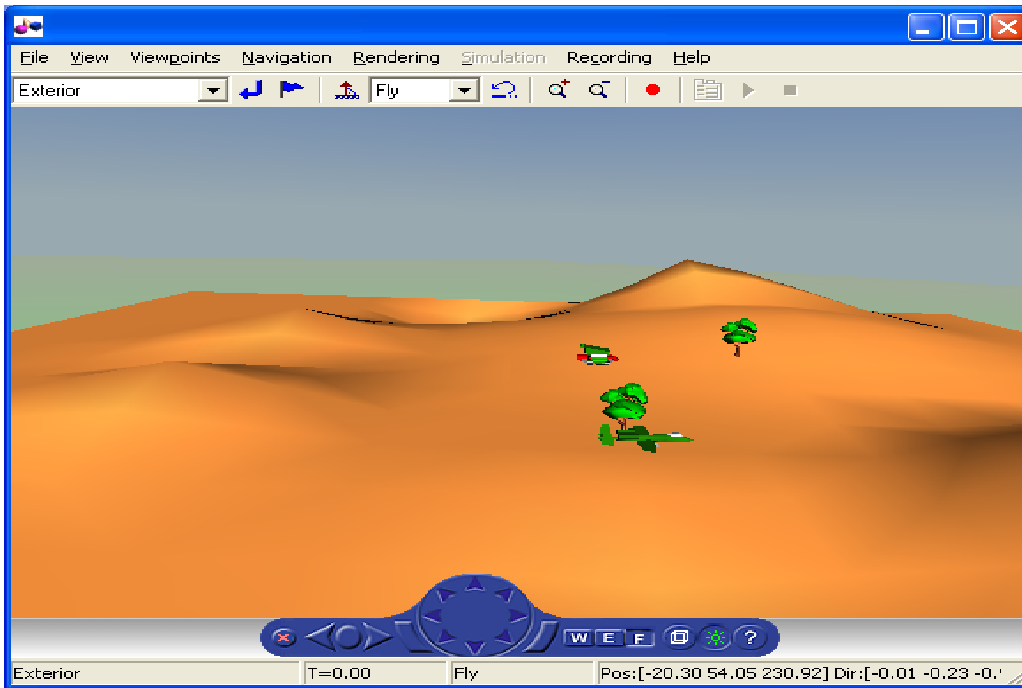
A continuación mostramos una captura de pantalla del fichero inicial, con el que empezamos a trabajar, se puede ver el avión en la parte superior, hacia el cual va dirigido un misil, ya en tierra vemos un coche (a la derecha), y un lanzamisiles (a la izquierda de la pantalla).



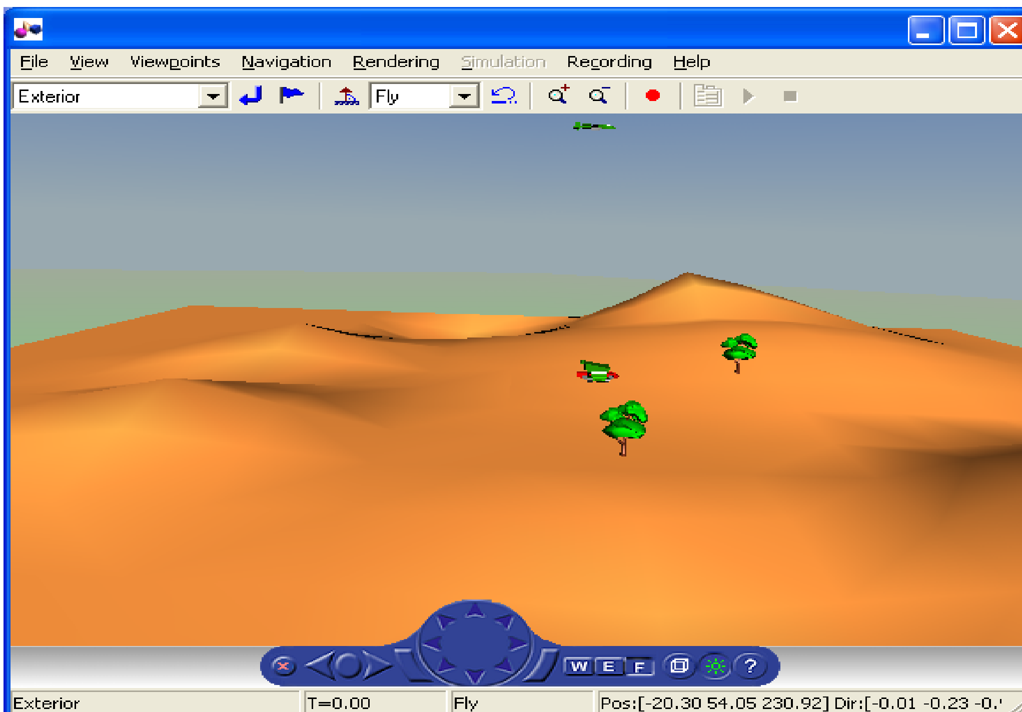
La interacción JAVA-MATLAB nos dio bastantes problemas, derivados del entorno y de la toolbox necesaria. La creación de objetos, ejecución del programa principal y obtención de parámetros y variables fue sencillo debido a las facilidades que proporciona MATLAB.

Volviendo a la representación gráfica, únicamente conseguimos representar el avión de manera estática, pero no con un movimiento significativo. Teníamos errores del *Canvas3D*; depurando el programa paso a paso comprobamos que los valores de las variables eran correctos, y que teníamos en el “workspace” todo lo necesario.

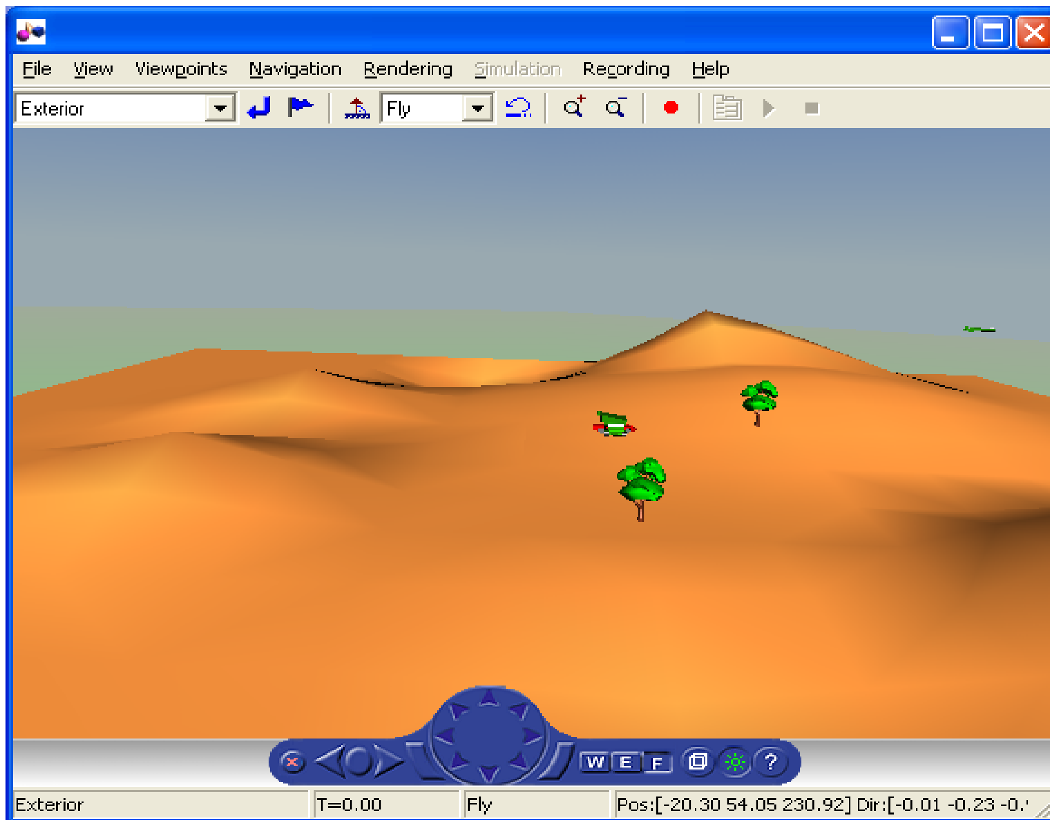
En la siguiente imagen vemos al avión en una posición fija. Esta fue la primera prueba que realizamos y, para posteriormente fuimos complicándolo (para más detalles ver avion.m):



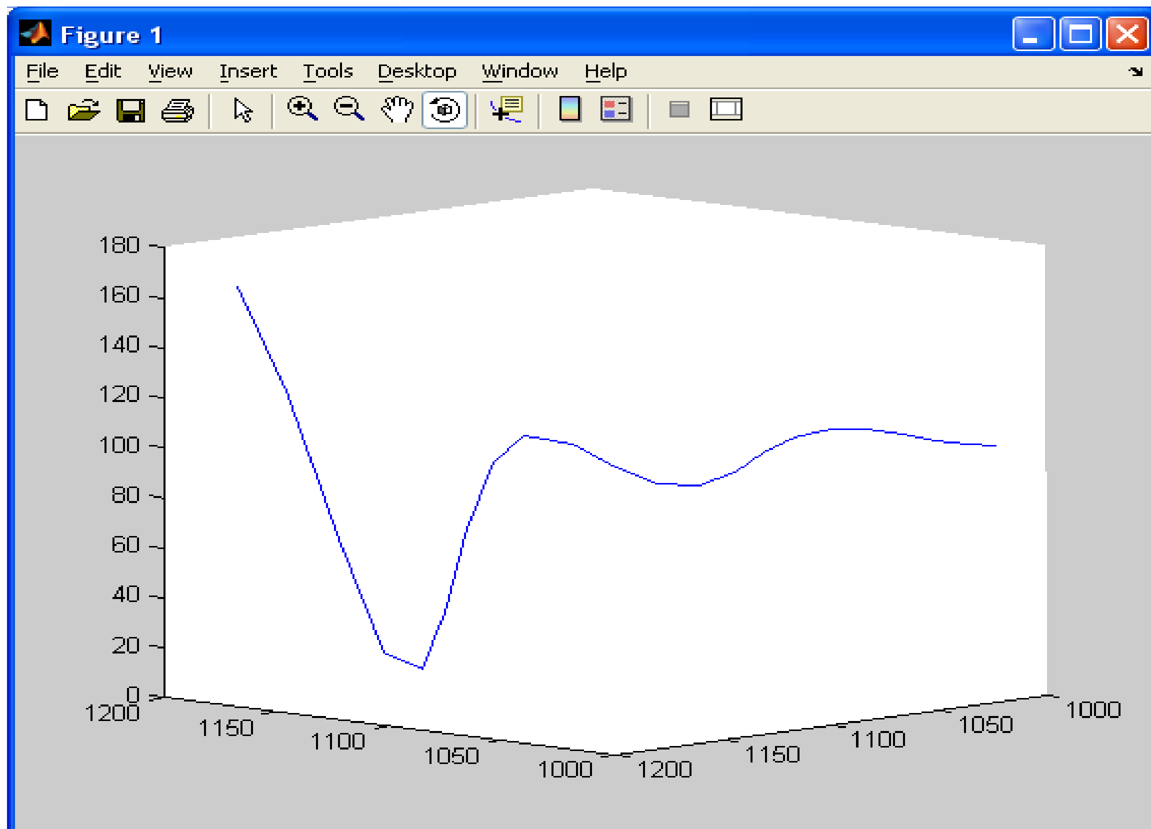
El siguiente paso fue intentar ver el avión en movimiento, lo podemos ver en la parte superior de la imagen, pero el movimiento era prácticamente nulo (para más detalles ver avion2.m):



Finalmente intentamos meter los parámetros calculados en el modelo RCAM, para ver el movimiento real del avión, pero en este paso nos atascamos y no conseguimos hacer que funcionara correctamente (para ver los detalles del código ver avion3.m):



También hicimos pruebas con MATLAB (sin VRML) para ver cómo se movía nuestro agente, hicimos un programa muy sencillo, que simplemente utiliza la función '*plot3* (...)' y representa los valores pasados mediante vectores en los ejes x's, y's y z's. Uno de los resultados obtenidos fue el siguiente (para más detalles ver graf.m):



Llegados a éste punto, y dado los pocos resultados aparentes que habíamos obtenido y las dificultades que nos planteaba MATLAB:

- Los archivos.m no siempre se ejecutaban correctamente
- Errores en la carga
- Pantalla del entorno completamente gris

Decidimos dejar apartada esta parte y comenzar a trabajar con JAVA3D, creando de este modo algo más vistoso. Además tenemos todo el proyecto escrito en el mismo lenguaje y podemos visualizar el programa en tiempo real a la ejecución.

5.3.2. Representación con Java 3D

5.3.2.1.¿Qué es el "API 3D" de JAVA?

Es una serie de paquetes de clases para crear y manipular gráficos tridimensionalmente que son renderizados en un universo virtual. Tiene como ventaja que aísla al programador del sistema de renderizado, que, pasa a estar gestionado automáticamente por JAVA3D.

Un programa en JAVA3D crea objetos que sitúa en una estructura llamada escenario gráfico. Este escenario gráfico tiene una estructura de árbol y es una representación del contenido de un universo virtual y de cómo esté universo es renderizado.

5.3.2.2.El "API 3D" de JAVA

Se utilizan 4 paquetes:

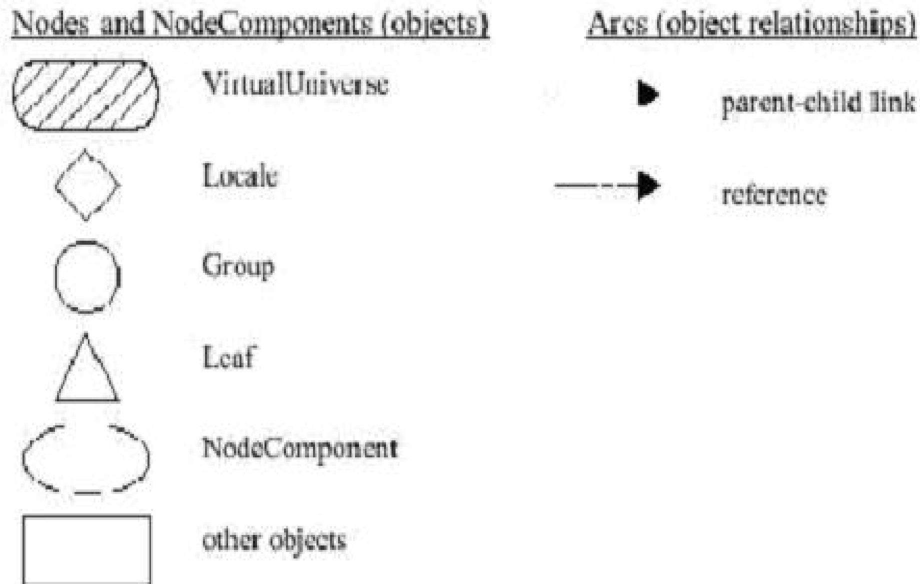
- javax.media.j3d: Es el paquete corazón de Java3D y contiene las clases de bajo nivel.
- com.sun.j3d.utils: Este paquete incluye utilidades adicionales a las clases del primer paquete, entre ellas podemos encontrar, cargadores de contenidos, ayudas a la construcción del escenario gráfico, clases de geometría, utilidades de conveniencia,…
- javax.vecmath: Incluye puntos, vectores, matrices y otros objetos matemáticos.
- java.awt: Se utiliza para crear una ventana y mostrar el renderizado del universo virtual.

5.3.2.3. Construir un escenario gráfico

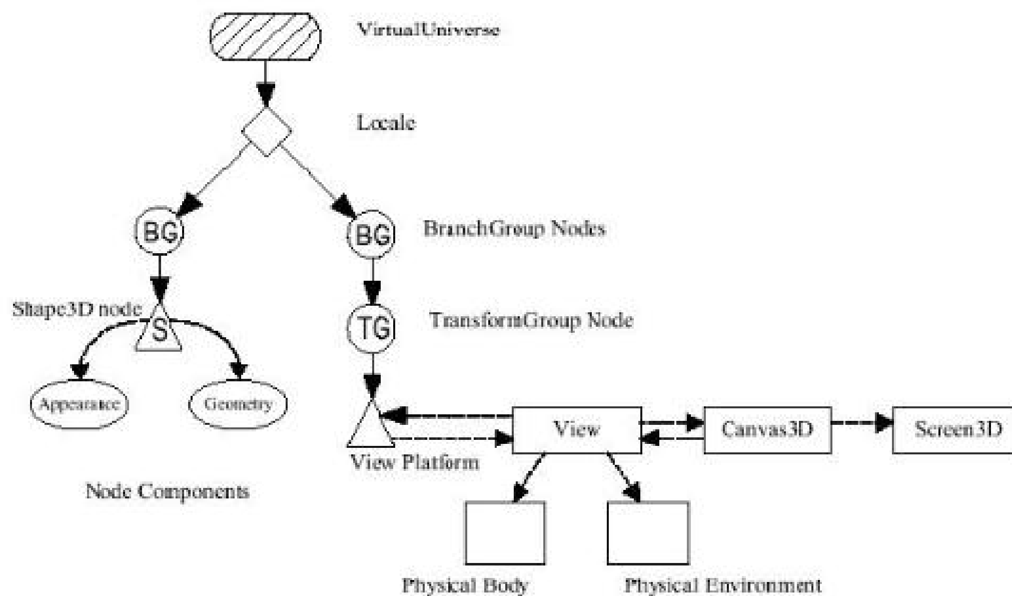
El grafo incluye en una rama tanto los elementos que forman parte de la escena como las transformaciones que les aplica. Se inserta en otra rama los elementos relacionados con el punto de vista del usuario, a este grafo se le llama *SceneGraph*.

El grafo de escena (*SceneGraph*) contiene una descripción de la escena (geometría, información de los atributos, información de visualización necesaria para renderizar la escena desde algún punto de vista). Además permite diseñar la escena basándose en objetos geométricos y permite programar centrándonos en la escena y su composición y no en obtener un código para renderizar con eficiencia.

Ejemplo de los nodos que nos podemos encontrar en el grafo de escena:



Una posible representación gráfica de un grafo de escena puede ser la siguiente:



Objetos del grafo escena

Nodo VirtualUniverse**:** Lista de objetos *Locale* que contienen una serie de nodos del grafo de escena que existen en el universo.

Nodo Locale**:** Conjunto de subgrafos de escena (*BranchGroup*). Se une de forma implícita a un universo virtual cuando se construye. Puede referenciar varios *BranchGroup*, pero no tiene hijos explícitos. Define la localización del universo virtual utilizando coordenadas de alta resolución (*HiResCoord*). Las coordenadas de todos los objetos del grafo de escena, son relativas al *HiResCoord* del *Locale* en el que se encuentran. Las operaciones del objeto *Locale* incluyen establecer y obtener el *HiResCoord* y añadir y eliminar subgrafos.

Los nodos se corresponden con instancias de clases JAVA 3D. A los nodos padre, se les denomina *nodos grupo*. Un arco es una relación entre nodos, estos arcos representan dos tipos de relaciones entre las instancias de JAVA 3D:

- Relación padre – hijo.
- Referencia: Asocia un objeto del tipo *NodeComponent* con un nodo del grafo de escena.

Objetos de agrupación de nodos

Todos los nodos de agrupación pueden tener otros nodos de agrupación y nodos hoja como hijos.

Nodo Group**:** Agrupación de propósito general. Tiene las siguientes operaciones:

1. Añadir hijos.
2. Eliminar hijos
3. Enumerar los hijos del grupo.

Nodo BranchGroup: Raíz de un subgrafo (escena). Tiene las siguientes acciones:

1. Compilarse.
2. Insertarse dentro de un universo.
3. Desconectarse del subgrafo.
4. Cambiarse de padre.

Nodo TransformGroup**:** Especifican una serie de transformaciones espaciales sencillas utilizando objetos *Transform3D*.*

Objetos de nodos hoja

Nodo Leaf**:** Clase abstracta de la que heredan todos los nodos del grafo de escena que no tienen hijos.

