Facultad de Informática
Universidad Complutense de Madrid

PROYECTO FIN DE CARRERA

SISTEMAS INFORMÁTICOS

2007-2008

SISTEMA MULTIAGENTE CON CAPACIDAD DE PERCEPCIÓN VISUAL Y TOMA DE DECISIONES EN UN ENTORNO 3D

Alumnos: Pedro Javier Herrera Caro
Pablo Martín de Blas Sánchez
Ignacio Rubio Díaz

Director: Gonzalo Rubén Méndez Pozo

Colaborador: Raquel Hervás Ballesteros

Javier

A mis padres y a mi hermana, por estar siempre ahí.

Y a mis abuelos, tíos y primos, que siempre creyeron en mí.

Pablo

A mi familia.

A mis padres, en especial por todo su apoyo.

A todos los que de una u otra manera me han ayudado durante la carrera.

Ignacio

A mi familia, por todo su cariño y apoyo desde siempre.

Agradecimientos

Queremos expresar nuestro profundo agradecimiento a Gonzalo, por su ayuda y disponibilidad, siempre que hemos tenido algún problema (que no han sido pocos), para que este proyecto llegara a buen puerto.

Javier:

Quiero expresar mi agradecimiento a mis compañeros de prácticas, y a todos aquellos que de una forma u otra me ha ayudado a lo largo de la carrera. Gracias Carlos, Alvarito, Jesús, Ana, Pedro, Miguel, Cris y Conchi. Y gracias Pablo, Nacho y Pablete, habéis sido como unos hermanos para mí estos dos últimos años. Gracias por ayudar a que los días en la facultad pasaran de la mejor forma posible. También quería mostrar mi sincero agradecimiento a todos aquellos profesores que me han marcado a lo largo de la carrera. La lista sería larga pero al menos me gustaría que figuraran aquí los siguientes nombres: Román Hermida, Marta Físico, Rafael Caballero y Katzalin Olcoz. Y por último, agradecer a Gonzalo Pajares toda su ayuda este último año.

Pablo:

Las primeras líneas a quienes deseo dirigir mi agradecimiento son para mis dos compañeros de proyecto, por su dedicación, motivación y esfuerzo por que todo esto saliese adelante. Pero no sólo por esto, si no por los ratos que hemos pasado juntos a lo largo de la carrera. Gracias en especial a David Romero, Alfonso, Pablete y David Moya por la amistad que me une a ellos. Muestro mi agradecimiento también a esa buena gente que he ido conociendo en la facultad, con la que he compartido risas, cenas y otros ratos divertidos. Gracias también a los profesores que han atendido mis dudas y me han hecho aprender en uno u otro sentido, así como los que han realizado mayores esfuerzos para facilitar a los alumnos un mejor estudio de su asignatura. Doy a gracias a Dios porque puedo decir con satisfacción que esta etapa llega a su fin y se abre ante mí una nueva aventura.

Ignacio:

Quiero estar agradecido a mis compañeros de proyecto Javi y Pablo, por su buen humor y sin los cuales esto no habría sido lo mismo. A Pablete, gran amigo y compañero de fatigas para lo que haga falta. A David y a Bego, por ser como son y porque sin ellos las interminables horas de biblioteca no serían lo mismo.

Resumen

Se ha diseñado un sistema de agentes software en un entorno 3D, donde se simulan situaciones basadas en las acciones y reacciones de agentes hostiles entre sí, de manera que se establecen comportamientos de persecución y huida como las acciones fundamentales.

Con el fin de modelar los agentes mediante un comportamiento más realista, se les ha dotado de un sistema de percepción visual, de tal forma que no puedan conocer la situación de cada agente en el entorno, sino únicamente la de aquellos agentes que estén en ese momento en su campo de visión.

Mediante el uso de técnicas de Inteligencia Artificial los agentes son capaces de responder de manera inteligente a los sucesos del entorno, perseguir metas, o interactuar con otros agentes de manera similar a como lo haría un humano.

Palabras clave: Sistema Multiagente, Java 3D, Claridad de Percepción,
Toma de Decisiones Multicriterio.

Abstract

An agent-based system has been designed within a 3D environment, where a range of situations have been simulated based on actions and reactions of agents hostile to each other, in order to establish fundamental actions of chase and flight.

With the aim of modelling agents with a more realistic behaviour, they have been equipped with a visual perception mechanism, so that they cannot know the status of any agent in the environment, but only those who are at that moment in their field of view.

Using techniques of Artificial Intelligence, agents are able to respond intelligently to the events within the environment, pursue goals, or interact with others agents in a human-like way.

Keywords: Multi-Agent System, Java 3D, Clarity of Perception,
Multi-Criteria Decision Making.

Se autoriza a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales, tanto esta memoria, como el código y el prototipo desarrollado.

Los autores:

Pedro Javier Herrera Caro

Pablo Martín de Blas Sánchez

Ignacio Rubio Díaz

Índice

PARTE 1. INTRODUCCIÓN 1
CAPÍTULO 1. INTRODUCCIÓN 1
1.1 Motivación 2
1.1.1 ¿Por qué dotarles de percepción visual? 3
1.1.2 ¿Por qué tomar decisiones? 3
1.2 Objetivos iniciales del proyecto 4
1.3 Organización de la memoria 5

PARTE 2. DESARROLLO 7
CAPÍTULO 2. PLAN DE FASE 7
2.1 Fases 7
2.2 Personal 8
CAPÍTULO 3. RIESGOS 17
3.1 Riesgos iniciales del proyecto 17
3.1.1 Identificación de riesgos 17
3.1.2 Análisis de riesgos 20
3.1.3 Planificación de riesgos 23
3.2 Riesgos finales del proyecto 28
3.2.1 Identificación de riesgos 28
3.2.2 Análisis de riesgos 29
3.2.3 Planificación de riesgos 29
CAPÍTULO 4. PLATAFORMA DE AGENTES 31
4.1 Agentes 31
4.2 Comportamientos 31
4.3 Mensajes ACL 33
4.4 Coordenadas y orientación 34
4.5 Trayectorias de los agentes 34
4.6 Páginas Amarillas 35
CAPÍTULO 5. DISEÑO DEL MODELO DE PERCEPCIÓN 37
CAPÍTULO 6. DETECCIÓN DE COLISIONES 41
6.1 Comportamientos: La base para la interacción 41
6.2 Escribir una clase *Behaviour* 42
6.3 Región Programada 42
6.4 Riesgos de programación al crear clases *Behaviour* 43
6.5 Riesgos de programación al usar objetos *Behaviour* 43
6.6 Recomendaciones de diseño para la clase *Behaviour* 44
6.7 Condiciones de disparo de los comportamientos 44
6.8 Árbol de escena 46
CAPÍTULO 7. TOMA DE DECISIONES 49
7.1 Enfoque general MCDM Fuzzy 49
7.1.1 Formulación del problema y definiciones 49
7.1.2 Normalización 50
7.1.3 Añadiendo pesos a los criterios 51
7.1.4 Elección de alternativas 51
7.2 Criterios que intervienen en la toma de decisiones 52
7.2.1 Criterio 1: Tipo de agente 52
7.2.2 Criterio 2: Orientación 53
7.2.3 Criterio 3: Claridad de percepción 55
7.2.4 Criterio 4: Distancia 56

7.3 Alternativas que intervienen en la toma de decisiones 56
7.3.1 Alternativas para el agente *Gato* 56
7.3.2 Alternativas para el agente *Ratón* 57
CAPÍTULO 8. COMUNICACIÓN MEDIANTE SOCKETS 59
8.1 Necesidades del proyecto 59
8.2 Protocolo de comunicación 60
8.2.1 Elementos de la arquitectura Cliente-Servidor 60
8.2.2 Características del modelo Cliente-Servidor 61
8.2.3 Ventajas de la arquitectura Cliente-Servidor 63
8.3 Diferencias entre plataformas 64
CAPÍTULO 9. ENTORNO VIRTUAL 65
9.1 Descripción 65
9.2 Árbol de escena 65
9.3 Modelos 3D 66
CAPÍTULO 10. DIAGRAMAS UML 71
10.1 Comportamientos 71
10.2 Java3D 75
10.3 Colisión Matemática 78
10.4 Trayectorias 80
10.5 Entorno Virtual 82
10.6 Socket 85
CAPÍTULO 11. PROTOTIPOS REALIZADOS 87
11.1 Sockets 87
11.1.1 Primer prototipo cliente Java- servidor Java 87
11.1.2 Primer prototipo cliente C - servidor C 88
11.1.3 Segundo prototipo cliente C - servidor C 88
11.1.4 Segundo prototipo cliente Java - servidor Java 89
11.1.5 Tercer prototipo cliente C - servidor C 89
11.2 JADE 90
11.3 OSG 90
11.4 Java3D 93

PARTE 3. RESULTADOS 99
CAPÍTULO 12. ANÁLISIS DE RESULTADOS 99
CAPÍTULO 13. CONCLUSIONES 109
13.1 Cumplimiento de los objetivos propuestos 110
13.2 Trabajos futuros 112

PARTE 4. ANEXOS 115
ANEXO A. JAVA 3D 115
A.1 Qué es Java 3D 115
A.2 Objetivos 116
A.3 Grafo de escena 116
A.4 Paradigma de programación 117
A.4.1. Modelo de programación del grafo de escena 118
A.4.2. Aplicaciones y applets 118
A.5 Instalar Java 3D 119
A.6 Obtener Java 3D 120
A.7 Proceso de instalación 120
ANEXO B. JADE (JAVA AGENT DEVELOPMENT FRAMEWORK) 121
B.1 Características 124
B.2 Instalación 124
ANEXO C. ECLIPSE IDE 127
ANEXO D. COMPILAR Y EJECUTAR JADE CON ECLIPSE 135

ANEXO E. CVS BERLIOS CON ECLIPSE 137
E.1 Introducción 137
E.2 Eclipse 138
E.2.1. Crear un módulo 138
E.2.2. Bajarse el proyecto del CVS 139
E.2.3. Subir el proyecto al CVS 140
E.2.4. Conflictos al hacer "Synchronize with Repository" 140
E.3 Enlaces de interés 141
ANEXO F. OPENSCENEGRAPH 143
F.1 Descripción 143
F.2 Características 144
ANEXO G. CONFIGURACIÓN VISUAL STUDIO 2005 147
G.1 Cómo ejecutar proyectos de OSG 147
ANEXO H. LIBRERÍAS LOG4J 155
H.1 Niveles de prioridad 155
H.2 Configuración 156
H.3 Uso 156

PARTE 5. BIBLIOGRAFÍA 159

Parte 1. Introducción

Capítulo 1. Introducción

En la actualidad, los Entornos Virtuales se están convirtiendo en una herramienta cada vez más habitual en diversos ámbitos, como el educativo, el diseño industrial, el entrenamiento militar o las visitas virtuales a museos o edificios históricos. Uno de los requisitos que comienzan a ser comunes consiste en dotar a los personajes virtuales de comportamientos similares a los humanos para crear aplicaciones con mayor sensación de realismo.

En numerosos entornos de simulación resulta necesario tomar algún tipo de decisión por parte de los agentes involucrados. Se hace necesario recurrir a algún tipo de estrategia basada en el paradigma de la Teoría de la Decisión. Existen diversos métodos y estrategias dentro de este paradigma para decidir sobre la alternativa a elegir en la decisión. En los entornos reales, las variables de decisión toman sus valores de acuerdo a la situación real del momento. Sin embargo, en simulación la realidad pasa a ser simulación y ésta a su vez es controlada por un supervisor o instructor, de forma que las decisiones serán deterministas desde el punto de vista de este instructor. Incluso cuando se induce aleatoriedad, la función de densidad de probabilidad aleatoria elegida nos proporcionará los valores previstos tras una serie suficientemente alta de carreras de Monte Carlo.

Se ha desarrollado, por parte de uno de los integrantes del proyecto, un trabajo de investigación que se presenta como proyecto fin de Master del Master en Investigación en Informática, en el que se pretende establecer las bases para la aplicación del enfoque conocido como Teoría de la Decisión Multicriterio (MultiCriteria-Decision Making, MCDM) bajo la perspectiva de la lógica borrosa o fuzzy. Con ello se consigue introducir un componente aleatorio en el entorno de simulación donde el aspecto determinista de las decisiones se sustituye por una componente predominantemente aleatoria.

Se ha aplicado esta teoría al entorno de simulación desarrollado, con agentes dotados de una cierta capacidad perceptual, de forma que las variables de decisión están condicionadas a la percepción visual obtenida por los diferentes agentes.

Como se ha mencionado anteriormente, el entorno se ha diseñado con unas pautas tales que permitan la verificación del método MCDM empleado. De este modo, se incluyen varios agentes, donde una serie de ellos es hostil al resto. Debido a ese comportamiento hostil unos u otros deben reaccionar ante determinados estímulos. Los agentes se mueven libremente por un entorno 3D.

Aunque en el entorno sólo actúan un número limitado de agentes del tipo *Gato* y *Ratón* que simulan las acciones de acción y reacción descritas, el planteamiento que se hace resulta ser fácilmente extensible a entornos con un mayor número de agentes sin más que incrementar o modificar las variables de decisión e incluso los propios criterios.

Los agentes se modelan en el sistema desarrollado como gatos y ratones, donde cada uno de ellos tiene un rol determinado. Por el momento, no se incluye coordinación entre ellos, si bien en el futuro podría hacerse sin más que añadir más variables de decisión.

1.1 Motivación

La idea de elaborar un sistema multiagente surge de reuniones iniciales en las que se barajan diversas posibilidades decantándose finalmente por la escogida.

Se pretendía aprovechar el trabajo paralelo que uno de los integrantes del proyecto estaba realizando como proyecto fin de Master para el Master en Investigación en Informática, con lo que se decidió diseñar un sistema en el que tuviera cabida la teoría comentada con anterioridad.

Los agentes que se encuentran en el entorno *virtual*, gatos y ratones, tienen unos objetivos definidos: los gatos deben cazar a los ratones, y éstos impedirlo.

Gracias al mecanismo de percepción visual los gatos sólo pueden perseguir a los ratones que perciben en cada momento, mientras que los ratones sólo pueden huir de aquellos gatos que estén próximos a ellos.

Una vez un agente percibiera a otro, cabrían varias alternativas posibles en función del tipo de agente. La conjunción de variables de decisión y alternativas a tomar conduce a la elección de un mecanismo basado en la Teoría de la Decisión.

1.1.1 ¿Por qué dotarles de percepción visual?

Desde el primer momento se mostró un interés por que los agentes que iban a interaccionar en el entorno realizado a tal efecto, no tuvieran conocimiento del mismo en todo momento, sino una parte de él, el que estuvieran percibiendo en cada momento.

Muchos trabajos en Sistemas Multiagentes Virtuales Inteligentes (mAVIs) en Entornos Virtuales (EV) tienen como objetivo proveer a los agentes de un alto grado de realismo en cuanto a su representación física o incluso su comportamiento. Pero, sin embargo, muy pocos estudios se han realizado centrándose en la percepción.

Los trabajos realizados por Herrero y De Antonio (2002 y 2003) han influido en gran medida para decantarnos por este mecanismo de percepción para dotar a los agentes de mayor realismo.

El objetivo de un Agente Virtual Inteligente (AVI) percibiendo un agente es llegar a tener consciencia de dicho agente.

En este proyecto se ha pretendido dotar a los AVIs de un mecanismo de percepción que les permitieran tener un conocimiento realista del entorno. Para ello hemos empleado el modelo de percepción para AVIs que introduce mayor similitud con el sistema de percepción humano[1].

1.1.2 ¿Por qué tomar decisiones?

Como consecuencia de incorporar el comportamiento anterior, surgió la necesidad de elaborar un método de toma de decisiones que permitiera capacitar a los agentes para responder de manera inteligente a los sucesos del entorno.

Uno de los principales problemas derivados del hecho de utilizar un entorno virtual surge porque el modelado de la incertidumbre no resulta trivial. O visto desde otra

[1] Véase Capítulo 5

perspectiva puede resultar tan trivial que carece de incertidumbre. En los grandes sistemas de simulación suele ser habitual tomar una decisión en función de si entre dos objetos existe línea de vista o no, es decir hay obstáculos entre ellos o se encuentran libres de obstáculos. En este caso la incertidumbre resulta difícilmente modelable y nos encontraríamos ante un sistema de decisión determinista sin incertidumbre.

Siguiendo la filosofía expuesta en Ríos y col. (2002) respecto de la incertidumbre y su modelado existen en la realidad multitud de situaciones en las que el decisor no tiene seguridad sobre lo que ocurrirá cuando elija una determinada alternativa de entre varias que se ofrecen. En los sistemas de simulación con decidores deterministas estas situaciones o bien no existen o están minimizadas. En efecto, dependiendo de las variables que introduzca el instructor el proceso de simulación realizará una acción que puede preverse de antemano.

El siguiente paso consistió en elegir un mecanismo apropiado con el fin de poder introducir esa incertidumbre en un entorno de simulación. Uno de tales procedimientos nos lo proporciona la denominada MCDM fuzzy (Chen, 2000; Jiang y Chen, 2005; Ribeiro, 1996; Wang y Fenton, 2008). El análisis multicriterio es una metodología de toma de decisiones que se ha impuesto como la idónea en multitud de campos de aplicación. El importante subcaso en el que hay que decidir entre varias alternativas, desde unas pocas a algunos centenares, teniendo en cuenta diversos criterios o puntos de vista, surge frecuentemente. A este tipo de problemas se dedica la llamada Decisión Multicriterio, cuyo grado de madurez científica está ya hoy sólidamente establecido. No sólo en su faceta puramente teórica, en donde cuenta con un notable cuerpo de propuestas, resultados y vías abiertas de investigación, sino en la aplicada, dada su extensa gama de aplicaciones en muy diversos contextos.

1.2 Objetivos iniciales del proyecto

Bajo la apariencia de un juego, se pretendía desarrollar una aplicación piloto, basada en agentes, que mediante el uso de técnicas de Inteligencia Artificial fuera capaz de responder de manera inteligente a los sucesos del entorno, perseguir metas, o interactuar con el usuario o con otros agentes de manera similar a como lo haría un humano.

Como objetivo adicional, se pretendía que los componentes desarrollados fueran reutilizables y fácilmente modificables, de manera que pudieran ser utilizados en otras aplicaciones sin necesidad de implementarlos de nuevo.

Entrando en detalle, se enumeran los objetivos iniciales que perseguía el proyecto:

1. Diseñar e implementar un sistema multiagente.
2. Dotar a los agentes de la capacidad de percibir el entorno visualmente.
3. Dotarles de la capacidad de responder de manera inteligente a los sucesos del entorno (tomar decisiones).
4. Diseñar e implementar un entorno virtual 3D.
5. Diseñar e implementar un sistema de comunicación entre la plataforma de agentes y el entorno virtual, convirtiéndolo en un sistema independiente de la interfaz gráfica de usuario (GUI).

1.3 Organización de la memoria

La memoria consta de cuatro partes claramente diferenciadas. Una primera parte que incluye una introducción en la que se da una visión global de la aplicación que se ha desarrollado, así como de las motivaciones y objetivos iniciales del proyecto. En la segunda parte se presenta el grueso del proyecto. Aquí se detalla el Plan de Fase, los riesgos iniciales y finales, así como un apartado por cada uno de los módulos principales en los que podemos dividir nuestro sistema: la plataforma de agentes, el diseño del modelo de percepción, detección de colisiones, toma de decisiones, entorno virtual y la comunicación mediante sockets entre la plataforma de agentes implementada en Java y el entorno 3D implementado en C++. Además se incluye un apartado explicando los diversos prototipos que se fueron elaborando durante el transcurso del proyecto. En la tercera parte se analizan los resultados obtenidos a partir de las pruebas realizadas sobre el sistema. En la cuarta parte se comentan las conclusiones obtenidas, objetivos cumplidos, así como trabajos futuros. Por último, se adjuntan diversos anexos en los que se explican las diversas tecnologías y herramientas utilizadas para llevar a buen puerto este proyecto.

Parte 2. Desarrollo

Capítulo 2. Plan de fase

2.1 Fases

Para el desarrollo del proyecto a lo largo de los ocho meses de trabajo se diseñó un Plan de Fase dividido en cuatro Fases principales (Inicio, Elaboración, Construcción y Transición) y se establecieron unas fechas tentativas de inicio y finalización de cada una de ellas, conforme a las estimaciones iniciales acerca de la carga de trabajo que cada una de ellas podría acarrear.

En principio, la planificación fue la siguiente:

Fase	Fecha de inicio	Fecha de Finalización
Inicio	15 de octubre	1 de noviembre
Elaboración	2 de noviembre	29 de febrero
Construcción	1 de marzo	15 de mayo
Transición	16 de mayo	Fecha entrega

Tabla 2.1 Planificación inicial

Esta planificación fue sufriendo modificaciones a lo largo del desarrollo del proyecto, debido en unas ocasiones a la necesidad de ampliar plazos para poder terminar el trabajo concertado para esa fase, o para mejorar el resultado final de lo desarrollado, y en otras la posibilidad de acortar plazos gracias a una redistribución más eficiente de las tareas y una estimación demasiado pesimista en cuanto a carga de trabajo. El resultado final, tal y como se terminó por realizar en el curso, es el siguiente:

Fase	Fecha de inicio	Fecha de Finalización
Inicio	15 de octubre	15 de diciembre
Elaboración	16 de diciembre	29 de febrero
Construcción	1 de marzo	15 de mayo
Transición	16 de mayo	Fecha entrega

Tabla 2.2 Planificación final

Los objetivos establecidos para cada fase, los cuales no han cambiado a lo largo de todo el proyecto, han sido los siguientes:

- **Fase de inicio**: Se especifica la visión del proyecto. La idea inicial para el desarrollo se lleva al punto de estar (al menos internamente) suficientemente bien fundamentada para garantizar la entrada en la fase de elaboración.
- **Fase de elaboración**: Se definen la visión del producto y su arquitectura. Se expresan con claridad los requisitos del sistema, se establecen las prioridades entre ellos, y son utilizados para crear una sólida base arquitectónica. Se planifican las actividades y los recursos necesarios.
- **Fase de construcción**: Se construye el producto mediante una serie de iteraciones incrementales. Se lleva el software desde una base arquitectónica ejecutable hasta su disponibilidad para la comunidad de usuarios.
- **Fase de transición**: El software es puesto en manos de la comunidad de usuarios. Manufactura. Entrega. Formación...

2.2 Personal

La distribución del trabajo al inicio, y una vez se definieron claramente cuales iban a ser las partes principales del proyecto, fue el siguiente:

15 octubre – 31 octubre. Inicialmente, estuvimos documentándonos acerca de JADE[2] y su funcionamiento probando pequeños ejemplos y realizando un pequeño prototipo.

1 noviembre – 15 noviembre. Se decidió repartir tareas para resolver cuanto antes los distintos riesgos tecnológicos con los que se enfrentaba nuestro proyecto:

- La plataforma de agentes con **JADE** donde se encuentra el motor de **detección de colisiones** y **toma de decisiones**. Javier se encargó de documentarse y buscar la manera de realizar la detección de colisiones entre los *focus* y *nimbus*[3] de los agentes/objetos que interactúan en el entorno 3D. Inicialmente se pensó en

[2] Véase Anexo II.
[3] Véase Apartado 2.4.

implementarlo nosotros mismos, pero íbamos a encontrarnos con más dificultades que ventajas por lo que se optó por utilizar Java3D[4] que incluía un motor para la detección de colisiones.

- La parte gráfica o entorno virtual con *OpenSceneGraph* (**OSG**)[5]. Nacho se dedicó a intentar compilar los códigos fuente de los ejemplos del OSG con el **Visual Studio**[6]. Se encontraron muchos problemas ya que la configuración de la herramienta era más complicada de lo que en principio parecía. Visual Studio se utilizó para desarrollar el entorno 3D.

- La **comunicación** entre la plataforma de agentes y el entorno virtual. Pablo se encargó de esta tarea y consideró varias posibilidades:

    - Hacer uso de **JNI** (*Java Native Interface*), mecanismo que permite ejecutar código nativo (como pueden ser funciones C y C++) desde Java. Posee una interfaz bidireccional que permite a las aplicaciones Java llamar a código nativo y viceversa.

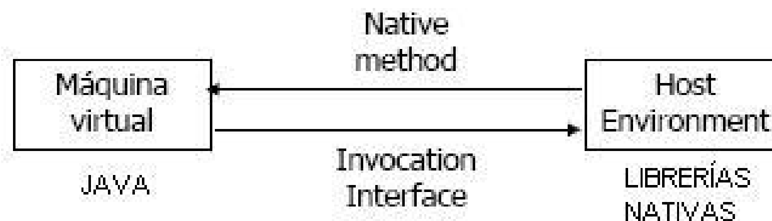


Figura 2.1 Interfaz bidireccional de JNI

    - Servirse de **CORBA** (*Common Object Request Broker Architecture),* un estándar que establece una plataforma de desarrollo de sistemas distribuidos.

[4] Véase Anexo I.
[5] Véase Anexo VI.
[6] Véase Anexo VII.

- Comunicar ambas entidades mediante **sockets**. Los sockets permiten implementar una arquitectura cliente-servidor. La idea es que una plataforma actué como programa cliente y la otra como programa servidor, y viceversa.

  Esta fue la opción escogida finalmente.

El reparto inicial se puede ver en la figura 2.2:

Figura 2.2 Reparto inicial

16 noviembre – 31 diciembre. Debido a problemas a la hora de conseguir la comunicación por sockets entre la plataforma de agentes (en Java), y el entorno 3D (en C++), Nacho pasó a prestar ayuda a Pablo para intentar solucionar cuanto antes este riesgo tecnológico. Por otro lado, Javier también se encontró con dificultades a la hora de implementar la detección de colisiones ya que no encontraba ejemplos parecidos a lo que nosotros pretendíamos realizar.

Figura 2.3 Segundo reparto

1 enero – 31 enero. Una vez resuelto el riesgo tecnológico de la comunicación, Nacho siguió investigando con OSG y Visual Studio, realizando pequeños prototipos que a la larga se convertirían en el código final. Pablo estuvo documentando toda la parte de la comunicación y Javier siguió investigando, realizando pequeños prototipos que ayudaran a familiarizarse con la herramienta.

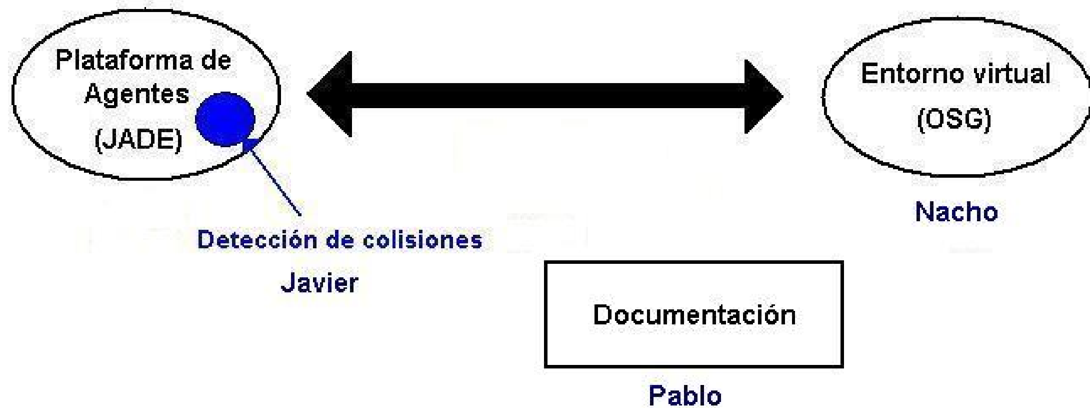


Figura 2.4 Tercer reparto

1 febrero – 29 febrero. La investigación empieza a dar sus frutos. Se consigue integrar la parte de la comunicación con el entorno virtual, consiguiendo enviar coordenadas (x, y, z) desde un servidor Java a un cliente C++ y que este las pinte en el lugar adecuado. También se encuentran unos modelos 3D de un gato y un ratón que cumplen los requisitos que queremos. Se consigue realizar la detección de colisiones entre un cono (*focus*) y una esfera (*nimbus*), y se implementa la función que se encarga de calcular la claridad de percepción[7] y que será de gran importancia en la toma de decisiones.

1 marzo – 15 marzo. Se entra de lleno a resolver el último riesgo tecnológico que queda: la plataforma de agentes con JADE. Se realiza un agente de comunicación que es el que envía las coordenadas a la parte C++ a través de socket. Se añade un agente que es el que envía las coordenadas al agente de comunicación. Finalmente se implementan dos

[7] Véase Apartado 2.4.

agentes que generan coordenadas diferentes. Estas son enviadas al agente de comunicación que envía las coordenadas a través del socket al cliente C++ que pinta los dos agentes en las coordenadas. Se realiza un protocolo de intercambio de mensajes ACL entre agentes, y de intercambio de mensajes entre el agente de comunicación y el cliente C++.

Figura 2.5 Cuarto reparto

16 marzo- 15 abril. El proyecto se centra más en la parte de detección de colisiones. Se crean nuevos agentes para los comportamientos de los gatos y los ratones y para la detección de colisiones. Se incluyen servicios de páginas amarillas en la plataforma de agentes para aprovechar el uso de esta tecnología .Se investiga una forma de evitar plasmar mensajes de salida por consola y, en su lugar, volcarlos a un fichero mediante Log4J. Mientras tanto, se estructura el código del entorno virtual, diferenciando la parte encargada de la comunicación (para la recepción de coordenadas de los agentes) y la parte gráfica. Se comienza a investigar cómo escalar modelos gráficos para adaptarlos a nuestras necesidades.

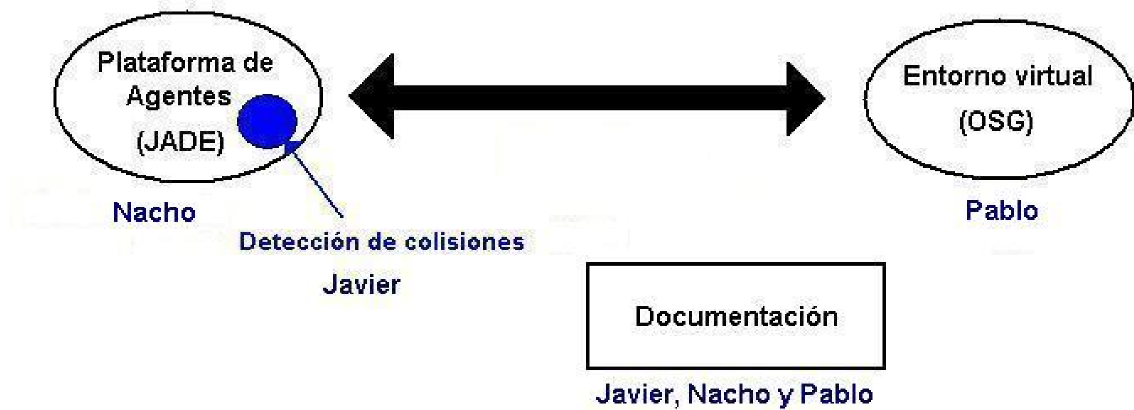


Figura 2.6 Quinto reparto

16 abril – 15 mayo. Comienza la implementación de la toma de decisiones. Inclusión de agentes para este cometido. Ligera modificación de los criterios en la toma de decisiones y de factores que influyen en la claridad de percepción. Se consigue que en la GUI de Java3d, se pinten las esferas y los conos a la par que se van generando nuevas coordenadas desde los agentes. Problemas con la detección de colisiones. Se detecta que la orientación del cono de visión en Java 3D, es opuesta a la implementación del cono de referencia para construir las cajas que han de detectar las colisiones. Se crean nuevos agentes que controlan diferentes trayectorias a seguir por los agentes gato y ratón. Se consiguen escalar los modelos gráficos. Se sigue documentando.

Figura 2.7 Sexto reparto

16 mayo-15 junio. Continúan los problemas para integrar la toma de decisiones y la detección de colisiones. Finalmente se observa que el método elegido para detectar las colisiones es el originario del problema por lo que se procede a cambiarlo. Se prueba la toma de decisiones por separado observando que los resultados son satisfactorios. Se reduce el número de polígonos de los modelos 3D para liberar la carga de la aplicación. Se comienza a elaborar la memoria a partir de los diversos documentos que se han ido elaborando a lo largo del transcurso del proyecto.

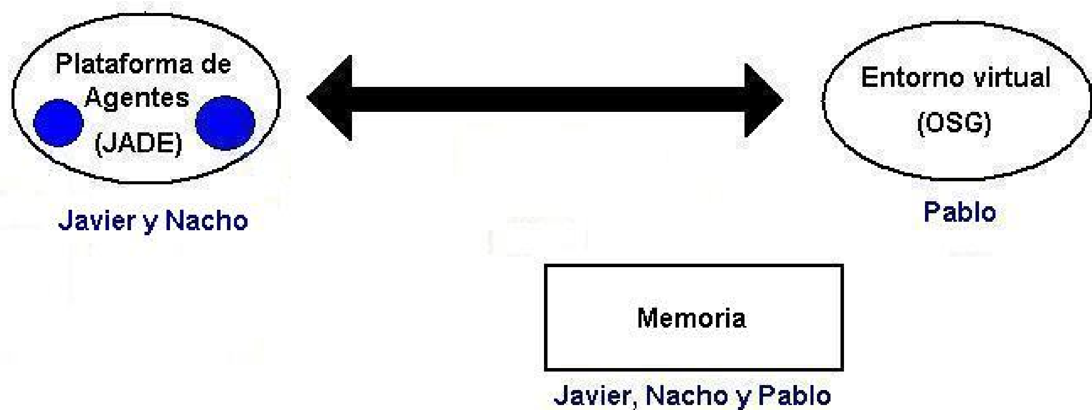


Figura 2.8 Séptimo reparto

16 junio-4 julio. Se termina de implementar la GUI en OSG. Se termina de implementar la detección de colisiones optando por utilizar Java 3D para una detección de colisiones simple, y cálculos geométricos para resolver los errores de detección provocados por la propia herramienta. Se integran ambos códigos. Estos a su vez se integran con la toma de decisiones en la plataforma de agentes obteniéndose resultados satisfactorios. Se termina de elaborar la memoria.