Nodo Shape3D**:** Da soporte a la creación de objetos geométricos. Componentes:

- Geometry
- Appearence (color, material, textura...)

Nodo ViewPlatform**:** Ayuda a especificar la localización del punto de vista y hacia qué
dirección está orientado.

Nodo Behavior**:** Permiten que una aplicación modifique el grafo de escena en tiempo de ejecución.

Nodo Background

Nodo Light: Sirve para indicar la iluminación sobre el grafo de escena (*DireccionalLigth,…*)

Otros objetos utilizados

Rama de visualización. Son objetos que se unen al objeto *ViewPlatform*.

Nodo View**:** Este es el objeto principal de visualización.

Nodo Canvas3D**:** Representa una ventana en la que Java 3D dibujará las imágenes.

Nodo Screen3D**:** Información relativa a las propiedades físicas de la pantalla.

Nodo NodeComponent**:** No forman parte de la estructura de árbol, son los responsables de que una figura tenga una cierta geometría bajo una cierta apariencia (color, textura...). Se asocian a través de relaciones de referencia con objetos Leaf para los cuales define la apariencia y la geometría.

5.3.2.4. Construir un escenario gráfico

1. Crear un objeto *Canvas3D*
2. Crear un objeto *SimpleUniverse* que haga referencia al objeto *Canvas3D*
3. Personalizar *SimpleUniverse*
4. Construir la rama de contenido (tiene como raíz un *BranchGroup*)
    - Hacer viva una rama.
    - Compilar una rama.
5. Compilar el gráfico de la rama de contenido
6. Insertar las estructuras de la rama de contenido en el objeto *Locale* del nodo *SimpleUniverse*.

Un objeto *BranchGroup* siempre es la raíz de un subgráfico o rama, existen dos tipos:

Rama de vista gráfica: Indica los parámetros de visualización, como la posición de visualización y la dirección.

Rama de contenido gráfico: Indica el contenido del universo virtual, entre otros, geometrías, apariencias, comportamientos, localizaciones, sonidos y luces.

VENTAJAS

1. Más sencillo que otros procedimientos

INCONVENIENTES

1. El Objeto *SimpleUniverse* no permite tener varias vistas de un Universo virtual.
2. El Objeto *SimpleUniverse* crea una rama de vista gráfica completa. Esta rama incluye un marco donde se proyecta la imagen renderizada, en nuestro caso ese marco es el objeto *Canvas3D*. El renderizado es una proyección del universo sobre el marco, inicialmente este marco esta centrado en el punto (0,0,0). Un programa típico de JAVA3D movería la vista hacia atrás (haciendo Z positivo) para hacer que los objetos se acerquen.

3. El Objeto *SimpleUniverse* tiene un objeto propio llamado *ViewingPlatform*. Utilizamos el método *setNominalViewingTransform* de esta clase para seleccionar la distancia nominal de la vista a 2,42m. De esta manera los objetos con 2 metros entraran en el marco de la imagen.

4. El paso 4 del procedimiento de construcción no es tan automático como la construcción de la rama de la vista gráfica. Este es el paso que mayor esfuerzo de programación requiere. Para aliviar el esfuerzo necesario en este paso nos hemos basado en la utilización de cargadores de archivos VRML. Estos archivos VRML contienen información sobre los objetos que queremos tener en nuestro Universo Virtual, y los cargadores de este tipo de archivo se encargan de procesar la información que contienen y generan la de nodos necesaria para representar los objetos en nuestro marco de JAVA3D (*Canvas3D*).

Hacer viva a una rama de contenido gráfico

Una rama de contenido gráfico se hace viva cuando se inserta al objeto *Locale*. Todos los objetos de la rama también estarán vivos y antes de renderizar una rama hay que hacerla viva. Los parámetros de los objetos vivos no se pueden modificar a no ser que se haya especificado esta propiedad antes de hacer viva la rama.

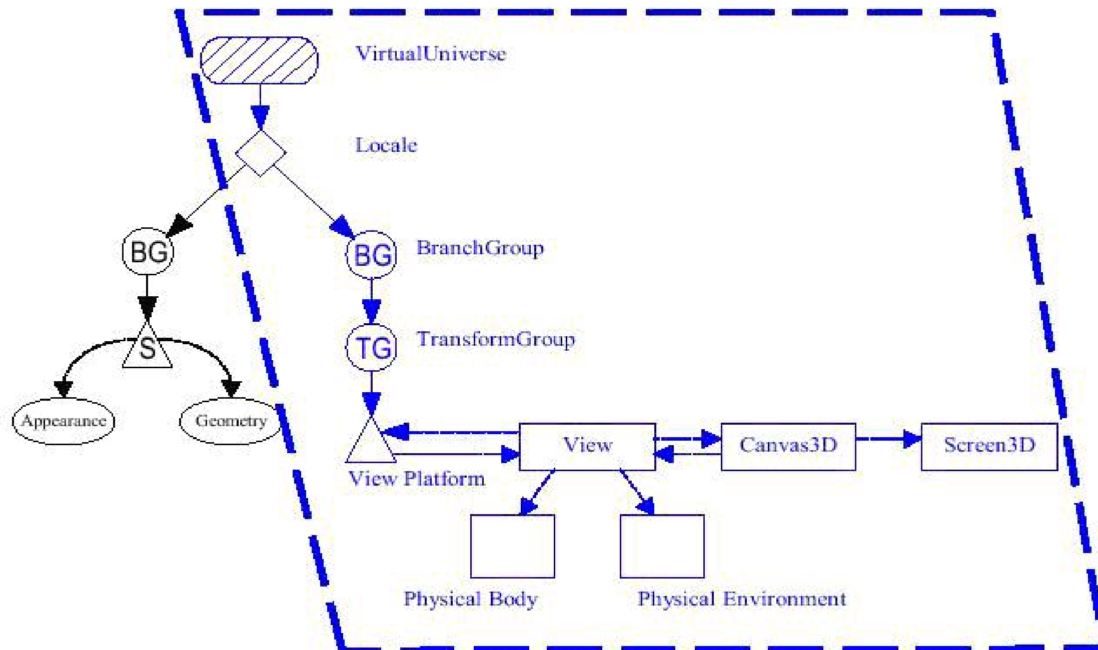
El método utilizado es *addBranchGraph*.

Compilar una rama de contenido gráfico

Una rama se compila cuando se compila el objeto *BranchGroup* del que cuelga la rama. Cuando una rama se compila, adopta una representación interna que hará más eficiente el renderizado. Compilar una rama suele ser el último paso antes de hacer viva una rama y es conveniente compilar únicamente aquellas ramas que van a estar vivas y que en consecuencia se van a renderizar.
Los métodos utilizados son:

- void compile()
- boolean isCompiled()
- boolean isLive()

A continuación podemos observar un ejemplo de la construcción de un árbol de escena:



5.3.2.5. Cargadores

Los cargadores toman como entrada un modelo tridimensional (fichero), producido por otro software (3D-STUDIO, AUTOCAD, SOLID WORKS, etc.), estos cargadores analizan el fichero y lo traducen a estructuras propias de JAVA3D que pueden ser incorporadas a un Universo Virtual de JAVA3D.

Existen multitud de formatos de ficheros J3D (3DS, OBJ, OCX, WRL,…), en nuestro caso utilizaremos el formato WRL, estos archivos son creados mediante las toolbox de *Virtual Reality Model* que incluye MATLAB. Y para nuestro proyecto han sido proporcionados por el profesor Segundo Esteban San Román, que imparte la asignatura de *Informática Gráfica* en la Universidad Complutense en la carrera de *Informática Técnica*.

Todos los cargadores implementan el interfaz *Loador*, este interfaz se encuentra dentro del paquete *com.sun.j3d.loaders*, y cada cargador se hace para un formato determinado. Existen multitud de cargadores disponibles públicamente y que ya han desarrollado otras personas, grupos o proyectos

El programador es el responsable de hacer el cargador y para ello debe desarrollar una clase que implementa el interfaz *Loador*. JAVA3D no proporciona cargadores, simplemente proporciona el interfaz que define los métodos que tiene que implementar cualquier cargador.

Cómo se usa un cargador

Sin una clase que realmente lea el fichero, no es posible cargar su contenido. Con una clase *Loader* es sencillo. La siguiente lista presenta los pasos a seguir para utilizar un cargador:

1. Encontrar un cargador (si no hay ninguno disponible, los escribimos)
2. Importar la clase cargador para nuestro formato de fichero
3. Importar otras clases necesarias
4. Declarar una variable *Scene* (no usar el constructor)
5. Crear un objeto *loader*
6. Cargar el fichero en un bloque *try*, asignar el resultado a la variable *Scene*
7. Insertar el *Scene* dentro del escenario gráfico

Cargadores disponible públicamente

En JAVA3D existen varias clases cargadoras. La siguiente tabla lista los formatos de ficheros cuyos cargadores están disponibles públicamente. En el momento de escribir esto, al menos hay disponible una clase cargador por cada uno de estos formatos de fichero:

Formato de Fichero	Descripción
3DS	3D-Studio
COB	Caligari trueSpace
DEM	Digital Elevation Map
DXF	AutoCAD Drawing Interchange File
IOB	Imagine
LWS	Lightwave Scene Format
NFF	WorldToolKit NFF format
OBJ	Wavefront
PDB	Protein Data Bank
PLAY	PLAY
SLD	Solid Works (prt and asm files)
VRT	Superscape VRT
VTK	Visual Toolkit
WRL	Virtual Reality Modeling Language

Interfaces y clases base del paquete *loader*

Esta gran variedad de cargadores existe para hacer más sencilla la escritura de cargadores para los diseñadores JAVA3D. Las clases *Loader* son implementaciones del interfaz *Loader* que baja el nivel de dificultad para escribir un cargador. Como en el ejemplo, un programa que carga un fichero 3D realmente usa un cargador y un objeto escena. El cargador lee, analiza y crea la representación JAVA3D de los contenidos del fichero. El objeto escena almacena el escenario grafico creado por el cargador. Es posible cargar escenas desde más de un fichero (del mismo formato) usando el mismo objeto cargador y crear múltiples objetos escena. Los ficheros de diferentes formatos pueden combinarse en un programa JAVA3D usando las clases cargadoras apropiadas.

Resumen de Interfaces del paquete *com.sun.j3d.loaders*:

- *Loador*: El interfaz *Loader* se usa para especificar la localización y los elementos de un formato de fichero a cargar.
- *Scene*: El interfaz *Scene* es un conjunto de métodos usado para extraer información de escenario gráfico JAVA3D de una utilidad cargador de ficheros.
-

Resumen de Clases del paquete *com.sun.j3d.loaders*:

- *LoaderBase*: Esta clase implementa el interfaz *Loader* y añade constructores. Esta clase es extendida por los autores para especificar clases cargadoras.
- *SceneBase*: Esta clase implementa el interfaz *Scene* y añade métodos usados por los cargadores. Esta clase también es usada por los programas que usan clases cargadoras.

Resumen de Métodos del Interfaz *Loader*:

El interfaz *Loader* se usa para especificar la localización y los elementos de un formato de fichero a cargar. Este interfaz se utiliza para darle a los cargadores de varios formatos de ficheros un interfaz público común. Idealmente el interfaz *Scene* será implementado para darle al usuario un interfaz consistente para extraer los datos.

- *Scene load (java.io.Reader reader)*: Este método carga el *Reader* y devuelve el objeto *Scene* que contiene la escena.
- *Scene load (java.lang.String fileName)*: Este método carga el fichero nombrado y devuelve el objeto *Scene* que contiene la escena.
- *Scene load (java.net.URL url):* Este método carga el fichero nombrado y devuelve el objeto *Scene* que contiene la escena.

- *void setBasePath (java.lang.String pathName)*: Este método selecciona el nombre del *path* base para los ficheros de datos asociados con el fichero pasado en el método *load(String)*.
- *void setBaseUr l(java.net.URL url):* Este método selecciona el nombre de la URL base para los ficheros de datos asociados con el fichero pasado en el método *load(String).*
- *void setFlags(int flags)*

Resumen de Constructores de la Clase LoaderBase

La clase *LoaderBase* proporciona una implementación para cada uno de los tres métodos *load()* del interfacz *Loader*. *LoaderBase* también implementa dos constructores. Observa que los tres métodos cargadores devuelven un objeto *Scene*.

Esta clase implementa el interfaz *Loader*. El autor de un cargador de ficheros debería extender esta clase. El usuario de un cargador de ficheros debería usar estos métodos.

- *LoaderBase():* Construye un *Loader* con los valores por defecto para todas las variables.
- *LoaderBase(int flags):* Construye un *loader* con las banderas especificadas.

Lista Parcial (métodos usados por usuarios) de la Clase *SceneBase:*

- *Background[] getBackgroundNodes()*
- *Behavior[] getBehaviorNodes()*
- *java.lang.String getDescription()*
- *Fog[] getFogNodes()*
- *float[] getHorizontalFOVs()*
- *Light[] getLightNodes()*
- *java.util.Hashtable getNamedObjects()*
- *BranchGroup getSceneGroup()*
- *Sound[] getSoundNodes()*
- *TransformGroup[] getViewGroups()*

Como se mencionó arriba, la característica más importante de los cargadores es que podemos escribir el nuestro propio, lo que significa que todos los usuarios de JAVA3D también pueden hacerlo.

Para escribir un cargador, debemos extender la clase *LoaderBase* definida en el paquete *com.sun.j3d.loaders*. El nuevo cargador usará la clase *Scene* del mismo paquete.

Escribir un cargador de ficheros puede ser bastante complejo dependiendo de la complejidad del formato del fichero. La parte más dura es analizar el fichero. Por supuesto, tenemos que empezar con la documentación del formato de fichero para el que queremos escribir la clase *Loader*. Una vez que se entiende el formato, empezamos leyendo la clase *LoaderBase* y *sceneBase*. La nueva clase *Loader* extenderá la clase *LoaderBase* y usará la clase *sceneBase*.

En la extensión de la clase *LoaderBase*, la mayoría del trabajo será escribir métodos que reconozcan los distintos tipos de contenidos que se pueden representar en el formato del fichero. Cada uno de esos métodos crea el correspondiente componente JAVA3D del escenario gráfico y lo almacena en un objeto *scene*.

Resumen de Constructores de la Clase *SceneBase*

Esta clase implementa el interfaz *Scene* y lo amplía para incorporar utilidades que podrían usar los cargadores. Esta clase es responsable del almacenamiento y recuperación de los datos de la escena.

Cargador utilizado

Finalmente, una vez estudiadas todas las posibilidades hemos decidido utilizar un cargador de archivos WRL, de los disponibles públicamente hemos decidido utilizar el cargador desarrollado por el grupo **Web3D Consortium** y descargado desde su la página web del proyecto **http://www.xj3d.org.**

5.3.2.6. Conclusiones

- JAVA3D presenta una forma muy elegante de representar escenas.
- JAVA3D está integrada en el lenguaje JAVA por eso funciona muy bien con *swing* y *awt*, y por eso resulta menos costoso de aprender.
- Con menos código se consiguen hacer más cosas.
- No hay ningún detalle, a priori, de bajo nivel.
- El renderizador se ejecuta en paralelo a la aplicación.
- Es portable.
- Tiene como defecto, el tiempo de ejecución, puesto que es alto.

5.3.2.7.Representación en JAVA 3D del modeloRCAM

Después de esta breve introducción a JAVA3D, describimos brevemente la clase donde hemos implementado la simulación en tres dimensiones.

Como ya hemos mencionado antes, tenemos dos clases distintas para la simulación, *Simulación* en la que simulamos el movimiento del avión y *SimulacionBatalla* en la cual se simula la batalla entre dos equipos de aviones. Ambas están guardadas en el paquete ***Java3D.*** Tenemos ahí guardado también la clase *datosBatalla* en la cual se almacenan los datos que serán necesarios para la simulación.

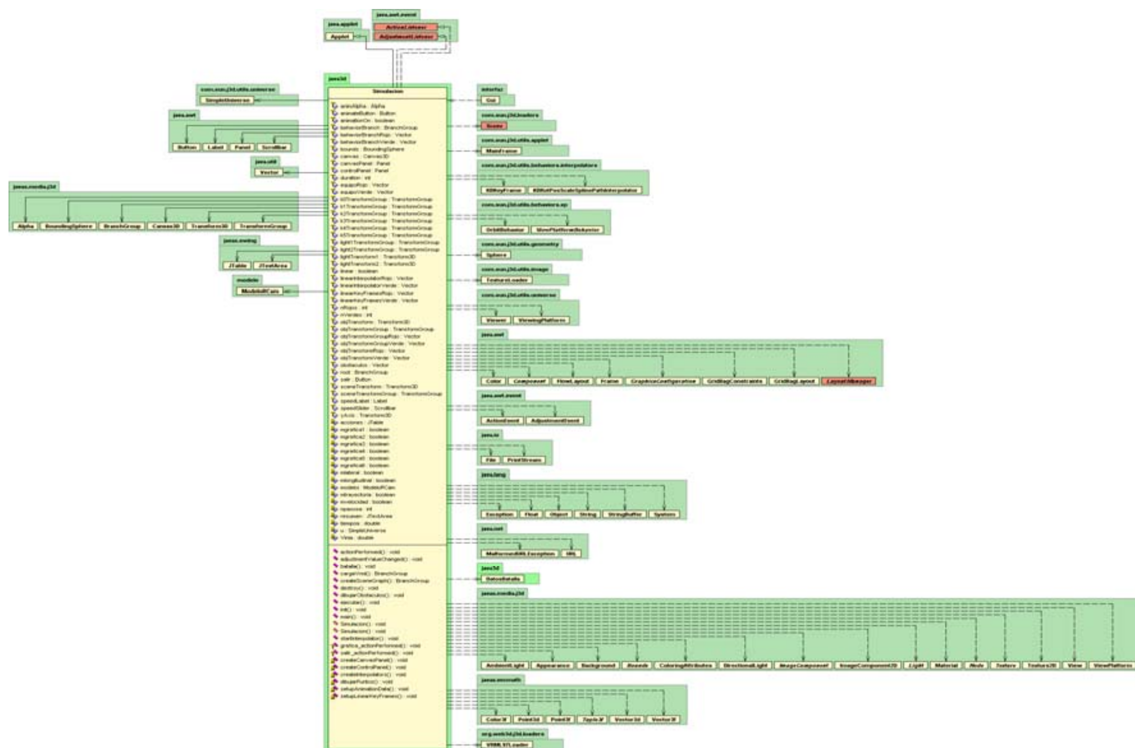
De las dos clases de simulación destacamos:

- El metodo *batalla*, que es donde se asignan los valores guardados del modeloRCAM a cada avion a representar.

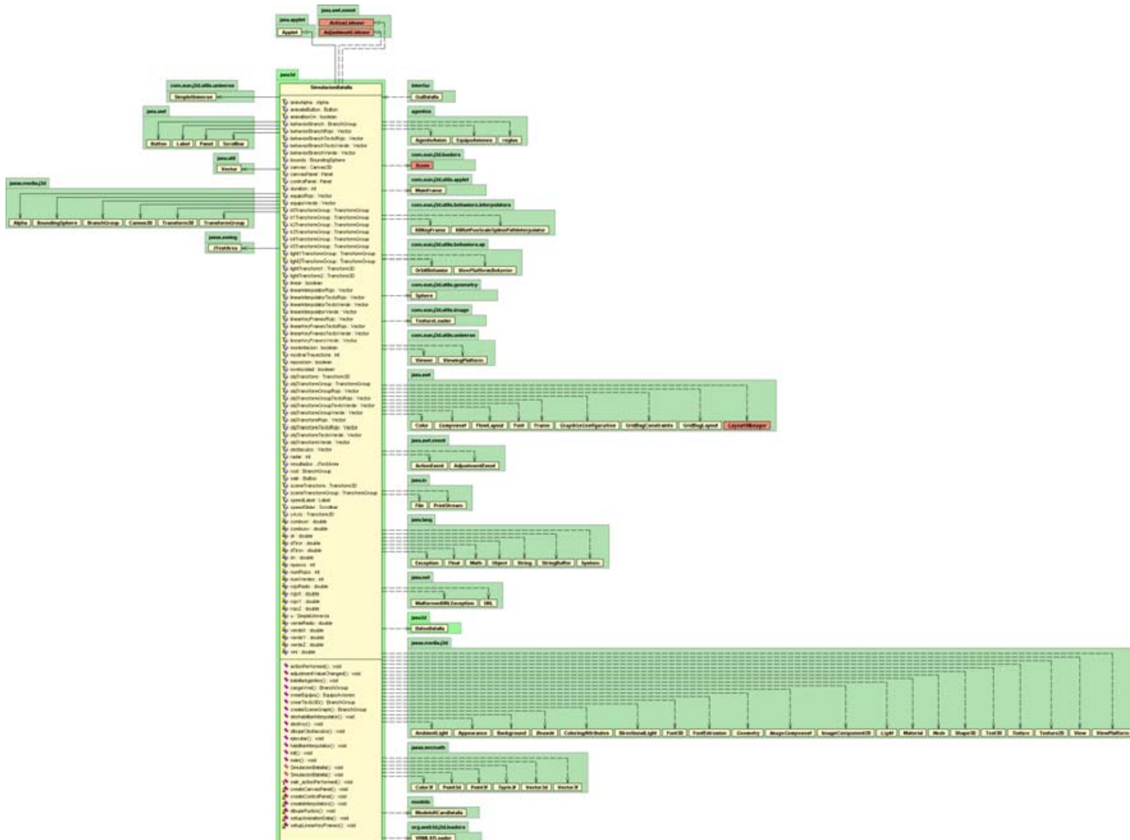
- El metodo *crearEquipo*, de *SimulacionBatalla*, que es donde se crean los equipos de aviones según los parámetros dados.

Mostramos, a continuación los diagramas de clases correspondientes:

1. Diagrama de la clase *Simulación*



2. Diagrama de la clase *SimulacionBatalla*



5.4. Cuarta Fase:

Con la simulación en tres dimensiones, ya hecha, pensamos que sería buena idea, para comprobar que el modelo funcionaba perfectamente, añadir unas gráficas donde demostrar cómo varían una serie de parámetros. Y si dichos valores coincidían con los valores que se obtienen del modeloRCAM de MATLAB.

Las graficas que hemos decidido mostrar son:

- Ángulos de Euler: Los ángulos de Euler son:
    - Ángulo de alabeo, PHI: nos dice el ángulo de giro de las alas del avión. Varía al hacer un cambio de rumbo.
    - Ángulo de guiñada, PSI: nos dice el ángulo que tomará la cabeza del avión al hacer un cambio de rumbo.
    - Ángulo de cabeceo, THETA: nos dice el ángulo que tomará la cabeza del avión al hacer un cambio de altura.

- Ángulos Alpha-Gamma: Los definimos:
    - Alpha: atan2 (WB/ $\sqrt{}$(UB^2+VB^2)).
    - Gamma: atan2 (WV/ $\sqrt{}$(UV^2+V^2)).

- Señales de control Lateral:
    - DA: deflexión del alerón.
    - DR: deflexión del timón.

- Señales de control Longitudinal:
    - DT: deflexión del alerón trasero
    - TH: posición de aceleración de los motores.

- Posiciones del avión:
    - X: Coordenada X.
    - Y: Coordenada Y.
    - Z: Coordenada Z.

- Variación de la velocidad: Nos indica cómo varia la velocidad al introducirle, los cambios deseados.

Para implementar estas graficas reutilizamos el paquete ***fundamentos***, que contiene varias clases, de las cuales sólo nos interesa *Graficas*.

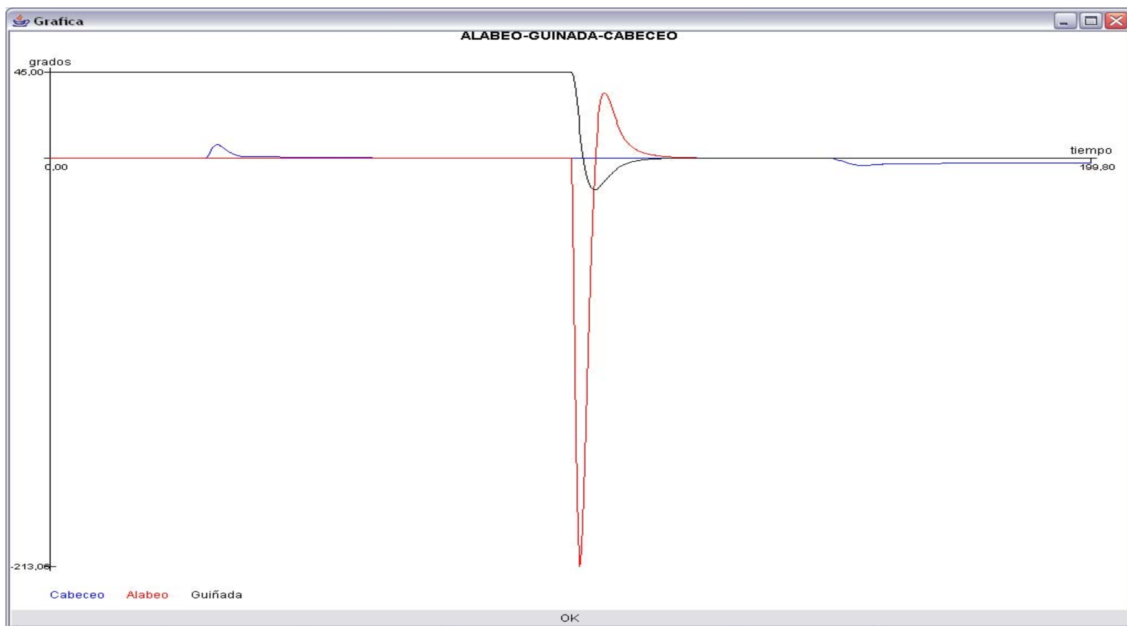
A continuación mostramos unos ejemplos de estas gráficas, y al lado la gráfica que se obtiene en MATLAB de dichos valores. Comprobamos que ambas coinciden, lo que demuestra que la traducción del modelo que hemos hecho es correcta.

Las órdenes que hemos utilizado para realizar las gráficas fueron las siguientes:

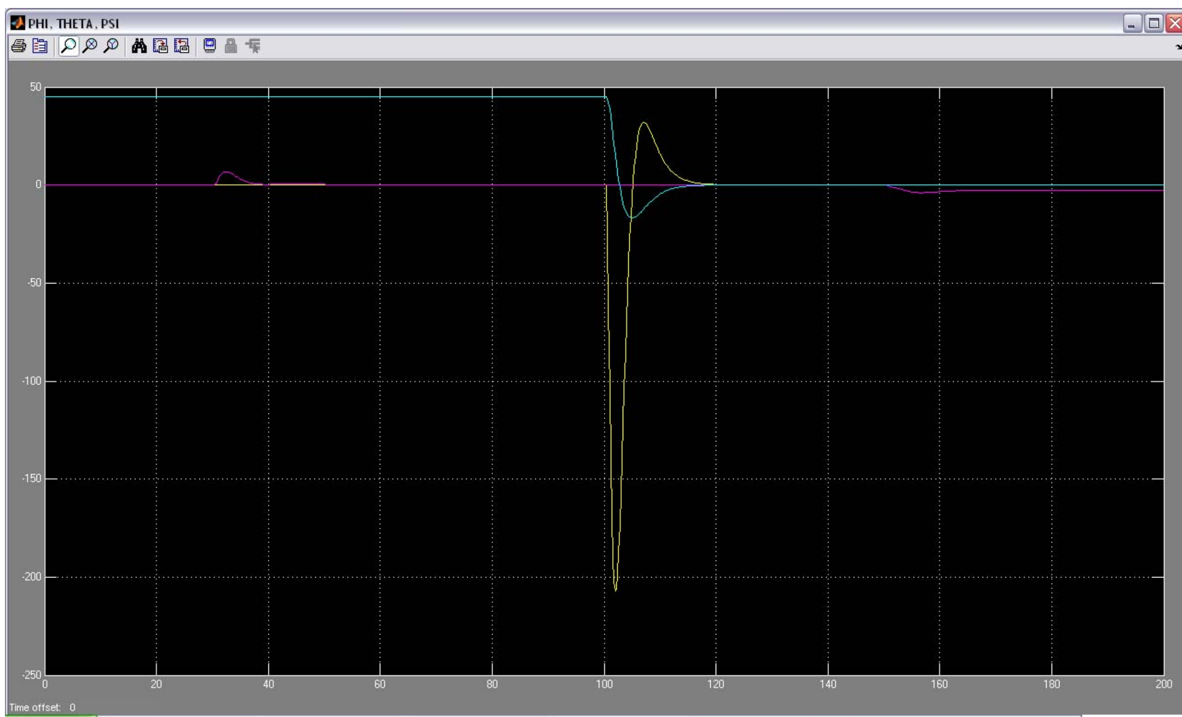
Orden	Tiempo	Valor
altura	30	50
rumbo	100	0
velocidad	150	20

1. Ángulos de Euler:

a. JAVA

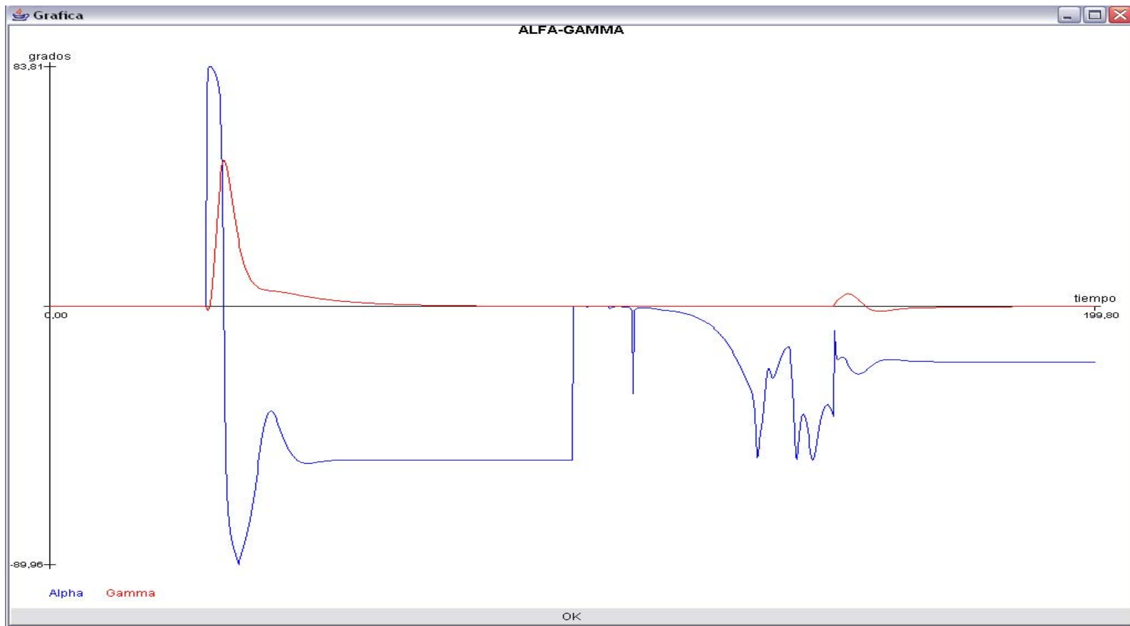


b. MATLAB

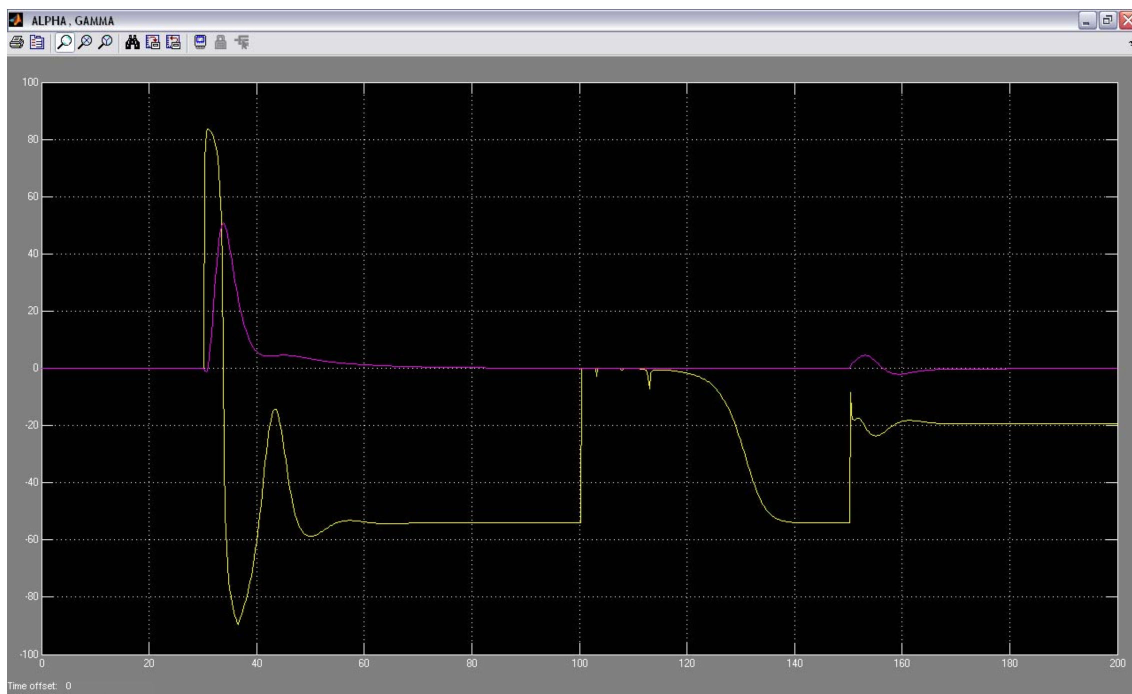


2. Ángulos Alpha-Gamma:

a. JAVA

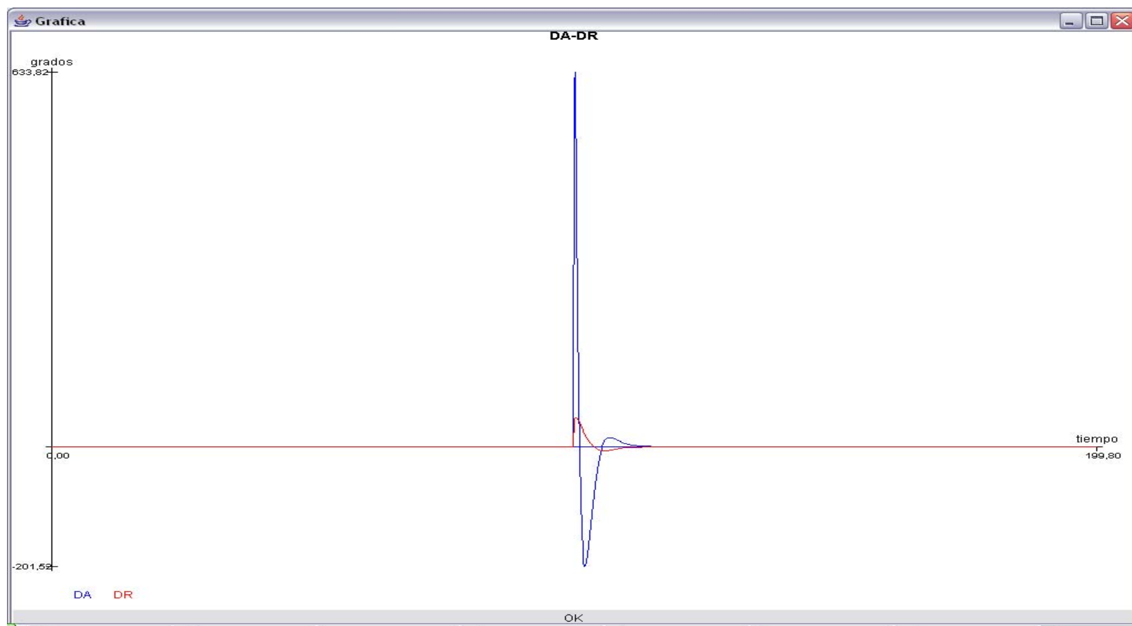


b. MATLAB

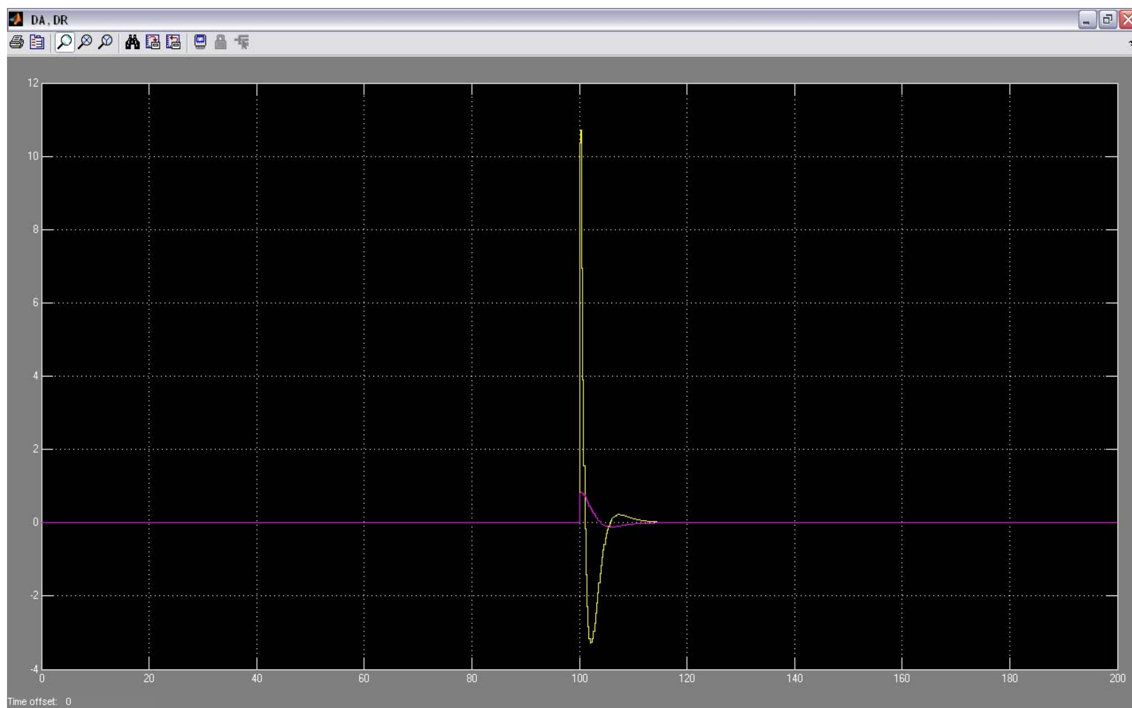


3. Señales de control Lateral:

a. JAVA

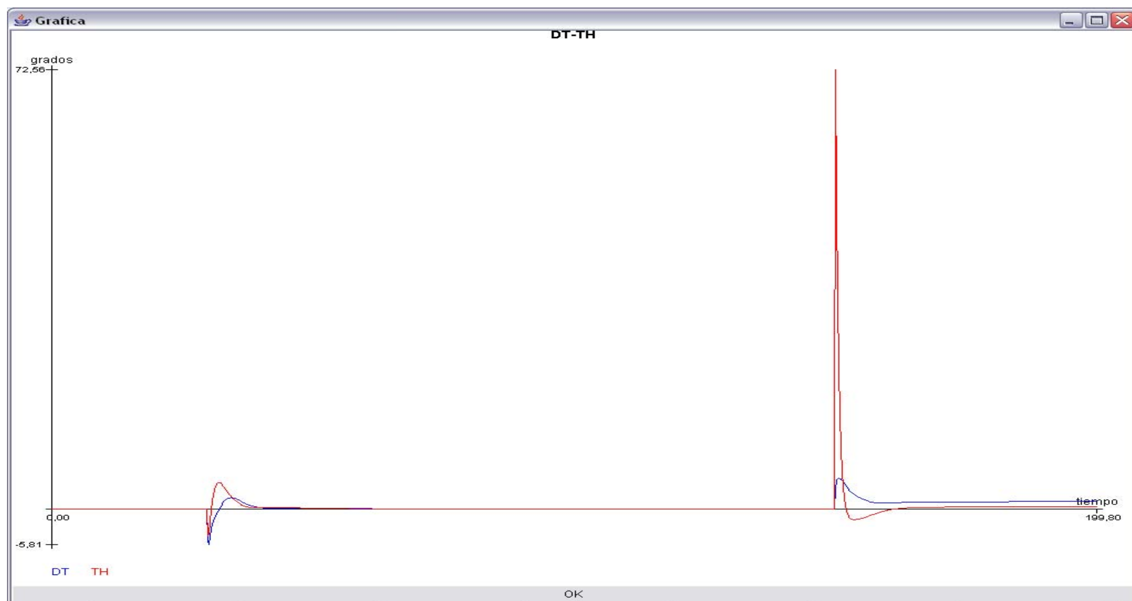


b. MATLAB

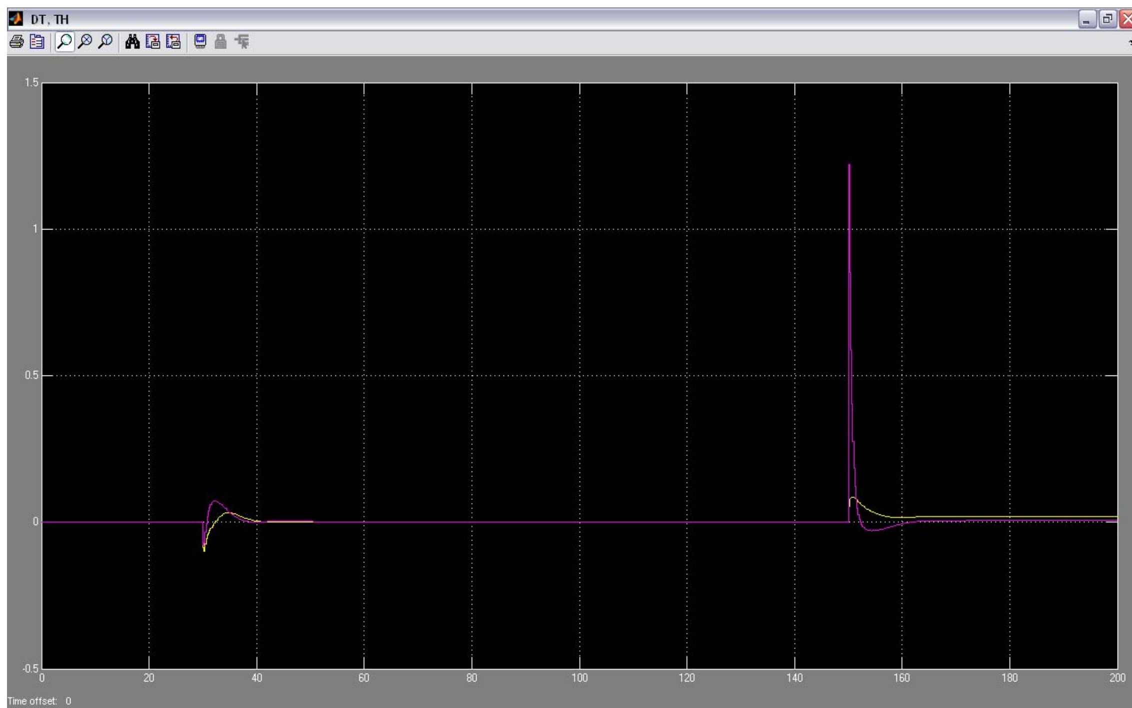


4. Señales de control Longitudinal:

a. JAVA

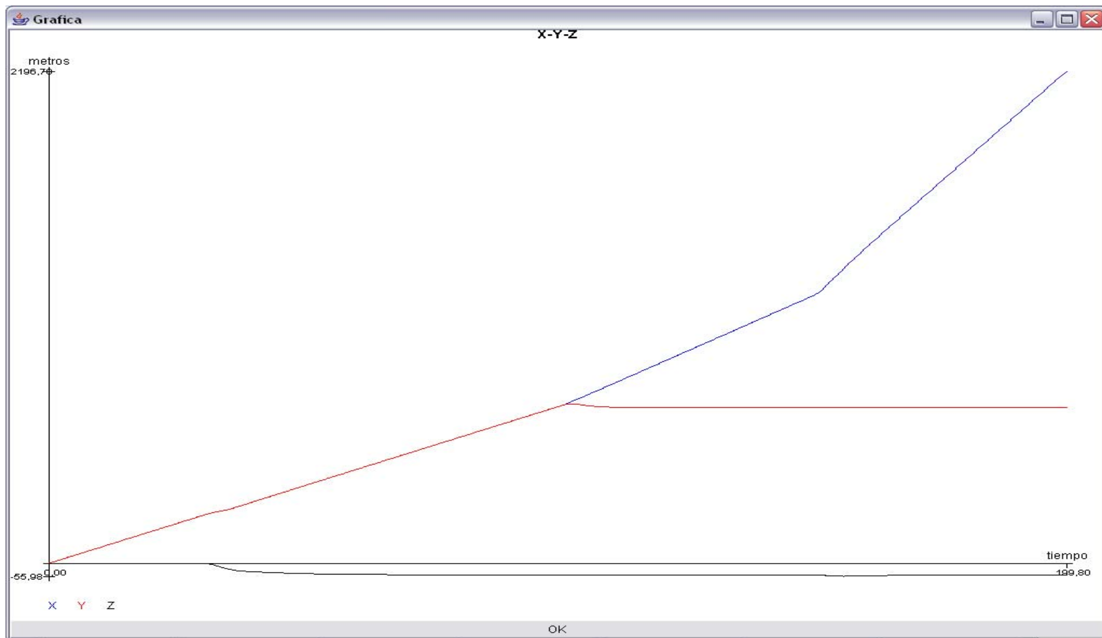


b. MATLAB

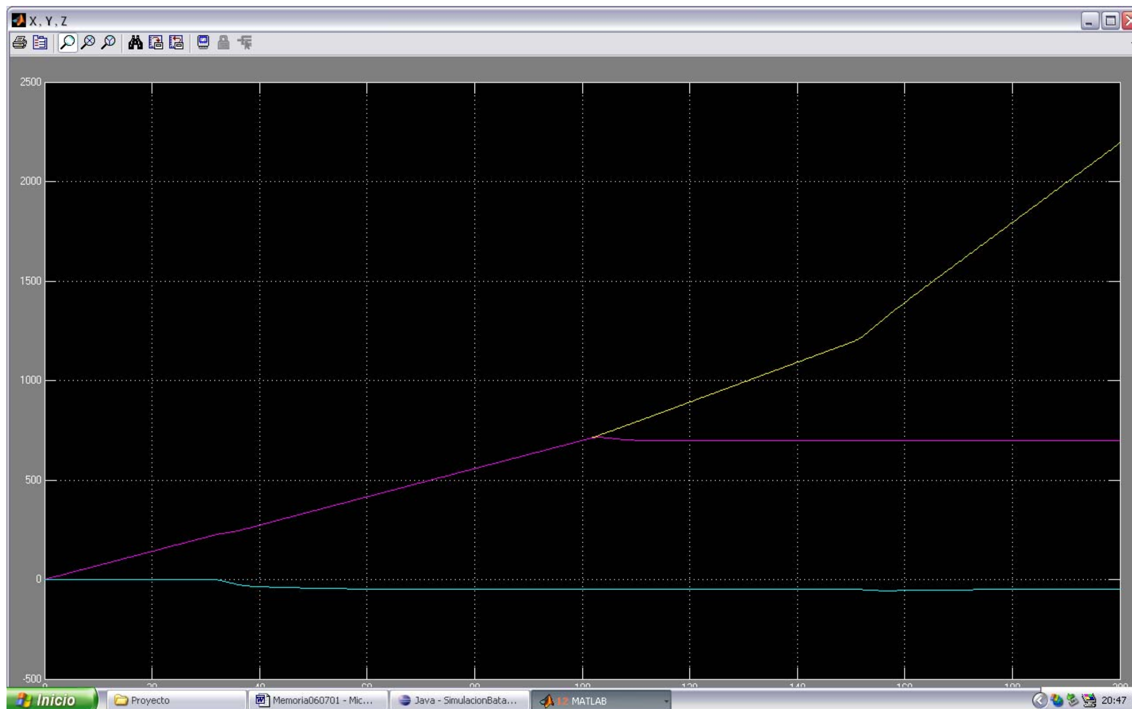


5. Posiciones del avión:

a. JAVA

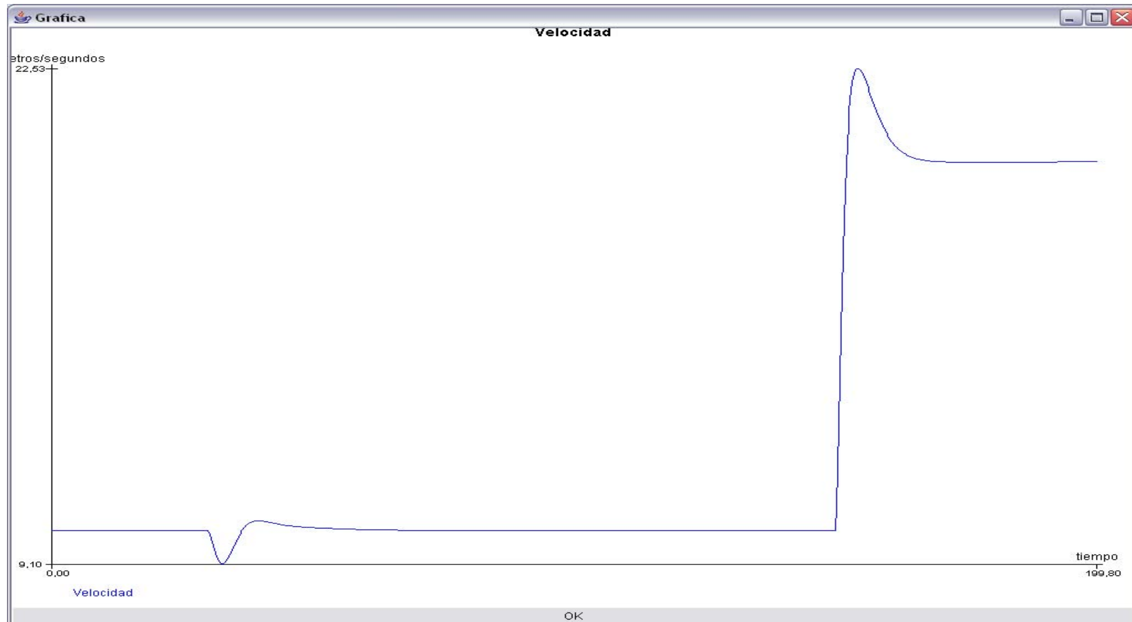


b. MATLAB

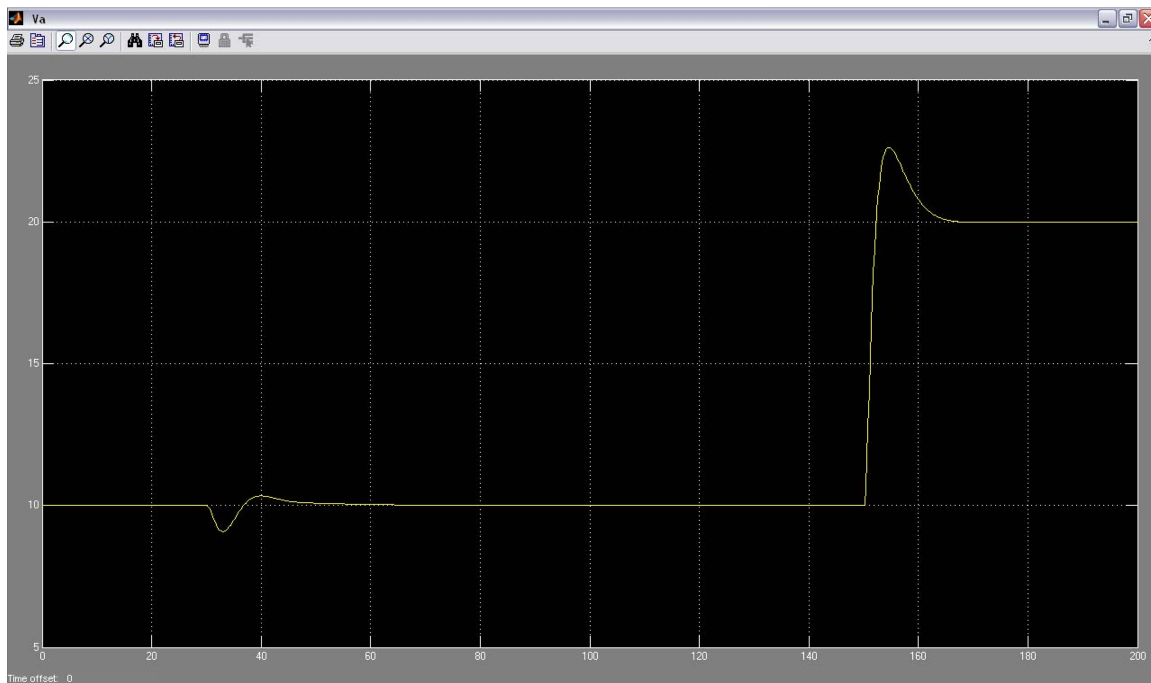


6. Variación de la velocidad

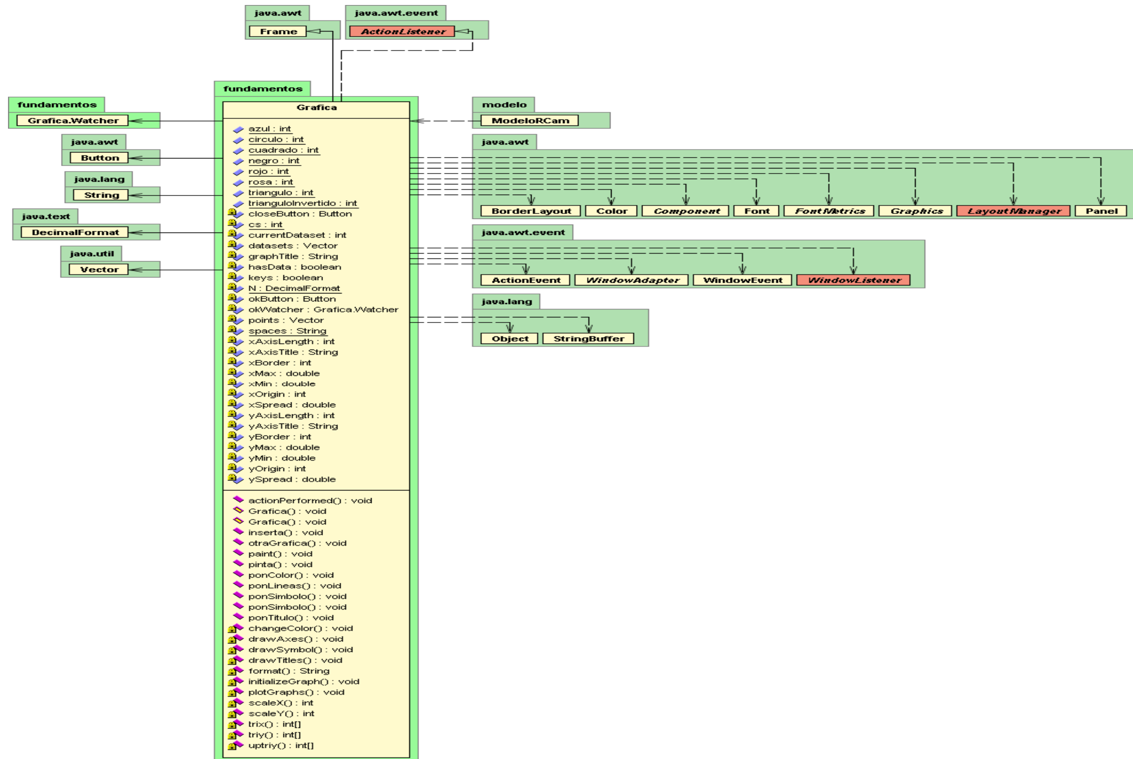
a. JAVA



b. MATLAB



Mostramos el diagrama de clases de ésta:



5.5. Quinta Fase:

Para hacer, el programa, más intuitivo y fácil de manejar, decidimos hacer un interfaz gráfico, desde la cual el usuario puede acceder la opción de visualización que prefiera. Como ya hemos dicho varias veces, tenemos dos opciones:

1. Modo avión: En esta opción el consumidor puede decidir que ordenes dar al avión y ver su posterior ejecución. También puede elegir que graficas desea visualizar y los datos que quiere mostrar.

2. Modo batalla: Si se elige esta posibilidad, el usuario, ve la simulación de dos equipos batallando. Teniendo la opción de decidir las características de cada equipo.

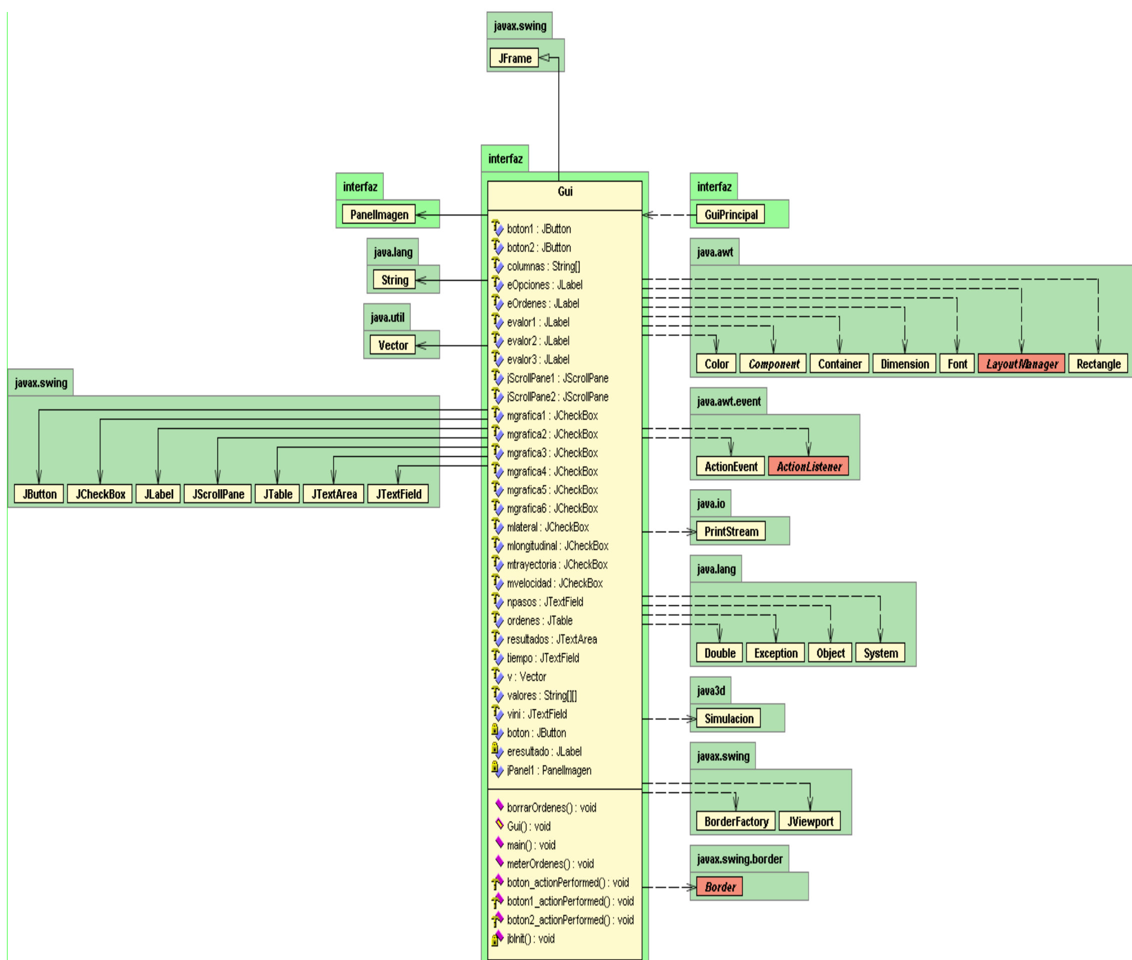
Decidimos agrupar todo en un paquete llamado interfaz, donde encontramos las siguientes clases:

- GuiPrincipal: Clase que diseña el formulario principal, que aparece nada más comenzar la simulación. Y que permitirá acceder a las dos.