Finalmente, añadir que se ha hecho uso de la herramienta **Google Groups** y **Gmail** para mantener comunicación periódica entre los integrantes del proyecto y los profesores, con casi 300 mensajes de correo, más de 60 hilos de discusión y más de 30 documentos.

Se han mantenido reuniones semanales con el director del proyecto con el objetivo de llevar un seguimiento periódico de la evolución del mismo, así como resolver cualquier tipo de problema originado, fundamentalmente, por la cantidad de riesgos tecnológicos con los que se enfrentaba nuestro proyecto.

Se ha utilizado un repositorio CVS de código (**Berlios**), que nos ha permitido mantener un control de versiones, así como aprovechas las ventajas que ofrece este servicio una vez las dimensiones del código generado empezaron a incrementarse.

Capítulo 3. Riesgos

3.1 Riesgos iniciales del proyecto

3.1.1 Identificación de riesgos

ID	Riesgo	Tipo de riesgo	Descripción
GEN1	Conocimiento previo insuficiente.	Personal / Conocido	Ninguno de los integrantes del proyecto aúna los conocimientos necesarios en las diferentes tecnologías que se van a utilizar.
GEN2	Tiempo de estudio insuficiente	Personal/ Conocido	No hay tiempo suficiente para estudiar documentarse lo necesario como para realizar un buen diseño e implementación de la aplicación
GEN3	Pérdida de personal	Personal/ Impredecible	Alguno de los integrantes del proyecto lo abandona.
GEN4	Reestructuración de personal	Organización/ Impredecible	Problemas de organización y/o necesidades del proyecto imponen que alguno de los integrantes abandone su parte para reforzar otra.
GEN5	Falta de coordinación en diseño general	Organización/ Predecible	Las distintas partes que interactúan entre si no lo hacen correctamente debido a una interfaz errónea o a una mala utilización de ésta.
GEN6	Falta de documentación	Organización/ Predecible	Alguno de los integrantes que se ha incorporado a otra parte del proyecto, carece de la documentación necesaria de lo que se estaba haciendo hasta ese momento para poder continuar el trabajo.
GEN7	Cambio de los requisitos	Requisitos/ Impredecible	Cambios en los requisitos suponen volver a rehacer el diseño de las distintas partes en que se compone la aplicación.

GEN8	Diseño incorrecto	Requisitos/ Predecible	Fallo en el diseño supone que no hacemos lo que nos habíamos comprometido a hacer.
GEN9	Fallo en la planificación del tiempo	Estimación/ Conocido	Habíamos subestimado el tiempo necesario para documentarse, realizar un buen diseño, llevar a la práctica y "encajar" las distintas partes de la aplicación en el proyecto.
GEN10	Software no disponible	Tecnológico/ Predecible	Para desarrollar la aplicación se necesita cierto software con el que no se va a poder contar.
GEN11	Pérdida de trabajo	Tecnológico/ Impredecible	Durante el desarrollo de la aplicación se sufre una pérdida, total o parcial, por cualquier motivo del código fuente, planificación o cualquier otra documentación del proyecto.
GEN12	Tiempo de solución errores	Estimación/ Predecible	El tiempo necesitado para arreglar los errores y riesgos es superior a lo que se había pensado en un principio.
GEN13	No se implementa la plataforma de agentes	Tecnológico/ Conocido	Por desconocimiento de la herramienta a utilizar o motivos inherentes a su uso no se consigue implementar la plataforma de agentes en JADE.
GEN14	No se consigue realizar la detección de colisiones	Tecnológico/ Conocido	Por desconocimiento de la herramienta a utilizar o motivos inherentes a su uso no se consigue implementar la detección de colisiones con Java 3D.
GEN15	No se puede realizar la comunicación mediante sockets	Tecnológico/ Conocido	Por desconocimiento del mecanismo de comunicación por sockets entre parte Java y C++ no se consigue realizar esta parte.

GEN16	No se puede realizar la GUI con OSG	Tecnológico/ Conocido	Por desconocimiento de la herramienta *OpenSceneGraph* no se consigue realizar la GUI.
SOC1	Representación distinta de los datos en ambos extremos de la comunicación	Tecnológico/ Conocido	Dificultad en el tratamiento e interpretación de los datos por diferencias de representación entre Java y C.
SOC2	Tiempo de comunicación entre plataforma de agentes y OSG inaceptable	Tecnológico/ Predecible	Excesivo tiempo transcurrido en el envío y recepción de datos, o tiempo de espera indeterminado en uno de los extremos de la comunicación.
SOC3	Seguridad e integridad de los datos.	Tecnológico/ Predecible	Envío y recepción de datos de manera íntegra y sin pérdida parcial o total de ellos.
SOC4	Congestión en el tráfico de los datos	Tecnológico/ Impredecible	Bloqueo indefinido de la comunicación o saturación de recursos
SOC5	Mantenimiento difícil	Tecnológico/ Predecible	Garantía de correcto funcionamiento cuando el proyecto avance.
SOC6	Accesibilidad limitada	Tecnológico/ Predecible	Funcionamiento no deseado si se intercambia información con muchos clientes.
AGE1	Uso indiscriminado de agentes	Tecnológico/ Predecible	Sobrecarga del sistema con una gran cantidad de agentes comunicándose entre sí.
AGE2	No explotar el esquema multiagente	Tecnológico/ Predecible	No conseguir que cada agente tenga una funcionalidad específica y recaigan sobre él excesivas responsabilidades
AGE3	Mecanismo de paso de mensajes ACL	Tecnológico/ Predecible	Incomprensión y mal uso de la comunicación entre agentes.
AGE4	Inanición en el paso de mensajes	Tecnológico/ Predecible	Tiempo empleado en la comunicación entre agentes poco razonable.

AGE5	Inadecuada distribución de los comportamientos	Tecnológico/ Predecible	Posible complicación en la implementación por una mala distribución de comportamientos.
AGE6	Mal uso de las páginas amarillas	Tecnológico/ Conocido	No aprovechar la utilidad de los servicios que ofrece cada agente en las páginas amarillas
OSG1	Excesivo volumen de los programas en tiempo de ejecución	Tecnológico/ Impredecible	Los modelos y el código que los maneja puede pesar demasiado mientras se ejecuta la aplicación
OSG2	Falta de los modelos necesarios para la presentación gráfica final	Tecnológico/ Predecible	Se requieren modelos gráficos específicos que sean de código libre y de un formato legible para OSG
OSG3	Modelos excesivamente grandes	Tecnológico/ Impredecible	Los modelos no son válidos por sus dimensiones, ya que aparecen incoherencias de presentación
OSG4	Necesidad de conversión de formato de modelos 3D	Tecnológico/ Predecible	Los modelos obtenidos de la Web tienen un formato incompatible con OSG
OSG5	Necesidad de escalado de modelos 3D	Tecnológico/ Impredecible	Los modelos tienen demasiados polígonos, es decir, son demasiado precisos.

Tabla 3.1 Identificación de riesgos

3.1.2 Análisis de riesgos

ID	**Riesgo**	**Tipo de riesgo**	**Probabilidad**	**Impacto**	**Prioridad**
GEN1	Conocimiento previo insuficiente.	Personal / Conocido	Alta	Catastróficos	2
GEN2	Tiempo de estudio insuficiente	Personal/ Conocido	Moderada	Catastróficos	3
GEN3	Pérdida de personal	Personal/ Impredecible	Baja	Serios	4
GEN4	Reestructuración de personal	Organización/ Impredecible	Alta	Serios	2

GEN5	Falta de coordinación en diseño general	Organización/ Predecible	Alta	Serios	1
GEN6	Falta de documentación	Organización/ Predecible	Baja	Serios	1
GEN7	Cambio de los requisitos	Requisitos/ Impredecible	Moderada	Serios	2
GEN8	Diseño incorrecto	Requisitos/ Predecible	Moderada	Catastróficos	1
GEN9	Fallo en la planificación del tiempo	Estimación/ Conocido	Alta	Serios	2
GEN10	Software no disponible	Tecnológico/ Predecible	Baja	Moderado	5
GEN11	Pérdida de trabajo	Tecnológico/ Impredecible	Media	Catastrófico	1
GEN12	Tiempo de solución errores	Estimación/ Predecible	Muy alta	Catastrófico	1
GEN13	No se implementa la plataforma de agentes	Tecnológico/ Conocido	Moderada	Catastrófico	1
GEN14	No se consigue realizar la detección de colisiones	Tecnológico/ Conocido	Moderada	Catastrófico	1
GEN15	No se puede realizar la comunicación mediante sockets	Tecnológico/ Conocido	Moderada	Catastrófico	1
GEN16	No se puede realizar la GUI con OSG	Tecnológico/ Conocido	Moderada	Catastrófico	1
SOC1	Representación distinta de los datos en ambos extremos de la comunicación	Tecnológico/ Conocido	Muy alta	Serio	1

SOC2	Tiempo de comunicación entre plataforma de agentes y OSG inaceptable	Tecnológico/ Predecible	Medio	Moderado	4
SOC3	Seguridad e integridad de los datos.	Tecnológico/ Predecible	Alta	Catastrófico	1
SOC4	Congestión en el tráfico de los datos	Tecnológico/ Impredecible	Baja	Serio	3
SOC5	Mantenimiento difícil	Tecnológico/ Predecible	Media	Moderado	4
SOC6	Accesibilidad limitada	Tecnológico/ Predecible	Alta	Moderado	4
AGE1	Uso indiscriminado de agentes	Tecnológico/ Predecible	Media	Moderado	4
AGE2	No explotar el esquema multiagente	Tecnológico/ Predecible	Media	Serio	3
AGE3	Mecanismo de paso de mensajes ACL	Tecnológico/ Predecible	Alta	Serio	2
AGE4	Inanición en el paso de mensajes	Tecnológico/ Predecible	Media	Moderado	5
AGE5	Inadecuada distribución de los comportamientos	Tecnológico/ Predecible	Media	Serio	3
AGE6	Mal uso de las páginas amarillas	Tecnológico/ Conocido	Media	Moderado	4
OSG1	Excesivo volumen de los programas en tiempo de ejecución	Tecnológico/ Impredecible	Media	Moderado	4
OSG2	Falta de los modelos necesarios para la presentación gráfica final	Tecnológico/ Predecible	Medio	Moderado	3

OSG3	Modelos excesivamente grandes	Tecnológico/ Impredecible	Alta	Moderado	4
OSG4	Necesidad de conversión de formato de modelos 3D	Tecnológico/ Predecible	Alta	Moderado	4
OSG5	Necesidad de escalado de modelos 3D	Tecnológico/ Impredecible	Media	Moderado	4

Tabla 3.2 Análisis de riesgos

3.1.3 Planificación de riesgos

ID	Riesgo	Estrategia
GEN1	Conocimiento previo insuficiente.	*Evitar/Minimizar:* Que se encargue de cada parte el que más familiarizado esté con el tipo de labor a realizar.
		Gestión/Plan de contingencia/Acción: Documentarse, realizar los prototipos necesarios y generar documentación de todo lo realizado.
GEN2	Tiempo de estudio insuficiente	*Evitar/Minimizar:* Dedicar el máximo "tiempo libre" posible a la investigación y estudio de las técnicas necesarias para desarrollar la aplicación.
		Gestión/Plan de contingencia/Acción: Utilizar tiempo dedicado a otros fines y pedir ayuda a gente experimentada y con conocimientos en las diferentes técnicas.
GEN3	Pérdida de personal	*Evitar/Minimizar:* Compromiso por parte de los integrantes del proyecto.
		Gestión/Plan de contingencia/Acción: Generar una buena documentación que facilite la incorporación de nuevos integrantes al grupo en el menor tiempo posible.
GEN4	Reestructuración de personal	*Evitar/Minimizar:* Repartir las tareas intentando dar prioridad a las preferencias de cada integrante del proyecto.
		Gestión/Plan de contingencia/Acción: Generar una buena documentación que facilite la incorporación de nuevos integrantes al grupo en el menor tiempo posible.

GEN5	Falta de coordinación en diseño general	*Evitar/Minimizar:* Al iniciar el proyecto, todos los componentes deben especificar unas interfaces de uso de cada parte del proyecto, para una buena integración de las mismas.
		Gestión/Plan de contingencia/Acción: Reunión entre los integrantes para coordinar las acciones pertinentes para rectificar errores en los diseños y modificar estos en la medida que sea necesario.
GEN6	Falta de documentación	*Evitar/Minimizar:* Generar una buena documentación inicial y durante las múltiples fases de desarrollo de las distintas partes.
		Gestión/Plan de contingencia/Acción: Que los antiguos integrantes del grupo se pongan en contacto con los nuevos, para poder explicarles su trabajo anterior, y poder generar una documentación.
GEN7	Cambio de los requisitos	*Evitar/Minimizar:* Intentar hacer un diseño flexible que a la hora de realizar cambios pueda soportarlos fácilmente.
		Gestión/Plan de contingencia/Acción: Intentar modificar el diseño lo mínimo posible para satisfacer los nuevos requisitos.
GEN8	Diseño incorrecto	*Evitar/Minimizar:* Dedicar el tiempo necesario al estudio de las distintas disciplinas, que permita realizar un buen diseño.
		Gestión/Plan de contingencia/Acción: Pedir ayuda a gente con conocimientos específicos en las diferentes partes.
GEN9	Fallo en la planificación del tiempo	*Evitar/Minimizar:* Conviene estimar correctamente la complejidad de las distintas partes, para aproximar el tiempo necesario para su implementación.
		Gestión/Plan de contingencia/Acción: Reducir los requisitos, lo cual obligaría a cambiar el diseño si este no fuera lo suficientemente flexible como para soportar dichos cambios, para así simplificar la implementación y que nos diera tiempo.
GEN10	Software no disponible	*Evitar/Minimizar:* Intentar utilizar o adaptarse a las aplicaciones existentes en el laboratorio.
		Gestión/Plan de contingencia/Acción: Pedir que nos instalen el software necesario, y si no fuera posible, instalar dicho software en portátiles particulares.

GEN11	Pérdida de trabajo	*Evitar/Minimizar:* Tener una copia del trabajo en otro dispositivo de almacenamiento o servidor. Utilizar un repositorio para actualizaciones, etc (disponibles 24h, más posibilidad de backup)
		Gestión/Plan de contingencia/Acción: Continuar con lo que se tenga (último backup).
GEN12	Tiempo de solución errores	*Evitar/Minimizar:* Realizar pruebas completas y correctas. Re-planificar y dar mas tiempo a las pruebas.
		Gestión/Plan de contingencia/Acción: Intentar producir opciones algorítmicas menos costosas, intentando evitar cambiar la especificación mientras sea posible. Si no es viable, será necesario cambiar la especificación del problema.
GEN13	No se implementa la plataforma de agentes	*Evitar/Minimizar:* Buscar tutoriales, ejemplos, realizar prototipos.
		Gestión/Plan de contingencia/Acción: Estudiar la posibilidad de reducir los requisitos del proyecto.
GEN14	No se consigue realizar la detección de colisiones	*Evitar/Minimizar:* Buscar tutoriales, ejemplos, realizar prototipos.
		Gestión/Plan de contingencia/Acción: Estudiar la posibilidad de reducir los requisitos del proyecto.
GEN15	No se puede realizar la comunicación mediante sockets	*Evitar/Minimizar:* Buscar tutoriales, ejemplos, realizar prototipos.
		Gestión/Plan de contingencia/Acción: Realizar la comunicación mediante JNI, CORBA, o estudiar la posibilidad de reducir los requisitos.
GEN16	No se puede realizar la GUI con OSG	*Evitar/Minimizar:* Buscar tutoriales, ejemplos, realizar prototipos.
		Gestión/Plan de contingencia/Acción: Estudiar la posibilidad de reducir los requisitos del proyecto o realizar la GUI con otra herramienta distinta a OSG y ya conocida.
SOC1	Representación distinta de los datos en ambos extremos de la comunicación	*Evitar/Minimizar:* Buscar tutoriales, ejemplos, realizar prototipos.
		Gestión/Plan de contingencia/Acción: Realizar envío y recepción con tipos de datos simples, con el mayor parecido posible entre Java y C, que faciliten la solución de este riesgo.

SOC2	Tiempo de comunicación entre plataforma de agentes y OSG inaceptable	*Evitar/Minimizar:* Simplificar el código.
		Gestión/Plan de contingencia/Acción: Reducción de código o envío y recepción de datos en fragmentos de menor tamaño.
SOC3	Seguridad e integridad de los datos.	*Evitar/Minimizar:* Establecer un protocolo de seguridad.
		Gestión/Plan de contingencia/Acción: Gestionar errores con manejo de excepciones y ajuste a un patrón determinado para que los datos se transmitan íntegros o no se transmitan.
SOC4	Congestión en el tráfico de los datos	*Evitar/Minimizar:* Evitar saturación de recursos del computador.
		Gestión/Plan de contingencia/Acción: Reducir la complejidad de la aplicación.
SOC5	Mantenimiento difícil	*Evitar/Minimizar:* Comprobación continua.
		Gestión/Plan de contingencia/Acción: Realizar pruebas cada vez que se introduzca nueva funcionalidad en el proyecto.
SOC6	Accesibilidad limitada	*Evitar/Minimizar:* Impedir comunicación con múltiples clientes,
		Gestión/Plan de contingencia/Acción: Realizar el intercambio de información desde un solo cliente en la plataforma de agentes que intercambiará información con los agentes correspondientes.
AGE1	Uso indiscriminado de agentes	*Evitar/Minimizar:* Consultar al profesor y buscar proyectos similares.
		Gestión/Plan de contingencia/Acción: Revisión de lo que se lleva implementado hasta el momento.
AGE2	No explotar el esquema multiagente	*Evitar/Minimizar:* Consultar al profesor y buscar potencialidades de los agentes.
		Gestión/Plan de contingencia/Acción: Introducir nuevas funcionalidades y verificar que su incorporación aporta una mejora al proyecto.
AGE3	Mecanismo de paso de mensajes ACL	*Evitar/Minimizar:* Consultar tutoriales y ejemplos.
		Gestión/Plan de contingencia/Acción: Hacer prototipos e implementar a partir de ellos.
AGE4	Inanición en el paso de mensajes	*Evitar/Minimizar:* Comprobar las ejecuciones.
		Gestión/Plan de contingencia/Acción: Depurar si es necesario.

AGE5	Inadecuada distribución de los comportamientos	*Evitar/Minimizar:* Revisar si hacen lo que tienen que hacer.
		Gestión/Plan de contingencia/Acción: En caso de ser necesario rectificar modificando/añadiendo/eliminando comportamientos
AGE6	Mal uso de las páginas amarillas	*Evitar/Minimizar:* Consultar si es conveniente incluir este servicio en el sistema.
		Gestión/Plan de contingencia/Acción: Buscar ejemplos e implementar los servicios necesarios.
OSG1	Excesivo volumen de los programas en tiempo de ejecución	*Evitar/Minimizar:* Simplificar la aplicación
		Gestión/Plan de contingencia/Acción: Reducir volumen de los modelos, probar con otros, ejecutar la aplicación con cliente/servidor alojados en lugares diferentes.
OSG2	Falta de los modelos necesarios para la presentación gráfica final	*Evitar/Minimizar:* Buscar dichos modelos.
		Gestión/Plan de contingencia/Acción: Obtener dichos modelos de la Web.
OSG3	Modelos excesivamente grandes	*Evitar/Minimizar:* Probar modelos de menor tamaño.
		Gestión/Plan de contingencia/Acción: Obtener otros modelos de la Web o reducir los encontrados.
OSG4	Necesidad de conversión de formato de modelos 3D	*Evitar/Minimizar:* Convertirlos.
		Gestión/Plan de contingencia/Acción: Obtener métodos/programas de conversión.
OSG5	Necesidad de escalado de modelos 3D	*Evitar/Minimizar:* Escalarlos
		Gestión/Plan de contingencia/Acción: Obtener métodos/programas de escalado.

Tabla 3.3 Planificación de riesgos

Leyenda:

Clasificación de riesgos

- GEN – Riesgos Generales
- SOC – Comunicación por Sockets.
- AGE – Plataforma de Agentes.
- OSG – Parte gráfica.

Rangos de los campos de las tablas

- **Tipo:** Personal, Organización, Requisitos, Estimación, Tecnológico / Predecible, Impredecible, Conocido
- **Probabilidad:** Muy Alta, Alta, Moderada, Baja, Muy Baja
- **Impacto:** Catastrófico, Serio, Moderado, Tolerable, Insignificante
- **Prioridad**: 1 a 5, de más prioritario a menos.

3.2 Riesgos finales del proyecto

Tras terminar la aplicación y revisar el documento inicial de riesgos, finalmente los riesgos que permanecen son los siguientes:

3.2.1 Identificación de riesgos

ID	Riesgo	Tipo de riesgo	Descripción
SOC4	Congestión en el tráfico de los datos	Tecnológico/ Impredecible	Bloqueo indefinido de la comunicación o saturación de recursos

Tabla 3.4 Identificación de riesgos

3.2.2 Análisis de riesgos

ID	Riesgo	Tipo de riesgo	Probabilidad	Impacto	Prioridad
SOC4	Congestión en el tráfico de los datos	Tecnológico/ Impredecible	Baja	Serio	3

Tabla 3.5 Análisis de riesgos

3.2.3 Planificación de riesgos

ID	Riesgo	Estrategia
SOC4	Congestión en el tráfico de los datos	*Evitar/Minimizar:* Evitar saturación de recursos del computador.
		Gestión/Plan de contingencia/Acción: Reducir la complejidad de la aplicación.

Tabla 3.6 Planificación de riesgos

Leyenda:

<u>Clasificación de riesgos</u>

- GEN – Riesgos Generales
- SOC – Comunicación por Sockets.
- AGE – Plataforma de Agentes.
- OSG – Parte gráfica.

<u>Rangos de los campos de las tablas</u>

- **Tipo:** Personal, Organización, Requisitos, Estimación, Tecnológico / Predecible, Impredecible, Conocido
- **Probabilidad:** Muy Alta, Alta, Moderada, Baja, Muy Baja
- **Impacto:** Catastrófico, Serio, Moderado, Tolerable, Insignificante

- **Prioridad**: 1 a 5, de más prioritario a menos.

Capítulo 4. Plataforma de agentes

La herramienta usada para desarrollar la plataforma de agentes ha sido JADE, que está escrita completamente en Java. A continuación se detalla su uso en el sistema:

4.1 Agentes

Hay cuatro tipos de agentes, cada uno con unas características y comportamientos propios. Los gatos y ratones serán los agentes que se muevan por la escena, pero también hay otros agentes "invisibles" para el usuario y cuya importancia es máxima: el agente encargado del envío de coordenadas por el socket y el agente encargado de la detección de colisiones.

4.2 Comportamientos

Lo que caracteriza a un agente es el comportamiento que lleva asociado. Por esa razón, una vez creado el agente se le dota de uno o varios comportamientos específicos, que le indicarán las acciones a realizar.

Es posible que dependiendo del tipo de agente se desee que tenga un comportamiento cíclico o repetitivo o, por el contrario, que sea un comportamiento que se ejecute una única vez y termine. En este caso es recomendable matar al agente para liberar los recursos o bien bloquearlo si interesa seguir teniéndolo en el sistema. En esta aplicación se han usado comportamientos cíclicos para los agentes gato, ratón, detección de colisiones y agente de comunicaciones, éste último también tiene dos comportamientos que se ejecutan una sola vez.

A continuación se describen los comportamientos utilizados en cada agente:

El agente encargado de las **comunicaciones** tiene tres comportamientos.

El primero se encarga de activar a los demás agentes (gato, ratón, detección de colisiones). Este comportamiento es importante, ya que es posible que se lance la plataforma de agentes y el entorno gráfico todavía no se haya ejecutado (recordar que ambas aplicaciones se lanzan por separado). Así se evita que los agentes comiencen a

realizar sus funciones antes de que el entorno 3D esté listo. Una vez se detecta petición por el socket desde la parte gráfica el comportamiento de activación de agentes hace que todos se activen y no se vuelva a ejecutar dicho comportamiento.

El segundo comportamiento es el más importante para este tipo de agente. Es de ejecución única y se encarga de recibir mensajes ACL de los agentes gato y ratón con toda la información necesaria para enviar al entorno 3D y que se dibujen en las posiciones correctas. La cadena enviada tiene el siguiente formato:

*:numGatos:numRatones:agenteAMover:orientación(x,y,z)**

Toda esta información es parseada al otro lado del socket y tratada debidamente para su correcta representación gráfica. Si no se recibe mensaje alguno el agente se bloquea y queda a la espera de un mensaje entrante.

El último comportamiento es el que incluye a los otros dos comportamientos descritos, y es de tipo periódico.

Los agentes **gato** y **ratón** tienen el mismo tipo de comportamiento: seguir una trayectoria generada al azar cuando se crea el agente. Además de generar nuevas coordenadas en cada paso de ejecución también se encargan de mandar esa información generada al agente encargado de las comunicaciones con el entorno gráfico y al agente que controla la detección de colisiones.

El agente **detección de colisiones** tiene un comportamiento cíclico que comprueba, a intervalos de tiempo fijados a priori, si se han producido colisiones entre los conos de visión de los agentes y las esferas que rodean a los mismos. En caso de colisión manda un mensaje de tipo ACL a los agentes involucrados para que entre en juego la toma de decisiones y actúen en consecuencia.

Ciclo de ejecución de un agente:

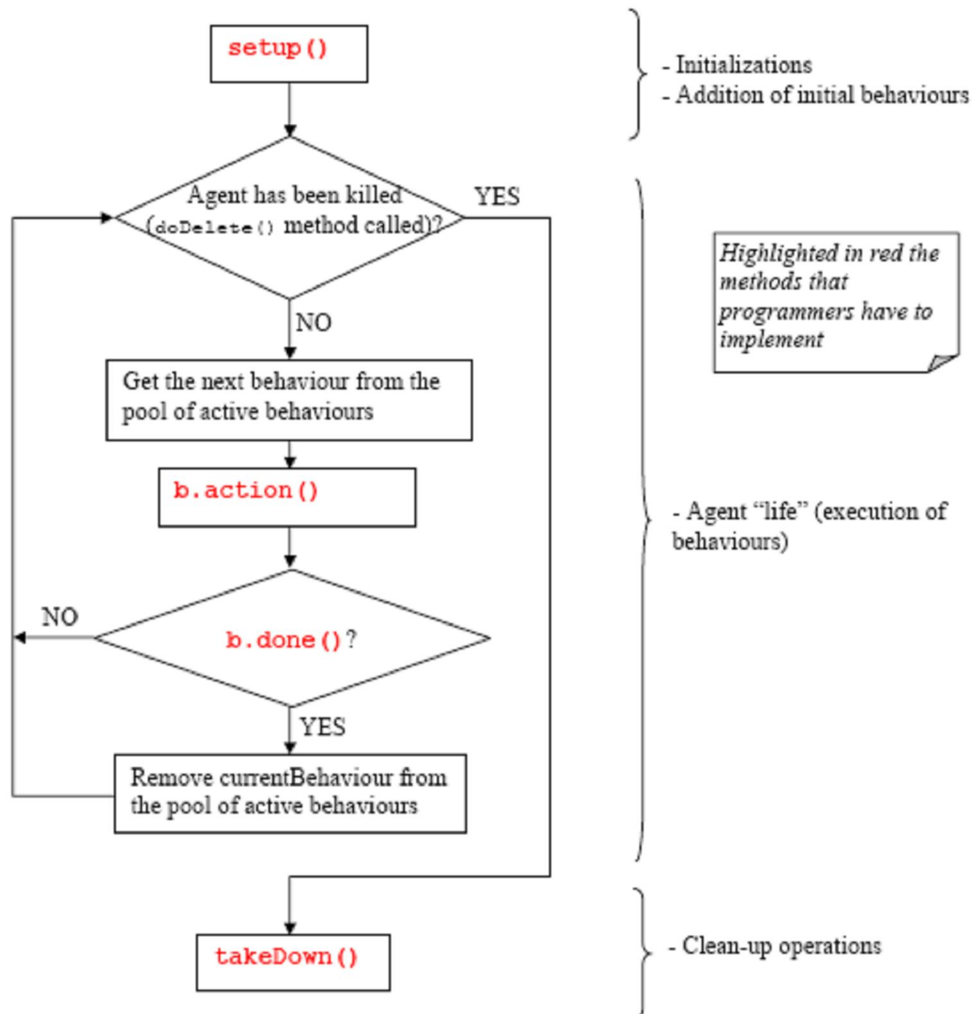


Figura 4.1. Camino del hilo de ejecución de un agente

4.3 Mensajes ACL

Uno de los mecanismos que permite a los agentes intercambiar información es el sistema de mensajería ACL. El paradigma de comunicación adoptado es el paso de mensajes asíncrono. Cada agente tiene un buzón de mensajes (el cual funciona como una cola) donde la plataforma JADE coloca los mensajes enviados por otros agentes. Cuando llega un mensaje al buzón se le notifica al agente.

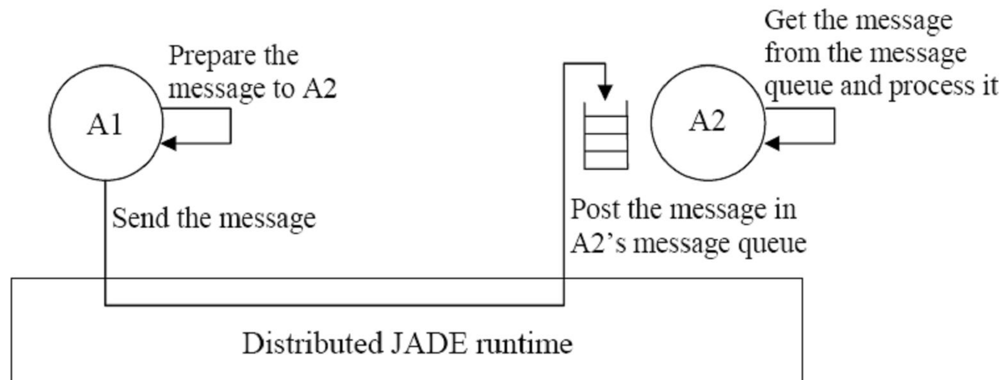


Figura 4.2. Paradigma de paso de mensaje asíncrono de JADE

4.4 Coordenadas y orientación

Un aspecto fundamental con el que deben trabajar las trayectorias de los agentes son las coordenadas nuevas que se generan cada vez que se mueve el objeto y la orientación que tiene en cada momento. Esta información es crucial, ya que será lo que se mande al agente de comunicaciones para que se lo transmita al entorno gráfico, donde se representará visualmente al usuario. Las coordenadas generadas variarán en los ejes X, Y permaneciendo la coordenada Z siempre a cero. La orientación puede tomar los ocho puntos cardinales: *N, S, E, O, NE, NO, SE, SO.*

4.5 Trayectorias de los agentes

Cuando se crea el comportamiento de un agente gato o ratón se selecciona una trayectoria aleatoria entre todas las que hay disponibles. Para ello se implementó un patrón strategy que permite crear las distintas trayectorias que van a seguir los agentes gato y ratón. De esta forma se facilita en gran medida la adición o supresión de trayectorias al sistema. En todas ellas se permite variar la distancia que recorre el agente así como cuánto avanza cada vez que se mueve. Actualmente, hay cinco trayectorias diferentes:

- Diamante
- Octogonal
- Cuadrada
- Triangular
- Zigzag

4.6 Páginas Amarillas

JADE ofrece un servicio muy útil denominado Páginas Amarillas, donde se pueden registrar los agentes que ofrecen un servicio. La forma de funcionar es muy simple: cuando se crea un agente se hace que se registre en las Páginas Amarillas ofreciendo un tipo de servicio. Cuando un agente quiere interaccionar con otros agentes no tiene más que buscar en las Páginas Amarillas el servicio que ofrecen dichos agentes, pudiendo elegir con quién de ellos se comunica. En cuanto se desee matar al agente se procede a desregistrarlo de las Páginas Amarillas. De este modo, los agentes gato y ratón se registran ofreciendo un servicio de tipo "generación-coordenadas", ya que con sus comportamientos de trayectorias irán generando nuevas coordenadas de posición según se muevan por el entorno. Por otro lado, el agente que envía los datos por el socket se registra ofreciendo un servicio de tipo "envío-coordenadas", debido a que se encarga de mandar la información relevante de cada agente hacia el entorno gráfico de OSG. Por último, el agente encargado de detectar las colisiones se registra en las Páginas Amarillas ofreciendo un servicio de tipo "detección-colisiones", ya que se encarga de gestionar las colisiones que se produzcan entre los agentes.

Capítulo 5. Diseño del modelo de percepción

Como se ha mencionado con anterioridad, se pretendía dotar a los agentes de la propiedad de la percepción visual, de tal manera que sólo pudieran conocer la zona del entorno que en cada momento les fuera "visible". Por ello, introdujimos conceptos como *focus*, *nimbus* y claridad de percepción (Herrero y De Antonio, 2003) que han sido clave a la hora de desarrollar nuestro Modelo Espacial de Interacción (Benford y Fahlén, 1993; Benford y col., 1995).