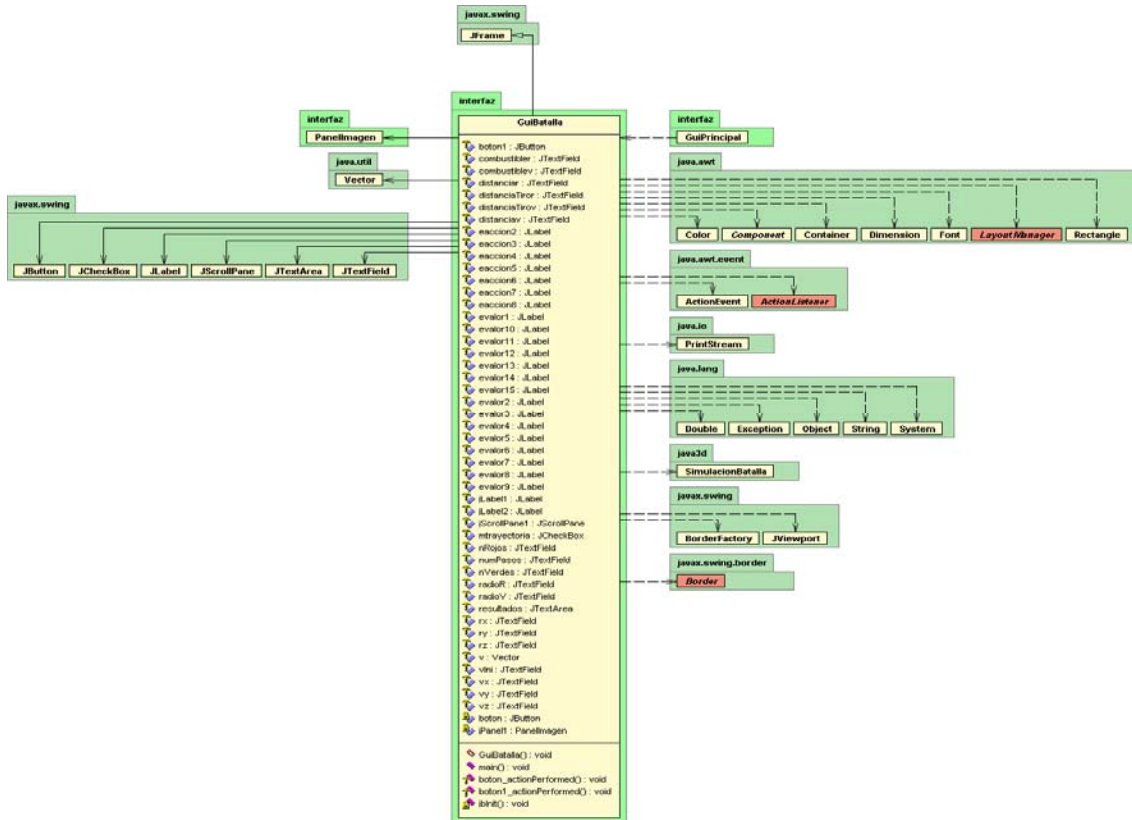
- Gui: Esta clase corresponde al interfaz del modo avión. Esta formada por un panel que mostrará los datos de la simulación que el usuario elija y por una tabla donde se irán insertando las ordenes que se pretenda que el avión realice.
- GuiBatalla: Diseña la GUI del interfaz de batalla. El consumidor decide cómo quiere que estén configurados los equipos. Contiene además un panel donde se muestra que hace cada avión de cada equipo.
- PanelImagen: Clase auxiliar que se utiliza para dar añadir un fondo a cada formulario.

A continuación mostramos los diagramas de clases:

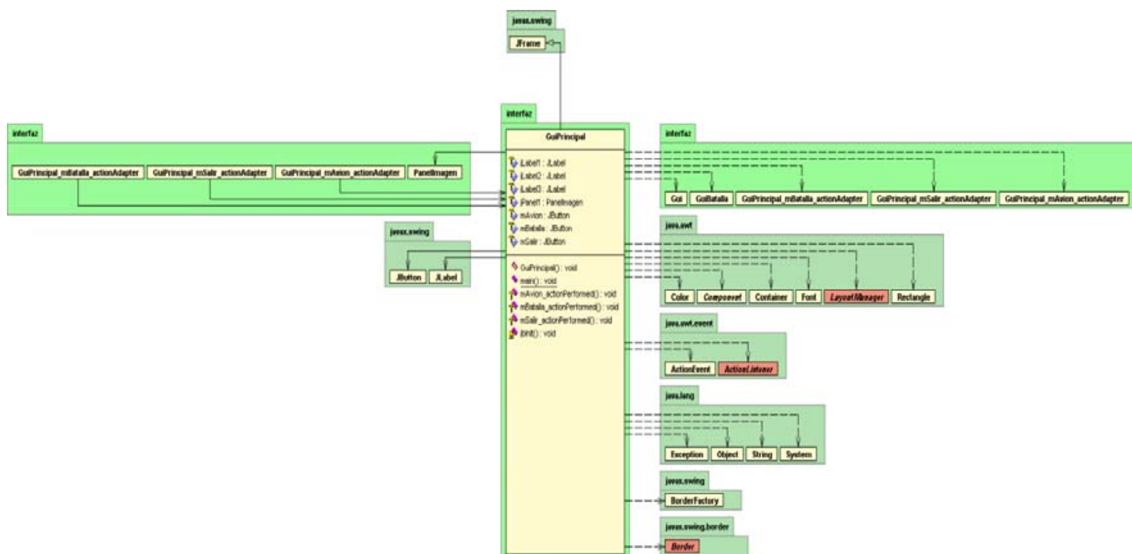
1. Diagrama de *Gui*



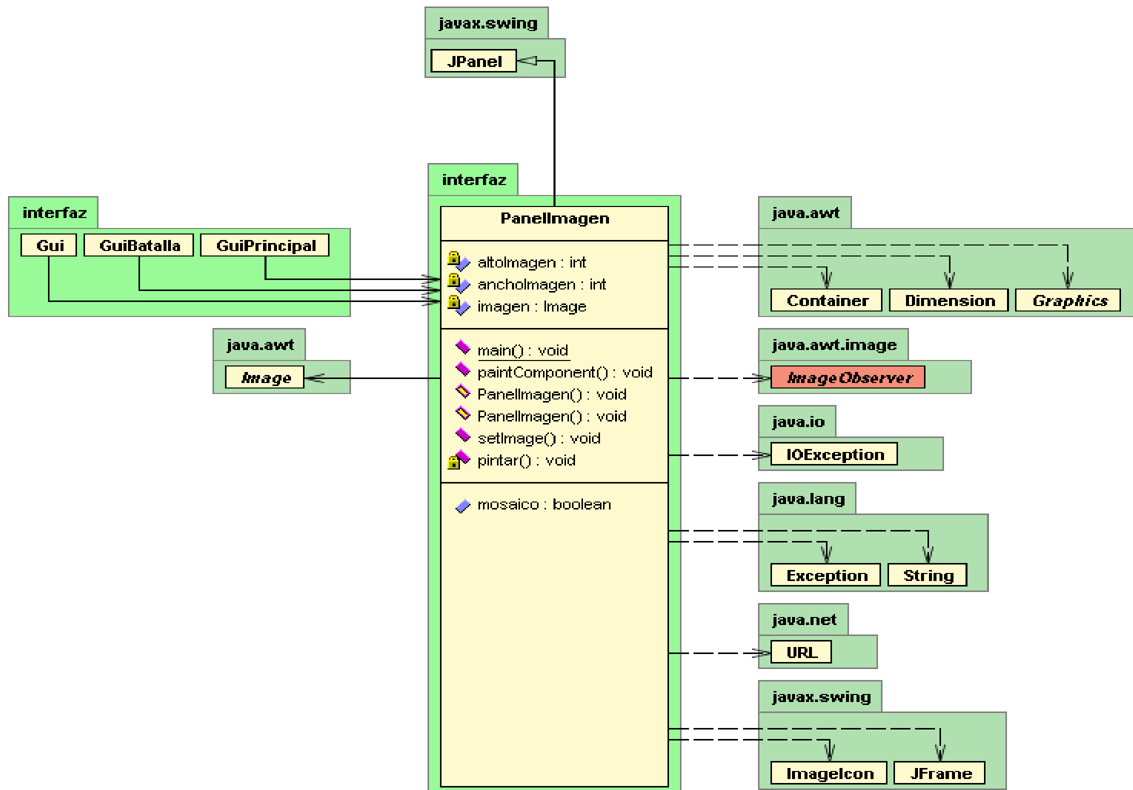
2. Diagrama de *GuiBatalla*



3. Diagrama de *GuiPrincipal*:



4. Diagrama de *PanelImagen*



5.6. Sexta Fase:

Esta fase consiste en dotar de inteligencia a los agentes. En principio, teníamos pensado trabajar siguiendo el esquema dado por el [Bigus01], tanto para representar a los agentes como para dotarlos de inteligencia.

Al final, decimos cambiar de idea y optamos por seguir un esquema mucho más sencillo e intuitivo para nosotros.

Como lenguaje de programación, continuamos con JAVA, lenguaje que ya usamos para obtener las ecuaciones del movimiento de nuestros aviones.

Tenemos que reseñar, que la implementación definitiva escogida para representar a los agentes, tiene como objetivo seguir un guión de juego de combate entre dos equipos de aviones. A continuación, conviene que se expliquen las normas principales que hemos escogido para el desarrollo del juego.

5.6.1. Dinámica del juego

En el juego tendremos dos equipos de aviones, el usuario podrá elegir el número de aviones que formaran cada equipo y las posiciones iniciales de los mismos.

Los aviones se irán moviendo por el tablero de juego, siguiendo la dinámica del modelo RCAM, antes explicado; por lo que sus movimientos seguirán unos pasos determinados.

Destacamos que los aviones son sabedores de la situación que tienen el resto de componentes de su propio equipo (posición, combustible, estado) pero que sólo conocen las posiciones de los aviones enemigos que están dentro del radio de alcance de su radar (dato que también introduce el jugador).

Una vez colocados en sus respectivas posiciones, los aviones tendrán que decidir ellos mismos, que hacer dentro del combate. En esta parte, es donde entra en juego la inteligencia de los agentes.

Los aviones, conscientes de su situación y la de los miembros de su equipo, actuaran de un modo u otro, siguiendo unas reglas, que detallaremos más adelante.

Las acciones, que pueden llevar a cabo son:

- Mover: Los aviones, se mueven como ya hemos citado, siguiendo el modelo RCAM, y esquivando obstáculos que tendrá repartidos en su camino.

- Repostar: Los aviones, tendrán un depósito con cierta cantidad de combustible. El nivel de gasolina irá bajando según se van moviendo y si sobrepasa cierto límite el avión se estrellaría por falta de combustible. Por lo que cada cierto tiempo, el avión debe parar a llenar el deposito.

- Huir: Los aviones, sabedores de las condiciones en que se encuentran (inferioridad numérica, falta de combustible), en determinadas ocasiones puede valorar el huir del enemigo y buscar una situación más apropiada para entra en combate.

- Perseguir Enemigo: Los aviones, una vez que detectan en su radar algún enemigo se dirigen hacia él, si las condiciones lo permiten, para buscar una situación apropiada para entrar en combate.

- Atacar: Los aviones, pueden atacar en cualquier momento, aunque saben que hay situaciones donde es más propicio el éxito. Deberán aprender de sus errores y deducir cuanto es el mejor momento para ejecutar un ataque certero.

Explicamos a continuación cómo hemos definido nosotros un combate. En la batalla luchan dos aviones de bandos contrarios. Para que el ataque tenga alguna posibilidad de éxito dichos aviones deben estar separados por una distancia mínima de tiro. Calcularemos para cada avión una probabilidad de éxito (los dos aviones luchan, aunque no sea el que ataque, el otro se defiende). Dicha probabilidad la definimos así:

Probabilidad éxito avion1 = velocidad(avion1)/(velocidad(avion1)+velocidad(avion2))
Probabilidad éxito avion2 = velocidad(avion2)/(velocidad(avion1)+velocidad(avion2)).

Tras calcular estos valores, se lanza un dado y se decide cual de los dos aviones gana. El avión derrotado muere.

El juego terminará cuando todos los aviones de un equipo hayan muerto.

5.6.2. Implementación de los agentes

Como ya dijimos antes, hemos escogido JAVA como lenguaje de implementación para nuestro proyecto. Además modificamos la idea original que teníamos en cuanto a la representación de los agentes y las reglas, por un planteamiento mucho más claro y sencillo para nosotros.

En el programa que se adjunta, se puede ver que toda la parte de representación de agentes, inteligencia y entorno está englobada en un paquete que se llama ***agentes.***

Dicho paquete está compuesto por cuatro clases distintas que pasamos a detallar a continuación*:*

- *AgenteAvion:* En esta clase representamos a los agentes. Con sus atributos y sus métodos.

- *EquipoAviones*: Se definen los aviones que formaran parte de los equipos que entraran en combate.

- *Reglas:* En esta clase representamos la inteligencia de nuestros aviones.

5.6.2.1.Agentes

Para representa los agentes hemos escogido la clase *AgenteAvion*, en la cual a través de unos atributos y varios métodos implementamos un avión.

Como atributos tenemos:

- *Avión*: Este atributo es muy importante, pues es el que lleva implícito todo el modelo RCAM, gracias a él conocemos la velocidad y el rumbo que tiene el avión, así como sus posiciones. Es del tipo *ModeloRCAM*, clase implementado en el paquete ***modelo***, explicado con anterioridad.

- *Combustible*: Atributo que nos indica el nivel de gasolina que tiene el avión, en un momento dado. Es de tipo double.

- *Vivo*: Atributo booleano que nos indicará si el avión esta vivo o no.

- *Identificador*: Atributo que nos dice define el avión. Gracias a él, sabremos a cual de los aviones nos referimos.

- *Aliados*: Vector en el cual guardamos la lista de aviones que forman parte de nuestro equipo. Gracias a este atributo, conocemos la situación de los aliados.

- *Enemigos*: Vector que contiene los aviones enemigos, que visualiza un avión en su radar. De ellos sólo conocemos su posición.

- *CoordenadaX*: Coordenada X del avión.

- *CoordenadaY*: Coordenada Y del avión.

- *CoordenadaZ*: Coordenada Z del avión.

- *Atacando*: Atributo booleano que nos indica cuando el avión está atacando; es de utilidad para saber el estado actual del avión.

- *Repostando*: Atributo booleano que nos indica cuando el avión está repostando; es de utilidad para saber el estado actual del avión.

- *Moviendo*: Atributo booleano que nos indica cuando el avión está moviéndose; es de utilidad para saber el estado actual del avión.

- *Huyendo*: Atributo booleano que nos indica cuando el avión está huyendo de algún enemigo; es de utilidad para saber el estado actual del avión.

- *Persiguiendo*: Atributo booleano que nos indica cuando el avión está persiguiendo a algún enemigo; es de utilidad para saber el estado actual del avión.

- *Regla*: Como ya explicaremos más adelante, los agentes en cada movimiento aplican una regla. Este atributo nos indica la regla que está aplicando el avión en ese instante.

En cuanto a los métodos, tenemos varios, aquí sólo nos limitaremos a explicar los más interesantes:

- *AvionesEnemigos*: Este método nos devuelve un vector, con la lista de aviones enemigos, que visualizamos con el radar. En esta lista sólo guardamos los aviones que siguen vivos, ya que los muertos, como es de suponer, no nos interesan. Para implementarlo usamos los métodos auxiliares: *hayAvionEnemigo* y *calculaDistanciaAviones*.

- *enemigoMasCercano*: Este método devuelve el avión enemigo más cercano del avión actual. Gracias a este método, sabemos cual es la posición del enemigo permitiendo así que nuestro avión se aproxime a él.

- *eligeRumboObstaculo*: El avión, detecta gracias a los métodos, *hayObstaculo* y *calculaDistanciaObstaculo* , si tiene un obstáculo en su recorrido. Gracias a este método modificará su rumbo, con el fin de esquivar el obstáculo.

- *mismoEnemigo*: Este método, nos dice si dos aviones tienes el mismo enemigo. Devuelve, en el caso de que haya varios aviones con el mismo enemigo, el aliado que tenga el enemigo más cercano.

Distinguimos, a continuación seis métodos, que se diferencian de los demás, por tratarse de los hechos que culminaran la acción de las reglas que describen la actuación de los agentes. Estas reglas las detallaremos más adelante.

Estos métodos, devuelven la puntuación final, que se otorga a la regla a la que pertenecen. Esta puntuación es muy importante, pues será decisiva para que el agente decida que regla aplicar:

- *Mover*: El avión se mueve por el tablero, como ya hemos dicho anteriormente.

- *Atacar*: El avión ataca a un avión del bando enemigo. Puede atacar siempre que quiera, pero dependiendo del resultado del ataque obtendrá mejor o peor puntuación.

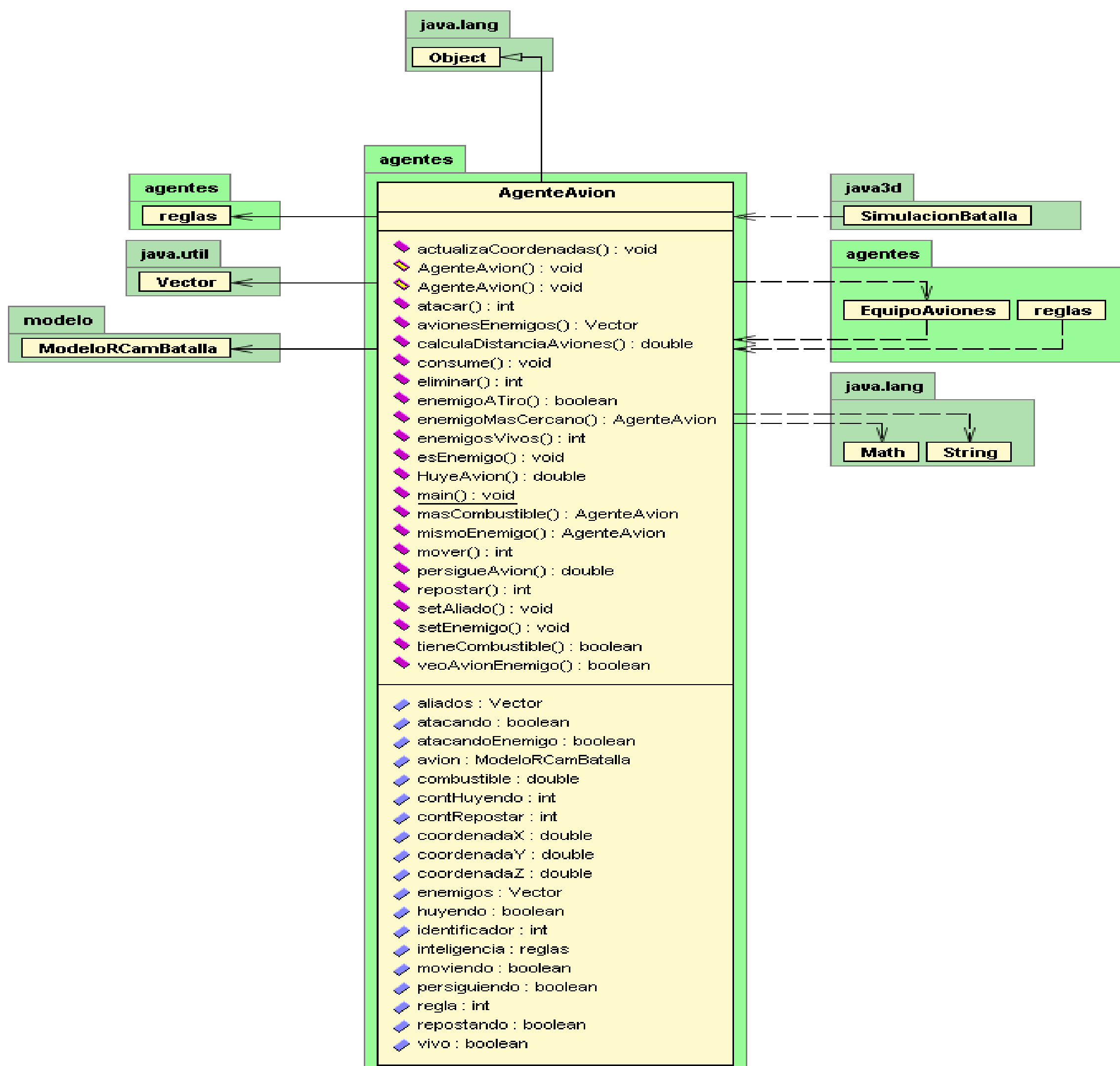
- *Repostar*: El avión repostará cuando su nivel de combustible sea inferior a un límite dado. Puede repostar cuando quiera, pero obtendrá mejor puntuación si lo hace en el momento oportuno.

- *Eliminar*: Se elimina a un avión cuando ha muerto y desaparece del juego.

- *persigueAvion*: El avión, conoce la posición del avión enemigo más cercano, gracias a este método, cambiará su rumbo con la idea de acercarse lo más posible al enemigo.

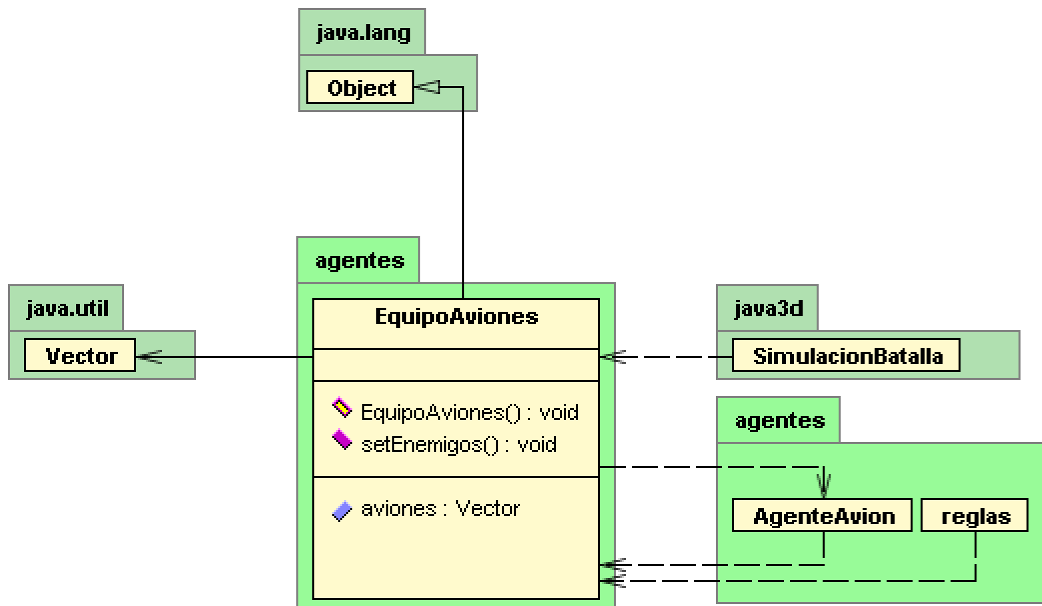
- *HuyeAvion*: Método semejante al anterior, en el cual el avión en lugar de acercarse al enemigo, se aleja de él.

A continuación mostramos el diagrama de clases.



5.6.2.2.Equipos

La clase, que describe cómo serán los equipos es *EquipoAviones*. Esta clase es muy sencillita y en ella sólo destacamos que cada equipo, está formado por un vector de aviones.



5.6.2.3.Reglas

La inteligencia de nuestros agentes, la implementamos en una clase llamada *reglas*. Esta clase es muy importante, pues en ella se decidirá que comportamiento tendrá el avión en cada paso del juego.

Nos llevó bastante tiempo, decidir cual seria la estrategia que emplearíamos para implementar el conocimiento. Valoramos varias opciones:

1. Utilizar lógica Fuzzy.
2. Utilizar el esquema de back-tracking del [Bigus01].
3. Utilizar algoritmos genéticos.
4. Utilizar una base de conocimiento donde se almacenen en tablas los distintos hechos.

Aunque al final, nos decantamos por implementar unas reglas, que desencadenan en unos hechos, y que devuelven a su vez una puntuación dependiendo si el agente ha obrado bien, mal o regular.

Mediante este sistema, nuestros aviones, aprenden a base de prueba y error. Si hacen un movimiento malo o desafortunado obtendrán una puntuación negativa, que hará que la siguiente vez, que se enfrenten a una situación similar opten por otra opción. De esta manera, después de varios intentos fallidos al final sólo actuarán correctamente, ejecutando en cada situación la regla que les lleve a obtener la máxima puntuación.

Al principio, suponemos, que los agentes no saben nada y actuaran de manera incierta, eligiendo de forma aleatoria la regla que ejecutaran. Pueden de este modo acertar, incluso a la primera, en su comportamiento.

Por simplicidad, suponemos que todos los aviones operan siguiendo el mismo esquema de reglas. Es decir todos los aviones comparten este tipo de inteligencia.

A continuación mostramos las reglas que hemos introducido, y que determinarán el comportamiento de los agentes. Se ven en mayúsculas los hechos que devolverán una puntuación a la regla que van asociados.

Destacamos, que para agilizar la batalla, decidimos añadir al final de cada hecho un *Mover*, para que la ejecución fuese más rápida.

Regla 1: Con esta regla, comenzamos la secuencia de reglas que deberán aplicar los aviones. Lo primero es comprobar si el avión está vivo, en cuyo caso podrá seguir avanzando, sino desaparecerá del juego. Si el avión está vivo decidirá que hacer aleatoriamente, aunque según vaya aprendiendo, según van pasando iteraciones, verá que la mejor opción es aplicar la regla 2.

Esquema de la regla1:

Si avión vivo:

a. Regla 2
b. ATACAR
c. REPOSTAR

Si avión muerto → ELIMINAR.

Regla 2: Con esta regla comprobamos si el avión tiene combustible, si no tiene debe parar a repostar. Y si tiene, entonces pasará a aplicar el resto de reglas. Lo más destacado de estas reglas es que el avión busca sus enemigos, si resulta que tiene enemigos puede elegir que hacer, sino deberá seguir avanzando con el fin de encontrar algún avión rival.

Mostramos el esquema de la regla2

Si combustible < 10 → REPOSTAR

Si no
Si hay avión enemigo:

a. Regla 3
b. ATACAR
c. Regla 4

Si no → MOVER

Regla 3: Esta regla se caracteriza por la batalla, el avión llega a esta regla sólo si tiene algún enemigo. Entonces deberá mirar si dicho enemigo está a tiro, en cuyo caso podrá atacar o aplicar otras reglas. Sino tiene al rival a tiro podrá hacer lo mismo que en la opción anterior, pero si decide atacar se le dará una puntuación mala.

Si enemigo a tiro:
a. ATACAR
b. Regla 5
c. Regla 4

Si no:

a. ATACAR
b. Regla 5
c. Regla 4
d. Regla 6

Regla 4: El avión aplica esta regla en el caso de que lo mejor, dado las circunstancias en que se encuentra, sea huir del enemigo.

HUIR; Regla 2.

Regla 5: Con está regla lo que buscamos es que el avión tenga en cuenta al resto de sus equipo a la hora de atacar. Lo que hace es mirar si hay algún aliado que tenga el mismo enemigo que él, y si se da esta circunstancia mirará quien de los dos tiene más combustible y ese será el avión que ataque.

Esquema de la regla5:

¡

Si *aliado mismo enemigo*:

Si tu más combustible → ATACAR

Si no → Regla 4

Si no regla 3

Regla 6: Esta regla se aplica correctamente en el caso de tener enemigos, pero éstos todavía no están a tiro, por lo que es más conveniente que nos acerquemos más al rival.

PERSEGUIR; Regla 2

Las reglas, están implementadas en la clase citada anteriormente. Como atributos tenemos:

- *Puntuaciones*: Matriz que guarda la puntuación que corresponde a cada regla. En la primera componente guardamos la regla que estamos tratando y en la segunda la condición que aplica. Por ejemplo en puntuaciones [0][0] guardamos la puntuación que se le da a la regla 1 cuando aplica la regla 2, porque el avión está vivo.

- *Siguiente*: Es un atributo, que nos dice, en el caso de que se tenga que aplicar otra regla y no un hecho, cual será la siguiente regla que ejecutan los agentes. Esto es interesante, pues nos permite seguir los pasos que va dando el avión y nos permite conocer la puntuación definitiva que tendrá la regla inicial.

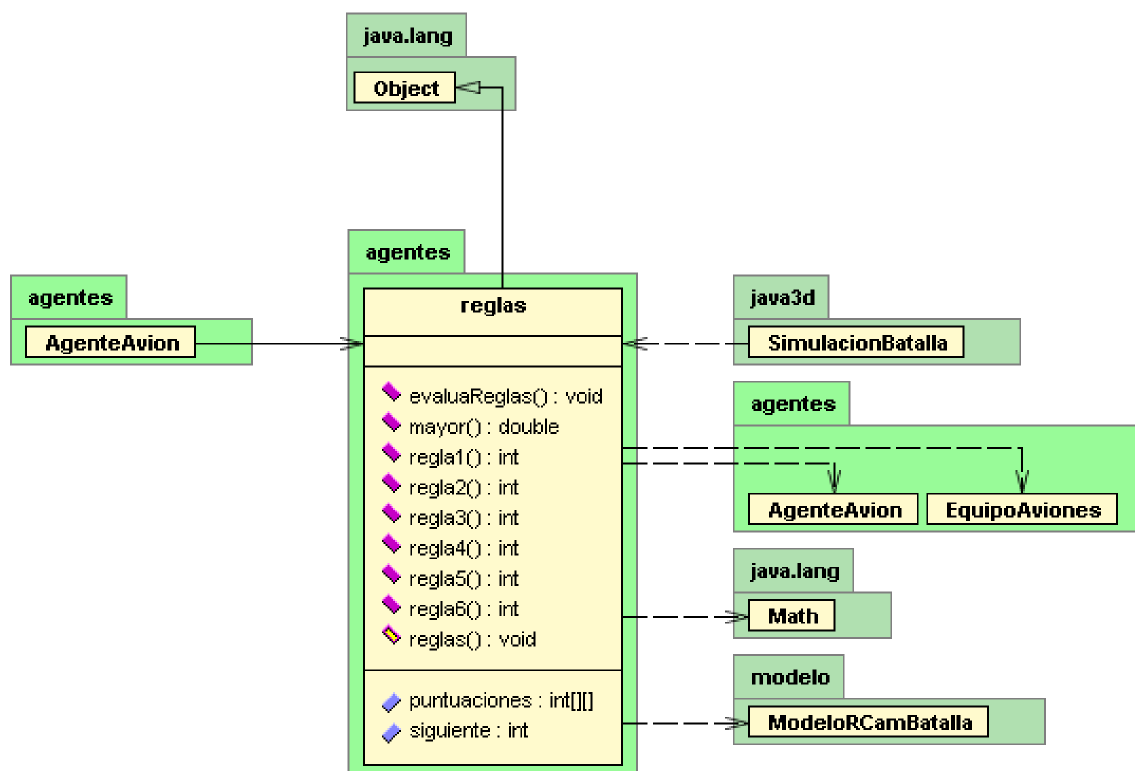
Entre los métodos, que tiene la clase, conviene destacar dos grupos:

1. Por un lado tenemos los métodos, que implementan las reglas que hemos explicado hace un momento.
2. Por otro el evaluador de reglas, que es el encargado de dar la puntuación que le corresponde, según el contexto en que se aplican, a cada regla.

Los métodos que implementan las reglas, siguen un esquema similar. Si es la primera vez que se accede a una regla, ésta elige el camino a seguir, de manera aleatoria. Si, por el contrario, el agente ya tiene experiencia y ya ha usado con anterioridad esa regla, seguirá el camino que tenga mejor puntuación. De esta forma, nos aseguramos que los aviones vayan aprendiendo según van aplicando las reglas y basándose en experiencias pasadas.

En cuanto al método *evaluaReglas*, destacamos que es muy importante, pues es en él donde se rellena la matriz *puntuaciones*, a través de la cual sabemos la puntuación que tiene cada regla. Dicha puntuación, como ya hemos dicho antes, es vital para que los agentes decidan que regla aplicar, pues será el camino de mayor puntuación de cada regla el que elijan los aviones.

El diagrama de clases que le corresponde a esta clase es el siguiente:



6. Manual de Usuario

En éste apartado vamos a describir el funcionamiento del interfaz que pone en marcha el programa de simulación del modelo de un avión, y la simulación de una batalla. Para ello vamos a realizar capturas de pantalla e ir explicando paso a paso la funcionalidad de los diferentes botones así como el resultado obtenido.

6.1. Pantalla Inicial

Ésta es la primera pantalla que sale al ejecutar el programa, en ella podemos ver tres botones que se encargan de poner en marcha los dos métodos de funcionamiento de los que disponemos. El tercer botón es el botón *salir*, que nos permite cerrar todas las ventanas que hayan sido abiertas por nuestro programa.

Con el botón "*Modo Avión*" ejecutamos la simulación de un único avión dirigido por su propio modeloRCAM; sirve de prueba inicial para comprobar el correcto funcionamiento del modelo. De este modo podemos prestar más atención a los movimientos del avión, y modificar los parámetros que influyen en el mismo.

Con el botón "*Modo Batalla*" simulamos una batalla entre agentes, utilizando el modeloRCAM, y las reglas.

6.2. Modo avión

Panel que aparece al ejecutar "*Modo Avión*", en él tenemos una tabla "*Órdenes*" en la que podemos meter las instrucciones que queremos que lleve a cabo nuestro avión, y que más tarde podremos visualizar con diferentes características para comprobar su funcionamiento. La tabla contiene tres campos:

- Orden: Referente a cada uno de los tres parámetros que podemos variar, *altura*, *rumbo* y *velocidad*, en metros, grados y metros por segundo, como valores iniciales tenemos altura cero, posición todo cero, rumbo 45º (este), y como velocidad inicial la que introduzcamos por pantalla.
- Tiempo: Instante de tiempo de la ejecución que queremos que se produzca el cambio correspondiente a la orden.
- Valor: Valor correspondiente al cambio que hemos introducido en el campo orden (cambio de altura, de rumbo o de velocidad), trabajamos con valores absolutos no incrementos sobre los valores iniciales. A continuación veremos un ejemplo que puede aclarar alguna duda.

En el centro del panel vemos la parte de "*Opciones*", en la columna de la izquierda se encuentran los *checkBox* para seleccionar los datos que queremos mostrar en el *TextArea* de la derecha, junto con los tiempos en los que se han ejecutado, y en la columna de la derecha tenemos los *checkBox* para que se nos muestren los gráficos de las variables que hemos considerado más interesantes comprobar

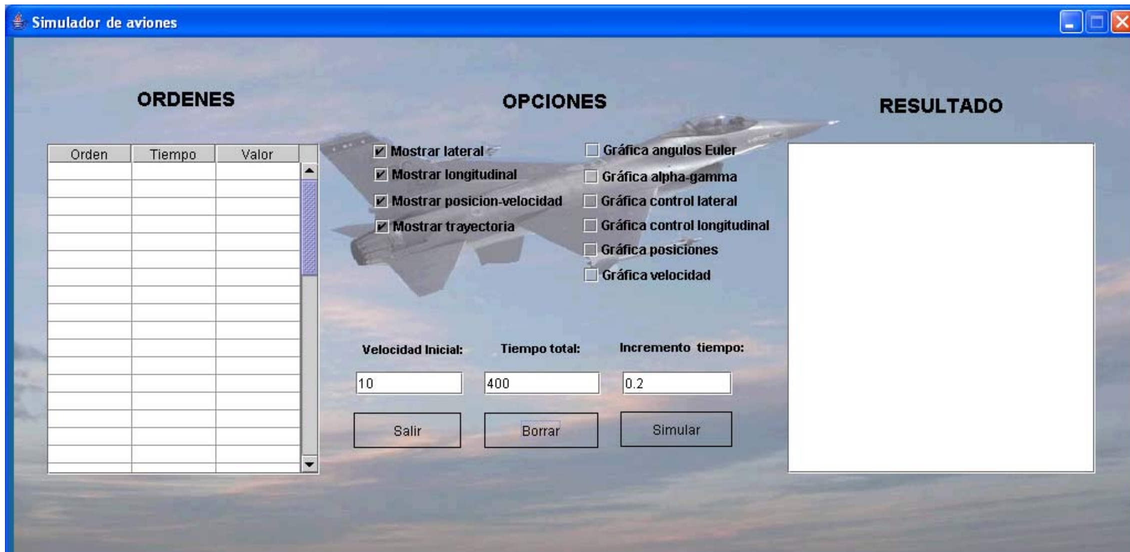
En la parte inferior introducimos los valores iniciales:

1. La *velocidad inicial*.
2. El *tiempo total de simulación*.
3. El *incremento de tiempo de cada paso*.

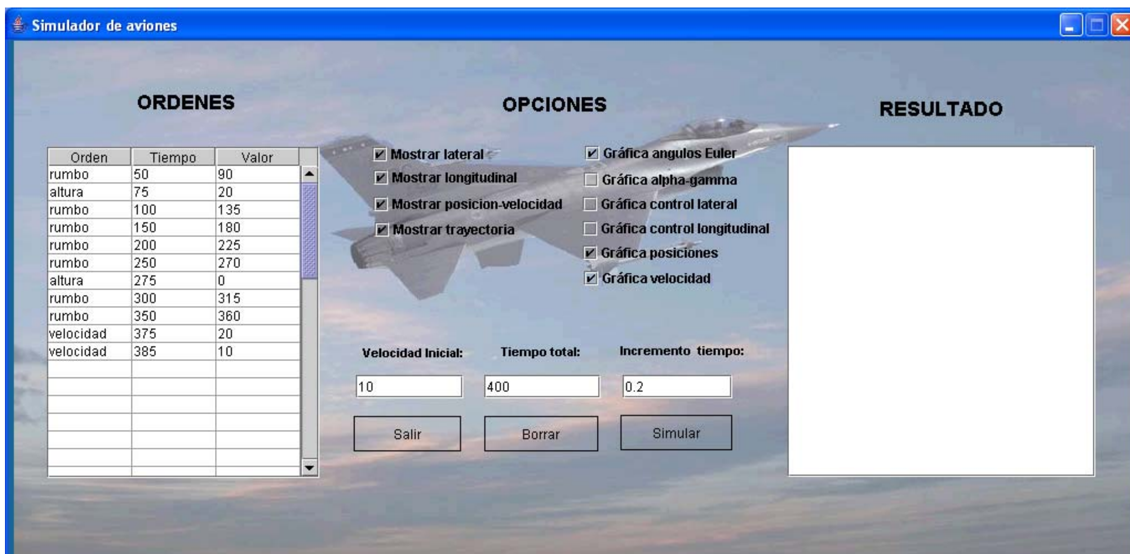
También tenemos los botones:

- "*Salir*" que cierra la ventana.
- "*Borrar*" que limpia la tabla de ordenes.
- "*Simular*" que realiza la simulación del avión.