El Modelo Espacial de Interacción fue el primer modelo de *consciencia* propuesto, en 1993 por Benford y Fahlén. El modelo espacial se diseñó de forma que permitiera obtener información del entorno que rodea al usuario en entornos colaborativos, a través de una serie de mecanismos como son el *focus*, *nimbus* o la *consciencia*, que fueron definidos para ese modelo y que permiten gobernar la interacción en un entorno virtual.

El término de *consciencia* es muy familiar dentro del campo del Trabajo Colaborativo (Computer Supported Cooperative Work, también conocido por las siglas CSCW).

Probablemente la mejor definición de *consciencia* en CSCW fue proporcionada por Dourish y Bellotti (1992). Ellos definieron el concepto el *consciencia* como "una comprensión de las actividades de los demás, las cuales proporcionan un contexto para tu propia actividad".

El *focus* determina la zona del espacio desde la cual un objeto recibe información, y el *nimbus* determina la zona del espacio en la cual un objeto proyecta su información.

El *nimbus*, como vemos en la figura 5.1, representa la zona del espacio en la que el agente y el objeto pueden ser percibidos por otro agente.

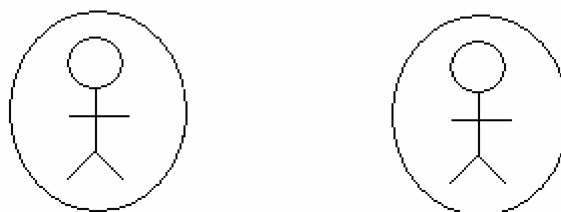


Figura 5.1. *Nimbus* alrededor de dos agentes

El *focus* o cono de visión, como vemos en la figura 5.2, lo representaremos mediante un cono que comprende la zona del espacio que el agente percibe.

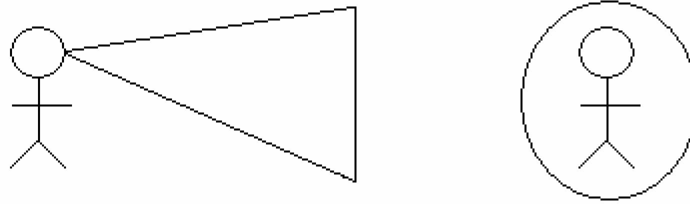


Figura 5.2. *Focus* y *nimbus*

Si el *focus* del agente entra en contacto con el *nimbus* de otro, el agente lo percibe. Esto queda reflejado en la figura 5.3.

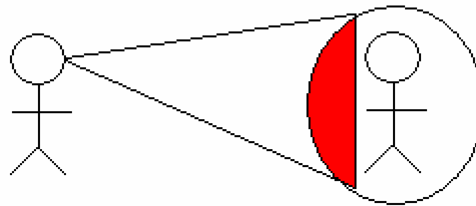


Figura 5.3. *Focus* intersecando con el *nimbus* de otro agente

Cuanto más cerca se encuentren los agentes, con más claridad percibirá el primero al segundo, es decir, mayor será la intersección entre el *focus* y el *nimbus*.

La claridad de percepción es una medida de la habilidad que un agente posee para percibir un objeto que se encuentra presente en su *focus* de percepción. Para ello se ha utilizado el concepto de *consciencia* tal y como fue introducido en sus orígenes por el Modelo Espacial de Interacción.

Así pues, si la percepción que se realiza es visual, una vez que el *nimbus* del objeto que se desea percibir interseca con el *focus* del agente que percibe el entorno, la percepción sensitiva del agente entra en juego y calcula la claridad de percepción que el agente tiene de ese objeto.

Matemáticamente, la claridad de percepción vendría dada por las funciones matemáticas:

$$\begin{aligned} &0.0 \leq d \leq d_1, \quad \mathrm{CP}(d) = \lambda d \\ &d_1 \leq d \leq d_2, \quad \mathrm{CP}(d) = \mathrm{CP}_{\max} \\ &d \geq d_2, \quad \mathrm{CP}(d) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(d-d_2)^2}{2\sigma^2}\right\} \end{aligned} \tag{5.1}$$

siendo:

d_1: La distancia mínima entre el objeto y el ojo para que dicho objeto pueda ser percibido.

d_2: La distancia máxima entre el objeto y el ojo para que dicho objeto pueda ser percibido con un elevado nivel de detalles.

Capítulo 6. Detección de colisiones

La detección de colisiones ha sido una parte clave del proyecto. Su finalidad era la de detectar las intersecciones entre los *focus* y *nimbus* de los agentes que se encontraban en el entorno.

No solo era importante saber si se había producido colisión o no, sino que había que averiguar con que parte del cono se había producido y cuanto había colisionado, ya que esta información iba a resultar vital a la hora de tomar una decisión o no tomarla, como se explicará mas adelante.

Para llevar a cabo la detección de colisiones se utilizó la herramienta Java 3D, la cual a priori nos iba a permitir facilitar esta tarea, gracias a la gran variedad de funciones y utilidades que componen su API[8].

La funcionalidad de Java 3D que íbamos a aprovechar en el proyecto fue la interacción.

6.1 Comportamientos: La base para la interacción

En Java 3D, la clase ***Behaviour*** (*comportamiento*) permite especificar interacciones con objetos visuales. Se trata de una clase abstracta, de la cual se debe heredar para poder incluir código que produzca modificaciones en el escenario gráfico.

El propósito de un objeto *Behaviour* dentro un escenario gráfico es la modificación del propio escenario gráfico o de los objetos contenidos en él, en respuesta a ciertos estímulos. Un *estímulo* en nuestro caso es la colisión de objetos pero puede ser también una pulsación de tecla, el movimiento del ratón, el paso del tiempo, o la combinación de varios de éstos estímulos. Los *cambios* que se producen como consecuencia de la ejecución del comportamiento incluyen la adicción o eliminación de objetos al escenario gráfico, cambios en sus atributos, o varias de estas acciones a la vez.

---

[8] http://download.java.net/media/java3d/javadoc/1.5.1/index.html

6.2 Escribir una clase *Behaviour*

Una clase de comportamiento personalizado implementa los métodos *initialize* (inicialización) y *processStimulus* de la clase abstracta *Behaviour*, además del constructor y otros métodos de utilidad. La mayoría de los comportamientos actúan sobre uno o varios objetos del escenario gráfico. El objeto(s) sobre el que actúa un comportamiento se conoce como el **objeto del cambio**.

El **método *initialize*** se invoca cuando el escenario gráfico que contiene la clase de comportamiento se activa. Este método de iniciación es responsable de seleccionar el evento de disparo inicial para el comportamiento y seleccionar la condición inicial de las variables de estado del comportamiento. El disparo se especifica como un objeto *WakeupCondition*, o una combinación de objetos *WakeupCondition*.

El **método *processStimulus*** se invoca cuando ocurre el evento de disparo especificado para el comportamiento. Este método es responsable de responder al evento. Como se pueden codificar muchos eventos en un sólo objeto *WakeupCondition*, aquí se incluye la descodificación del evento. El método *processStimulus* responde al estímulo, normalmente modificando el objeto de cambio, y, cuando es apropiado, reseteando el disparo.

A continuación se muestra una guía para escribir una clase *Behaviour* personalizada:

1. escribir constructor (al menos uno) y almacenar una referencia al objeto del cambio.
2. sobreescribir *public void initialize()* y especificar el criterio de disparo inicial
3. sobreescribir *public void processStimulus()*, decodificar la condición de disparo, actuar de acuerdo a la condición de disparo, y por último resetear el disparo si es apropiado

6.3 Región Programada

En un universo virtual con *muchos comportamientos*, se necesita una significante potencia de *cálculo para manejarlos*. Como tanto el renderizador como el comportamiento usan el mismo procesador, es posible que la potencia de cálculo que necesita el comportamiento degrade el rendimiento del renderizado. Java 3D permite al programador manejar este problema especificando un límite espacial para que el comportamiento tenga

lugar. Este límite se llama ***región programada***. Un comportamiento no está activo a menos que el volumen de activación de *ViewPlatform* (el observador) interseccione con una región programada del *Behaviour.* El uso de regiones programadas hace más eficiente a Java 3D en el manejo de universos virtuales con muchos comportamientos.

6.4 Riesgos de programación al crear clases *Behaviour*

Al crear una clase *Behaviour* personalizada, los dos errores más comunes son:

1. ***Olvidarse de seleccionar y resetear el disparo del comportamiento***. Obviamente, si no se selecciona el disparo inicial en el método *initialize(),* el comportamiento nunca será invocado. Un poco menos obvio es que el disparo debe seleccionarse de nuevo en el método *processStimulus()* si se desea un comportamiento repetido, como es nuestro caso.

2. ***No volver de los métodos de la clase Behaviour.*** Como estos dos métodos (*initialize* y *processStimulus)* son llamados por el sistema Java 3D, deben volver para permitir que continúe el renderizado.

6.5 Riesgos de programación al usar objetos *Behaviour*

La siguiente lista muestra una guía con los pasos para usar un objeto *Behaviour*.

1. preparar el escenario gráfico (añadiendo un *TransformGroup* u otros objetos necesarios)

2. insertar el objeto *Behaviour* en el escenario gráfico, referenciando el objeto del cambio

3. especificar los límites

4. seleccionar las capacidades de escritura (y lectura) del objeto fuente (según sea apropiado)

Los dos errores más comunes son:

1. ***No especificar (correctamente) los límites***. La intersección de los límites de un *Behaviour* con el volumen de activación de una vista determina si el evento Java 3D considera el disparo del estímulo para el *Behaviour*. Java 3D no avisará si no se ponen los límites y el comportamiento nunca se disparará.

2. ***No añadir un Behaviour al escenario gráfico.*** Un objeto *Behaviour* que no forma parte de un escenario gráfico será considerado basura y será eliminado en el siguiente ciclo del recolector de basura. Esto, también sucederá sin errores ni avisos.

6.6 Recomendaciones de diseño para la clase *Behaviour*

El mecanismo de escritura de un comportamiento personalizado es sencillo. Sin embargo, habrá que tener en cuenta que un comportamiento mal escrito puede degradar el rendimiento del renderizado. A la hora de crear comportamientos, habrá dos aspectos que será importante evitar:

1. ***quemar la memoria.*** 'quemar la memoria' es el término que se aplica a la creación de objetos innecesarios en Java. La quema de memoria excesiva causará la recolección de basura. Las pausas ocasionales en el renderizado son típicas de la quema de memoria ya que durante la recolección de basura, el renderizado se parará. Los métodos de la clase *Behaviour* normalmente son los responsables de crear problemas de quema de memoria. Por ejemplo, usar un *'new'* en el método *processStimulus* hace que se cree un nuevo objeto cada vez que se invoca a este método.

   Los problemas potenciales de la quema de memoria son fáciles de identificar y evitar. Por ejemplo, basta con buscar cualquier uso de *'new'* en el código, para encontrar la fuente de estos tipos de problemas. Para solucionarlos, se reemplazará, siempre que sea posible, el uso de *'new'* por código que reutilice un objeto.

2. ***condiciones de disparo innecesarios***.

6.7 Condiciones de disparo de los comportamientos

Los comportamientos activados se disparan por la ocurrencia de uno o más estímulos especificados. El estímulo de disparo para un comportamiento se especifica usando

descendientes de la clase abstracta *WakeupCondition*. También se permite la composición de múltiples condiciones de disparo en una única condición de disparo.

La tabla 6.1 presenta las 14 clases *WakeupCriterion* específicas. Estas clases se usan para especificar las condiciones de disparo de los objetos *behaviour*. Los ejemplares de esta clase se usan individualmente o en combinaciones.

Clase Criterio	**Disparo**
WakeupOnActivation	en la primera detección de una intersección del volumen de activación de un **ViewPlatform** con la región límite del objeto.
WakeupOnAWTEvent	cuando ocurre un evento AWT específico
WakeupOnBehaviourPost	cuando un objeto **Behaviour** envía un evento específico
WakeupOnCollisionEntry	en la primera detección de colisión del objeto especificado con otro objeto del escenario gráfico.
WakeupOnCollisionExit	cuando el objeto especifico no colisiona con ningún otro objeto del escenario gráfico.
WakeupOnCollisionMovement	cuando el objeto especificado se mueve mientras colisiona con otro objeto del escenario gráfico
WakeupOnDeactivation	cuando el volumen de activación de un **ViewPlatform** deja de intereseccionar con los límites del objeto
WakeupOnElapsedFrames	cuando ha pasado un número determinado de frames
WakeupOnElapsedTime	cuando ha pasado un número de segundos determinado

WakeupOnSensorEntry	en la primera detección de cualquier sensor que intersecciona con los límites especificados
WakeupOnSensorExit	cuando un sensor que interseccionaba con los límites del objeto deja de interseccionar con los límites especificados
WakeupOnTransformChange	cuando cambia la transformación dentro de un **TransformGroup** especificado
WakeupOnViewPlatformEntry	en la primera detección de intersección del volumen de activación de un **ViewPlatform** con los límites especificados
WakeupOnViewPlatformExit	cuando el volumen de activación de una vista deja de intereseccionar con los límites especificados

Tabla 6.1 Clases WakeupCriterion específicas proporcionadas por Java3d

En nuestro caso, el criterio utilizado fue WakeupOnCollisionMovement.

6.8 Árbol de escena

En la figura 6.1 representamos el grafo de escena de un agente *Gato* o *Ratón*.

La rama izquierda hace referencia al *Nimbus* del agente.

La rama derecha hace referencia al Focus. Esta se compone de una esfera de tamaño 1 que es la que manipulamos cuando el agente se desplaza o rota. El cono y el objeto *J3dCollisionDetectionBehaviour*, que hereda de la clase *Behaviour*, son hijos de la esfera de tamaño 1, con lo que cualquier modificación sobre esta afectará a sus hijos. Es más sencillo trabajar con una esfera lo que justifica su uso.

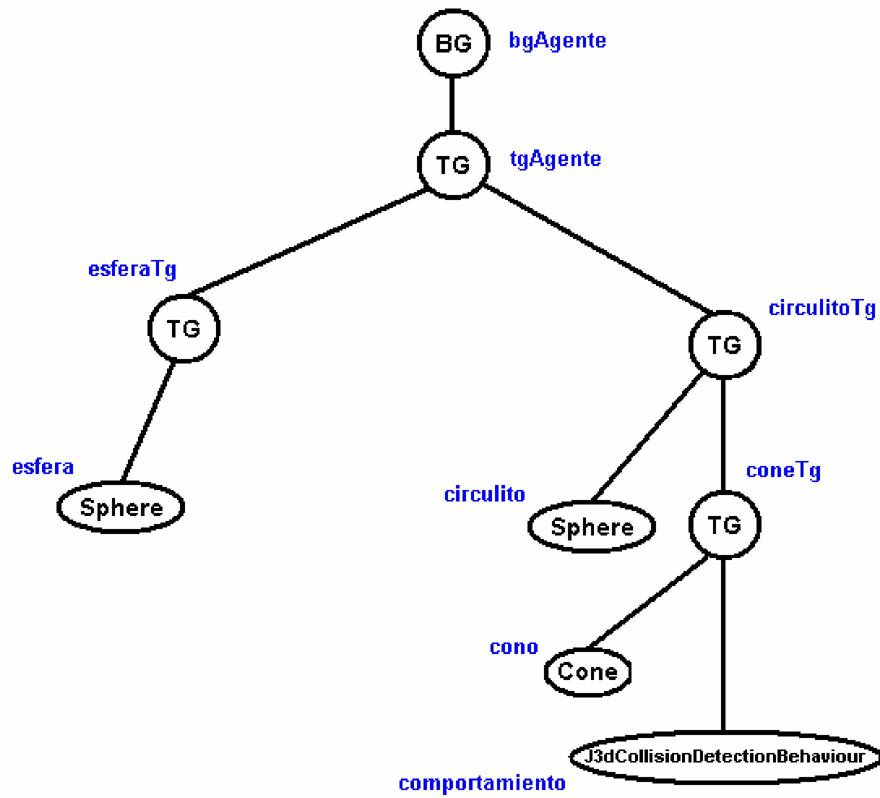


Figura 6.1 Grafo de escena de un agente *Gato* o *Ratón*

Capítulo 7. Toma de decisiones

En numerosos entornos de simulación resulta necesario tomar algún tipo de decisión por parte de los agentes involucrados. Se hace necesario recurrir a algún tipo de estrategia basada en el paradigma de la Teoría de la Decisión. Existen diversos métodos y estrategias dentro de este paradigma para decidir sobre la alternativa a elegir en la decisión.

En este proyecto se ha aplicado el enfoque conocido como Teoría de la Decisión Multicriterio (MultiCriteria-Decision Making, MCDM) bajo la perspectiva de la lógica borrosa o fuzzy. Con ello se consigue introducir un componente aleatorio en el entorno de simulación donde el aspecto determinista de las decisiones se sustituye por una componente predominantemente aleatoria.

7.1 Enfoque general MCDM Fuzzy

7.1.1 Formulación del problema y definiciones

Un problema general de decisión multicriterio con m alternativas A_i (i = 1,…, m) y n criterios C_j (j = 1,…, n) se puede expresar de la siguiente manera:

$$D = \lfloor x_{ij} \rfloor \text{ y } W = (w_j), \text{donde } i = 1, ..., m \text{ y } j = 1, ..., n. \quad (7.1)$$

Aquí D hace referencia a la *matriz de decisión* (donde la entrada x_{ij} representa el valor para la alternativa A_i con respecto al criterio C_j), y W como el *vector de pesos* (donde w_j representa el peso del criterio C_j). En general se clasifican los criterios como sigue:

- *criterios de beneficio* (donde el mejor valor para la toma de decisiones es el valor mas alto de x_{ij}) o
- *criterios de coste* (donde el mejor valor para la toma de decisiones es el valor mas bajo de x_{ij}).

Como se desea considerar a los valores en D y W fuzzy en lugar de no fuzzy, se usará la misma notación, teniendo en cuenta que a partir de este momento son valores fuzzy:

$$D = \lfloor x_{ij} \rfloor \text{ y } W = (w_j) \tag{7.2}$$

Donde x_{ij} representa el valor fuzzy para la alternativa A_i con respecto al criterio C_j, y w_j representa el peso fuzzy del criterio C_j. En particular, un acercamiento por intuición fácil y eficaz a la captura de la incertidumbre del experto sobre el valor de un número desconocido es un número triangular fuzzy:

Definición: Un número triangular fuzzy a se define por una tupla (a_1, a_2, a_3). La función que los relaciona se define como sigue (Kaufmann y Gupta, 1985):

$$\boldsymbol{\mu}_{\boldsymbol{a}}(x) = \begin{cases} (x - a_1)/(a_2 - a_1), & a_1 \le x \le a_2, \\ (a_3 - x)/(a_3 - a_2), & a_2 \le x \le a_3, \\ 0, & \text{en otro caso.} \end{cases} \tag{7.3}$$

El número triangular fuzzy se basa en el juicio de tres valores: el menor valor posible a_1, el valor más posible a_2 y el máximo valor posible a_3.

7.1.2 Normalización

Para tratar con criterios de diferentes escalas, se aplica un proceso de normalización. Específicamente, se normalizan los números fuzzy en la matriz de decisión como la matriz de *performance*:

$$P = \lfloor p_{ij} \rfloor,$$

$$\text{donde } p_{ij} = \begin{cases} \left(\dfrac{x_{ij1}}{M}, \dfrac{x_{ij2}}{M}, \dfrac{x_{ij3}}{M}\right), & \text{siendo } M = \max_i x_{ij3}, \\ & C_j \text{ es un criterio de beneficio} \\ \left(\dfrac{N - x_{ij3}}{N}, \dfrac{N - x_{ij2}}{N}, \dfrac{N - x_{ij1}}{N}\right), & \text{siendo } N = \max_i x_{ij3}, \\ & C_j \text{ es un criterio de coste} \end{cases} \tag{7.4}$$

Este método preserva los números triangulares fuzzy normalizados al rango [0, 1].

7.1.3 Añadiendo pesos a los criterios

Se construye la matriz de *performance* promediada multiplicando el vector de pesos por la matriz de decisión como:

$$P^w = \left[p_{ij}^w\right],$$

$$\text{donde } p_{ij1}^w = w_{j1}p_{ij1},\ p_{ij2}^w = w_{j2}p_{ij2},\ p_{ij3}^w = w_{j3}p_{ij3},\ i = 1, 2, ..., m, \text{ y } j = 1, 2, ..., n. \quad (7.5)$$

7.1.4 Elección de alternativas

Se utiliza el *método del vértice* (Chen, 2000) para calcular el índice de referencia de las alternativas para tratar soluciones ideales (Hwang y Yoon, 1981). La mejor alternativa debería tener la distancia mas corta a la solución ideal positiva, y la distancia mas larga a la solución ideal negativa.

Definición: Sean $a = (a_1, a_2, a_3)$ y $b = (b_1, b_2, b_3)$ dos números triangulares fuzzy, entonces el *método del vértice* define la distancia entre ellos como:

$$d(a,b) = \left\{\left[(a_1 - b_1)^2 + (a_2 - b_2)^2 + (a_3 - b_3)^2\right]/3\right\}^{1/2} \quad (7.6)$$

Para matriz de funcionamiento *fuzzy* normalizada, se define la solución ideal positiva $p_j^+ = (1, 1, 1)$ y la solución ideal negativa $p_j^- = (0, 0, 0)$ bajo los criterios como referencias para medir el funcionamiento de las alternativas (Chen, 2000). Por el *método del vértice*, la distancia entre cada alternativa y la solución ideal positiva y la solución ideal negativa se calcula como:

$$d_i^+ = \sum_{j=1}^{n} d\left(p_{ij}^w, p_j^+\right) \quad (7.7)$$

$$d_i^- = \sum_{j=1}^{n} d\left(p_{ij}^w, p_j^-\right) \tag{7.8}$$

donde $i = 1, ..., m$ y $j = 1, ..., n$.

Se calcula el índice de *performance* para cada alternativa como:

$$p_i = \frac{d_i^- + n - d_i^+}{2n} \tag{7.9}$$

donde $i = 1, ..., m$, y n es el número de criterios. La p_i obtenida que se aproxime más a 1 será la mejor alternativa.

7.2 Criterios que intervienen en la toma de decisiones

La *matriz de decisión* se puede representar a partir de la siguiente tabla:

Criterios/ Alternativas	w_1 c_1	w_2 c_2	……..	w_n c_n
A_1	(a_{11},a_{21}, a_{31})	(b_{11},b_{21},b_{31})		
A_2	(a_{12},a_{22},a_{32})	(b_{12},b_{22},b_{32})		
A_3	(a_{13},a_{23},a_{33})	(b_{13},b_{23},b_{33})		

Tabla 7.1 Matriz de decisión

Cada fila representa una de las posibles decisiones que pueden tomar cada uno de los agentes que intervienen en el entorno (en nuestro caso los gatos y los ratones).

Estas decisiones dependen de una serie de criterios (c_1, c_2,..., c_n), que se corresponden con las columnas que tenga la tabla.

A continuación se entra en detalle de cada uno de los criterios que se han definido:

7.2.1 Criterio 1: Tipo de agente

El primer criterio hace referencia al tipo de agente que se percibe. El peso de este criterio con respecto al resto debe ser mayor ya que condiciona mucho la alternativa. En situaciones similares tomaremos alternativas diferentes en función del tipo de agente.

Los tipos de agente que hemos definido han sido dos, *Gato* y *Ratón*, pero se podría extender si hubiera mas agentes en el entorno.

En la matriz de decisión de ambos agentes, este criterio es de *beneficio*, ya que a mayor valor mejor decisión tomaremos.

La diferencia entre ambos agentes es que en el caso del agente *Gato* la mejor decisión es que el tipo de agente sea *Ratón* y en el caso del agente *Ratón* es al contrario.

Los valores numéricos asignados a cada uno de los posibles valores de este criterio, para el caso del agente *Gato*, han sido los siguientes:

Ratón → 1

Gato → 0

Los valores numéricos asignados a cada uno de los posibles valores de este criterio, para el caso del agente *Ratón*, han sido los siguientes:

Gato → 1

Ratón → 0

Estos valores numéricos ya se encuentran normalizados en el rango [0, 1].

7.2.2 Criterio 2: Orientación

El segundo criterio hace referencia a la orientación con que un agente percibe al otro.

Los valores que puede tomar este criterio son ocho:

1. *Frente*
2. *Frente derecha*
3. *Frente izquierda*
4. *Lado derecha*
5. *Lado izquierda*

6. *Espaldas derecha*

7. *Espaldas izquierda*

8. *Espaldas*

En la matriz de decisión del agente *Gato*, este criterio es de *beneficio*, siendo el mejor valor "*espaldas*" y el peor "*frente*". El motivo es que el agente *Gato* (sin considerar el resto de criterios), tomará mejor la decisión de *perseguir* al agente *Ratón* con mayor posibilidad si este se encuentra de espaldas a él que si está de frente. Esto es porque estando de espaldas el agente *Gato* puede aproximarse sin ser visto, mientras que si el ratón percibiera al gato, con toda seguridad tomaría la decisión de *huir*.

En la matriz de decisión del agente *Ratón*, este criterio es de *coste*, siendo el mejor valor "*frente*" y el peor "espaldas". El motivo es que el agente *Ratón* (sin considerar el resto de criterios), tomará mejor la decisión de *huir* si el agente *Gato* se encuentra de frente a él que si está de espaldas, ya que como ocurre con el caso del gato, si un agente está de espaldas a otro no puede percibirle.

Los valores numéricos asignados a cada uno de los posibles valores de este criterio han sido los siguientes:

1. *Frente* → 0

2. *Frente derecha* → 0.5

3. *Frente izquierda* → 0.5

4. *Lado derecha* → 1

5. *Lado izquierda* → 1

6. *Espaldas derecha* → 1.5

7. *Espaldas izquierda* → 1.5

8. *Espaldas* → 2

Para normalizar estos valores en el rango [0, 1] simplemente dividimos entre 2.

7.2.3 Criterio 3: Claridad de percepción

El tercer criterio que interviene en la toma de decisiones, sigue el esquema de la figura 7.1 que representa la claridad de percepción de un agente en las distintas zonas de su *focus* o cono de visión:

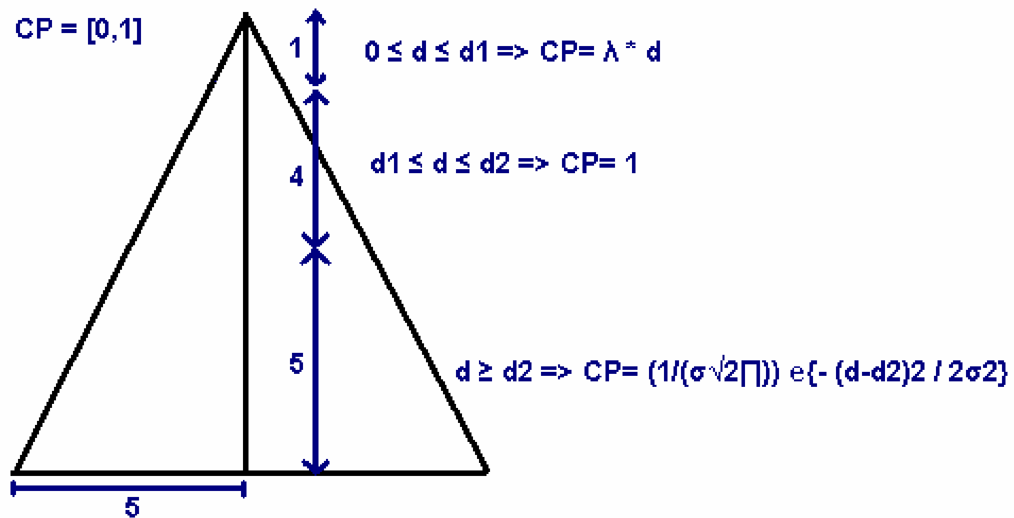


Figura 7.1 Claridad de percepción del agente dependiendo de la parte del *focus* con la que se colisione

En la región central, la claridad de percepción es máxima. Muy cerca del agente, no se percibe con tanta claridad, al igual que si nos alejamos del agente donde la claridad va disminuyendo siguiendo una curva Gaussiana.

d1 representa la distancia mínima entre el objeto y el ojo para que dicho objeto pueda ser percibido. Hemos tomado *d1* = 1.

d2 representa la distancia máxima entre el objeto y el ojo para que dicho objeto pueda ser percibido con un elevado nivel de detalle. Hemos tomado *d2* = 8.

Los valores de este criterio se encuentran normalizados entre [0, 1].

7.2.4 Criterio 4: Distancia

El cuarto y último criterio que interviene en la toma de decisiones es la distancia.

Este criterio representa la distancia entre dos agentes entre los que se ha producido al menos una intersección entre el *focus* de un agente y el *nimbus* del otro.

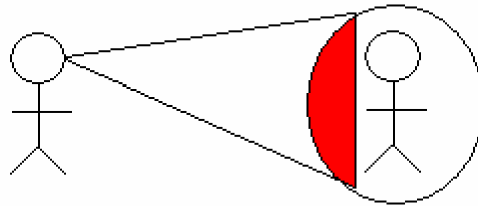


Figura 7.2 Intersección entre *focus* y *nimbus*.

Los valores que pueden tomar la distancia están dentro del rango [0, 15].

Dado que la altura del cono es de 10, el radio de la esfera que representa el *nimbus* es de 5, y que el modelo de percepción que hemos definido considera que a partir de una intersección del 50% entre el *focus* de un agente y el *nimbus* de otro, la claridad de percepción es máxima, la distancia entre los agentes cuando uno de ellos percibe al otro con toda claridad es de 10.

Para normalizar los valores de este criterio dentro del rango [0, 1] se divide entre 15.

7.3 Alternativas que intervienen en la toma de decisiones

7.3.1 Alternativas para el agente *Gato*

Las posibles alternativas que puede tomar un agente *Gato* en función del valor de los criterios y de los pesos asignados a estos, pueden ser tres:

- Alternativa 1: *No hacer nada.*
- Alternativa 2: *Perseguir*

- Alternativa 3: *Esquivar*

En el caso de que la decisión sea *no hacer nada*, el agente seguirá con la trayectoria que estaba siguiendo a este ese momento.

En el caso de que la decisión tomada sea *perseguir*, el agente variará su trayectoria con la intención de perseguir al agente que ha percibido, y del que conoce que tipo de agente es, su orientación y la distancia a la que se encuentra.

En el caso de que la decisión tomada sea *esquivar*, el agente deberá variar su trayectoria, ya que de lo contrario se chocará con otro agente que se dirige al mismo lugar que él. Un agente *Gato* solo tomará la decisión de *esquivar* si el otro agente no es un ratón, ya que de lo contrario hubiera tomado la decisión de *perseguir*.

7.3.2 Alternativas para el agente *Ratón*

Las posibles alternativas que puede tomar un agente *Ratón* en función del valor de los criterios y de los pesos asignados a estos, pueden ser tres:

- Alternativa 1: *No hacer nada.*
- Alternativa 2: *Huir*
- Alternativa 3: *Esquivar*

En el caso de que la decisión sea *no hacer nada*, el agente seguirá con la trayectoria que estaba siguiendo a este ese momento.

En el caso de que la decisión tomada sea *huir*, el agente variará su trayectoria con la intención de escapar del agente que ha percibido, y del que conoce que tipo de agente es, su orientación y la distancia a la que se encuentra.

En el caso de que la decisión tomada sea *esquivar*, el agente deberá variar su trayectoria, ya que de lo contrario se chocará con otro agente que se dirige al mismo lugar

que él. Un agente *Ratón* solo tomará la decisión de *esquivar* si el otro agente no es un gato, ya que de lo contrario hubiera tomado la decisión de *huir*.

Capítulo 8. Comunicación mediante sockets

8.1 Necesidades del proyecto

Una vez hecha la especificación del proyecto se da paso a la investigación acerca de la comunicación entre las plataformas Java y C, con las particularidades de cada una de ellas. En nuestro caso Java es el lenguaje a usar en la plataforma de agentes, y C el lenguaje a usar en el módulo de la interfaz Gráfica (desarrollo con OSG).

Surge la necesidad de buscar información sobre qué posibilidades de trabajo se tienen, con el fin de permitir la comunicación mencionada, además del envío bidireccional de estructuras de datos (para empezar, las coordenadas [x,y,z] de un objeto móvil) entre un módulo y otro.

Inicialmente se pretende que la plataforma de agentes pueda facilitar información al módulo gráfico para representar correctamente los objetos y agentes en un escenario 3D, así como que dicho escenario envíe información actualizada sobre los agentes que pasan a formar parte del escenario después de iniciada la aplicación.

Figura 8.1 Necesidad de comunicar la plataforma de agentes y el entorno virtual.

Una vez se han tenido en cuenta estos requerimientos, se hace de vital importancia para la implementación del proyecto encontrar un mecanismo que permita el intercambio de datos.

Como se indica con anterioridad, finalmente se optó por implementar una arquitectura software cliente/servidor mediante sockets, una vez descartadas la JNI y el estándar CORBA.

8.2 Protocolo de comunicación

Para entender un poco mejor el funcionamiento del modelo elegido, se detallan a continuación los aspectos básicos de dicha elección.