Vemos un ejemplo del interfaz que aparece al elegir el Modo Avión:



Inicialmente hemos introducido datos dentro de las órdenes para que sea más rápida la prueba de la simulación, y no tener que meter continuamente ocho o diez cambios para comprobar el correcto funcionamiento:



A continuación vamos a mostrar los resultados obtenidos de la ejecución de la anterior captura de pantalla, en la que hacemos cambios de los tres tipos, con lo que conseguimos que el avión de una vuelta desde el punto inicial, mostramos todas las variables en "Resultado" y mostramos los gráficos de los ángulos de Euler, la de posiciones y la de la velocidad.

Gráfica de los ángulos de Euler (alabeo, guiñada y cabeceo):

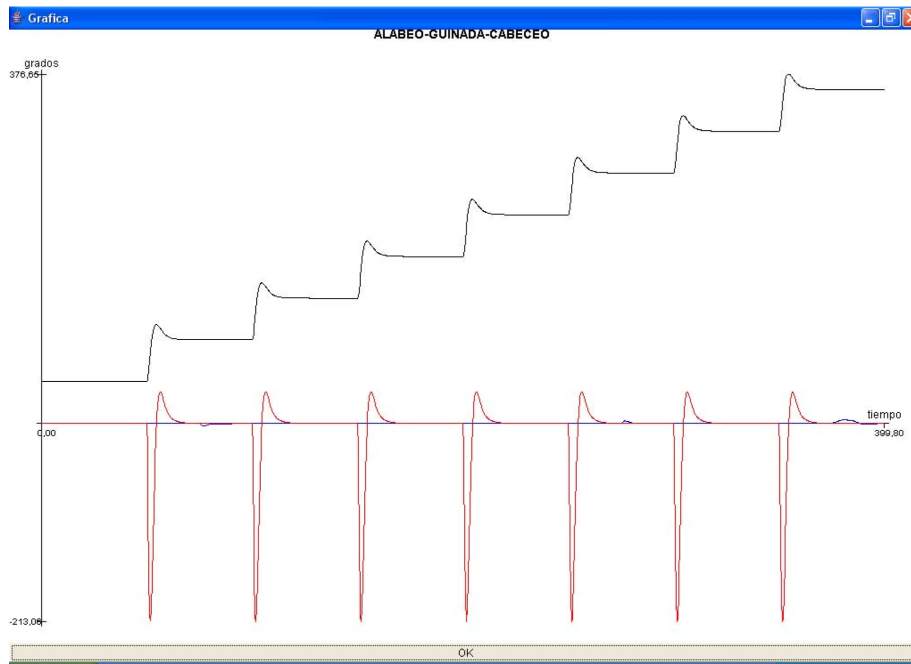


Gráfico de las posiciones (X, Y, Z). Se corresponde con un avión que lleva una trayectoria circular, podemos ver la evolución de la X y la Y, que suben y bajan o viceversa hasta volver a la coordenada inicial. La Z varía mínimamente en comparación con los valores que toman la X y la Y:

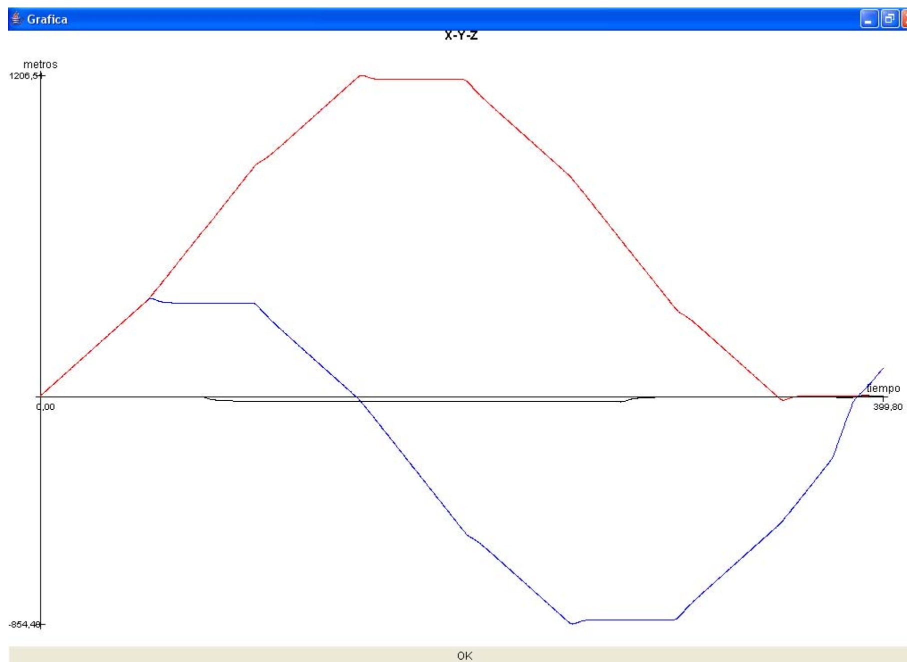
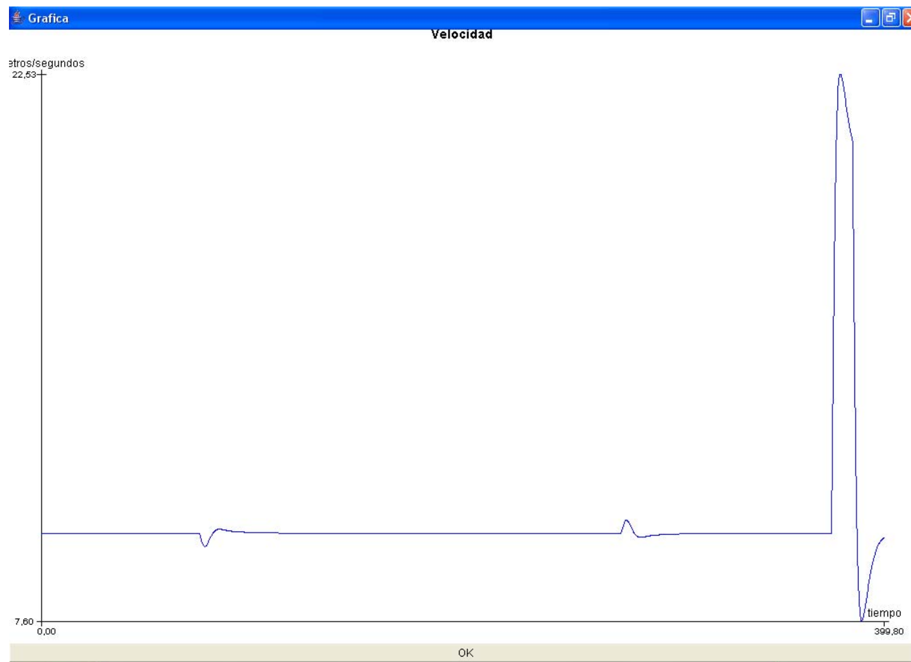


Gráfico de la velocidad:

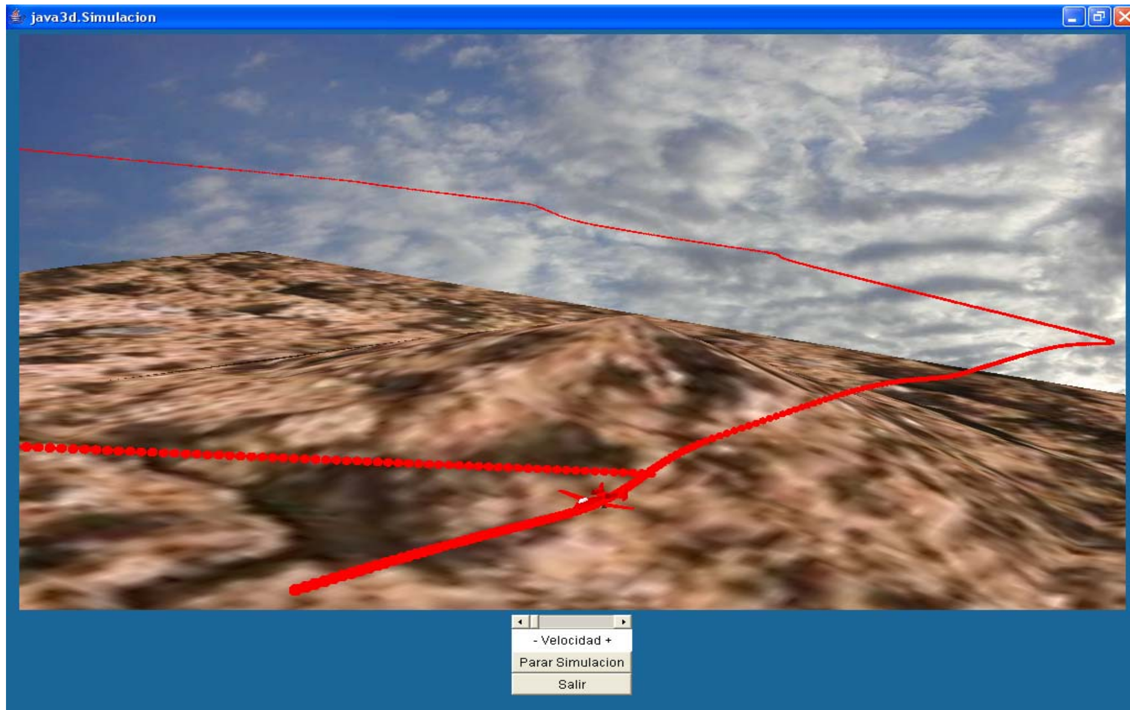


A continuación vemos la simulación en JAVA3D, el avión va siguiendo la trayectoria que le hemos definido, está representada por la línea de puntos (ésta opción se puede quitar desmarcando el '*mostrar trayectoria'*).

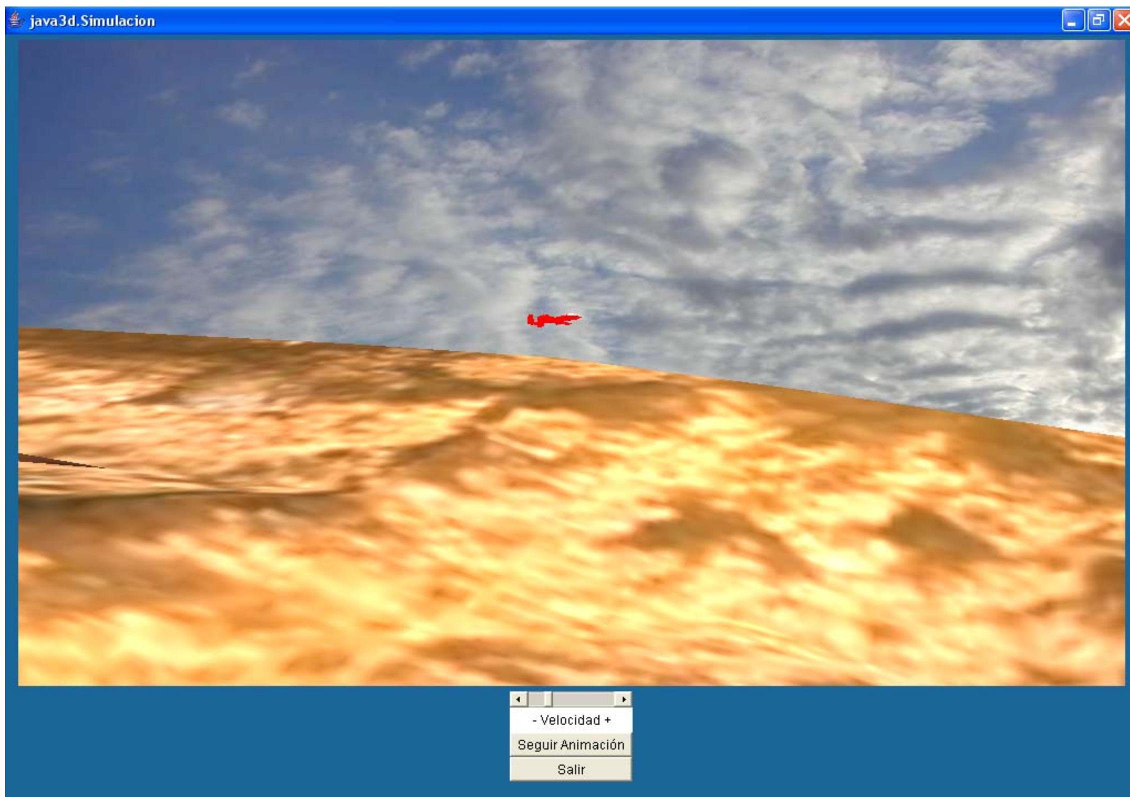
En la pantalla disponemos de un '*scroll bar*' para modificar la velocidad de simulación, más rápido o más lento, independientemente de la velocidad a la que estemos simulado (varía la velocidad de la representación, no la del modelo).

Existe otro botón que te permite parar o seguir la simulación.

Con el botón de "*Salir*", salimos de este modo y volvemos al interfaz inicial.



Ahora vemos la misma simulación pero sin la opción '*Mostrar Trayectoria*' activada, y con la animación detenida:



6.3. Modo Batalla

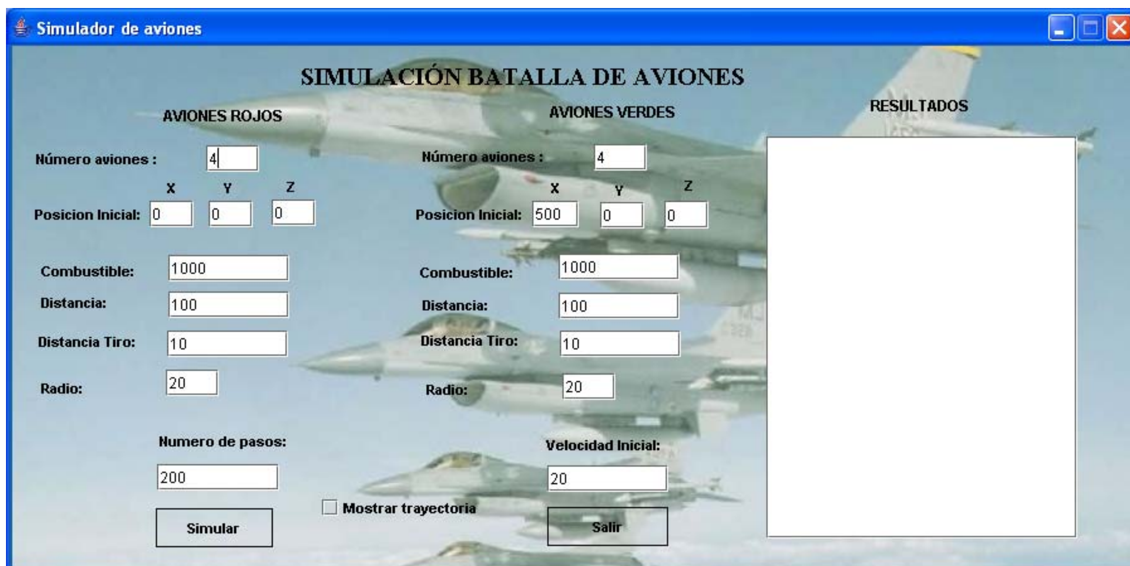
Al pulsar el botón correspondiente al modo Batalla aparece el interfaz dónde introducir los datos iniciales para el desarrollo del combate.

Disponemos de dos equipos, el rojo y el verde. El usuario tiene la opción de modificar los parámetros de estos equipos a su gusto e independientemente el uno del otro. Estos parámetros son:

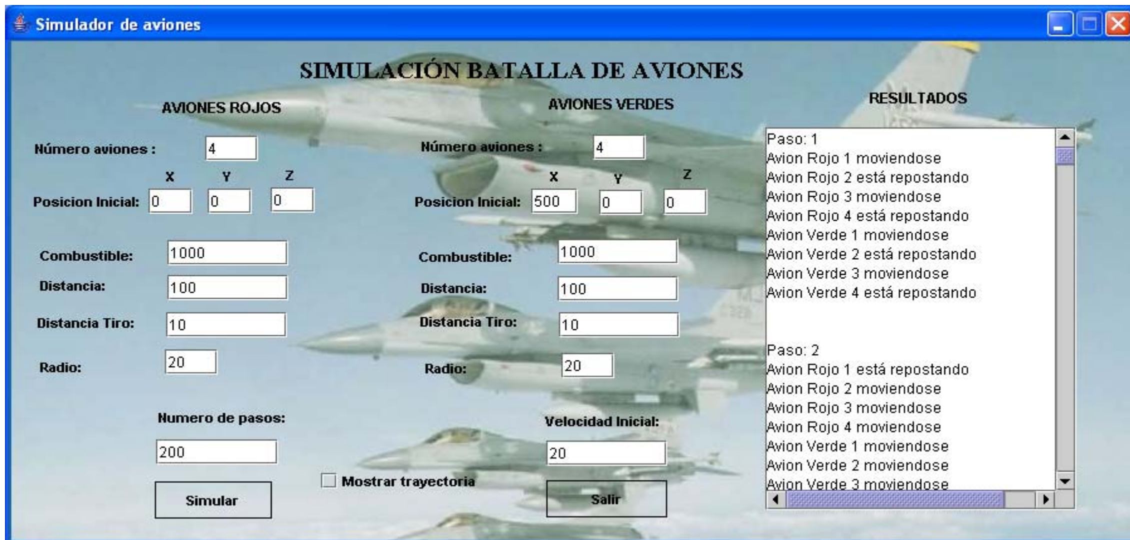
- Número de aviones de cada equipo.
- Posición inicial a partir de la cual se distribuir todos los aviones.
- Combustible del que disponen.
- Distancia del alcance del radar: Distancia mínima desde la que un avión detecta a sus enemigos.
- Distancia de tiro: Distancia mínima para que el ataque sea efectivo.
- Radio: Distancia a la que están distribuidos los aviones de la coordenada inicial

El número de pasos de la simulación y la velocidad inicial de los agentes serán parámetros comunes a los dos equipos.

También disponemos de un '*check box'* para ver la simulación con la trayectoria de cada avión o sin ella. Y dos botones, el de "*Simulación*" para simular la batalla, y el de "*Salir*" para abandonar el interfaz. Asimismo mostramos en un panel la acción que está ejecutando cada avión en un determinado instante.

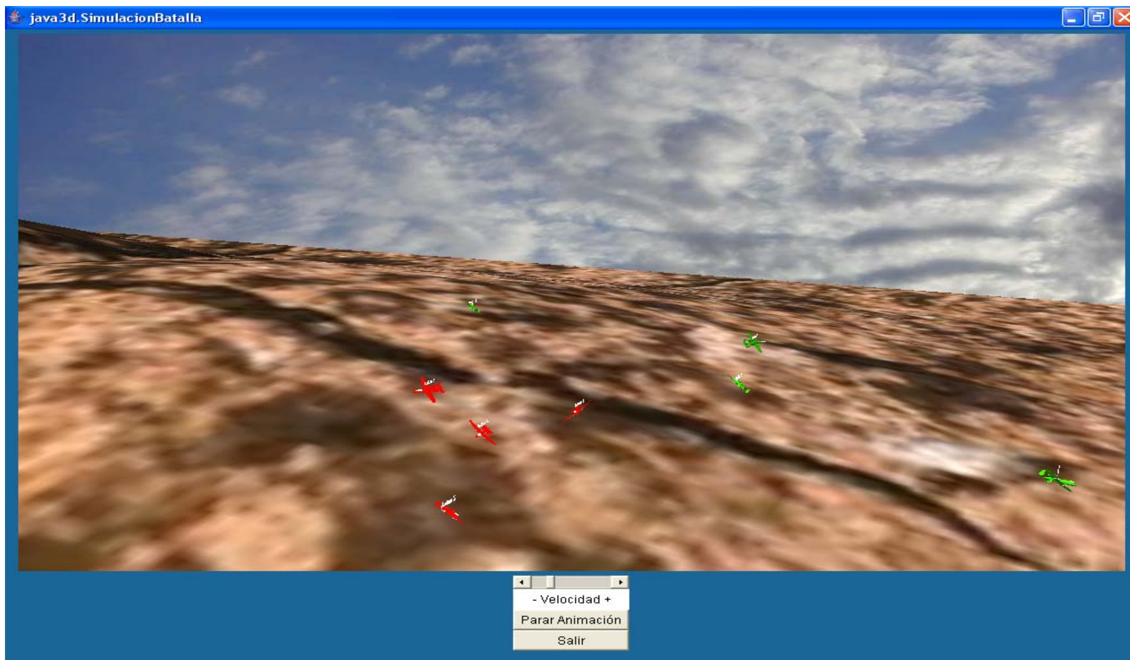


Comprobamos cómo en el panel de la derecha se visualizan las acciones de cada avión.