8.2.1 Elementos de la arquitectura Cliente-Servidor

Con el objetivo de definir y delimitar el modelo de referencia de una arquitectura Cliente/Servidor, se han de identificar los componentes que permiten articular dicha arquitectura, considerando que toda aplicación de un sistema de información está caracterizada por tres componentes básicos:

- Presentación/Captación de Información
- Procesos
- Almacenamiento de la información

Los cuales se suelen distribuir tal como se presenta en la figura:

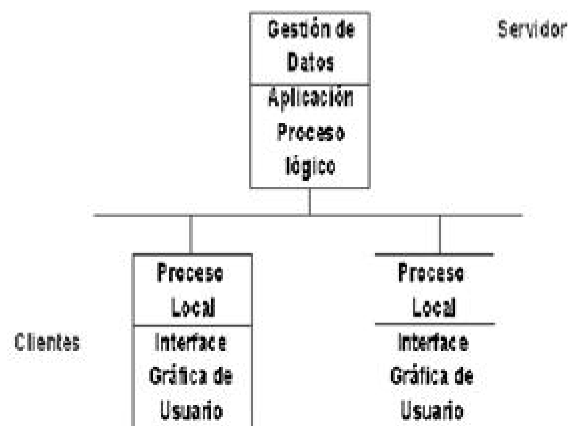


Figura 8.2 Componentes básicos de la arquitectura Cliente-Servidor

Se integran en una arquitectura Cliente-Servidor en base a los elementos que caracterizan dicha arquitectura, es decir:

- Puestos de Trabajo
- Comunicaciones

- Servidores

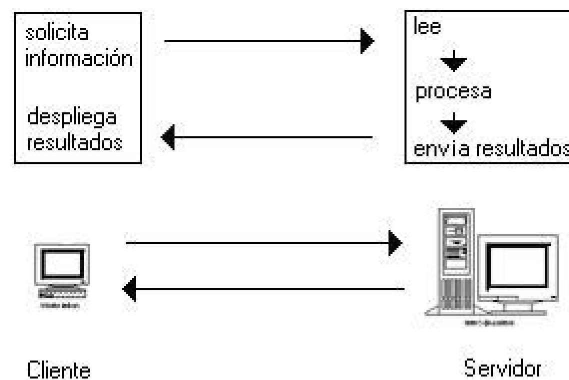


Figura 8.3 Arquitectura Cliente-Servidor

8.2.2 Características del modelo Cliente-Servidor

El modelo Cliente-Servidor presenta las siguientes características:

1. El Cliente y el Servidor pueden actuar como una sola entidad y también pueden actuar como entidades separadas, realizando actividades o tareas independientes.

2. Las funciones de Cliente y Servidor pueden estar en plataformas separadas, o en la misma plataforma.

3. Un servidor da servicio a múltiples clientes de forma concurrente.

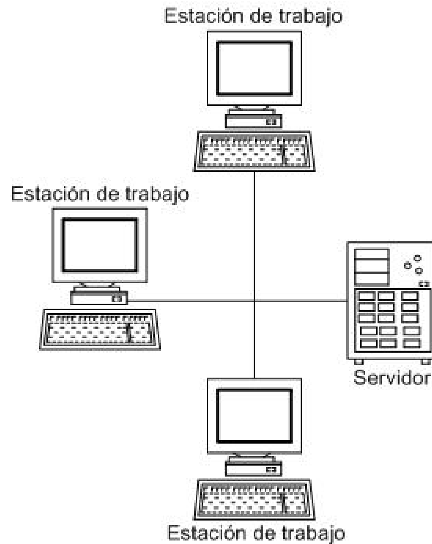


Figura 8.4 Un servidor da servicio a múltiples clientes de forma concurrente

4. Cada plataforma puede ser escalable independientemente. Los cambios realizados en las plataformas de los Clientes o de los Servidores, ya sean por actualización o por reemplazo tecnológico, se realizan de una manera transparente para el usuario final.

5. La interrelación entre el hardware y el software están basados en una infraestructura poderosa, de tal forma que el acceso a los recursos de la red no muestra la complejidad de los diferentes tipos de formatos de datos y de los protocolos.

6. Un sistema de servidores realiza múltiples funciones al mismo tiempo que presenta una imagen de un solo sistema a las estaciones Clientes**.** Esto se logra combinando los recursos de cómputo que se encuentran físicamente separados en un solo sistema lógico, proporcionando de esta manera el servicio más efectivo para el usuario final.

   También es importante hacer notar que las funciones Cliente-Servidor pueden ser dinámicas. Ejemplo, un servidor puede convertirse en cliente cuando realiza la solicitud de servicios a otras plataformas dentro de la red.

7. Además se constituye como el nexo de unión mas adecuado para reconciliar los sistemas de información basados en *mainframes* o minicomputadores, con aquellos otros sustentados en entornos informáticos pequeños y estaciones de trabajo.

8. Designa un modelo de construcción de sistemas informáticos de carácter distribuido.

En conclusión, el modelo Cliente-Servidor puede incluir múltiples plataformas, bases de datos, redes y sistemas operativos. Estos pueden ser de distintos proveedores, en arquitecturas propietarias y no propietarias y funcionando todos al mismo tiempo.

8.2.3 Ventajas de la arquitectura Cliente-Servidor

Algunas de las ventajas que proporciona la arquitectura Cliente-Servidor son:

- Centralización del control: los accesos, recursos y la integridad de los datos son controlados por el servidor de forma que un programa cliente defectuoso o no autorizado no pueda dañar el sistema.
- Escalabilidad: se puede aumentar la capacidad de clientes y servidores por separado.
- La interfaz del usuario aprovecha bien la GUI y la LAN.
- La aplicación aprovecha el Host.
- Adecuada para algunos tipos de aplicaciones de apoyo a la toma de decisiones.

Por el contrario, algunas de las desventajas que presenta son:

- Las aplicaciones pueden ser complejas de desarrollar.
- Los programas de la aplicación siguen en el Host.
- El alto volumen de tráfico en la red puede hacer difícil la operación de aplicaciones muy pesadas.

El estilo de arquitectura a seguir es el de presentación remota, en el que:

- La interfaz para el usuario esta completamente en los clientes (OSG).
- La aplicación y los datos están en el servidor (Plataforma JADE).

8.3 Diferencias entre plataformas

El problema es que cada microprocesador define los tipos de datos de una manera concreta. Lo normal es que un entero, por ejemplo, sean cuatro bytes (32 bits), aunque algunos microprocesadores antiguos eran de 2 bytes (16 bits) y los más modernos empiezan a ser de 8 bytes (64 bits).

Si se envía un entero a través de un socket, se están enviando cuatro bytes. Si los microprocesadores a ambos lados del socket tienen el mismo orden para los bytes, no hay problema, pero si tienen distinto, el entero se interpretará incorrectamente.

La máquina virtual Java, por ejemplo, define los char como de 2 bytes (para poder utilizar caracteres UNICODE), mientras que en el resto de los microprocesadores habituales suele ser de un byte. Si se envía desde Java un carácter a través de un socket, se mandan 2 bytes, mientras que si se lee del socket desde un programa en C, sólo se leerá un byte, dejando el otro "pendiente" de lectura.

En C de Windows (WinSocket), se dispone de la familia de funciones htonl.

En el apartado de prototipos se detallan los pasos realizados.

Capítulo 9. Entorno Virtual

9.1 Descripción

El proyecto necesita de un módulo que se encargue de ilustrar las interacciones entre los agentes virtuales y con el mundo donde se producen las mismas. Para ello se ha elegido el conjunto de herramientas gráficas que ofrece OSG, por lo que el desarrollo de este entorno se ha realizado en el lenguaje C++.

El uso de OSG permite obtener buenos resultados visuales: gran movilidad por el entorno (acercar/alejar, girar…), texturas opacas (modelos que representan los agentes) y transparentes (modelos que representan el *focus* y el *nimbus* de un objeto).

9.2 Árbol de escena

OSG trabaja con árboles de escena. El entorno virtual dispone de funciones para la creación de conos y esfera, así como para la gestión de los nodos del árbol del que cuelgan los componentes gráficos.

El grafo para representar y mover a los agentes tiene la siguiente estructura:

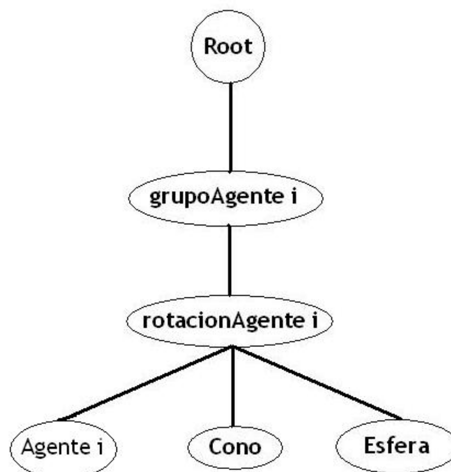


Figura 9.1. Estructura del grafo para representar y mover los agentes

- ***Root***: Este es el nodo principal (raíz) del que cuelga todo lo que va a aparecer en la escena. Se encuentra situado a profundidad 0.

- ***grupoAgente i***: Este nodo es hijo del nodo principal Root, y tiene la función de desplazar los agentes por el entorno virtual. Es decir, todas las traslaciones se harán sobre este nodo, ya que cada traslación afectará a todos sus hijos produciendo el efecto de movimiento. *Agente* es un nombre significativo que se le otorga al nodo para poder localizarlo fácilmente, y puede ser Gato ó Ratón. El índice *i* se utiliza para ir numerando los agentes, ya que dependiendo de la ejecución el número de agentes en escena puede variar. De esta forma, todos los nodos en la escena tienen un identificador único, a excepción de los conos y esferas. Se encuentra situado a profundidad 1.
- ***rotaciónAgente i:*** Este nodo es hijo de grupo*Agente i*, y tiene la función de rotar a los agentes cuando se produzcan cambios de orientación provocados por las distintas trayectorias o por la toma de decisiones. Es decir, todas las operaciones de rotación se harán sobre este nodo, para que todos sus hijos roten a la vez y con el mismo ángulo produciendo el efecto de giro. Al igual que en su nodo padre, se utiliza el mismo sistema para nombrar a los nodos de este tipo. Se encuentra situado a profundidad 2.
- ***Agente i:*** Este nodo es hijo de rotación*Agente i,* y representa el modelo 3D del gato ó ratón. Se utiliza el mismo sistema de nombrado que en sus nodos ancestros. Al ser descendiente de rotación*Agente i* y de grupo*Agente i* le afectarán todas las traslaciones y rotaciones realizadas sobre dichos nodos. Se encuentra situado a profundidad 3.
- ***Cono***: Este nodo es hijo de rotación*Agente i* y representa el *focus,* es decir, el foco de visión del agente. Como a *Agente i,* también le afectarán todas las traslaciones y rotaciones realizadas sobre sus nodos antecesores. Se encuentra situado a profundidad 3.
- ***Esfera***: Este nodo es hijo de rotación*Agente i* y representa el *nimbus,* es decir, el espacio que ocupa el agente. Al igual que a sus hermanos, también le afectarán todas las traslaciones y rotaciones realizadas sobre sus nodos antecesores. Se encuentra situado a profundidad 3.

9.3 Modelos 3D

Los modelos 3D son elementos gráficos cuyo propósito es conseguir una proyección visual en dos dimensiones en una pantalla o sobre papel. Están creados mediante

aplicaciones que manejan colecciones de datos (puntos, polígonos..), ya sea a través de procesos de escaneado, de forma manual o algorítmicamente.

Este tipo de gráficos se origina mediante cálculos matemáticos sobre entidades geométricas tridimensionales producidas en un ordenador, y cuyo propósito es conseguir una proyección visual en dos dimensiones para ser mostrada en una pantalla o impresa en papel.

Hoy en día se utilizan en numerosos campos tales como la medicina, la industria del cine, videojuegos, la arquitectura, la ingeniería, la investigación espacial…

En el módulo gráfico del proyecto ha sido necesario disponer de modelos que representen tanto los agentes Gato y Ratón como las esferas que representan el *focus* y los conos que representan el *nimbus*

Para conseguir un aspecto transparente en los conos y las esferas de manera que representen estos conceptos pero no obstaculicen la correcta visión de los agentes, se ha utilizado la constante *GL_BLEND*, que se pasa como parámetro al invocar el método *setMode* sobre un objeto *osg::StateSet **.

Respecto a los modelos, es preciso indicar que ha sido necesario adaptarlos a nuestras necesidades. Estas adaptaciones se han realizado sobre el modelo de la rata, ya que tanto su tamaño como el número de polígonos de la malla 3D era excesivo y no se corresponde proporcionalmente con el tamaño deseado para el gato, que no fue necesario transformar.

Para ello se han utilizado herramientas de tratamiento de gráficos en 3D, entre las que destacar Blender[9], que es de libre distribución y se encuentra disponible en la Web.
El primer paso fue conseguir la conversión del modelo de la rata en formato ".max" a ".3ds". Con Blender se pueden importar y exportar ficheros de distintos formatos, entre ellos el ".3ds". Al importar fue necesario ajustar el tamaño del modelo mediante

---

[9] Consultar información en la bibliografía y en www.blender.org

"SizeConstraints" (escala el modelo por 10 hasta alcanzar el valor especificado), que después de varias pruebas se fijó a 250.

Una vez importado el modelo, se usaron distintas funciones de transformación (escala, rotación...) y desde la ventana de scripts, se seleccionó el PolyReducer correspondiente al tratamiento de la malla para hacer la reducción de los polígonos mencionada.

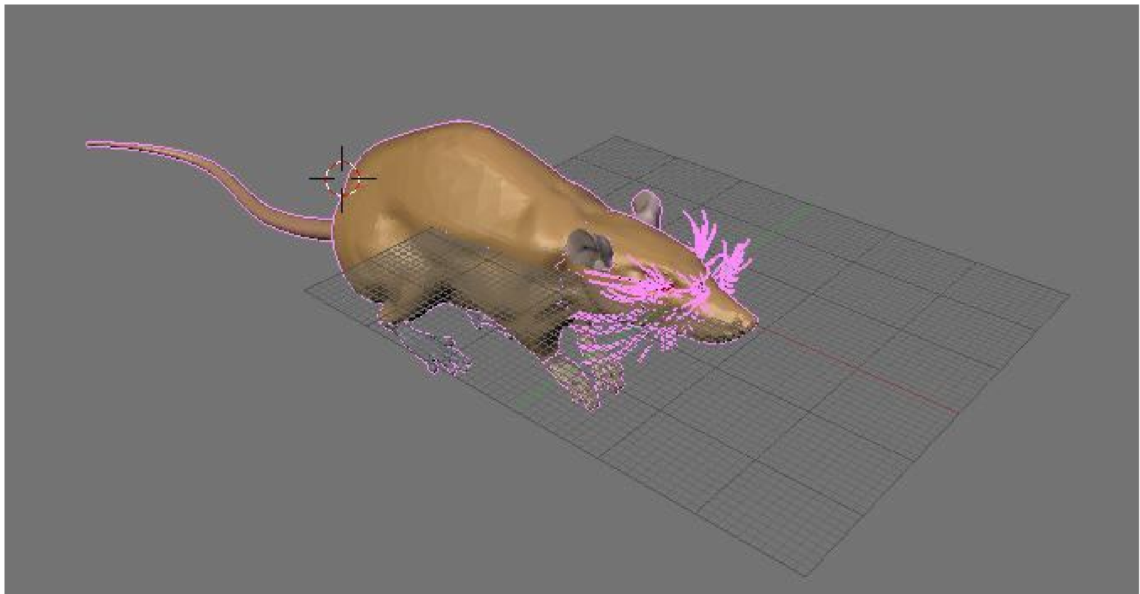


Figura 9.2. Modelo recién importado

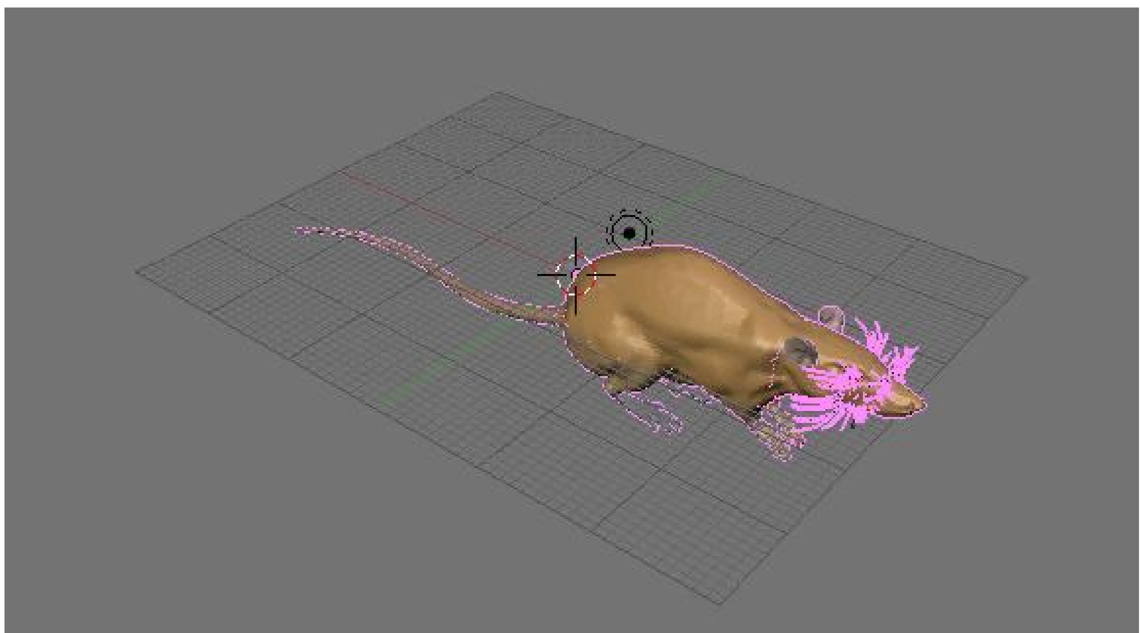


Figura 9.3. Modelo escalado

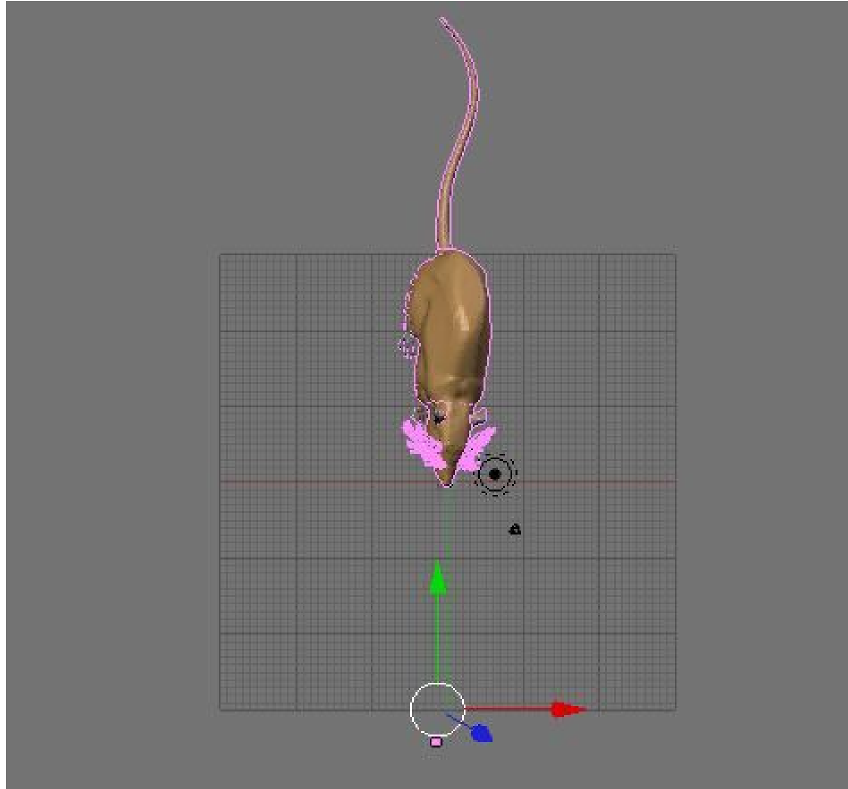


Figura 9.4. Modelo rotado localmente sobre el eje Z

Capítulo 10. Diagramas UML

10.1 Comportamientos

Los siguientes diagramas de clases muestran la organización de los comportamientos de los distintos agentes.

Como se puede apreciar, el comportamiento más complejo es del agente que controla la comunicación con el entorno gráfico. Consta a su vez de varios comportamientos.

ActivationAgentsBehaviour se encarga de activar a los agentes de percepción mandándoles un mensaje ACL cuando la comunicación con el entorno 3D esté lista.

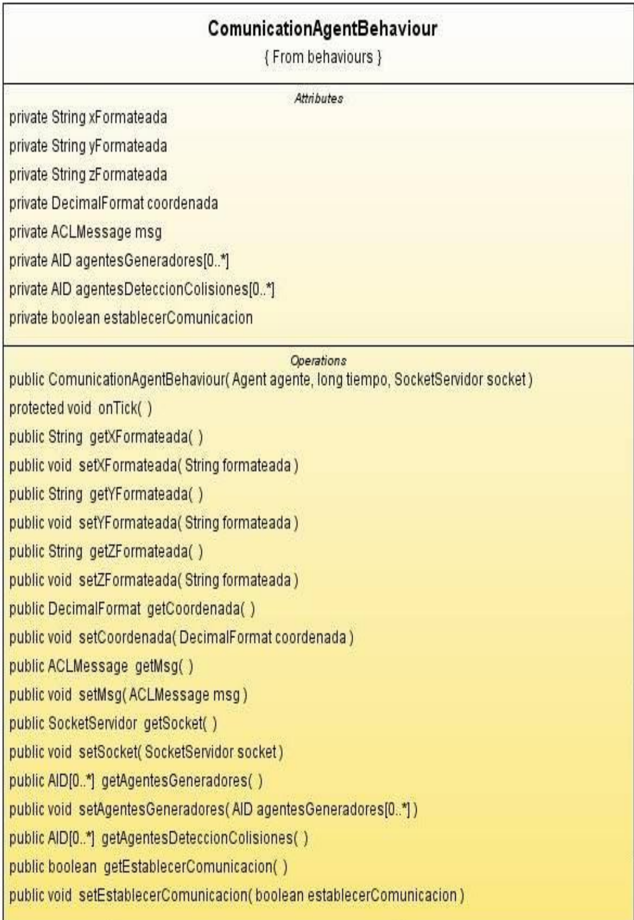
Por otro lado, el comportamiento *FormatAndSendBehaviour* trata las coordenadas que le han enviado los agentes Gato y Ratón para que tengan un formato de tres cifras decimales. Una vez formateadas las coordenadas las envía por el socket para que se dibujen los agentes en sus posiciones correspondientes.

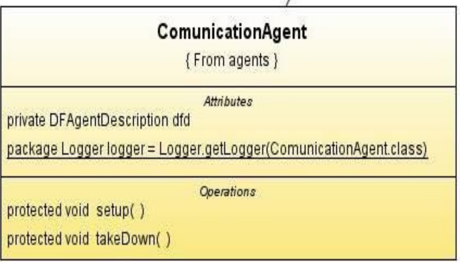
Mediante la clase *YellowPages* (Páginas Amarillas) se permite al agente de comunicaciones localizar cuántos agentes de detección de colisiones y de percepción hay actualmente en el sistema, para poder comunicarse con ellos en caso de que sea necesario.

CommunicationAgentBehaviour es el comportamiento principal de agente de comunicación (*CommunicationAgent*), el cual se encarga (usando los comportamientos descritos anteriormente) de buscar agentes en las Páginas Amarillas, activarlos al inicio, recibir mensajes de otros agentes y formatear y enviar las coordenadas a la parte gráfica.

Los comportamientos de los agentes Gato y Ratón son prácticamente iguales, ya que ambos generan una trayectoria de movimiento al azar cuando se crean. Sólo se diferencian en las decisiones tomadas en función de la información percibida por su campo de visión. Ambos disponen de métodos que permiten generar coordenadas dependiendo de cuál ha sido la decisión tomada. Mediante el servicio de Páginas Amarillas tienen conocimiento del agente de detección de colisiones y del agente de comunicación, para poder enviarles la información pertinente a cada uno.

El comportamiento *CollisionDetectionBehaviour* dispone de información (mediante las Páginas Amarillas) de cuántos agentes generadores hay actualmente activos (Gatos y Ratones) para que, una vez haya tratado las colisiones entre los agentes, les indique cuándo deben volver a generar coordenadas nuevas.

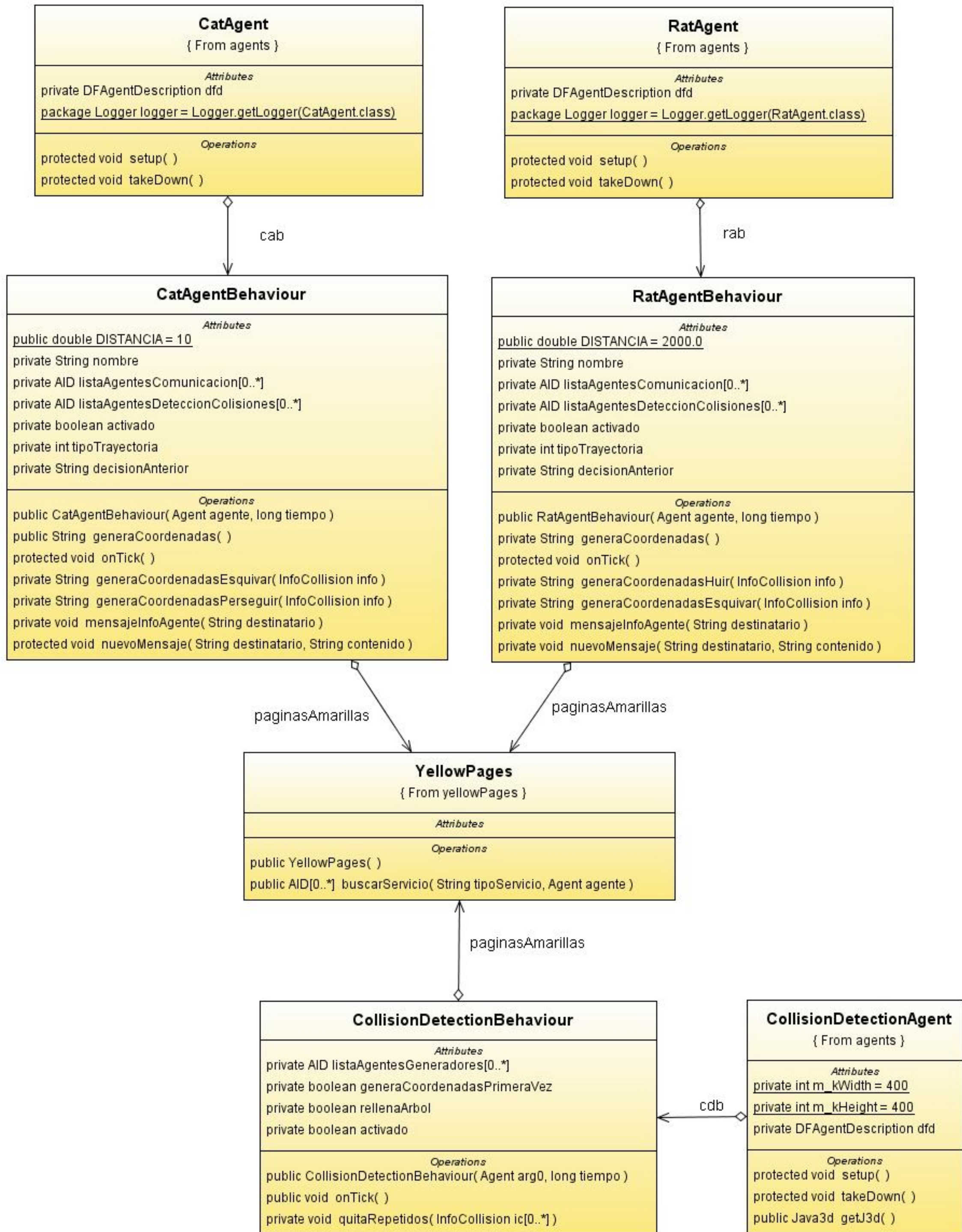

ComunicationAgentBehaviour
{ From behaviours }
Attributes
private String xFormateada
private String yFormateada
private String zFormateada
private DecimalFormat coordenada
private ACLMessage msg
private AID agentesGeneradores[0..*]
private AID agentesDeteccionColisiones[0..*]
private boolean establecerComunicacion
Operations
public ComunicationAgentBehaviour(Agent agente, long tiempo, SocketServidor socket)
protected void onTick( )
public String getXFormateada( )
public void setXFormateada(String formateada)
public String getYFormateada( )
public void setYFormateada(String formateada)
public String getZFormateada( )
public void setZFormateada(String formateada)
public DecimalFormat getCoordenada( )
public void setCoordenada(DecimalFormat coordenada)
public ACLMessage getMsg( )
public void setMsg(ACLMessage msg)
public SocketServidor getSocket( )
public void setSocket(SocketServidor socket)
public AID[0..*] getAgentesGeneradores( )
public void setAgentesGeneradores(AID agentesGeneradores[0..*])
public AID[0..*] getAgentesDeteccionColisiones( )
public boolean getEstablecerComunicacion( )
public void setEstablecerComunicacion(boolean establecerComunicacion)
paginasAmarillas
cab
cab
cab


ComunicationAgent
{ From agents }
Attributes
private DFAgentDescription dfd
package Logger logger = Logger.getLogger(ComunicationAgent.class)
Operations
protected void setup( )
protected void takeDown( )

FormatAndSendBehaviour
{ From behaviours }
Attributes
package Logger logger = Logger.getLogger(FormatAndSendBehaviour.class)
Operations
public FormatAndSendBehaviour(ComunicationAgentBehaviour c)
public void action( )

ActivationAgentsBehaviour
{ From behaviours }
Attributes
Operations
public ActivationAgentsBehaviour(ComunicationAgentBehaviour c)
public void action( )

YellowPages
Attributes
Operations
public YellowPages( )
public AID[0..*] buscarServicio(String tipoServicio, Agent agente)


CatAgent
{ From agents }
Attributes
private DFAgentDescription dfd
package Logger logger = Logger.getLogger(CatAgent.class)
Operations
protected void setup( )
protected void takeDown( )
RatAgent
{ From agents }
Attributes
private DFAgentDescription dfd
package Logger logger = Logger.getLogger(RatAgent.class)
Operations
protected void setup( )
protected void takeDown( )
cab
rab
CatAgentBehaviour
Attributes
public double DISTANCIA = 10
private String nombre
private AID listaAgentesComunicacion[0..*]
private AID listaAgentesDeteccionColisiones[0..*]
private boolean activado
private int tipoTrayectoria
private String decisionAnterior
Operations
public CatAgentBehaviour(Agent agente, long tiempo)
public String generaCoordenadas( )
protected void onTick( )
private String generaCoordenadasEsquivar(InfoCollision info)
private String generaCoordenadasPerseguir(InfoCollision info)
private void mensajeInfoAgente(String destinatario)
protected void nuevoMensaje(String destinatario, String contenido)
RatAgentBehaviour
Attributes
public double DISTANCIA = 2000.0
private String nombre
private AID listaAgentesComunicacion[0..*]
private AID listaAgentesDeteccionColisiones[0..*]
private boolean activado
private int tipoTrayectoria
private String decisionAnterior
Operations
public RatAgentBehaviour(Agent agente, long tiempo)
private String generaCoordenadas( )
protected void onTick( )
private String generaCoordenadasHuir(InfoCollision info)
private String generaCoordenadasEsquivar(InfoCollision info)
private void mensajeInfoAgente(String destinatario)
private void nuevoMensaje(String destinatario, String contenido)
paginasAmarillas
paginasAmarillas
YellowPages
{ From yellowPages }
Attributes
Operations
public YellowPages( )
public AID[0..*] buscarServicio(String tipoServicio, Agent agente)
paginasAmarillas
CollisionDetectionBehaviour
Attributes
private AID listaAgentesGeneradores[0..*]
private boolean generaCoordenadasPrimeraVez
private boolean rellenaArbol
private boolean activado
Operations
public CollisionDetectionBehaviour(Agent arg0, long tiempo)
public void onTick( )
private void quitaRepetidos(InfoCollision ic[0..*])
cdb
CollisionDetectionAgent
{ From agents }
Attributes
private int m_kWidth = 400
private int m_kHeight = 400
private DFAgentDescription dfd
Operations
protected void setup( )
protected void takeDown( )
public Java3d getJ3d( )

10.2 Java3D

Toda la parte relativa a la detección de colisiones con Java 3D se encuentra dentro del paquete *es.ucm.fdi.collisionDetection*. En él podemos encontrar las siguientes clases que pasamos a comentar brevemente.