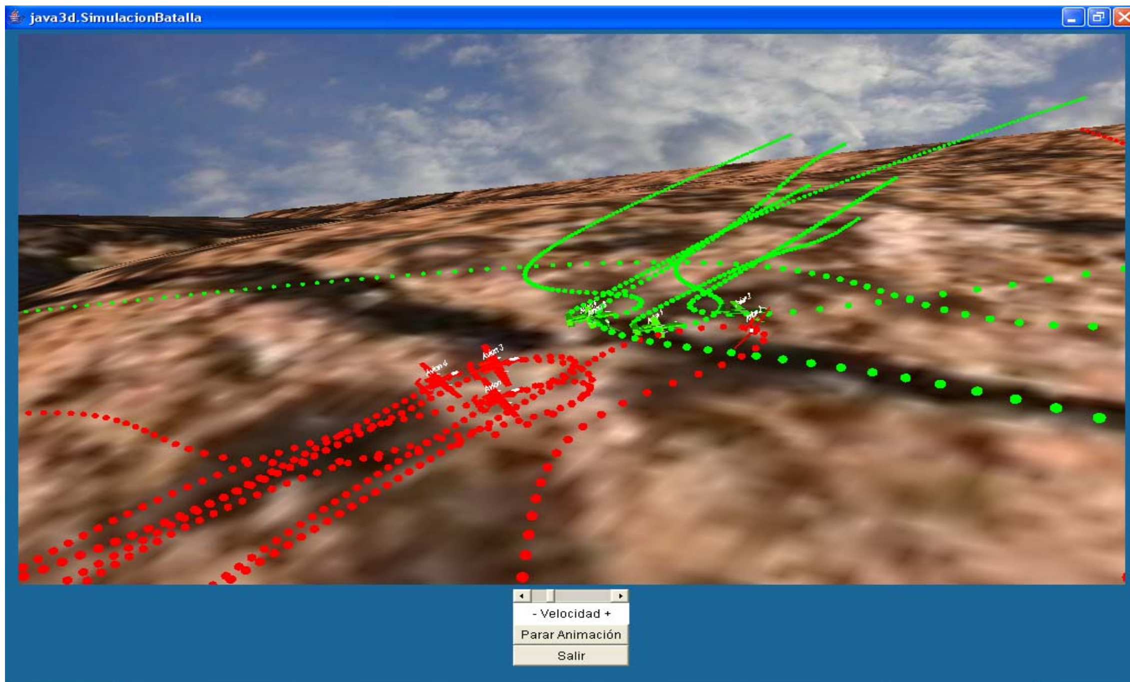


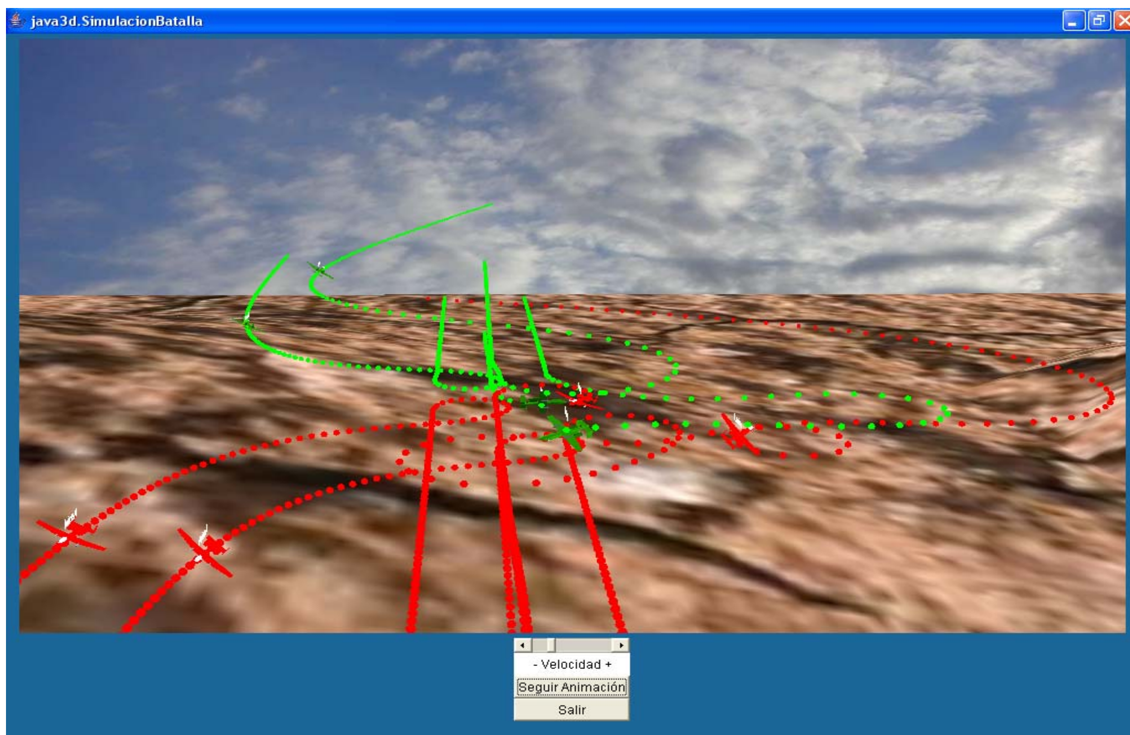
En la simulación vemos la evolución del combate, cómo unos aviones se persiguen, otros huyen, cómo luchan... Hemos decidido representar a los aviones muertos, estáticos en la posición en la cual han sido derribados. Cuando un avión está repostando se ve cómo permanece parado unos instantes y cómo luego reanuda el vuelo.

Para localizar mejor los aviones hemos puesto encima de cada uno de ellos su identificador en el color correspondiente a su equipo.



Vemos ahora la misma ejecución pero mostrando las trayectorias, para ver algo más claro el desarrollo del combate. La primera captura corresponde al inicio del combate y la segunda al final:





7. Conclusiones y Futuras Ampliaciones

7.1. Conclusiones

El objetivo principal de este Proyecto Fin de Carrera era la simulación de un modelo que reproduce las ecuaciones dinámicas reales de un avión y su representación gráfica en tres dimensiones.

Este objetivo principal ha quedado cubierto a través de los siguientes subobjetivos marcados al inicio del proyecto:

- Se ha implementado un modelo, en JAVA, que simula fielmente el movimiento de un avión real.
- Se ha diseñado e implementado un escenario para la representación gráfica del avión en tres dimensiones.
- Se ha validado el comportamiento de nuestro modelo, contrastando los valores que obteníamos en JAVA, con los valores proporcionados por el modelo real que nos han proporcionado.
- Se ha diseñado un interfaz de usuario, fácil e intuitivo para el consumidor.
- Se ha dotado a los aviones de cierta inteligencia, mediante un sistema de reglas, ideado por nosotros, consistente en prueba y error.
- Se ha creado un nuevo interfaz donde se visualiza la batalla entre dos equipos de aviones.

7.2. Futuras Ampliaciones

A pesar de que, como hemos visto, se han cumplido todos los objetivos que se plantearon para este Proyecto Fin de Carrera, se proponen una serie de ideas que se podrían realizar en un futuro, tomando nuestro proyecto como base.

Algunas de estás propuestas son las que se muestran a continuación:

- **Control de Formaciones**: Formar un escuadrón de batalla no es tan fácil como parece. Cuando varios agentes se ponen en formación, se deben tener en cuenta muchos factores, para que no se produzcan fallos o colisiones entre ellos. Una posible ampliación sería añadir una opción para que los aviones fuesen capaces deponerse en formación ellos solos y de una manera fiel a la realidad.

- **Aparatos Cooperativos**: En toda labor de equipo es imprescindible la colaboración y coordinación de sus miembros, para ello es necesario saber cómo opera u operaría cada unidad del mismo. El tener un modelo del comportamiento de cada componente facilitaría muchos datos relevantes y ayudaría a encontrar la mejor forma de cooperación entre ellos.

- **Aprendizaje de estrategias mediante simulación**: Sería interesante que cada vez que se simulase un comportamiento, los agentes aprendiesen del mismo y fuesen evolucionando en cada iteración. Una especie de combinación entre simulación y programación evolutiva.

Bibliografía

[Bigus01] Joseph P Bigus and Jennifer Bigus. *Constructing intelligent agents using JAVA*. Wiley. New York.2001

[Mestre04] J.L. Mestre, I. Luño and I. Morales. *Servidor de Agentes*. Proyecto de Sistemas Informáticos del curso 2003/2004.Facultad de Informática, Universidad Complutense de Madrid. 2004

[Gonz05] D. Gonzalez, M. Gomez, R.Vallejo. *Agentes capaces de deducir de forma cooperativa relaciones causa efecto en un entorno estructurado.* Proyecto de Sistemas Informáticos del curso 2004/2005. Facultad de Informática, Universidad Complutense de Madrid. 2005

[Black91] Blakelock J.H. *Automatic control of Aircraft and Missiles*. 2º Edition. John Wiley & Sons, 1991.

[CruzAr] J. M. de la Cruz Garcia, J.Aranda Almansa. *Ecuaciones dinámicas de aviones.* Dpto Arquitectura de Computadores, Universidad Complutense de Madrid.

[Brys94] Bryson A.E. *Control of Spacecraft and Aircraft*. Princeton University Press, 1994.

[Nelson] Nelson. *Flight Stability and Automatic Control*.

[MSS06] M. Santos Peñas. *Modelado y Simulación de Sistemas*. Dpto Arquitectura de Computadores, Universidad Complutense de Madrid.2006

[Matl97] Matlab/Simulink. User's Guide. Math Works, 1997

[Benn95] B. S. Bennet. Simulation Fundamental. Prentice-Hall. London, 1995

[Paj05] G. Pajares, M.Santos. *Inteligencia Artificial e Ingeniería del Conocimiento.* RA-MA, 2005

[Stu03] Stuart Russell, Peter Norvig. *Inteligencia Artificial un Enfoque Moderno.* Prentice Hall, 2003

[Iaj06] *Introducción a la API Java3D*. Disponible en la web: http://www.cricava.com/java/introducci_oacute_n_a_la_api_java3d

[Tj06] *Tutorial Java3D*
Autor: Sun Microsystem
Traductor: Juan Antonio Palos
Disponible en web:
http://www.programacion.com/java/tutorial/3d/

[Sun06] *Java Runtime Environment Version 5.0 Update 7.*
Disponible en web:
http://www.java.com/es/download/index.jsp

[W3d06] *The Virtual Reality Modeling Language.* Disponible en web:
http://www.web3d.org/

[Xj3d06] *Cargador de archivos* .WRL. Disponible en web:
http://www.xj3d.org/download.html

[J3d06] *Información general sobre Java3d.* Disponible en web
http://java3d.j3d.org/faq/vrml.html

[Aj06] *API Java3d, documentación y ejemplos*. Disponible en web:
https://java3d.dev.java.net/binary-builds.html

[Jg06] *Clases sencillas para entrada/salida gráfica en Java.*
Disponible en web:
http://www.ctr.unican.es/fundamentos

[Jh03] Programado por Sagran, 20 Julio 2003
Clase que proporciona la posibilidad de mostrar un fondo. Disponible en web:
http://www.javahispano.org/code/snippets/panelImagenFondo.txt

[Js06] "*Java Sun*" 1994-2006 Sun Microsystems, Inc. Software de libre distribución, maquinas virtuales: http://www.sun.com/java/

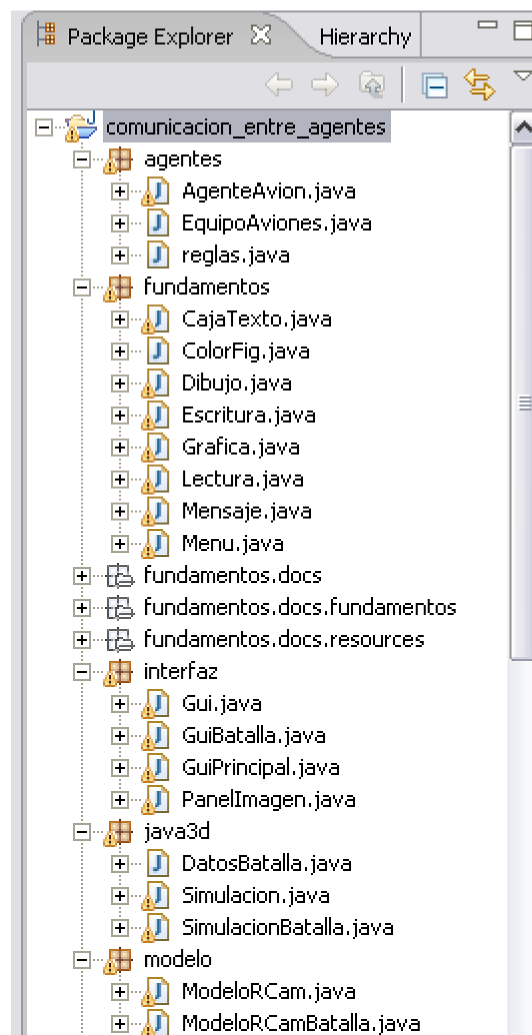
[Ck06] *Tutorial sobre commonkads*: http://www.commonkads.uva.nl/frameset-commonkads.html

[Kd06] "*Tutorial sobre KADS22 lenguaje de modelado*", University of Amsterdam 1999: http://hcs.science.uva.nl/projects/kads22/help/kads22/kads22.htm

8. Apéndice

Adjuntamos como apéndice las partes más relevantes del proyecto, que hemos realizado.

Estructura del programa:



modeloRCAM

referencias : double[]	getCHI()
th : Vector	getCHI_c()
time : double	getDA()
V0Ini : double	getDR()
VAIni : double	getDT()
velocidad : Vector	getGuiñada()
WVIni : double	getLatControl()
X : double	getLatDev()
XLat : double[]	getLongControl()
XLong : double[]	getNX()
xs : Vector	getNZ()
Y : double	getPHIX()
YLat : double[]	getPHIY()
YLong : double[]	getPSI()
ys : Vector	getPX()
Z : double	getPY()
zs : Vector	getQX()
ModeloRCam(double, double, doul	getQY()
aircraft()	getReferencias()
calculaXLat()	getRX()
calculaXLong()	getRY()
calculaYLat()	getTH()
calculaYLong()	getTHETA()
controlador()	getTime()
controladorLateral()	getUB()
controladorLongitudinal()	getV()
dameAlabeo()	getV0Ini()
dameCabeceo()	getVA_c()
dameGuiñada()	getVB()
dameVelocidad()	getWB()
dameX()	getWV()
dameY()	getWV_c()
dameZ()	getX_c()
datos(boolean, boolean, boolean,	getXA()
getAlabeo()	getXLat()
getBETA()	getXLong()
getCabeceo()	getXR()

getXTH()
getYLat()
getYLong()
getYs()
getZ_c()
getZs()
inicializaCtes()
integra(double, double, double)
integra1(double, double)
lazoInternoLateral(double, double
lazoInternoLongitudinal(double)
mostrarGraficaAG()
mostrarGraficaAGC()
mostrarGraficaDADR()
mostrarGraficaDTTH()
mostrarGraficaPos()
mostrarGraficaVel()
mover()
multiplicaMatrices(double[][], dout
multiplicaMatrizEscalar(double[][],
principal(double, double, double, c
señalesTodas()
setAlabeo(Vector)
setBETA(double)
setCabeceo(Vector)
setCHI(double)
setCHI_c(double)
setDA(double)
setDR(double)
setDT(double)
setGuiñada(Vector)
setLatControl(double[])
setLatDev(double)
setLongControl(double[])
setNX(double)
setNZ(double)
setPHIX(double)
setPHIX(double)
setPHIY(double)
setPSI(double)
setPX(double)
setPY(double)
setQX(double)
setQY(double)
setReferencias(double[])
setRX(double)
setRY(double)
setTH(double)
setTHETA(double)
setUB(double)
setV(double)
setV0Ini(double)
setVA_c(double)
setVB(double)
setWB(double)
setWV(double)
setWV_c(double)
setX_c(double)
setXA(double)
setXLat(double[])
setXLong(double[])
setXR(double)
setXs(Vector)
setXT(double)
setXTH(double)
setYLat(double[])
setYLong(double[])
setYs(Vector)
setZ_c(double)
setZs(Vector)
sumaMatrices(double[][], double[][]
tangente(double, double)
trayectory_generator()
trayectory_generatorIni(double, do

AgenteAvion

AgenteAvion
- main(String[])
- aliados : Vector
- atacando : boolean
- atacandoEnemigo : boolean
- avion : ModeloRCamBatalla
- combustible : double
- contHuyendo : int
- contRepostar : int
- coordenadaX : double
- coordenadaY : double
- coordenadaZ : double
- enemigos : Vector
- huyendo : boolean
- identificador : int
- inteligencia : reglas
- moviendo : boolean
- persiguiendo : boolean
- regla : int
- repostando : boolean
- vivo : boolean
- AgenteAvion(int)
- AgenteAvion(ModeloRCamBat
- actualizaCoordenadas()
- atacar(double)
- avionesEnemigos(EquipoAvion
- calculaDistanciaAviones(Agent
- consume()
- eliminar(EquipoAviones)
- enemigoATiro(AgenteAvion, d
- enemigoMasCercano()
- enemigosVivos()
- esEnemigo(EquipoAviones)
- getAliados()
- getAvion()
- getCombustible()
- getContHuyendo()
- getContHuyendo()
- getContRepostar()
- getCoordenadaX()
- getCoordenadaY()
- getCoordenadaZ()
- getEnemigos()
- getIdentificador()
- getInteligencia()
- getRegla()
- HuyeAvion(AgenteAvion)
- isAtacando()
- isAtacandoEnemigo()
- isHuyendo()
- isMoviendo()
- isPersiguiendo()
- isRepostando()
- isVivo()
- masCombustible(AgenteAvion)
- mismoEnemigo(AgenteAvion)
- mover(double, EquipoAviones)
- persigueAvion(AgenteAvion)
- repostar()
- setAliado(AgenteAvion)
- setAliados(Vector)
- setAtacando(boolean)
- setAtacandoEnemigo(boolean)
- setAvion(ModeloRCamBatalla)
- setCombustible(double)
- setContHuyendo(int)
- setContRepostar(int)
- setCoordenadaX(double)
- setCoordenadaY(double)
- setCoordenadaZ(double)
- setEnemigo(AgenteAvion)
- setEnemigos(Vector)
- setHuyendo(boolean)
- setIdentificador(int)
- setContRepostar(int)
- setCoordenadaX(double)
- setCoordenadaY(double)
- setCoordenadaZ(double)
- setEnemigo(AgenteAvion)
- setEnemigos(Vector)
- setHuyendo(boolean)
- setIdentificador(int)
- setInteligencia(reglas)
- setMoviendo(boolean)
- setPersiguiendo(boolean)
- setRegla(int)
- setRepostando(boolean)
- setVivo(boolean)
- tieneCombustible()
- veoAvionEnemigo(AgenteAvio

Reglas