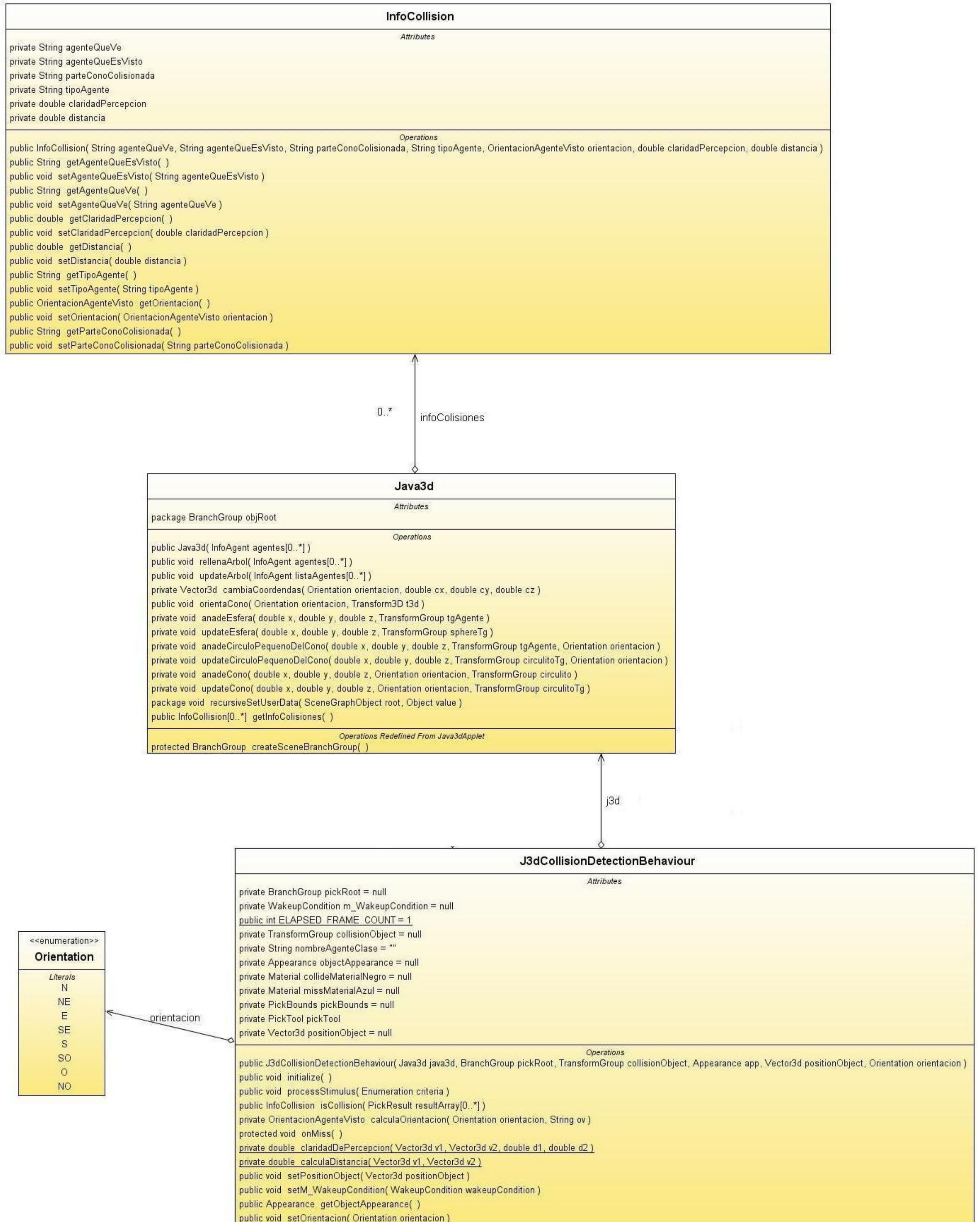
La clase principal es la que recibe el nombre *Java3D*. En ella creamos el árbol de escena y todos los nodos que lo componen. También en esta clase se actualiza cada uno de los nodos en cada una de las iteraciones (generación de nuevas coordenadas). La información relativa a los agentes que se encuentran en el entorno le llega encapsulada en objetos *InfoAgent*, que guardan el nombre del agente, las coordenadas absolutas y su orientación.

La clase *Java3D* hereda de la clase *Java3dApplet* que es la que se encarga de "pintar" la GUI de Java 3D, creada inicialmente con el objetivo de comprobar que se detectaban correctamente las colisiones, pero que finalmente ha pasado a ser una funcionalidad más de la aplicación.

Una de las clases más importantes es *J3dCollisionDetectionBehaviour*. Es clase hereda de la clase *Behaviour* de Java 3D, y se encarga de, en cada iteración, detectar qué colisiones se han producido en el entorno virtual y procesarlas. La detección y procesamiento de las colisiones se lleva a cabo en las funciones *processStimulus* e *isCollision*.

En caso de que se haya detectado colisión entre el *Focus* de un agente y el *Nimbus* de otro, se devuelve un objeto de la clase *InfoCollision*. Esta clase, como su nombre indica, guarda información relativa a la colisión: agente que ve, agente que es visto, parte del *Focus* colisionada, tipo de agente percibido, claridad con la que se le ha percibido, distancia a la que se encuentra del agente y su orientación. La orientación puede tomar ocho valores almacenados en el enumerado *Orientation*.

Java3dApplet
Attributes
public int m_kWidth = 300
public int m_kHeight = 300
protected String m_szCommandLineArray[0..*] = null
protected VirtualUniverse m_Universe = null
protected BranchGroup m_SceneBranchGroup = null
protected Bounds m_ApplicationBounds = null
Operations
public Java3dApplet( )
public boolean isApplet( )
public URL getWorkingDirectory( )
public VirtualUniverse getVirtualUniverse( )
public Locale getFirstLocale( )
protected Bounds getApplicationBounds( )
protected Bounds createApplicationBounds( )
protected Background createBackground( )
public void initJava3d( )
protected void onDoneInit( )
protected double getScale( )
public TransformGroup[0..*] getViewTransformGroupArray( )
protected void addViewBranchGroup(Locale locale, BranchGroup bg)
protected Locale createLocale(VirtualUniverse u)
protected BranchGroup createSceneBranchGroup( )
protected View createView(ViewPlatform vp)
protected PhysicalBody createPhysicalBody( )
protected AudioDevice createAudioDevice(PhysicalEnvironment pe)
protected PhysicalEnvironment createPhysicalEnvironment( )
protected float getViewPlatformActivationRadius( )
protected ViewPlatform createViewPlatform( )
protected Canvas3D createCanvas3D( )
protected int getCanvas3dWidth(Canvas3D c3d)
protected int getCanvas3dHeight(Canvas3D c3d)
protected double getBackClipDistance( )
protected double getFrontClipDistance( )
protected BranchGroup createViewBranchGroup(TransformGroup tgArray[0..*], ViewPlatform vp)
protected void addCanvas3D(Canvas3D c3d)
protected VirtualUniverse createVirtualUniverse( )
protected void saveCommandLineArguments(String szArgs[0..*])
protected String[0..*] getCommandLineArguments( )
Java3d
Attributes
package BranchGroup objRoot
Operations
public Java3d(InfoAgent agentes[0..*])
public void rellenaArbol(InfoAgent agentes[0..*])
public void updateArbol(InfoAgent listaAgentes[0..*])
private Vector3d cambiaCoordendas(Orientation orientacion, double cx, double cy, double cz)
public void orientaCono(Orientation orientacion, Transform3D t3d)
private void anadeEsfera(double x, double y, double z, TransformGroup tgAgente)
private void updateEsfera(double x, double y, double z, TransformGroup sphereTg)
private void anadeCirculoPequenoDelCono(double x, double y, double z, TransformGroup tgAgente, Orientation orientacion)
private void updateCirculoPequenoDelCono(double x, double y, double z, TransformGroup circulitoTg, Orientation orientacion)
private void anadeCono(double x, double y, double z, Orientation orientacion, TransformGroup circulito)
private void updateCono(double x, double y, double z, Orientation orientacion, TransformGroup circulitoTg)
package void recursiveSetUserData(SceneGraphObject root, Object value)
public InfoCollision[0..*] getInfoColisiones( )
Operations Redefined From Java3dApplet
protected BranchGroup createSceneBranchGroup( )
agentes
0..*
InfoAgent
Attributes
private String nombreAgente
private double x
private double y
private double z
Operations
public InfoAgent(String nombreAgente, double x, double y, double z, Orientation orientacion)
public InfoAgent( )
public String getNombreAgente( )
public void setNombreAgente(String nombreAgente)
public double getX( )
public void setX(double x)
public double getY( )
public void setY(double y)
public double getZ( )
public void setZ(double z)
public Orientation getOrientacion( )
public void setOrientacion(Orientation orientacion)
orientacion
<<enumeration>>
Orientation
Literals
N
NE
E
SE
S
SO
O
NO



InfoCollision
Attributes
private String agenteQueVe
private String agenteQueEsVisto
private String parteConoColisionada
private String tipoAgente
private double claridadPercepcion
private double distancia
Operations
public InfoCollision(String agenteQueVe, String agenteQueEsVisto, String parteConoColisionada, String tipoAgente, OrientacionAgenteVisto orientacion, double claridadPercepcion, double distancia)
public String getAgenteQueEsVisto( )
public void setAgenteQueEsVisto(String agenteQueEsVisto)
public String getAgenteQueVe( )
public void setAgenteQueVe(String agenteQueVe)
public double getClaridadPercepcion( )
public void setClaridadPercepcion(double claridadPercepcion)
public double getDistancia( )
public void setDistancia(double distancia)
public String getTipoAgente( )
public void setTipoAgente(String tipoAgente)
public OrientacionAgenteVisto getOrientacion( )
public void setOrientacion(OrientacionAgenteVisto orientacion)
public String getParteConoColisionada( )
public void setParteConoColisionada(String parteConoColisionada)
0..*
infoColisiones
Java3d
Attributes
package BranchGroup objRoot
Operations
public Java3d(InfoAgent agentes[0..*])
public void rellenaArbol(InfoAgent agentes[0..*])
public void updateArbol(InfoAgent listaAgentes[0..*])
private Vector3d cambiaCoordendas(Orientation orientacion, double cx, double cy, double cz)
public void orientaCono(Orientation orientacion, Transform3D t3d)
private void anadeEsfera(double x, double y, double z, TransformGroup tgAgente)
private void updateEsfera(double x, double y, double z, TransformGroup sphereTg)
private void anadeCirculoPequenoDelCono(double x, double y, double z, TransformGroup tgAgente, Orientation orientacion)
private void updateCirculoPequenoDelCono(double x, double y, double z, TransformGroup circulitoTg, Orientation orientacion)
private void anadeCono(double x, double y, double z, Orientation orientacion, TransformGroup circulito)
private void updateCono(double x, double y, double z, Orientation orientacion, TransformGroup circulitoTg)
package void recursiveSetUserData(SceneGraphObject root, Object value)
public InfoCollision[0..*] getInfoColisiones( )
Operations Redefined From Java3dApplet
protected BranchGroup createSceneBranchGroup( )
j3d
J3dCollisionDetectionBehaviour
Attributes
private BranchGroup pickRoot = null
private WakeupCondition m_WakeupCondition = null
public int ELAPSED_FRAME_COUNT = 1
private TransformGroup collisionObject = null
private String nombreAgenteClase = ""
private Appearance objectAppearance = null
private Material collideMaterialNegro = null
private Material missMaterialAzul = null
private PickBounds pickBounds = null
private PickTool pickTool
private Vector3d positionObject = null
Operations
public J3dCollisionDetectionBehaviour(Java3d java3d, BranchGroup pickRoot, TransformGroup collisionObject, Appearance app, Vector3d positionObject, Orientation orientacion)
public void initialize( )
public void processStimulus(Enumeration criteria)
public InfoCollision isCollision(PickResult resultArray[0..*])
private OrientacionAgenteVisto calculaOrientacion(Orientation orientacion, String ov)
protected void onMiss( )
private double claridadDePercepcion(Vector3d v1, Vector3d v2, double d1, double d2)
private double calculaDistancia(Vector3d v1, Vector3d v2)
public void setPositionObject(Vector3d positionObject)
public void setM_WakeupCondition(WakeupCondition wakeupCondition)
public Appearance getObjectAppearance( )
public void setOrientacion(Orientation orientacion)
orientacion
<<enumeration>>
Orientation
Literals
N
NE
E
SE
S
SO
O
NO

10.3 Colisión Matemática

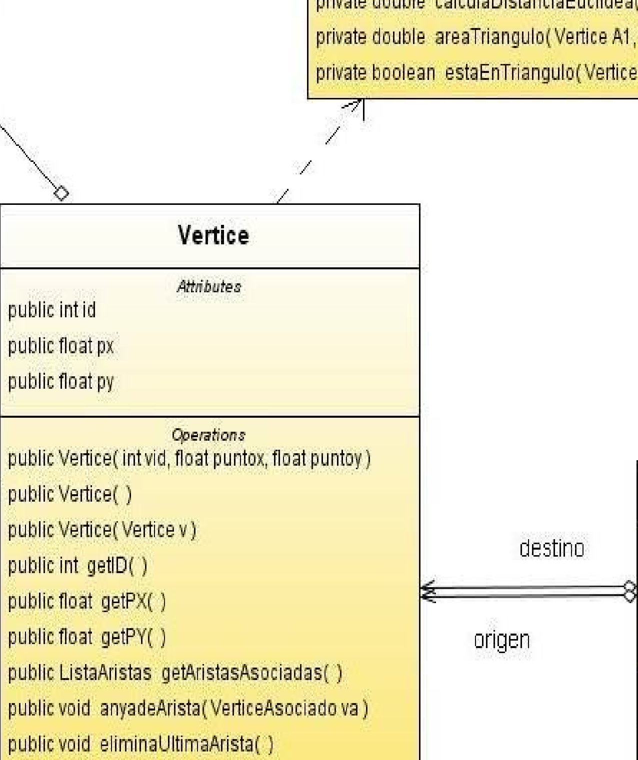
Este conjunto de clases se encuentra dentro del paquete *es.ucm.fdi.collisionDetection.mathCollision* y su objetivo es el de resolver las "falsas colisiones". Como se comentó con anterioridad, el cono de visión se encuentra recubierto por una *BoundingBox*, que no es sino una caja alrededor del cono, y es la que detecta las colisiones. Hay por tanto puntos de la caja que no se encuentran dentro del cono, y para evitar que colisiones con esos puntos induzcan a tomar decisiones equivocadas nos hemos visto obligados a implementar esta detección de colisiones mediante cálculos geométricos.

Gracias a la orientación del cono y a las coordenadas de su centro, podemos obtener las coordenadas de los vértices que lo forman (tratado en realidad como un triángulo en 2D).

La clase *ListaAristas* permite gestionar de una manera adecuada y eficiente las aristas asociadas a un determinado vértice.

Por otro lado, la clase *Recta* usará los vértices origen y destino para hallar la ecuación de la recta que une ambos puntos. Cada vértice almacena información de la coordenadas x e y que forman el punto, un número de identificación y una lista de aristas asociadas a dicho vértice.

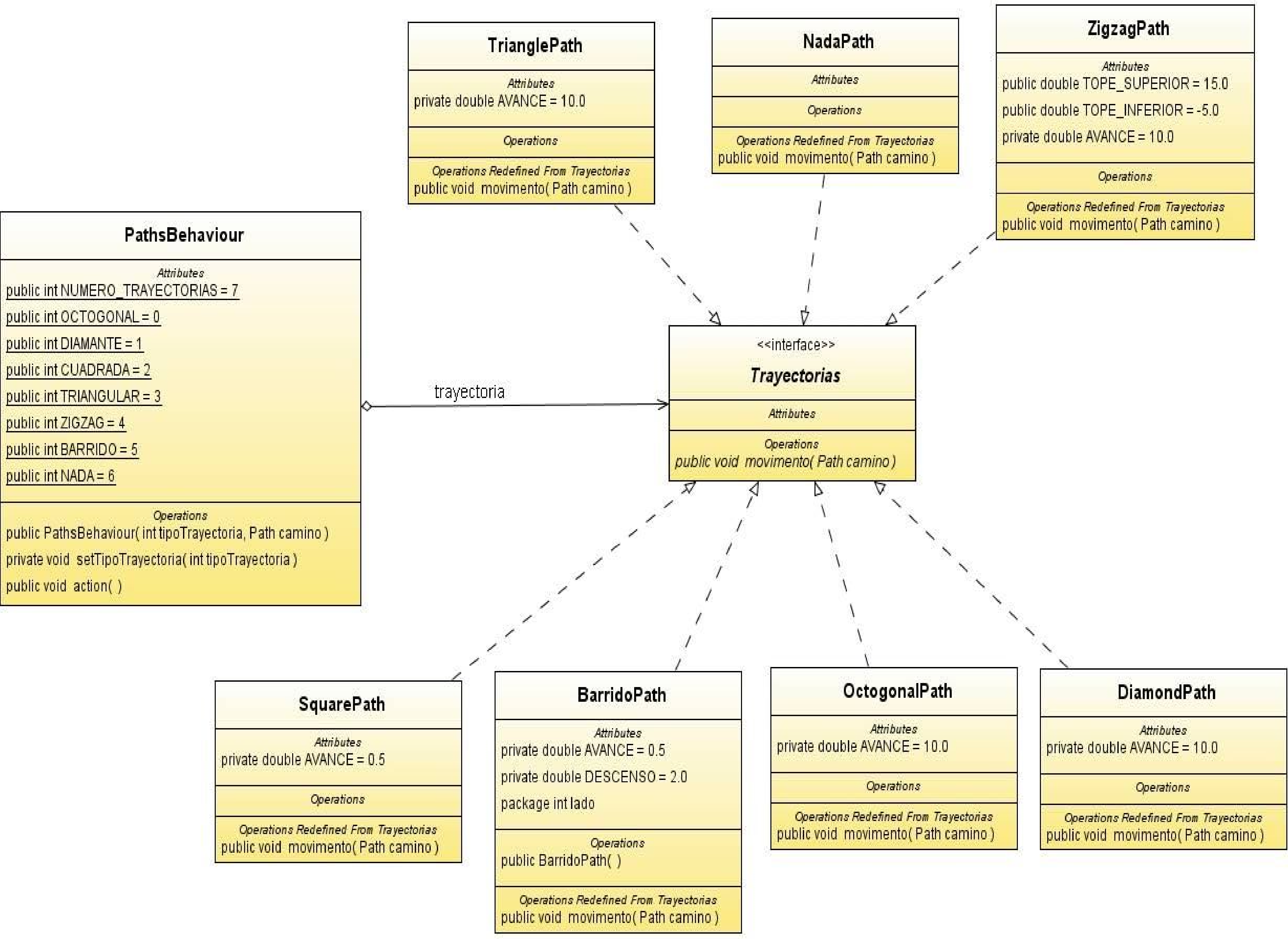
La clase *MathCollision* se encarga (en caso de que haya habido colisión) de calcular la recta que une los centros del cono y la esfera, y calcular los puntos de intersección con las aristas que forman el triángulo. Si alguno de estos puntos se encuentra dentro del cono, entonces comprobamos si la distancia entre este punto y el centro del círculo, o la distancia entre este punto y alguno de los vértices del cono es menor que el radio del círculo, y si es así ha habido colisión.


VerticeAsociado
Attributes
public int vertice
public float peso
Operations
public VerticeAsociado(int idVertice, float distancia)
public int getVertice( )
public float getPeso( )
MathCollision
Attributes
public double DISTANCIA1 = 3.54
public double DISTANCIA2 = 7.07
Operations
private Vector3d[0..*] dameVerticesCono(Orientation or, Vector3d centro)
public String detectaColision(Orientation or, Vector3d centroCono, Vector3d centroCirculo)
private String determinaOrientacion(Vertice v2, Vertice v3, Recta r2, Vertice cco, Vertice interR2)
private double calculaDistanciaEuclidea(Vertice v1, Vertice v2)
private double areaTriangulo(Vertice A1, Vertice A2, Vertice A3)
private boolean estaEnTriangulo(Vertice A1, Vertice A2, Vertice A3, Vertice P)
ListaAristas
Attributes
public Vector verticesAsociados
Operations
public ListaAristas( )
public ListaAristas(ListaAristas la)
public void anyadeVerticeAsociado(VerticeAsociado arista)
public void borraUltimaArista( )
public Vector getVerticesAsociados( )
public int lengthVerticesAsociados( )
public VerticeAsociado getVerticeAsociado(int p)
aristasAsociadas
Vertice
Attributes
public int id
public float px
public float py
Operations
public Vertice(int vid, float puntox, float puntoy)
public Vertice( )
public Vertice(Vertice v)
public int getID( )
public float getPX( )
public float getPY( )
public ListaAristas getAristasAsociadas( )
public void anyadeArista(VerticeAsociado va)
public void eliminaUltimaArista( )
destino
origen
Recta
Attributes
public float x
public float y
public float c
Operations
public Recta(Orientation or)
public Recta(Recta r, Orientation or)
public Recta(Vertice porigen, Vertice pdestino, Orientation or)
public void ecuacionGeneral(Orientation or)
public Float interseccionRectas(Recta r)
public float distancia(Float p)
public Vertice getP1( )
public Vertice getP2( )

10.4 Trayectorias

Se ha usado un patrón *Strategy* para las trayectorias que pueden seguir los agentes. De este modo se facilita en gran medida la inclusión o borrado de trayectorias para los agentes.

Todas las trayectorias disponen de una constante que indica cuánto van a avanzar los agentes cada vez que se muevan. Así se puede hacer que la velocidad con la que se desplazan aumente o disminuya.


TrianglePath
Attributes
private double AVANCE = 10.0
Operations
Operations Redefined From Trayectorias
public void movimento(Path camino)
NadaPath
Attributes
Operations
Operations Redefined From Trayectorias
public void movimento(Path camino)
ZigzagPath
Attributes
public double TOPE_SUPERIOR = 15.0
public double TOPE_INFERIOR = -5.0
private double AVANCE = 10.0
Operations
Operations Redefined From Trayectorias
public void movimento(Path camino)
PathsBehaviour
Attributes
public int NUMERO_TRAYECTORIAS = 7
public int OCTOGONAL = 0
public int DIAMANTE = 1
public int CUADRADA = 2
public int TRIANGULAR = 3
public int ZIGZAG = 4
public int BARRIDO = 5
public int NADA = 6
Operations
public PathsBehaviour(int tipoTrayectoria, Path camino)
private void setTipoTrayectoria(int tipoTrayectoria)
public void action( )
trayectoria
<<interface>>
Trayectorias
Attributes
Operations
public void movimento(Path camino)
SquarePath
Attributes
private double AVANCE = 0.5
Operations
Operations Redefined From Trayectorias
public void movimento(Path camino)
BarridoPath
Attributes
private double AVANCE = 0.5
private double DESCENSO = 2.0
package int lado
Operations
public BarridoPath( )
Operations Redefined From Trayectorias
public void movimento(Path camino)
OctogonalPath
Attributes
private double AVANCE = 10.0
Operations
Operations Redefined From Trayectorias
public void movimento(Path camino)
DiamondPath
Attributes
private double AVANCE = 10.0
Operations
Operations Redefined From Trayectorias
public void movimento(Path camino)

10.5 Entorno Virtual

La organización de clases del entorno gráfico es muy sencilla.

La clase *Cono* es la encargada de crear el cono de visión de los agentes, y tiene como atributos todo lo necesario para hacer las llamadas que proporciona OSG para la construcción de conos. También hay atributos (*ejeARotar, gradosRotacion*) que permitirán realizar rotaciones al cono en un determinado eje.

Se han usado algunos tipos no básicos y que proporciona el OSG como son los tipos osg::Vec3 y osg::Vec4, que sirven para representar vectores de tres y cuatro componentes, usados comúnmente en vectores de posición y para el color. En esta clase, también se ha usado el tipo *PositionAttitudeTransform* (al que se le aplican fácilmente operaciones de traslación), que es justamente lo que devuelve el método *dibujarCono()* encargado de pintar el cono en la escena.

La clase *Esfera* tiene como objetivo crear y dibujar la esfera que rodea a los agentes y que representa el *Nimbus*. Como atributos tiene el centro de la esfera, el radio y el color. El método *dibujarEsfera()* es el encargado de pintar la esfera en la escena y devuelve un puntero al tipo *Geode*, proporcionado por OSG.

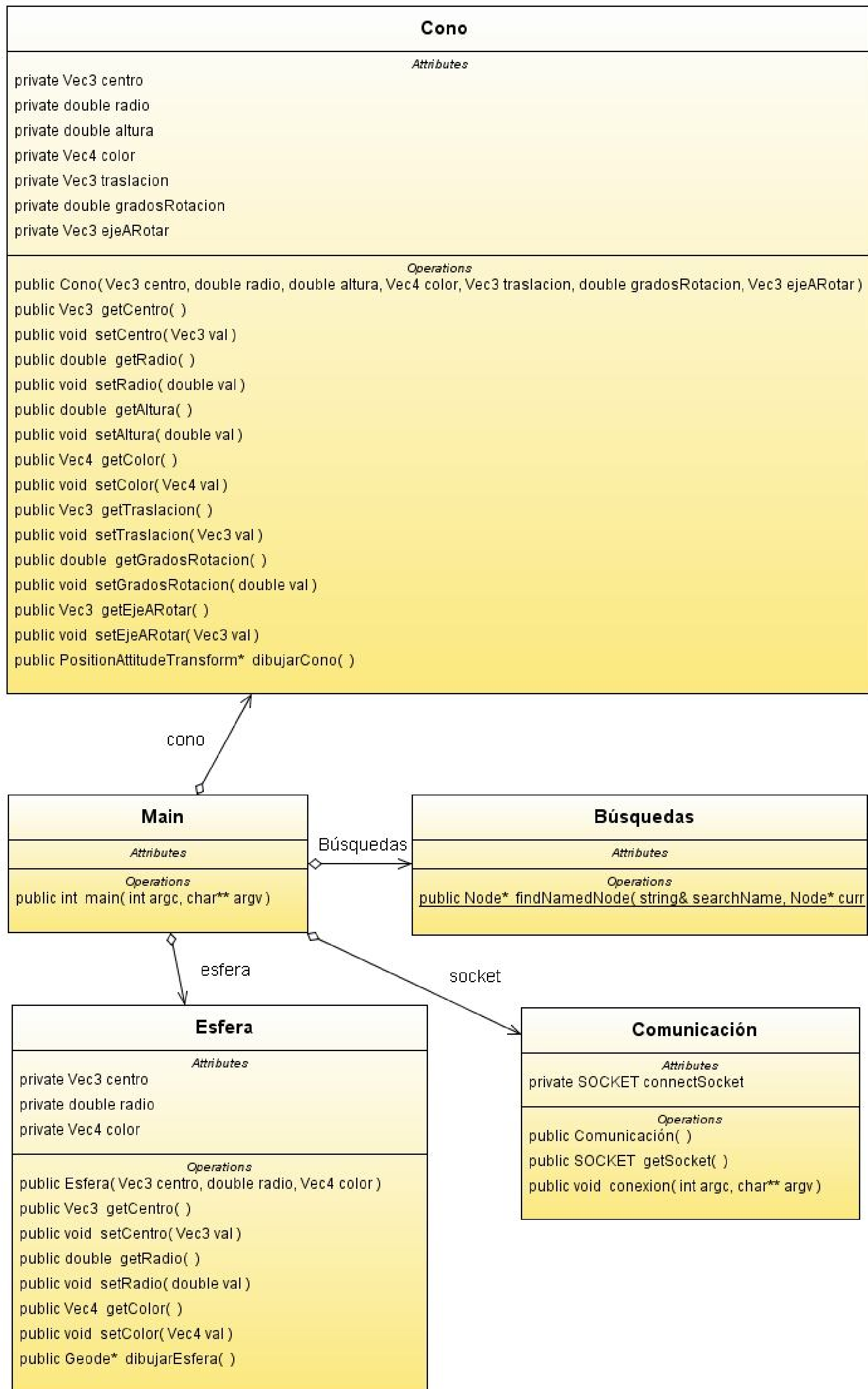
La clase *Búsquedas* tiene la función de proporcionar diferentes métodos de búsqueda de nodos dentro de un árbol de escena. Esto es necesario ya que en cada momento se desea mover un objeto en concreto y no todos los que forman parte de la escena. El método implementado necesita el nombre de un nodo en el árbol y la raíz del mismo.

La clase *Comunicación* es la encargada de crear el socket y de mandar la primera petición al servidor Java una vez se haya iniciado la parte gráfica. Sólo cuando se ha recibido dicha petición en Java comenzará el tráfico de datos desde el servidor Java hasta el cliente C++.

Por último, en la clase *Main* es dónde intervienen todas las clases descritas anteriormente. Primero se crea el socket y se empiezan a recibir los datos. En cuanto se ha reconocido el final de cadena (representado mediante un *) se procede a parsearla para extraer toda la información necesaria: número de gatos y de ratones, agente que hay que mover y su orientación y sus nuevas coordenadas de posición. Con toda esta información

almacenada se cargan los modelos 3D del gato y ratón, se crean los gatos y ratones, sus conos de visión y las esferas (*Nimbus)*.

A partir de ese momento lo único que se hará cada vez que se reciba una cadena será ver qué agente hay que desplazar, buscarlo en el árbol de escena y trasladarlo a las nuevas coordenadas. Cuando proceda, también será rotado, para que siempre mire a la dirección que indica su orientación.

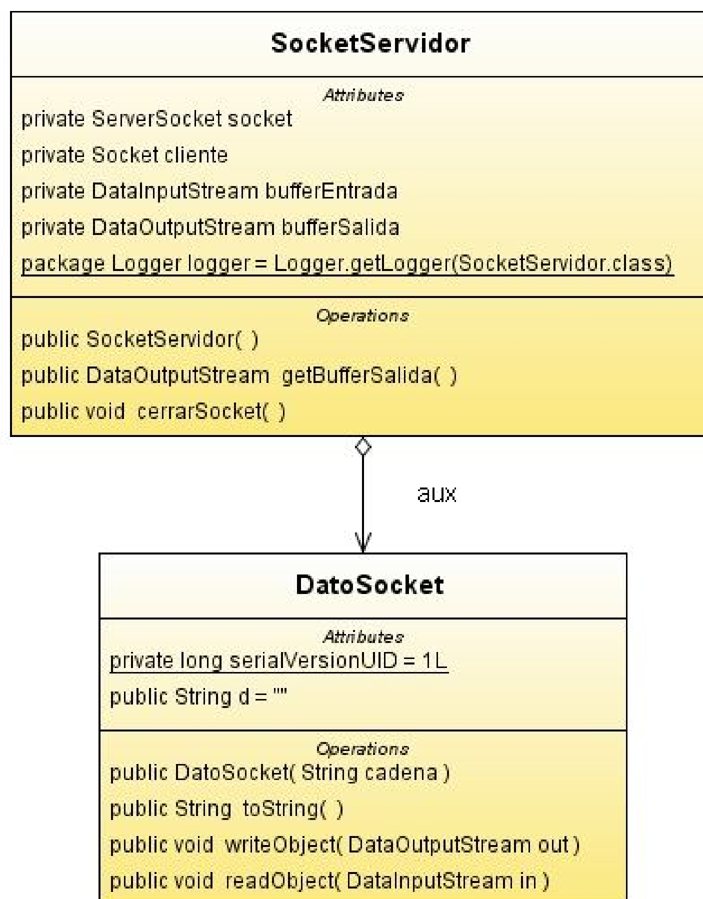

Cono
Attributes
private Vec3 centro
private double radio
private double altura
private Vec4 color
private Vec3 traslacion
private double gradosRotacion
private Vec3 ejeARotar
Operations
public Cono(Vec3 centro, double radio, double altura, Vec4 color, Vec3 traslacion, double gradosRotacion, Vec3 ejeARotar)
public Vec3 getCentro( )
public void setCentro(Vec3 val)
public double getRadio( )
public void setRadio(double val)
public double getAltura( )
public void setAltura(double val)
public Vec4 getColor( )
public void setColor(Vec4 val)
public Vec3 getTraslacion( )
public void setTraslacion(Vec3 val)
public double getGradosRotacion( )
public void setGradosRotacion(double val)
public Vec3 getEjeARotar( )
public void setEjeARotar(Vec3 val)
public PositionAttitudeTransform* dibujarCono( )
cono
Main
Attributes
Operations
public int main(int argc, char** argv)
Búsquedas
Búsquedas
Attributes
Operations
public Node* findNamedNode(string& searchName, Node* curr
esfera
socket
Esfera
Attributes
private Vec3 centro
private double radio
private Vec4 color
Operations
public Esfera(Vec3 centro, double radio, Vec4 color)
public Vec3 getCentro( )
public void setCentro(Vec3 val)
public double getRadio( )
public void setRadio(double val)
public Vec4 getColor( )
public void setColor(Vec4 val)
public Geode* dibujarEsfera( )
Comunicación
Attributes
private SOCKET connectSocket
Operations
public Comunicación( )
public SOCKET getSocket( )
public void conexion(int argc, char** argv)

10.6 Socket

Este diagrama UML muestra la organización de clases que conforman el socket. El agente de comunicación es el que crea el canal de comunicación mediante estos mecanismos.

La clase *DatoSocket* implementa la forma en la que se van a escribir y leer datos en el socket, mientras que la clase *SocketServidor* se encarga de estar a la escucha de peticiones de clientes (en este caso el entorno 3D) en el puerto establecido. Una vez establecida conexión usa los métodos de la clase *DatoSocket* para recibir y enviar datos.

SocketServidor también se encarga de cerrar el canal de comunicación cuando ya no se desea seguir enviando datos.



Capítulo 11. Prototipos realizados

11.1 Sockets

11.1.1 Primer prototipo cliente Java- servidor Java[10]

Ejemplo que establece un pequeño diálogo entre un programa servidor y sus clientes, que intercambian cadenas de información.

- Programa Cliente

El programa cliente se conecta a un servidor indicando el nombre de la máquina y el número puerto (tipo de servicio que solicita) en el que el servidor está instalado.

Una vez conectado, lee una cadena del servidor y la escribe en la pantalla.

- Programa Servidor

El programa servidor se instala en un puerto determinado, a la espera de conexiones, a las que tratará mediante un segundo *socket*.

Cada vez que se presenta un cliente, le saluda con una frase "Hola cliente N".

Este servidor sólo atenderá hasta tres clientes, y después finalizará su ejecución, pero es habitual utilizar bucles infinitos (*while(true)*) en los servidores, para que atiendan llamadas continuamente.

Tras atender cuatro clientes, el servidor deja de ofrecer su servicio.

- Ejecución

Aunque la ejecución de los *sockets* está diseñada para trabajar con ordenadores en red, en sistemas operativos multitarea (por ejemplo Windows y UNIX) se puede probar el correcto funcionamiento de un programa de *sockets* en una misma máquina.

---

[10] http://pisuerga.inf.ubu.es/lsi/Invest/Java/Tuto/V_2.htm

Cuando se lanza el cuarto de cliente, el servidor ya ha cortado la conexión, con lo que se lanza una excepción.

Observación: Tanto el cliente como el servidor pueden leer o escribir del *socket*.

11.1.2 Primer prototipo cliente C - servidor C

Se prueba en primer lugar un socket en C realizado por uno de los miembros del grupo en una asignatura cursada durante la carrera, y se comprueba su correcto funcionamiento.

11.1.3 Segundo prototipo cliente C - servidor C[11]

Se encuentra un ejemplo para C++Builder, que se decide probar por haber usado anteriormente esta herramienta.

Programa servidor que espera por una conexión, la acepta, hace eco de los datos que recibe (bytes), y luego vuelve a escuchar cuando el cliente termina la conexión.

Figura 11.1. Compilación del código

[11] http://tangentsoft.net/wskfaq/examples/basics/index.html

```
C:\WINDOWS\system32\cmd.exe - basic-server.exe 127.0.0.1 4242
Microsoft Windows XP [Versión 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\hlocal>cd cliente_servidor_C

C:\hlocal\cliente_servidor_C>basic-server.exe 127.0.0.1 4242
Establishing the listener...
Waiting for a connection...Accepted connection from 127.0.0.1:1748.
Received 136 bytes from client.
Sent 136 bytes back to client.
Connection closed by peer.
Shutting connection down...Connection is down.
Waiting for a connection...
```

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Versión 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\hlocal>cd cliente_servidor_C

C:\hlocal\cliente_servidor_C>basic-client.exe 127.0.0.1 4242
Looking up address...127.0.0.1:4242
Connecting to remote host...connected, socket 1952.
Sending echo packet (136 bytes)...
Reply packet matches what we sent!
Will shut down in 3 seconds... (one dot per second): ...
Shutting connection down...Connection is down.
All done!

C:\hlocal\cliente_servidor_C>_
```

Figura 11.2. Ejecución de prueba

11.1.4 Segundo prototipo cliente Java - servidor Java[12]

El servidor permanece a la espera enviando el String “Hola”, que es recibido por el cliente, el cual manda el String “Adios” al servidor, que una vez recibida la cadena de respuesta, lo notifica.

Nota: Este es el código Java que se usa definitivamente para el socket.

11.1.5 Tercer prototipo cliente C - servidor C[13]

Pruebas combinadas, que usan el código Java correspondiente a la segunda prueba (en el entorno de desarrollo Eclipse[14]) y el código C encontrado en MSDN (en el entorno de desarrollo Visual Studio 2005)

---

[12] http://www.chuidiang.com/java/sockets/
[13] Cliente: http://msdn2.microsoft.com/en-us/library/ms737591(VS.85).aspx, Servidor: http://msdn2.microsoft.com/en-us/library/ms737593(VS.85).aspx

- Cliente Java-Servidor C
- Cliente C-Servidor Java

Este código C es el que se utiliza finalmente en el proyecto.

11.2 JADE

El primer prototipo consistió en crear un agente asignándole un nombre y que lo mostrara por consola. Posteriormente se probó a matar el agente una vez finalizada su función. Después de haber entendido la creación y destrucción de agentes se procedió a añadir diferentes tipos de comportamientos para comprobar su efecto. Esto nos permitió poder elegir adecuadamente el tipo de comportamiento que tendrían los futuros agentes de la aplicación. Una vez se tuvo un agente con un comportamiento se procedió a comunicarlo con otro agente con otro comportamiento distinto mediante mensajería ACL. Simplemente se enviaban mensajes cuyo contenido era de tipo *String*. Por último se integró el servicio de Páginas Amarillas[15], donde se registró un agente ofreciendo un servicio, otro buscó agentes que ofrecieran ese servicio y por último le envió un mensaje.

Resumiendo, se puede decir que todo el desarrollo de la parte de agentes se llevó a cabo de forma incremental y muy bien organizada, empezando por lo básico y acabando por tener una estructura bastante compleja, teniendo varios agentes con distintos comportamientos y haciendo que interactúen entre ellos de la forma más eficiente posible para obtener el mejor resultado de la aplicación.

11.3 OSG

La idea inicial era empezar a desarrollar un entorno gráfico adecuado usando como base algunos ejemplos que incorporaba OSG.

Se comenzó modificando una demostración de una avioneta y una cometa sobre un tablero de ajedrez que giraban en la escena describiendo una circunferencia, para que realizaran otro tipo de movimiento y con otra velocidad. Una captura de este primer prototipo puede verse en la figura 11.3.

---

[14] Véase Anexo III.

[15] Véase sección 4.6 del capítulo 4

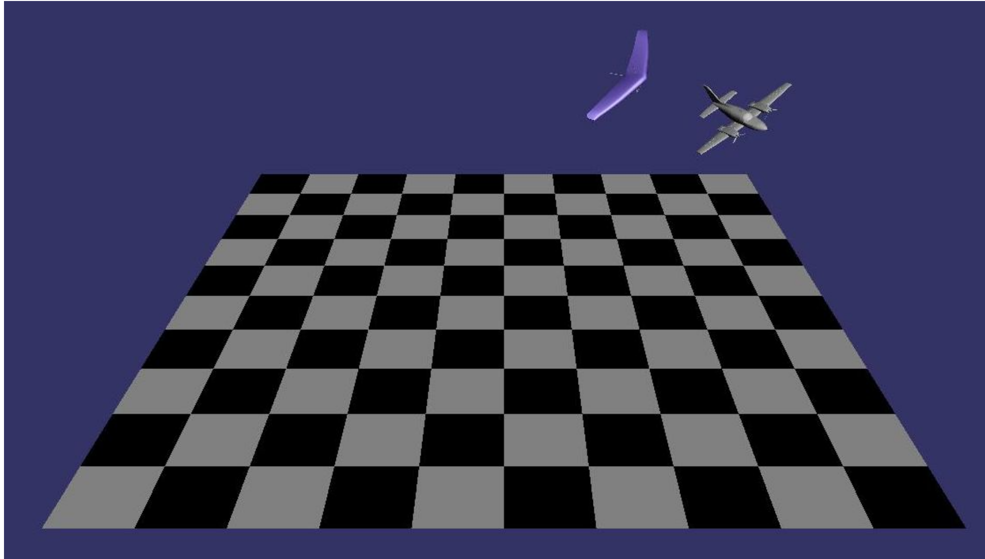


Figura 11.3. Ejemplo de una avioneta y una cometa sobre un tablero de ajedrez

Debido al desconocimiento de la aplicación y a lo complicado que resultaba entender el código se optó por algo más sencillo. La decisión fue empezar a desarrollar un entorno 3D en OSG desde cero, con ayuda de tutoriales y mucho más sencillo que las demostraciones incluidas en OSG. El objetivo fundamental era poder dibujar objetos en la escena, moverlos y rotarlos.

La primera aproximación fue dibujar una pirámide de base cuadrada, a la que se le aplicaron varios efectos de color.

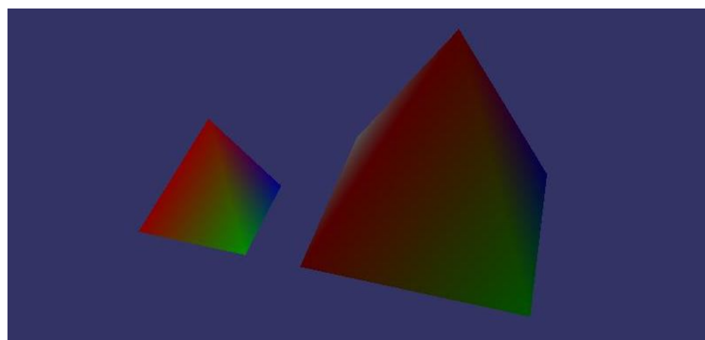


Figura 11.4. La primera aproximación fue dibujar una pirámide de base cuadrada

De la pirámide se pasó a otras formas geométricas muy básicas como son el cubo y la esfera.

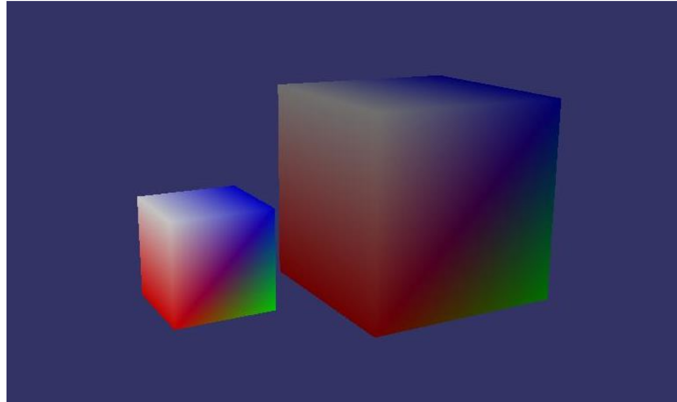


Figura 11.5. La segunda aproximación consistió en dibujar formas geométricas básicas como el cubo

Una vez se consiguió dominar el uso de formas geométricas se procedió a aplicar operaciones de traslación y rotación sobre los objetos.

Tras diversas pruebas satisfactorias se abordó la parte final, la cual consistió en sustituir las formas geométricas por modelos 3D. Internet fue el sitio donde se encontraron modelos 3D gratuitos de un gato y un ratón. Cuando todos los componentes de la escena estuvieron listos (cono de visión, esfera –*nimbus*-, modelo 3D del agente), se hicieron pruebas para comprobar que tanto las traslaciones como los giros se realizaban correctamente y que todos los elementos de un agente formaban parte de un todo, dando por fin la sensación de movimiento por la escena.

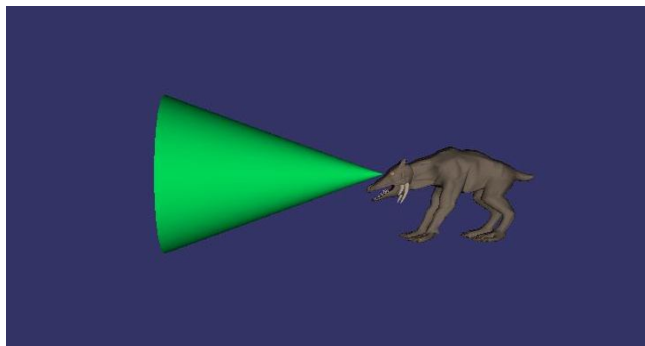


Figura 11.6. Aproximación con una figura que representa un agente, y su cono de visión

Figura 11.7. Modelo 3D escogido para representar los agentes Ratón

11.4 Java3D

Para llevar a cabo el módulo de detección de colisiones, como se ha comentado anteriormente, se optó por utilizar Java 3D.

Los primeros prototipos consistieron en utilizar ejemplos ya existentes en la Web con el objetivo de conocer las posibilidades que ofrecía la herramienta y como podíamos adecuarlas a los requerimientos de nuestro sistema.

El primero de los ejemplos probados fue *SimpleCollision*[16] que consistía en tres cubos, uno de los cuales podíamos moverlo con el ratón, de tal forma que al colisionar con el resto de cubos se mostraban mensajes indicando donde se han producido dichas colisiones.

[16] http://www.java-tips.org/other-api-tips/java3d/collision-detection-with-java3d-2-2.html

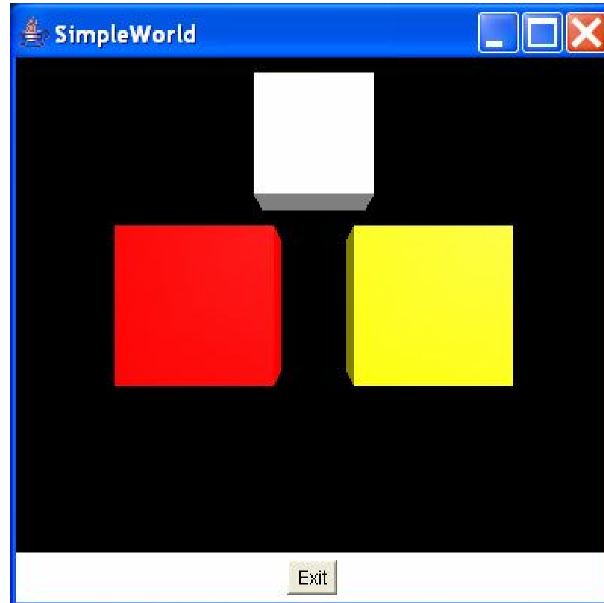


Figura 11.8 Ejemplo *SimpleCollision*

El segundo ejemplo que se probó fue *TickTockCollision*[17] que consistía en dos barras verticales, y un cubo que se iba moviendo de una barra a otra. Cuando el cubo tocaba una barra, esta cambiaba de color.

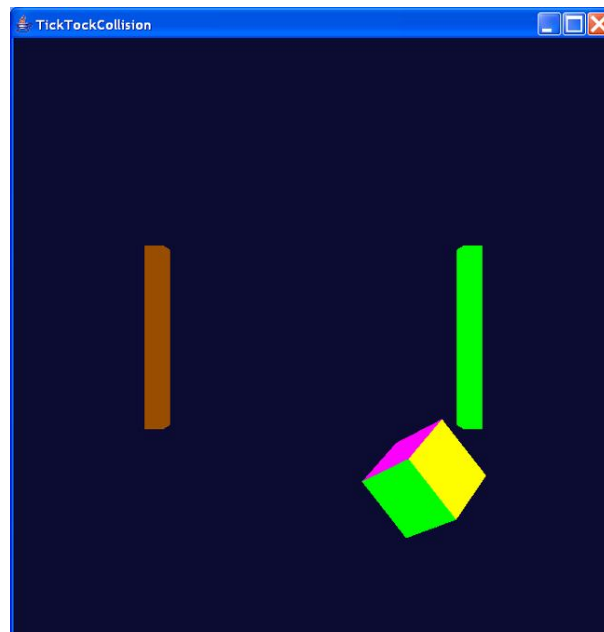


Figura 11.9 Ejemplo *TickTockCollision*

[17] http://www.java-tips.org/other-api-tips/java3d/collision-detection-with-java3d-3.html

El tercer ejemplo llamado *Spheres* consistía en una serie de esferas que iban moviéndose y rebotando con los límites del entorno.

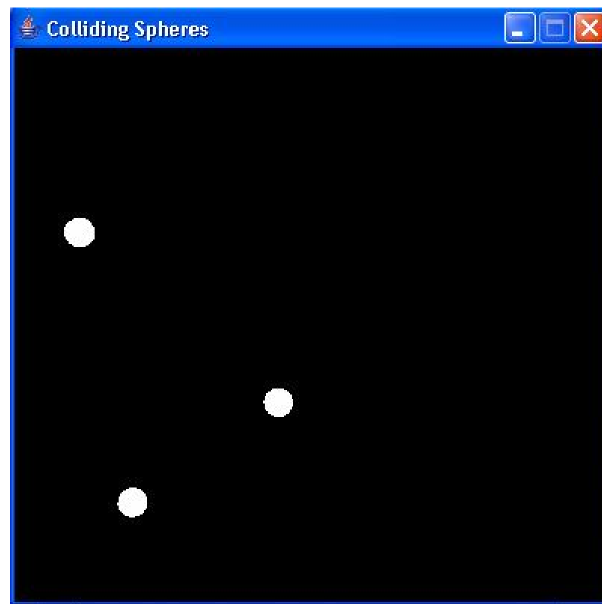


Figura 11.10 Ejemplo *Spheres*

El último ejemplo que mostramos *PickCollisionTest* es parecido al anterior, pero incorpora la detección entre esferas, de forma que rebotan cuando colisionan entre ellas. Este ejemplo es el que nos basamos para realizar los primeros prototipos.

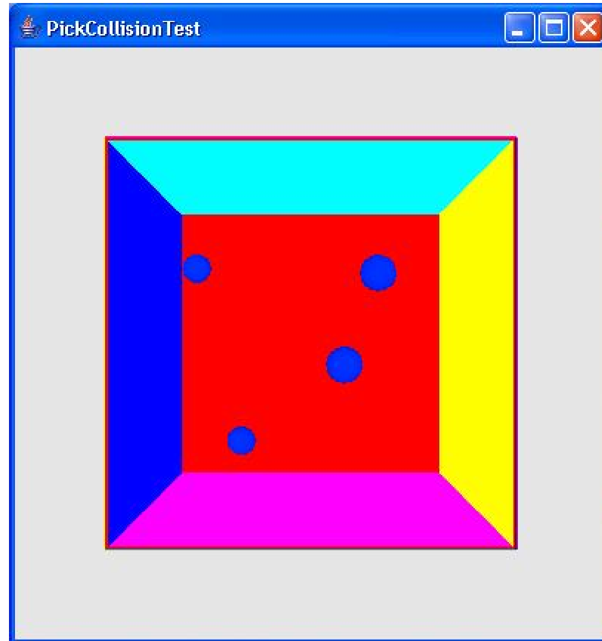


Figura 11.11 Ejemplo *PickCollisionTest*

El primer prototipo desarrollado consistió en un cono parado en el centro del entorno y una esfera moviéndose de izquierda a derecha. Cuando la esfera entraba en contacto con el cono, cambiaba de color. El cono representaría el *focus* o cono de visión mientras que la esfera correspondería al *nimbus*.

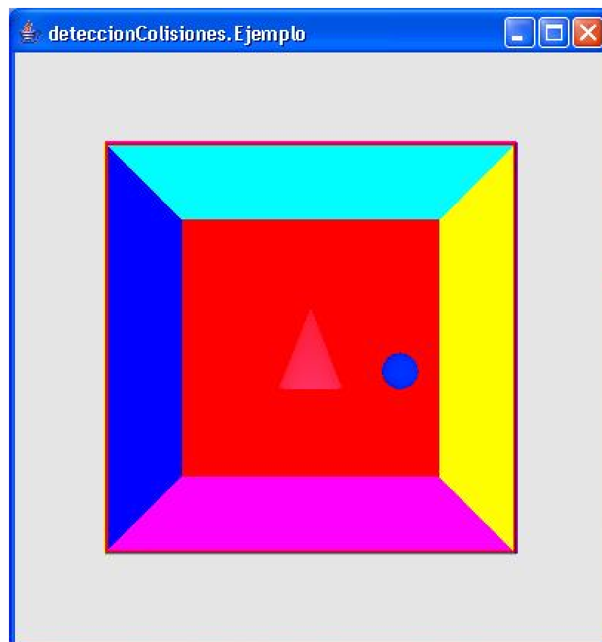


Figura 11.12 Primer prototipo

El segundo prototipo desarrollado consistió, basándonos en el anterior, en cambiar el color del cono cuando entraba en contacto con la esfera. Esto fue clave para darnos cuenta que el cono internamente estaba recubierto por una caja (*BoundingBox*) que era la que finalmente detectaba las colisiones.

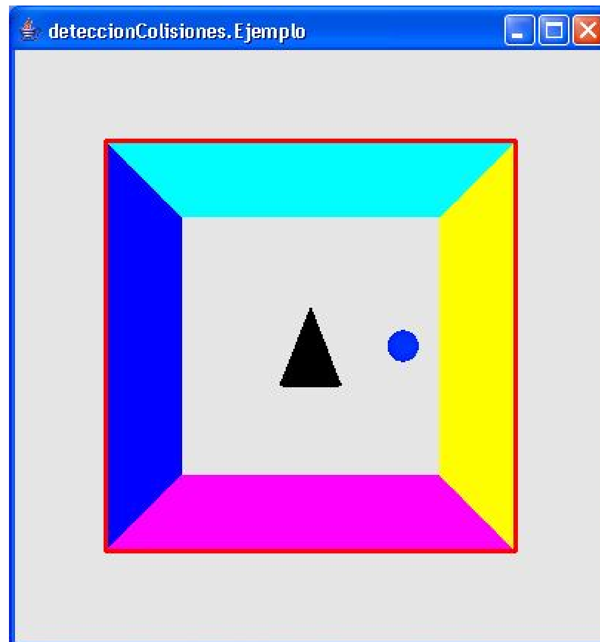


Figura 11.13 Segundo prototipo

Tras prototipos intermedios en los que se iban haciendo pruebas similares a estas y otras en las que barajaban distintas posibilidades para la detección de colisiones[18], finalmente el último prototipo incorpora detección de colisiones entre el *focus* de un agente y el *nimbus* de otro, teniendo en cuenta que no se deben detectar colisiones entre el *focus* y el *nimbus* del mismo agente.

[18] Véase Capítulo 6.

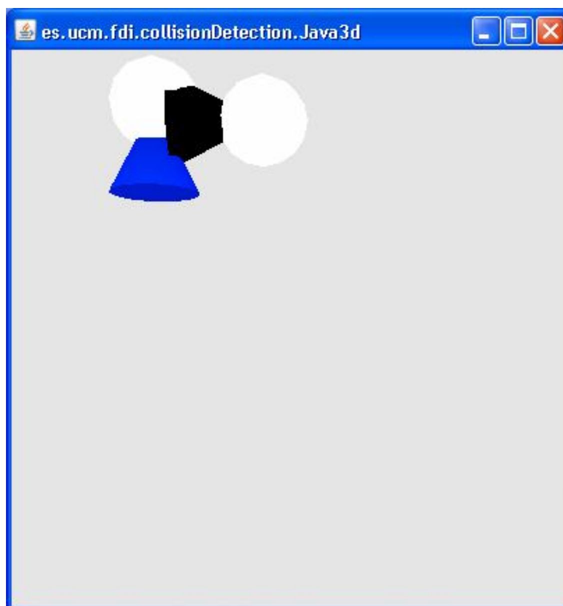


Figura 11.14 Prototipo final

Parte 3. Resultados

Capítulo 12. Análisis de Resultados

En este capítulo se pretende mostrar los resultados obtenidos por nuestra aplicación poniendo de manifiesto los módulos desarrollados así como la complejidad de algunos de ellos.

A continuación se muestran una serie de capturas en las que se pueden ver dos agentes (Gato y Ratón) en el entorno 3D OSG y la misma situación en la interfaz gráfica desarrollada en Java 3D con el objetivo de validar la detección de colisiones implementada.

En la figura 12.1 y 12.2 aparecen un agente Gato y agente Ratón junto a su *Focus* y su *Nimbus* en el entorno 3D realizado en OSG. En la figura 12.3 se representa la misma situación en el entorno 3D realizado en Java 3D.

Las figuras 12.4, 12.5 y 12.6 muestran la misma situación, la colisión entre el *Focus* de un agente Gato y el *Nimbus* de un agente Ratón, en los dos entornos 3D.

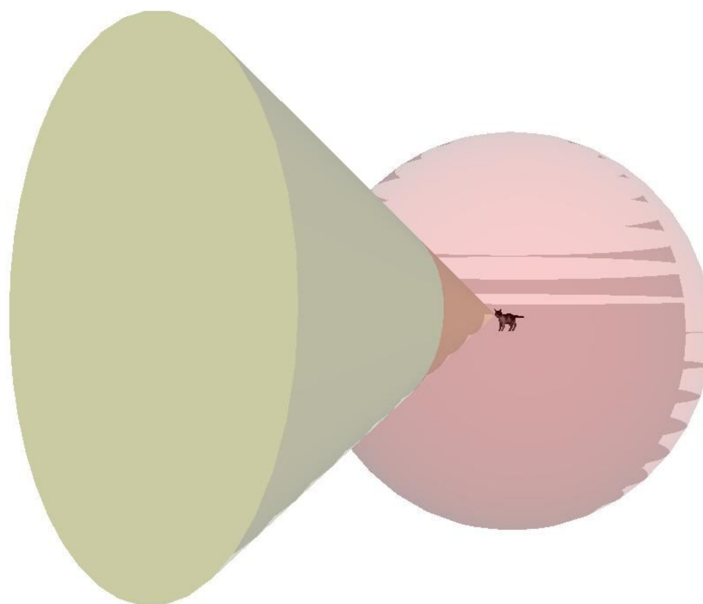


Figura 12.1 Agente Gato junto a su *Focus* y *Nimbus* en OSG

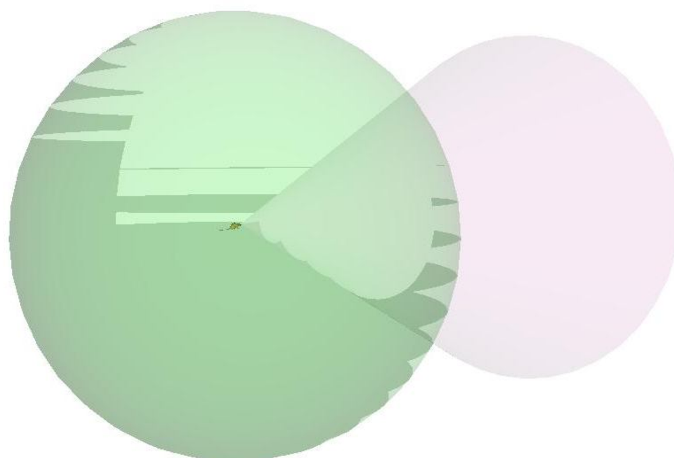


Figura 12.2 Agente Ratón junto a su *Focus* y *Nimbus* en OSG

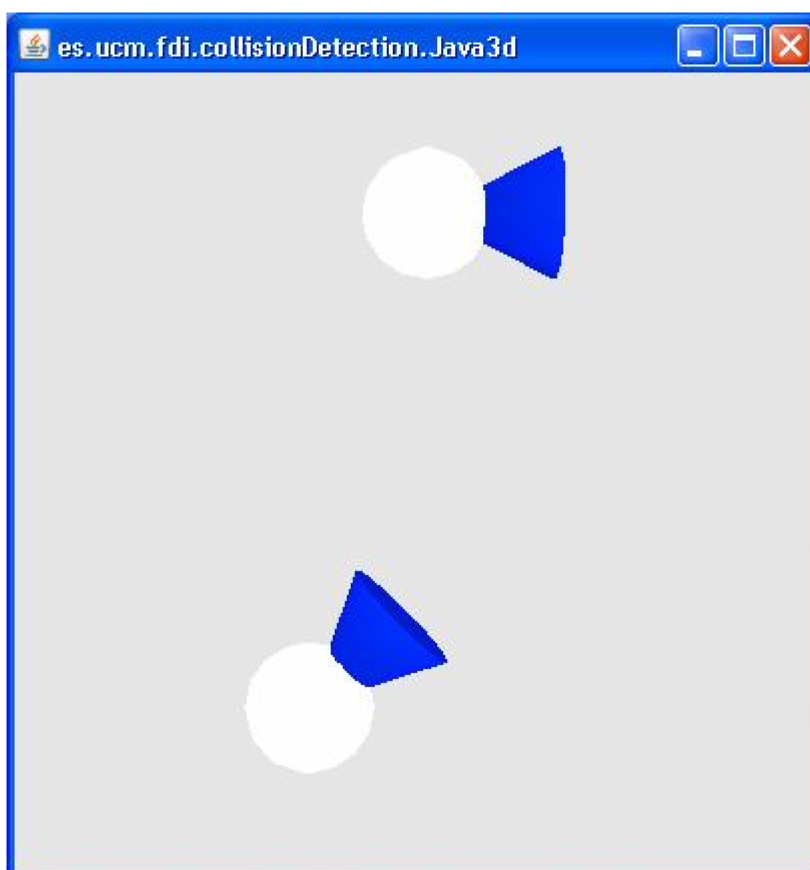


Figura 12.3 Agentes Gato y Ratón junto a sus *Focus* y *Nimbus* en Java 3D

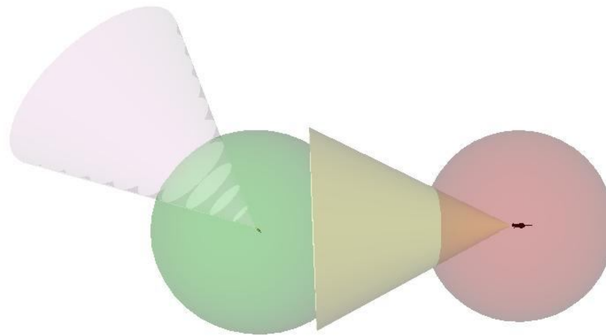


Figura 12.4 Colisión entre el *Focus* de un agente Gato y el *Nimbus* de un agente Ratón en OSG

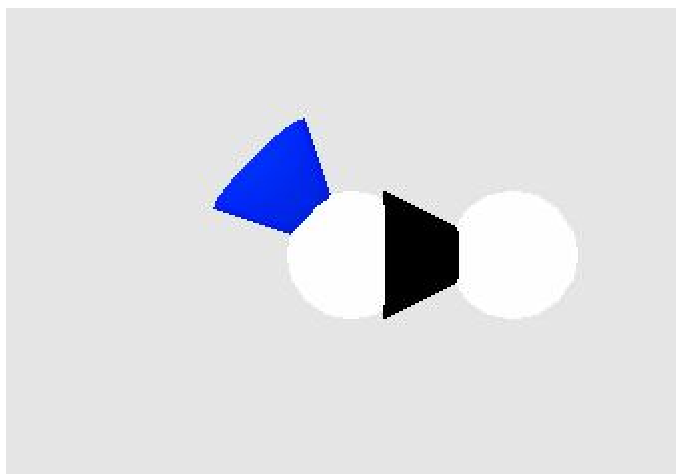


Figura 12.5 Misma colisión entre el *Focus* de un agente Gato y el *Nimbus* de un agente Ratón en Java 3D. El cono de color negro quiere decir que se ha detectado la colisión

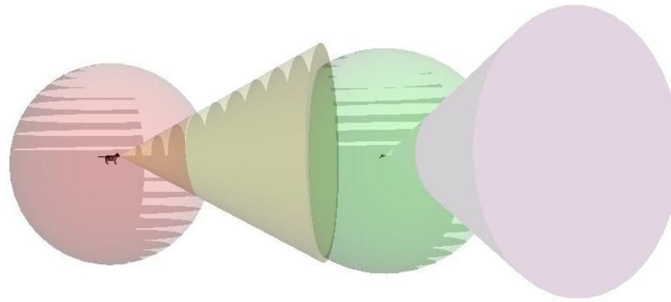


Figura 12.6 Misma colisión entre el *Focus* de un agente Gato y el *Nimbus* de un agente Ratón en OSG visto desde otra perspectiva

A continuación se muestran las pruebas realizados sobre los agentes Gato y Ratón. Han consistido en representar diferentes situaciones que se pueden dar en el entorno virtual creado para mostrar el comportamiento de los agentes, y ver las decisiones que tomaban.

Para cada prueba, se han realizado 20 iteraciones con los mismos valores de entrada para cada criterio. Con ello se pretende observar que en función de los valores de los criterios, unas veces la decisión tomada será determinista, pero esto no tiene por qué ocurrir siempre.

Los resultados siguientes pretenden representar situaciones en las que la decisión tomada se conoce de antemano, y ver si efectivamente, la decisión en base a la teoría MCDM se corresponde. En las pruebas cuya solución no es determinista, se explica cual es el motivo por el que esto ocurre, que no es otro que los valores que toman algunos de los criterios no se aproximan hacia una decisión en concreto, lo que sumado a la incertidumbre creada por los números triangulares *fuzzy*, desencadena en decisiones diferentes para los mismos valores de las entradas.

Todos las pruebas se han realizado con los mismos valores para los pesos, tal y como se refleja en la tabla 12.1.

	Tipo de agente	**Orientación**	**Claridad de percep.**	**Distancia**
Pesos	0.4	0.2	0.2	0.2

Tabla 12.1 Valores de los pesos para las pruebas

Resultados obtenidos con la toma de decisiones del agente Gato

- Resultado G1:

	Tipo de agente	Orientación	Claridad de percep.	Distancia
Criterios	Ratón	Frente	0.5	10

Tabla 12.2 Valores de los criterios para la prueba G1

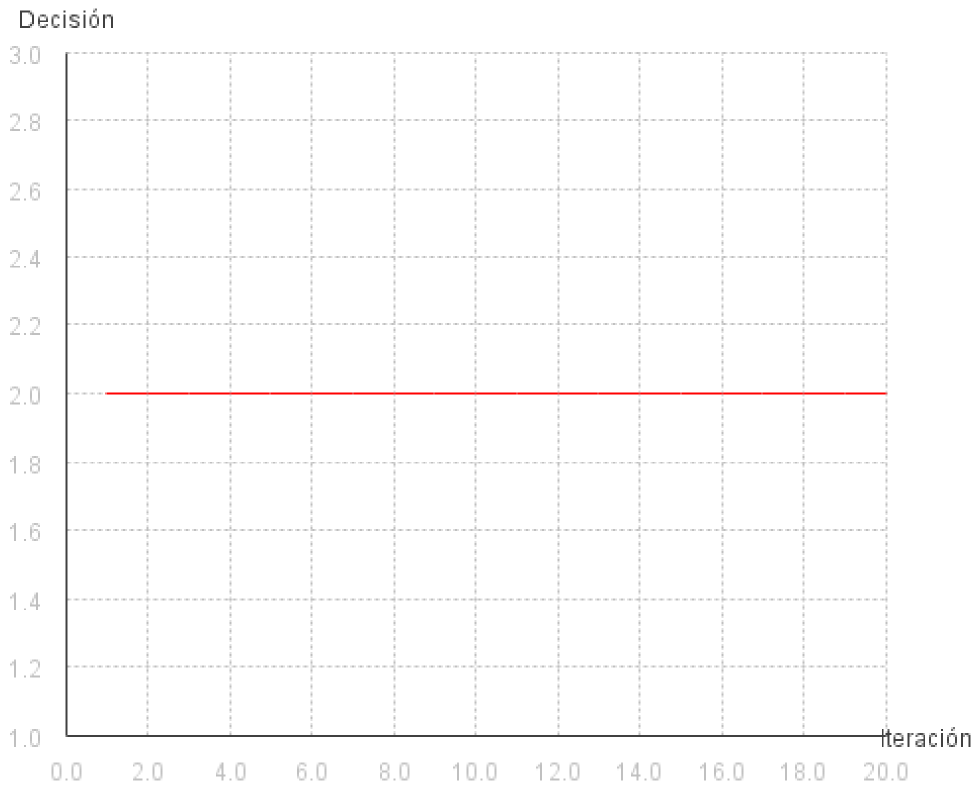


Figura 12.7 Decisiones tomadas para la prueba G1

Como podemos ver en la figura 12.7, para los valores de los criterios introducidos y los pesos asignados a ellos, la decisión ha sido la de *perseguir* al ratón en todos los casos. Para tomar esta decisión ha sido relevante el hecho de que el agente observado era un ratón a pesar de que la claridad de percepción era del 50% y el agente *Gato* se encontraba de

frente al gato. Esto es debido a que el peso asociado al criterio *tipo de agente* es del doble que el resto de criterios.

- Resultado G2:

	Tipo de agente	Orientación	Claridad de percep.	Distancia
Criterios	Ratón	Frente	0.0	15

Tabla 12.3 Valores de los criterios para la prueba G2

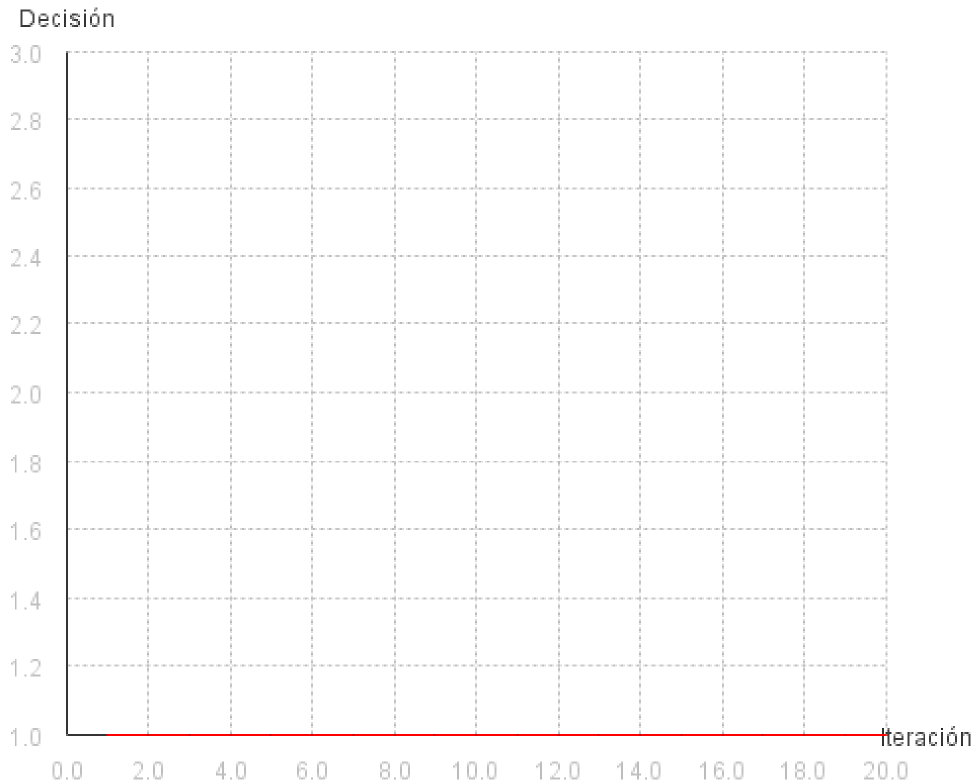


Figura 12.8 Decisiones tomadas para la prueba G2

Como podemos ver en la figura 12.8, para los valores de los criterios introducidos y los pesos asignados a ellos, la decisión ha sido la de *no hacer nada* en todos los casos. Para tomar esta decisión ha sido relevante el hecho de que el agente observado no era percibido con claridad.

- <u>Resultado G3:</u>

	Tipo de agente	**Orientación**	**Claridad de percep.**	**Distancia**
Criterios	Gato	Frente	1.0	5

Tabla 12.4 Valores de los criterios para la prueba G3

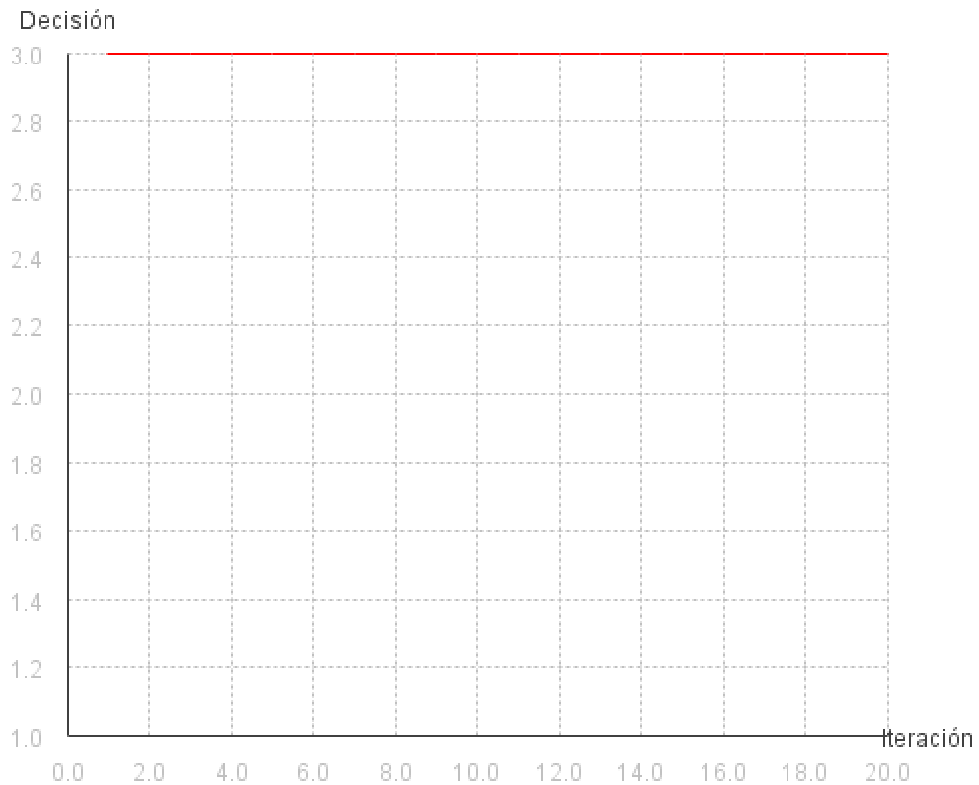


Figura 12.9 Decisiones tomadas para la prueba G3

Como podemos ver en la figura 12.9, para los valores de los criterios introducidos y los pesos asignados a ellos, la decisión ha sido *esquivar*. Ha influido en esta decisión que el tipo de agente observado fuera un gato, y que la claridad de percepción fuera máxima.

- Resultado G4

	Tipo de agente	Orientación	Claridad de percep.	Distancia
Criterios	Ratón	Frente	0.0000037	7

Tabla 12.5 Valores de los criterios para la prueba G4

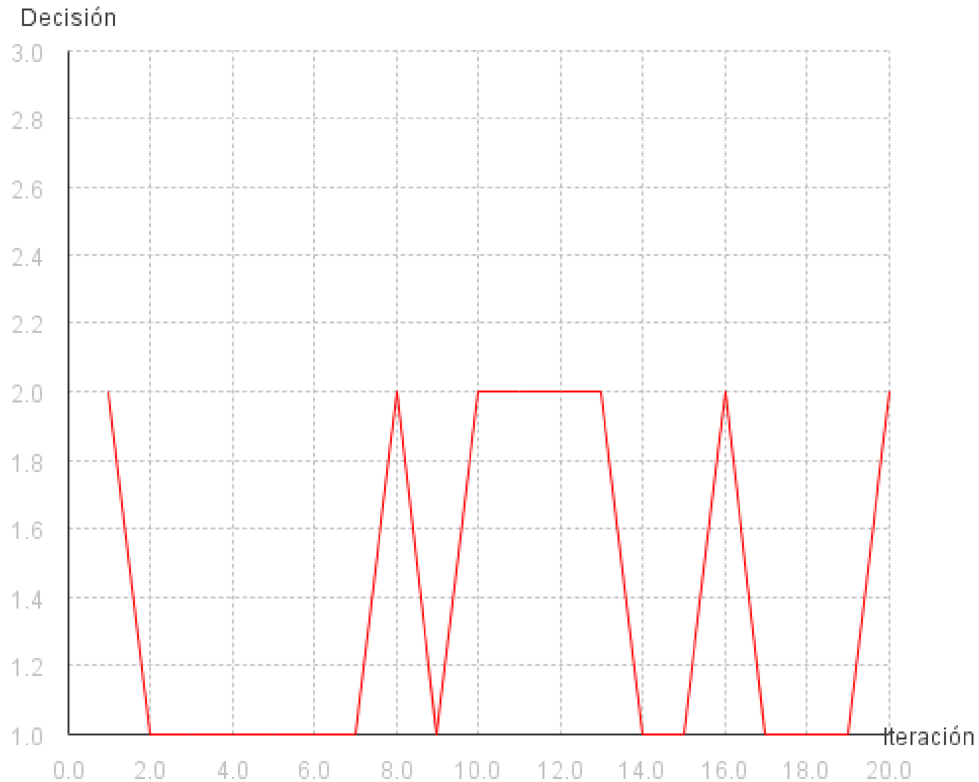


Figura 12.10 Decisiones tomadas para la prueba G4

Como podemos ver en la figura 12.10, para los valores de los criterios introducidos y los pesos asignados a ellos, la toma de decisiones no es determinista. De las 20 iteraciones realizadas, en 12 la decisión tomada es *no hacer nada*, mientras que en 8 es *perseguir*. El motivo por el que la decisión no es determinista se debe a que los valores que toman algunos de los criterios no se aproximan hacia una decisión en concreto, lo que sumado a la incertidumbre creada por los números triangulares *fuzzy*, desencadena en decisiones diferentes para los mismos valores de las entradas. En la práctica, esto es debido en gran parte a que la distancia al otro agente es grande y por tanto la claridad con la que lo percibe es muy pequeña. El hecho de que además el agente que está percibiendo se encuentre de

frente a él, influye para que la toma de decisiones se guíe hacia la decisión de no hacer nada en la mayoría de los casos, en vez de perseguir al agente que está percibiendo.

- Resultado G5:

	Tipo de agente	**Orientación**	**Claridad de percep.**	**Distancia**
Criterios	Gato	Frente	0.387	5.55

Tabla 12.6 Valores de los criterios para la prueba G5

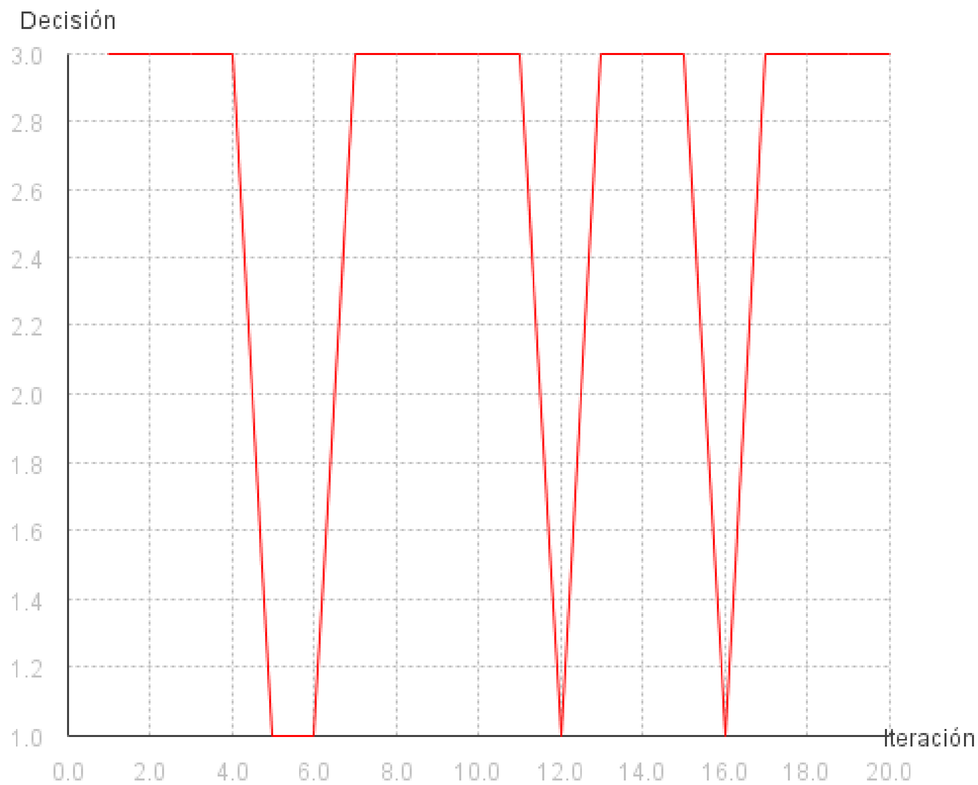


Figura 12.11 Decisiones tomadas para la prueba G5

Como podemos ver en la figura 12.11, para los valores de los criterios introducidos y los pesos asignados a ellos, la toma de decisiones no es determinista. En la mayoría de los casos la decisión tomada es *esquivar*, mientras que en 4 de las 20 iteraciones, la decisión es *no hacer nada*. El motivo por el que la decisión no es determinista se debe a que los valores que toman algunos de los criterios no se aproximan hacia una decisión en concreto,

lo que sumado a la incertidumbre creada por los números triangulares *fuzzy*, desencadena en decisiones diferentes para los mismos valores de las entradas. En la practica, podemos apreciar que con una claridad de percepción próxima al 40%, el agente sabe que está percibiendo un agente del mismo tipo, y que este viene de frente a él, pero como la distancia no es todavía ni muy próxima para esquivarle, ni muy lejana para no hacer nada, se produce la situación que hemos representado en esta prueba.

Resultados obtenidos con la toma de decisiones del agente Ratón

Los resultados obtenidos con la toma de decisiones del agente Ratón son muy similares a los del agente Gato, por lo que no se incluyen aquí. La única diferencia reside en que una de las tres posibles alternativas a tomar por un ratón es “huir” en lugar de “perseguir”. El resto de alternativas son equivalentes a la de los agentes Gato.

Capítulo 13. Conclusiones

En nuestro proyecto hemos tenido que afrontar y resolver cada uno de los riesgos tecnológicos con los que nos enfrentábamos inicialmente (Socket, Java 3D, JADE, OSG…). Dado que estas herramientas nos eran desconocidas, ha sido necesario investigar durante prácticamente todo el curso, limitando sensiblemente la parte de desarrollo final.

Se ha diseñado un entorno en el que se simulan una serie de situaciones basadas en las acciones y reacciones de agentes hostiles entre sí, de forma que se establecen comportamientos de persecución y huida como los mecanismos fundamentales de interacción entre los agentes.

Con el fin de modelar los agentes mediante un comportamiento más realista, se les dotó de un sistema de percepción visual, de tal forma que no pudieran conocer la situación de cada agente en el entorno, sino únicamente la de aquellos agentes que estaban en ese momento en su campo de visión.

El Plan de Fase realizado inicialmente ha sufrido cambios durante el proyecto, pero ha servido de referencia para no perder el rumbo, al intentar en la medida de lo posible cumplir con las fechas que nos íbamos imponiendo.

Aunque somos conscientes de los límites que tiene la aplicación final, haber podido resolver las dificultades encontradas (principalmente el desconocimiento de las herramientas), conseguir incorporar la detección de colisiones y la toma de decisiones en la plataforma de agentes, así como conectar ésta con el entorno 3D a través de un socket, ha sido para nosotros la aportación fundamental del proyecto.

Como contrapartida, el entorno 3D no es todo lo elaborado que hubiéramos deseado, nos hubiera gustado incorporar comunicación entre agentes y permitir interacción con el usuario a través de teclado, aunque todo esto es consecuencia de la gran labor de investigación que hemos tenido que llevar a cabo, realizando prototipos que finalmente no iban a ser útiles, barajando posibilidades que finalmente se descartaron, etc.

13.1 Cumplimiento de los objetivos propuestos

1. **Diseñar e implementar un sistema multiagente.**

Se ha implementado una plataforma de agentes con ayuda del *framework* para Java, JADE, que ha permitido establecer distintos roles y comportamientos para posibilitar la interacción de unos agentes con otros gracias al paso de mensajes ACL. Éstos desempeñan distintas tareas, las cuales ofrecen mediante un servicio de páginas amarillas. Los agentes Gato y Ratón serán los que se muevan por la escena, pero también hay otros agentes "invisibles" para el usuario pero no por ello carentes de importancia: el agente encargado del envío de coordenadas por el socket y el agente encargado de la detección de colisiones.

2. **Dotar a los agentes de la capacidad de percibir el entorno visualmente.**

Para ello se utilizó la herramienta Java 3D que nos permitió implementar el *focus* y *nimbus* de cada uno de los agentes que intervienen en el entorno, y detectar de una manera simple la colisión entre estos objetos (conos y esferas en la práctica). Se incorporó un módulo de detección de colisiones mediante cálculos geométricos que permite evitar errores en la detección de colisiones debidos a limitaciones inherentes a la propia herramienta. Estos se deben, por ejemplo, a que el cono de visión se encuentra recubierto por un rectángulo invisible que es el que detecta las colisiones. Por tanto, el rectángulo cubre zonas que no se corresponden con el cono, siendo necesario un mecanismo adicional con el objetivo de evitar errores en la detección.

3. **Dotarles de la capacidad de responder de manera inteligente a los sucesos del entorno.**

Esto en la práctica consistió en implementar un módulo de toma de decisiones propio para cada tipo de agente, que les permitiera "responder" a los estímulos que reciben del entorno.

El método de toma de decisiones empleado es el MCDM (*Multi-Criteria Decision Making*) bajo la perspectiva de la lógica *fuzzy*. Con ello se consigue introducir incertidumbre en un entorno simulado, donde el aspecto determinista de las decisiones se sustituye por una componente predominantemente aleatoria.

La toma de decisiones se lleva a cabo a partir de una serie de criterios: el tipo de agente percibido, la orientación con respecto al agente que lo percibe, la claridad con la que lo percibe y la distancia que los separa. Las posibles alternativas a tomar por los distintos tipos de agentes son:

Agente Gato: *no hacer nada / perseguir / esquivar*

Agente Ratón: *no hacer nada / huir / esquivar*

4. Diseñar e implementar un entorno virtual 3D.

Se ha realizado un entorno 3D sencillo compuesto por agentes y sus *focus* y *nimbus*, llevada a cabo gracias a la potencialidad de OSG, que con sus librerías proporciona medios para trabajar con modelos 3D y formas de adaptar las figuras geométricas necesarias (cono y esfera) a nuestras necesidades. Gran movilidad por el escenario, texturas opacas y transparentes…etc.

Nuestra aplicación dispone además de otra interfaz gráfica en 3D implementada haciendo uso de Java 3D.

5. Diseñar e implementar un sistema de comunicación entre la plataforma de agentes (Java) y el entorno 3D (C++).

También se ha cumplido este objetivo, convirtiendo nuestra aplicación en un sistema independiente de la interfaz gráfica desarrollada. El sistema de comunicación empleado fue un socket bidireccional, aunque el flujo principal es en un único sentido (Servidor Java – Cliente C++). Disponer de un socket bidireccional será de gran utilidad para desarrollar algunos de los trabajos futuros que se enumeran mas adelante.

Por tanto, se ha desarrollado una aplicación piloto, basada en agentes, que mediante el uso de técnicas de Inteligencia Artificial es capaz de responder de manera inteligente a los sucesos del entorno y perseguir metas de manera similar a como lo haría un humano.

Como objetivo adicional se pretendía que los componentes desarrollados fueran reutilizables y fácilmente modificables, de manera que pudieran ser utilizados en otras aplicaciones sin necesidad de implementarlos de nuevo.

Si queremos conseguir software reutilizable y extensible, es decir, modular, debemos hacer uso de algún método de diseño o de alguna técnica arquitectónica que favorezca la descomposición modular. Además se necesita que una vez hallados los subproblemas estos se puedan conectar entre si formando una estructura sencilla que sea la solución al problema de software mas general. Esta estructura sencilla la podemos conseguir entre otras cosas garantizando que la solución a cada subproblema sea lo mas independiente posible de las demás soluciones. En nuestro caso, la plataforma de agentes JADE nos ha facilitado esta labor. El mecanismo de percepción y toma de decisiones es un comportamiento JADE que podría ser fácilmente sustituido por otro. Es posible sustituir el método de toma de decisiones por otro método sin que afecte al resto de la aplicación. Lo mismo ocurre con el mecanismo de percepción visual incorporado en los agentes. Es posible sustituirlo por otro mecanismo de percepción, o incluso no sustituirlos, sino añadirle otro como, por ejemplo, percepción auditiva a los agentes. Esto se haría sin más que añadir un comportamiento nuevo a la plataforma de agentes desarrollada en JADE.

13.2 Trabajos futuros

Debido a la naturaleza de este proyecto y a las posibilidades que ofrece el campo de la Inteligencia Artificial, se pueden plantear numerosas mejoras y ampliaciones del mismo. Si bien podemos destacar algunas que resultarían bastante significativas:

- Añadir inteligencia a los agentes, ya que de momento sólo se guían mediante un campo de visión limitado, sin que hechos pasados tengan relevancia en sus decisiones. De esta forma, se podría tener un sistema multiagente en el que mediante tomas de decisiones conjuntas y la evaluación de casos anteriores se mejorara la utilidad de la aplicación, pudiéndose incluso aplicar a campos cuya complejidad empieza a ser un factor determinante.

- Un usuario pueda manejar uno o varios agentes mediante el teclado u otro periférico. Podría ser de gran utilidad a la hora de comprobar el grado de

inteligencia que tienen el resto de los agentes, su capacidad para aprender y el modo en que se relacionan con el entorno.

- Mejora del entorno 3D en OSG. Para ello, podría incluirse un escenario por el que moverse e interaccionar los agentes. Éstos, deberían incorporar nuevos criterios y quizá nuevas alternativas acordes a la nueva situación.
- Permitir comunicación/cooperación entre agentes. Añadir algoritmos de *Machine Learning* capacitaría a los agentes para aprender de situaciones anteriores. Al incorporar comunicación entre agentes de un mismo tipo, podrían desarrollarse estrategias de "caza" entre los agentes tipo Gato, por ejemplo.
- Mejora del sistema de percepción de los agentes. Estudiar la posibilidad de incluir, por ejemplo percepción auditiva (Herrero y De Antonio, 2003).

Parte 4. Anexos

Anexo A. Java 3D

A.1 Qué es Java 3D

La API (*Application Program Interface*) Java 3D es una interfaz de programación de aplicación utilizada para escribir aplicaciones y *applets* con gráficos en tres dimensiones.

Proporciona a los desarrolladores constructores de alto nivel para crear y manipular geometrías 3D y para construir las estructuras utilizadas en el renderizado de dichas geometrías. Se pueden describir grandes mundos virtuales utilizando estos constructores, que proporcionan a Java 3D la suficiente información como para renderizar dichos mundos de forma eficiente.

Java 3D proporciona a los desarrolladores de gráficos 3D la principal característica de Java: "escribe una vez y ejecútalo donde sea". Java 3D es parte del conjunto de APIs JavaMedia, lo cual hace que esté disponible en un gran número de plataformas. También, se integra correctamente con Internet ya que tanto los applets como las aplicaciones escritas utilizando Java 3D tienen acceso al conjunto completo de clases de Java.

La API Java 3D parte de otras APIs gráficas existentes así como de las nuevas tecnologías disponibles. Las construcciones gráficas de bajo nivel de Java 3D sintetizan las mejores ideas encontradas en otras APIs de bajo nivel como Direct3D, OpenGL, QuickDraw3D y XGL. Del mismo modo, sus construcciones de alto nivel integran las mejores características proporcionadas por varios sistemas de escenas basados en grafos. Además, Java 3D introduce algunos conceptos que no se consideran habitualmente como parte de los entornos gráficos, como el sonido espacial 3D. Las posibilidades de sonido de Java 3D permiten proporcionar una experiencia más realista al usuario.

Java 3D es una extensión estándar del JDK 2 de Java. Como se ha dicho, proporciona una colección de constructores de alto nivel para crear y manipular geometrías 3D y estructuras par dibujar esta geometría.

J3D se basa en objetos. Existen objetos para transformaciones, luces, sonidos, formas, sombras, etc. Éstos tienen una estructura de nodos y arcos. Un nodo se representa con un objeto de las clases J3D y un arco es una relación entre nodos.

A.2 Objetivos

Java 3D ha sido diseñado teniendo en cuenta diferentes objetivos; prestando especial atención en el buen rendimiento. Se tomaron diferentes decisiones relativas al diseño de tal forma que las implementaciones de Java 3D proporcionaran el mejor rendimiento posible a las aplicaciones de usuario. En particular, cuando se realizan distribuciones, se elige la alternativa que permite obtener mejores prestaciones en tiempo de ejecución.

Otros objetivos importantes de Java 3D son:

- Proporcionar un amplio conjunto de utilidades que permitan crear mundos en 3D interesantes. También se tuvo en cuenta no incluir características no esenciales u oscuras. No se incluyeron características que podrían colocarse directamente sobre Java 3D.
- Proporcionar un paradigma de programación orientado a objetos de alto nivel para permitir a los desarrolladores generar sofisticadas aplicaciones y *applets* de forma rápida.
- Proporcionar soporte a cargadores en tiempo de ejecución. Esto permite que Java 3D se adapte a un gran número de formatos de ficheros, como pueden ser formatos específicos de distintos fabricantes de CAD, formatos de intercambio o VRML 1.0 (*Virtual Reality Modelling Language*) y VRML 2.0.

A.3 Grafo de escena

Java 3D distribuye la información necesaria para representar objetos y escenas en tres dimensiones en una estructura de grafo. Siguiendo dicha estructura desde el nodo raíz hasta los distintos nodos hoja, se van viendo las distintas operaciones que se realizan para crear la escena final que se quiere conseguir. En dicho grafo se incluyen tanto los distintos elementos que forman parte de la escena como las transformaciones que se les aplica. Del

mismo modo, se insertan en el mismo grafo (aunque en otra rama distinta) diferentes elementos que tienen que ver con el punto de vista del usuario.

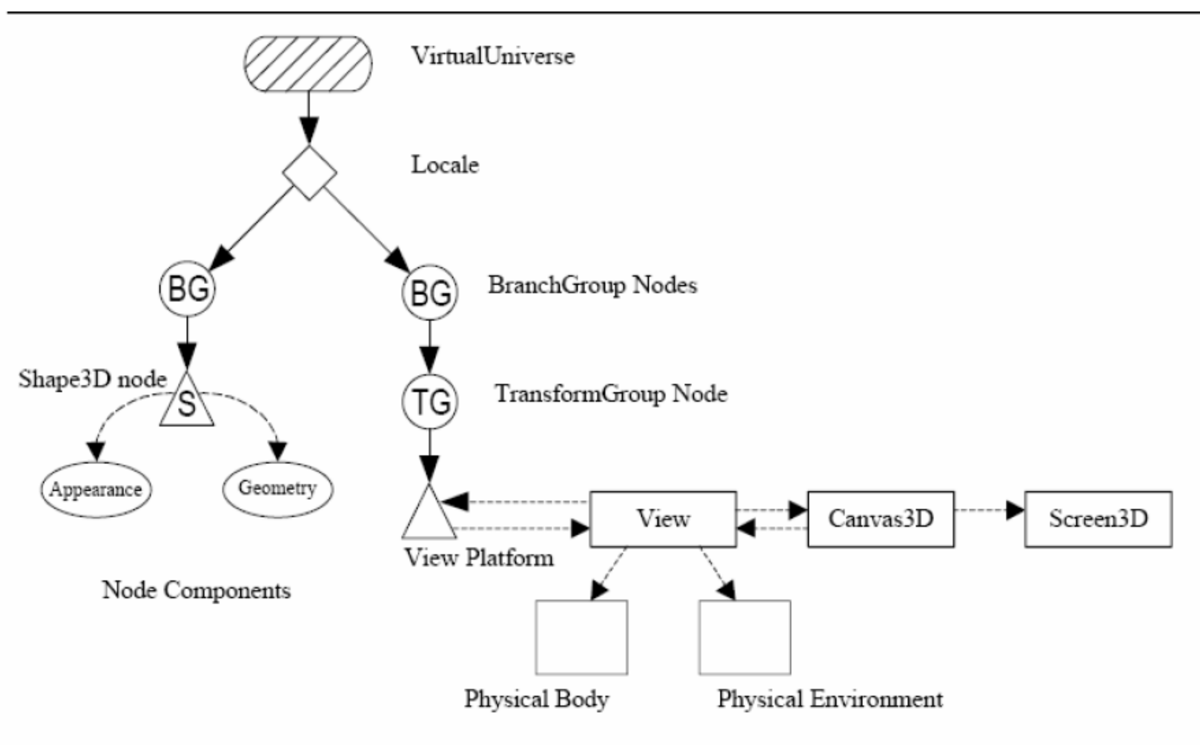


Figura A.1. Estructura general del grafo de escena

Al igual que en la figura A.1, el árbol necesario para generar imágenes en Java 3D va formando los distintos elementos de las mismas de manera progresiva, añadiendo en las diferentes ramas del árbol las características propias de la imagen que se genera. Por otro lado, la distribución de los elementos que forman la escena en una estructura jerárquica permite aislar convenientemente unos aspectos de otros para poder trabajar con ellos de forma más independiente.

A.4 Paradigma de programación

Java 3D es una API orientada a objetos. Las aplicaciones construyen los distintos elementos gráficos como objetos separados y los conectan unos con otros mediante una estructura en forma de árbol denominada *grafo de escena*. La aplicación manipula los

diferentes objetos utilizando los métodos de acceso, de modificación y de unión definidos en su interfaz.

A.4.1. Modelo de programación del grafo de escena

El modelo de programación basado en el grafo de escena de Java 3D proporciona un mecanismo sencillo y flexible para representar y renderizar escenas. El grafo de escena contiene una descripción completa de la escena o universo virtual. Esta descripción incluye datos sobre la geometría, información de los distintos atributos, así como información de visualización necesaria para renderizar la escena desde un punto de vista en particular.

La API Java 3D mejora algunas APIs previas eliminando algunas de las serias dependencias impuestas por éstas. Java 3D permite al programador diseñar su escena en base a objetos geométricos y no considerando triángulos. Le permite centrarse en la escena y en su composición y no en cómo escribir el código para renderizar eficientemente la escena en cuestión.

A.4.2. Aplicaciones y applets

Java 3D ni anticipa ni soporta directamente cualquier necesidad posible de 3D. En cambio, proporciona la base necesaria para añadir nuevas funcionalidades utilizando código Java. Objetos definidos utilizando un sistema de CAD o de animación se pueden incluir en una aplicación basada en Java 3D. Algunos paquetes modelados tienen distintos formatos externos que son, en muchos casos, propietarios. Los diseñadores pueden exportar geometrías a fichero utilizando modeladores externos. Java 3D puede utilizar toda esa información geométrica, pero sólo si la aplicación proporciona un método para leer y traducir la información del modelador en primitivas Java 3D.

De forma similar, cargadores VRML analizarán y traducirán ficheros VRML y generarán los objetos Java 3D apropiados y el código Java necesario para poder utilizar el contenido de los ficheros.

- **Navegadores**. Los navegadores actuales pueden aceptar contenido 3D pasando información a visores 3D (*plugins*) que la renderizan dentro de su propia ventana. Con el tiempo, serán los propios navegadores los que se encarguen, directamente, de la gestión de los elementos 3D.

- **Juegos**. Los desarrolladores de software para juegos 3D siempre han intentado aprovechar, en la medida de lo posible, hasta el límite las posibilidades del hardware disponible. Históricamente han utilizado optimizaciones específicas del hardware y, por supuesto, no portables. De esta forma trataban de lograr el mejor rendimiento posible. Por esta razón, la programación típica de juegos se realizaba a un nivel de abstracción menor que el sencillo nivel de Java 3D. Sin embargo, la tendencia actual en la creación de juegos 3D es usar aceleradores hardware 3D de propósito general y utilizar menos "trucos" para la renderización.

Así pues, aunque Java 3D no fue explícitamente diseñado para satisfacer las expectativas de los desarrolladores de juegos, sus sofisticadas técnicas de implementación proporcionan un rendimiento más que suficiente para realizar distintas aplicaciones para juegos. Puede criticarse, de cualquier forma, que las aplicaciones escritas con una API general, como puede ser Java 3D, pueden tener una ligera penalización en cuanto a rendimiento en comparación con otras que utilicen técnicas no portables. Sin embargo, otros factores como la propia portabilidad, el tiempo de desarrollo y el coste pueden también tenerse en cuenta para contrarrestar ese pequeño déficit de rendimiento.

A.5 Instalar Java 3D

Para poder usar Java3D, se necesita tener instalado el **J2SE** Software Development Kit (SDK). Si no lo tiene instalado bájelo de la página de Sun e instálelo: http://java.sun.com/javase/downloads/index.jsp

Para instalar Java 3D en un sistema Windows es necesario además tener correctamente instalada alguna de las APIs gráficas sobre las que corre Java 3D, actualmente DirectX u OpenGL. Podrá encontrar una versión actualizada de DirectX para su sistema en

http://www.microsoft.com/windows/directx/downloads/. Así mismo visite la página

http://www.opengl.org/ para obtener una versión de OpenGL actualizada.

Nota: *Es recomendable que obtenga e instale apropiadamente la última versión de los controladores de su dispositivo de vídeo.*

A.6 Obtener Java 3D

Puede encontrar la última versión de Java 3D en http://java3d.j3d.org/download.html.

Elija la versión Java 3D apropiada para su sistema.

A.7 Proceso de instalación

Ejecute el fichero de instalación que acaba de descargar y siga las instrucciones de la pantalla. Se le preguntará durante este proceso la ubicación de la máquina virtual de Java que desea utilizar, indique la ubicación correcta. Si instaló el SDK de Java, deberá indicar la ubicación del JRE y el SDK de Java por separado.

Anexo B. JADE (Java Agent DEvelopment Framework)

JADE (Java Agent Development) es una estructura software que simplifica el desarrollo de sistemas multiagente de acuerdo con las especificaciones FIPA para la interoperabilidad de sistemas de agentes inteligentes.

Es un software libre y de código abierto, que ha sido desarrollado por el CSELT (Centro Studi e Laboratori Telecomunicación) del grupo Telecom Italia, en parte dentro del proyecto de investigación europeo ACTS AC17 "FACTS".

Tanto JADE, como los agentes que el usuario define para una aplicación específica, utilizan el lenguaje de desarrollo JAVA, lo que aporta a la plataforma una total independencia del sistema o sistemas operativos empleados.

La plataforma de agentes puede estar distribuida entre distintas máquinas y su configuración puede ser cambiada en cualquier momento, ya que JADE permite la movilidad y clonación de agentes de una máquina a otra.

JADE, siguiendo las especificaciones FIPA, implementa aquellos aspectos de un sistema multiagente, que no son particularidades internas del agente y son independientes del tipo de aplicación. Con esta premisa, JADE nos proporciona la siguiente estructura base (figura B.1):

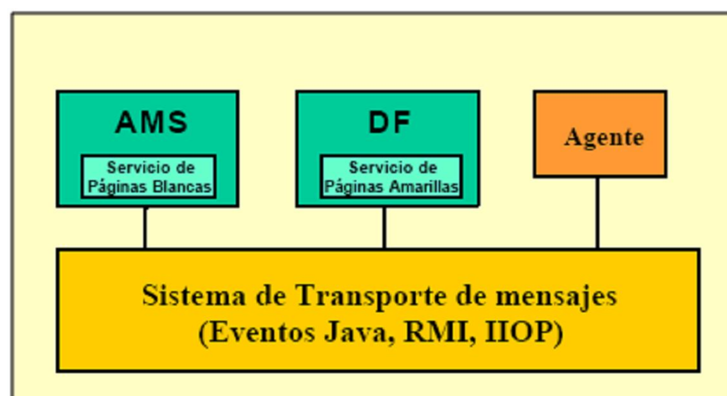


Figura B.1. Estructura básica de JADE.

Gestión de los agentes, que consiste fundamentalmente en la identificación, registro, ubicación, y estado de los agentes, así como en el registro de los servicios específicos que

realizan los agentes de la aplicación. Los dos tipos de agentes encargados de realizar estas funciones son:

- *AMS (Agent Management System)* es el agente que ejerce el control de supervisión sobre el acceso y uso de la plataforma de agentes. Proporciona, entre otros, el servicio de páginas blancas y ciclo de vida de los agentes, manteniendo el directorio de identificador de agentes y el estado de los agentes.
- *DF (Directory Facilitator)* es el agente que proporciona el servicio de páginas amarillas a otros agentes.

De acuerdo con las especificaciones FIPA, los agentes AMS y DF se comunican usando el lenguaje FIPA-SL0 y la ontología fipa-agentmanagement, ambos también están implementados en JADE.

El Sistema de Transporte de Mensajes, también ll*amado ACC (Agent Communication Channel)* es el componente software que controla todo el intercambio de mensajes dentro de la plataforma, incluyendo los mensajes hacia y desde plataformas remotas. El sistema es totalmente transparente al usuario, cuando un agente quiere enviar un mensaje basta con utilizar los métodos predefinidos y es JADE quien automáticamente selecciona la vía más adecuada para transportar ese mensaje, así, si los agentes emisor y receptor están ejecutándose sobre la misma máquina virtual de JAVA, utilizará los eventos JAVA, si están en distintas máquinas virtuales, RMI y si están en distintas plataformas, IIOP.

La *clase Agent,* contiene las estructuras básicas para que los agentes realicen interacciones esenciales con la plataforma (registro, configuración, administración, etc.) y un conjunto básico de métodos para personalizar las tareas del agente (envío y recepción de mensajes, protocolos de interacción estándar para la construcción de las conversaciones entre agentes, etc.)

Con todo esto, el diseñador de la aplicación sólo tiene que centrarse en determinar las características particulares de los agentes (objetivos, tareas a realizar) para la aplicación específica que se está desarrollando, esto se compone de dos partes esenciales:

- Definición de las Tareas específicas de los agentes, para desarrollar un agente el programador parte de la *clase* base, *Agent*. Cada agente JADE esta compuesto de

un único hilo de ejecución y cada tarea, funcionalidad, servicio o intención de un agente debe ser implementado mediante uno o varios *comportamientos*, que son unidades lógicas de actividad que pueden ser formadas de diversos modos (cíclicas, secuenciales, deterministas, no deterministas,...) para alcanzar complejos patrones de ejecución. JADE, de forma automática y transparente para el programador, gestiona los comportamientos, soportando la ejecución concurrente y paralela de múltiples tareas mediante una política de round-robin.

- Definición de la ontología u ontologías específicas de la aplicación, donde se engloben todos los conceptos, acciones, predicados y proposiciones que definen el dominio de la aplicación.

Por consiguiente, utilizar JADE como plataforma de partida, reduce drásticamente el esfuerzo que supone el diseño y desarrollo de un sistema multiagente. Por otra parte JADE es de fácil instalación y utilización, asimismo proporciona una serie de herramientas que simplifican la administración y desarrollo de la aplicación, como son:

- *RMA (Remote Monitoring Agent)*, es una consola gráfica para la administración y control de la plataforma. Permite explorar el servicio de páginas blancas tanto de la propia plataforma como de plataformas remotas, controlar el ciclo de vida de los agentes (creación remota, migración, etc).

- *Dummy Agent*, es una herramienta para la depuración del programa, permite enviar, recibir, cargar y almacenar mensajes ACL (Language Communication Agent).

- *Sniffer Agent*, permite capturar y visualizar los mensajes que se intercambian los agentes.

- *DF GUI*, permite explorar el servicio de páginas amarillas que proporcionan los agentes DF (Director Facilitator), así como gestionar complejos dominios y subdominios de DF’s, páginas amarillas.

- *Socket Proxy Agent*, es un agente que actúa como un puente entre la plataforma JADE y una conexión TCP/IP.

B.1 Características

JADE ofrece al programador:

- Plataforma distribuida.
- Interfaces graficas para administración remota de agentes.
- Herramientas de debugging.
- Movilidad de agentes inter-plataforma.
- Soporta la ejecución en paralelo de múltiples agentes.
- Transporte de mensajes ACL dentro de la plataforma.
- FIPA-compliant Agent Platform (incluye AMS, DF y ACC).
- Librería de protocolos de interacción FIPA.
- Servicio de nombres (GUID: Globally Unique Identifier).
- Interfaz para que aplicaciones externas inicien agentes autónomos.

B.2 Instalación

Los siguientes documentos y enlaces nos han resultado útiles para conocer esta plataforma:

- JADE tutorial. JADE programming for beginners. JADE 3.1.Giovanni Caire (TILAB, formerly CSELT) Diciembre 2003.
- JADE. A White Paper. F. Bellifemine, G. Caire, A. Poggi, G. Rimassa. Septiembre 2003.
- Experiencia en el desarrollo de un Sistema Multiagente. A. Alonso Álvarez, J. R. Villar Flecha, C. Benavides Cuellar, I. García Rodríguez, F. J. Rodríguez Sedano.

- http://www.dccia.ua.es/~pablo/tutorial_agentes/index.html
- http://www.geocities.com/paolaneriortiz/jade.htm
- http://www.iro.umontreal.ca/~vaucher/Agents/Jade/JadePrimer.html

Puede descargarse JADE de la página oficial http://jade.tilab.com/

Anexo C. Eclipse IDE

Un entorno de desarrollo integrado (IDE) puede hacernos el trabajo mucho más sencillo, sobretodo si nuestro desarrollo ya va manejando un buen número de Clases. Además estos entornos nos permiten mucha más versatilidad para depurar nuestros programas puesto que tienen depuradores mucho más avanzados.

Eclipse es un IDE de código abierto. Hay más herramientas similares de código abierto disponibles pero hemos decidido utilizar esta porque nos resultaba más familiar al haber tenido que usarla en otras asignaturas de la carrera.

Su instalación es muy sencilla, podemos descargárnoslo de www.eclipse.org en forma de archivo ZIP y solo tenemos que descomprimirlo en la carpeta donde queramos tenerlo instalado. Para ejecutarlo solo hay que arrancar el fichero *Eclipse.exe*. Una vez arrancado lo único que nos pedirá es que le demos la ruta por defecto donde queramos que Eclipse nos vaya guardando los proyectos que creemos:

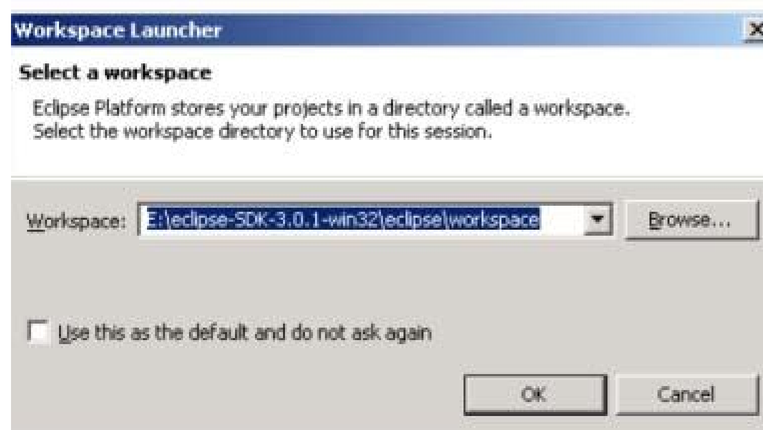


Figura C.1 Seleccionar un espacio de trabajo

Después de esto nos aparecerá la ventana principal de Eclipse:

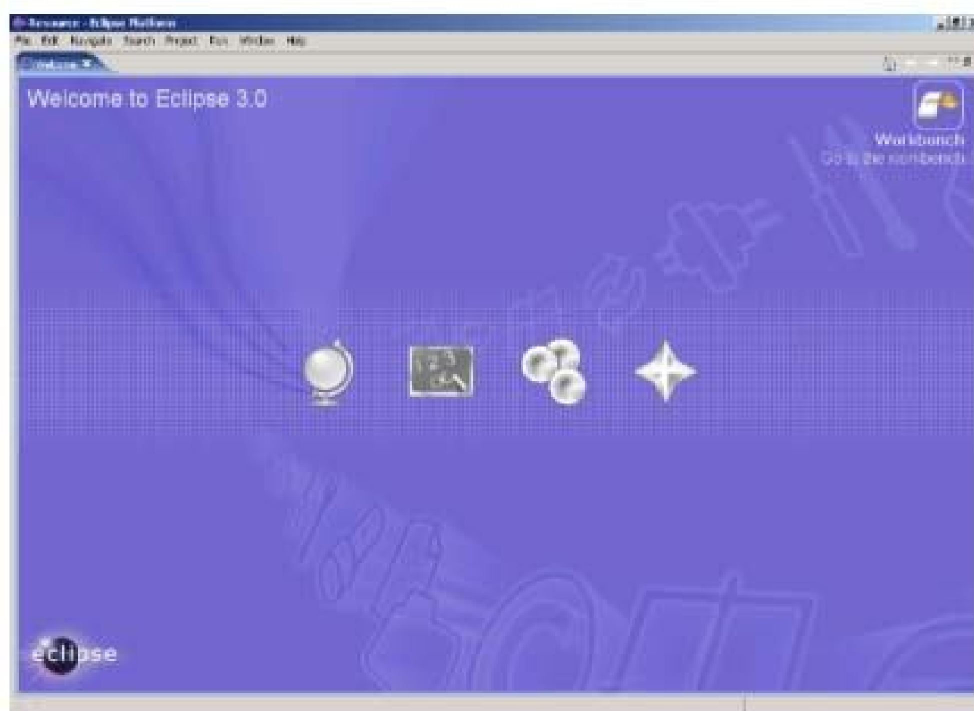


Figura C.2 Ventana principal de Eclipse

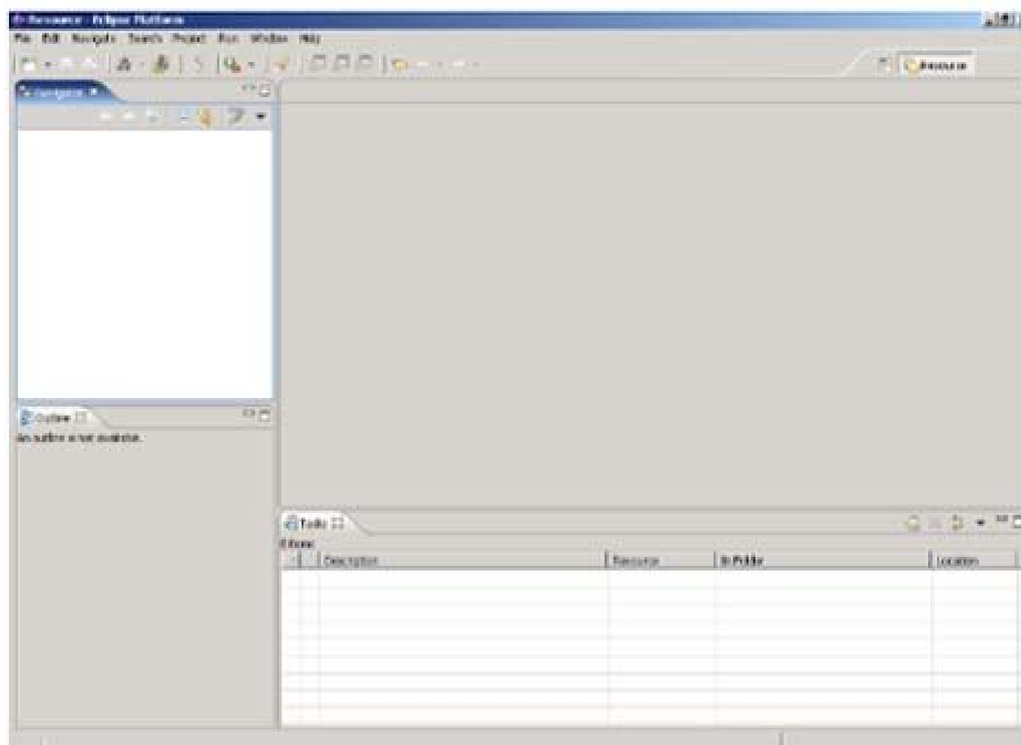


Figura C.3 Diferentes perspectivas de Eclipse

Eclipse puede usar varias perspectivas en su ventana principal dependiendo del tipo de desarrollo que vayamos a realizar. Ahora abriremos la perspectiva "*Java*":

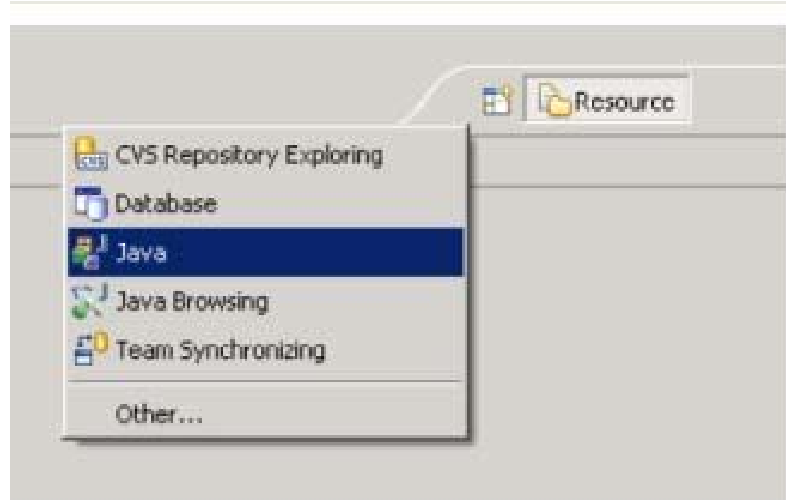


Figura C.4 Abrir la perspectiva Java

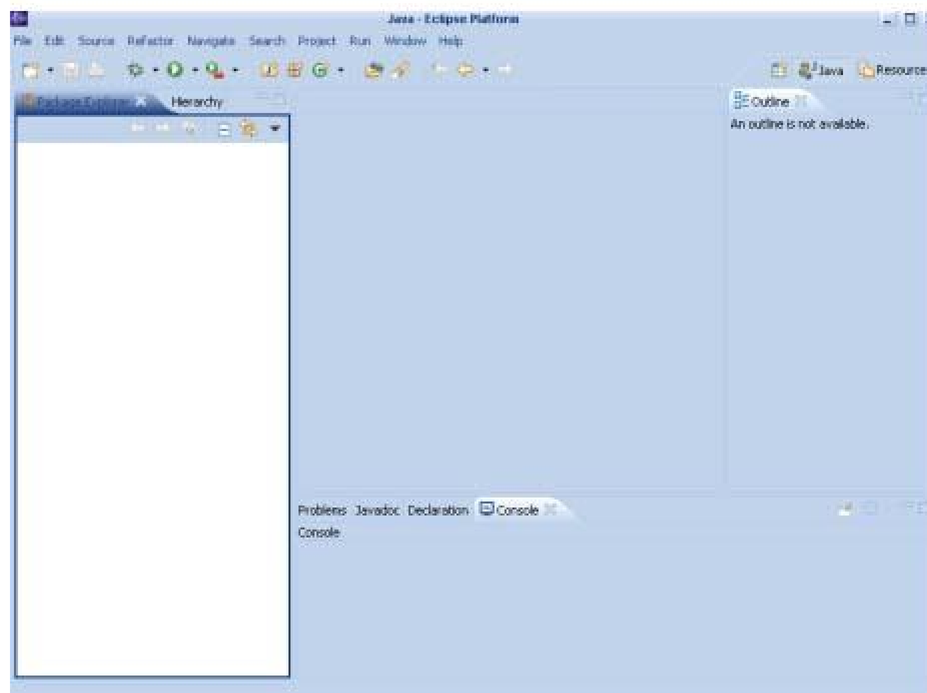


Figura C.5 Perspectiva Java

Es conveniente que todos los desarrollos que hagamos los hagamos dentro de un proyecto. Por lo tanto vamos a crear un proyecto para nuestro desarrollo:

Para esto le damos a "*File* → *New* → *Project*". Como podemos ver hay muchos tipos de proyectos para poder crear para nuestro desarrollo. Elegimos "*Java project*" y le damos a siguiente. En la ventana en la que estamos ahora podemos darle un nombre a nuestro proyecto y nos aparecen dos opciones relativas a la organización de nuestro proyecto. Las dejamos tal y como está para que simplemente nos coloque nuestros archivos .java y .class (fuentes y ejecutables java) en la carpeta que hemos escogido para el entorno de trabajo al arrancar Eclipse, y le damos a "*Finish*".

Figura C.6 Crear un nuevo proyecto

El siguiente paso es ir añadiendo nuestras clases al proyecto. Pinchando con el botón derecho en la carpeta del proyecto que se nos ha creado en la parte izquierda de la ventana principal podemos darle a "*New* → *Class*"

Esta ventana tiene varias opciones. Ahora simplemente pondremos los nombres de la clase, del paquete donde queramos incluirla (podemos dejar el paquete por defecto dejando este campo en blanco) y marcaremos las opciones que vemos en la ilustración.

Figura C.7 Crear una nueva clase

Como hemos podido comprobar al escribir los nombres de la clase y del paquete nos avisa de ciertas reglas para la nomenclatura de estos. Los nombres de las clases siempre empiezan en mayúscula y los de los paquetes en minúscula.

Al darle a finalizar nos crea una plantilla que podemos ver en el centro de la pantalla. Esta área es la que corresponde al editor y aquí es donde escribiremos nuestros programas en Java. La plantilla creada nos añade las líneas básicas en el tipo de clase Java que hemos creado con todos los pasos anteriores.

Al igual que en el ejemplo anterior cuando hemos hecho el habitual *hola mundo* escribimos lo que queda para que se nos quede tal y como en la figura. Es decir, introducimos System.out.println ("Hola Mundo\n"); dentro del método "*main*" de la clase.

Ya solo nos queda ejecutar el programa para ver que funciona. Para hacerlo funcionar podemos utilizar el menú "*run*" o directamente mediante los iconos de la barra de herramientas.

Al ejecutar el "*run*" un asistente nos dará a elegir el tipo de ejecución que queremos para nuestro código en Java. Simplemente escogemos "*Java Application*" en el menú con doble "clic" y nos creará un "apéndice" de configuración de ejecución para nuestro código en concreto, como podemos ver:

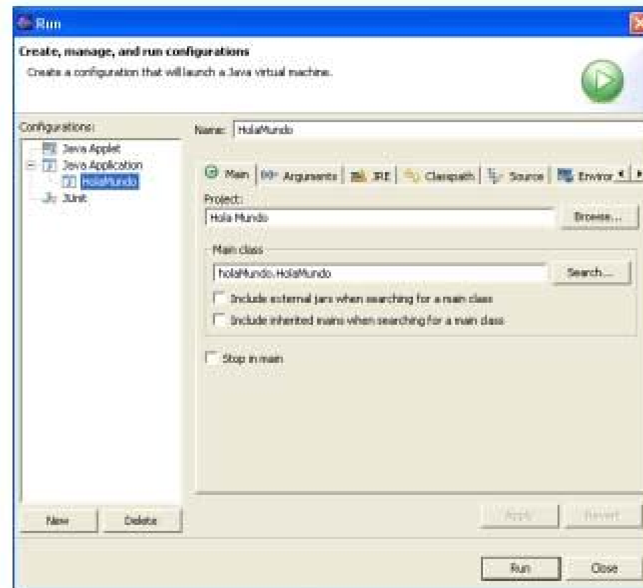


Figura C.8 Asistente para el tipo de ejecución

En principio y sin más detalles le damos a "*Run*" y vemos los resultados:

Figura C.9 Perspectiva de consola

Como podemos comprobar, abajo hay una pestaña que pone "*console*". Esa pestaña nos muestra una consola que hace las veces de la línea de comandos desde la que ejecutamos nuestro programa de Java en el primer ejemplo. Por tanto es en ese espacio en el que podemos ver la salida del programa: "Hola Mundo".

Con Eclipse tenemos un entorno completo para hacer nuestros “pequeños” desarrollos en Java.

Anexo D. Compilar y Ejecutar JADE con Eclipse

Las instrucciones para poder compilar y ejecutar la plataforma JADE en el entorno de desarrollo Eclipse son:

1. Abrir el **eclipse**.

2. Crear un **nuevo proyecto de Java**.

3. Seleccionar los siguientes .jar para que estén incluidos en el proyecto:

   - **http.jar**
   - **iiop.jar**
   - **jade.jar**
   - **jadeTools.jar**

4. Una vez creado el proyecto, crear dentro del mismo un paquete (el nombre del paquete deberá ser el que aparece en la cabecera del .java del ejemplo que queremos ejecutar).

5. Cuando hayamos creado el paquete copiamos y pegamos dentro el .java del ejemplo. Ya tenemos todo el proyecto creado.

6. A continuación procedemos a ejecutar el ejemplo. Pinchamos en el icono **"Run".**

7. Deberemos establecer los parámetros de ejecución dentro de la opción **"Java Application".** En **"Main class"** escribimos **jade.Boot**

8. En la pestaña **"arguments"** y concretamente en el recuadro de texto **"Program arguments"** ponemos lo siguiente:

   NombreAgente:nombrePaquete.archivoJava

   Ejemplo:

agente:examples.hello.HelloWorldAgent

9. Pinchar en el botón **"Run".** Deberían aparecer por consola los resultados de la ejecución.

10. Si después de todo esto no funcionara, probar lo siguiente:

    En lugar de poner: **NombreAgente:nombrePaquete.archivoJava**

    Escribir: **-nomtp NombreAgente:nombrePaquete.archivoJava**

    Ejemplo:

-nomtp agente:examples.hello.HelloWorldAgent

<u>IMPORTANTE:</u> En ocasiones no funciona de ninguna de las dos formas. En este caso cerrar el Eclipse, volver a abrirlo, abrir el proyecto y comenzar desde el paso **6**.

Funciona con la **versión 5.0 y 6.0 del jre**

Anexo E. CVS Berlios con Eclipse

E.1 Introducción

El **Concurrent Versions System** (CVS), también conocido como **Concurrent Version System** o **Concurrent Versioning System**, es una aplicación informática que implementa un sistema de control de versiones que mantiene un registro de todo el trabajo y los cambios en los ficheros (código fuente principalmente) que forman un proyecto (de programa) y permite que distintos desarrolladores colaboren.

Un Repositorio de código se compone de varios **Módulos,** cada uno de los cuales es un proyecto.

Dentro de cada Módulo se encuentran los distintos paquetes de los proyectos y los ficheros de proyecto.

Para conectar al repositorio se necesita la siguiente información.

- Host: **cvs.catsrats.berlios.de**
- Repository Path: **/cvsroot/catsrats**
- User: "**Vuestro nombre de usuario de Berlios**"
- Password: "**Vuestra contraseña de Berlios**"
- Connection type: **extssh** (en Eclipse), **ext** (en Netbeans).
- Usar puerto por defecto (**Default port**)

A continuación se explican algunos conceptos relativos a los CVS:

- **Share Project**: Compartir el proyecto. Esta opción se utiliza para crear módulos en el Repositorio, es decir, subir proyectos a él, cuando aun no existen. Si ya existe el proyecto en el repositorio no hará falta compartirlo, será suficiente sincronizar la versión que se tiene con la existente en el repositorio.

- **Synchronize**: Comprobar las diferencias o conflictos existentes entre la versión que se está usando en el ordenador de trabajo y que probablemente se haya modificado, y la versión existente en el repositorio.
- **Update**: Actualizar el proyecto con la última versión existente en el repositorio
- **Commit**: Actualizar el repositorio con una versión más nueva (a la que se haya incluido nuevo código), que será con la que se esté trabajando en el disco duro.
- **Conflict**: al hacer *commit* o *update*, existen diferencias importantes entre la propia versión con la que se ha trabajado y la del repositorio, es decir, esa versión tiene cosas que el repositorio no tiene, y el repositorio tiene código que no contiene los cambios. Esto puede deberse a que ha habido dos personas distintas escribiendo código en el mismo fichero *.java al mismo tiempo.

  A la hora de hacer un *update*, si no hay conflictos, (la versión del repositorio tiene que tener lo mismo que la versión de trabajo, y alguna línea más), se actualizan los ficheros en el PC con los cambios que hayan hecho los demás miembros registrados.

  A la hora de hacer un *commit*, si no hay conflictos, (la versión del disco duro ha de tener lo mismo que la versión del repositorio, y los últimos cambios que se hayan realizado), se actualizan los ficheros del repositorio con los cambios realizados, cambiando el número de versión del fichero a otro posterior (esto último lo hace automáticamente el repositorio).

IMPORTANTE (puede ser el motivo de pérdida de trabajo): antes de hacer un *commit*, hay que hacer un *update*, o sincronizar con el proyecto en el repositorio, ya que si aparecen conflictos por un cambio hecho por otra persona, se puede borrar todo su trabajo.

Peculiaridades: Los paquetes vac

E.2 Eclipse

E.2.1. Crear un módulo

Subir el proyecto por primera vez al CVS.

Para crear un modulo todavía no debe existir el proyecto en el CVS.

Seleccionar el proyecto en el **Package explorer,** y pulsar el botón derecho del ratón.

- Menú **Team** -> **Share Project**
- Seleccionar la opción **CVS** y **Next**
- Introducir la información que se dio en la introducción **Next**
- **Use project Name as Module name** o si se desea que el módulo se llame de otra forma: **Use specified module name -> Next**
- Aceptar todas las ventanas emergentes y seguir el asistente.

E.2.2. Bajarse el proyecto del CVS

Si ya existe una versión del proyecto en el disco:

Seleccionar el proyecto en el **Package explorer**, y pulsar el botón derecho del ratón.

Menu **Team** -> **Synchronize with Repository**. Si no hay cambios el Eclipse avisa, y no hace falta bajar nada, sino:

- Si se pide abrir la perspectiva de CVS, pulsar aceptar
- Desplegar el árbol de paquetes para ver los ficheros.
- Si un archivo tiene una flechita azul hacia la izquierda, significa que se puede hacer *update*.

Si todavía no se dispone del proyecto

- File -> Import
- Checkout projects from CVS -> Next

Si ya se había entrado antes en el CVS seleccionar el Repositorio en la lista, Sino:

- Introducir la información que se dio en la introducción -> **Next**
- Seleccionar **Use an existing module.** Aceptar las posibles ventanas emergentes y esperar.....
- Seleccionar de la lista la carpeta de proyecto que queráis abrir -> **Next**
- **Checkout as a project in the workspace** (No cambiar el nombre) -> **Next**
- Seleccionar la carpeta que queráis del disco -> **Finish**

E.2.3. Subir el proyecto al CVS

Seleccionar el proyecto en el **Package explorer,** y pulsar el botón derecho del ratón.

Menu **Team** -> **Synchronize with Repository.** Si no hay cambios el Eclipse avisa, y no hace falta bajar nada, sino:

- Si se pide abrir la perspectiva de CVS, pulsar aceptar
- Desplegar el árbol de paquetes para ver los ficheros.
- Si un archivo tiene una flechita gris hacia la derecha, significa que se puede hacer un *commit* sin peligro.

E.2.4. Conflictos al hacer "Synchronize with Repository"

Si en la vista del CVS, aparecen flechas rojas que señalan a derecha e izquierda en algunos archivos al desplegar el árbol de paquetes, significa que tienen conflictos.

Antes de hacer *commit* o *update*, hacer doble click en él y se abrirán los dos ficheros, tanto el propio como el del repositorio, mostrándose todas las diferencias entre los dos.

Entonces será preciso realizar manualmente una mezcla entre los dos, haciendo los cambios siempre en el fichero que se tiene en el disco, que será con el que luego se hará *commit*, para dejar la nueva versión con todo incluido finalmente en el repositorio.

E.3 Enlaces de interés

FAQ CVS eclipse

http://wiki.eclipse.org/index.php/CVS_FAQ#What_is_the_difference_between_ext_and_extssh.3F

TUTORIAL Eclipse (incluye carpeta Team Cvs Tutorial)

http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.user/gettingStarted/qs-60_team.htm

Anexo F. OpenSceneGraph

F.1 Descripción

OpenSceneGraph (OSG) es una librería gráfica de código abierto, que presenta independencia de la plataforma y utiliza como lenguaje de programación C++. Dispone de un conjunto de herramientas gráficas para el desarrollo de alto rendimiento de aplicaciones tales como simuladores de vuelo, juegos, realidad virtual y visualización científica.

OSG es de reciente desarrollo y encapsula las diferentes primitivas de OpenGL. Esto libera al desarrollador de la aplicación y optimización de bajo nivel de gráficos pide y ofrece muchas más utilidades para el rápido desarrollo de aplicaciones gráficas.

OSG emplea técnicas de grafos de escena para contener toda la información relativa a la escena generada. Un grafo de escena es una estructura de datos que permite crear una estructura jerárquica de la escena, de tal forma que se mantengan una serie de relaciones padre-hijo entre los distintos elementos. Por ejemplo, variaciones de posición y orientación en el nodo padre afectan a los nodos hijos.

Otra importante relación padre-hijo explotada por las técnicas de grafos de escena es la posibilidad de definir volúmenes envolventes que agrupen a elementos cercanos, de tal forma que durante el proceso de descarte de los elementos que se van a representar en pantalla no sea necesario recorrer los hijos de un nodo padre ya descartado.

La elección de OSG se ha basado fundamentalmente en su código abierto, su gratuidad, su independencia de la plataforma y, sobretodo, sus posibilidades de expansión. El principal inconveniente es la falta de documentación específica. Pero este problema es minimizado mediante una serie de ejemplos que aportan los conocimientos básicos de las distintas capacidades de la librería.

F.2 Características

Los principales puntos fuertes de OSG son su rendimiento, escalabilidad, portabilidad y las ganancias de productividad asociadas con el uso de un escenario completamente gráfico:

Rendimiento

Permite nodos de nivel de detalle (LOD), clasificación de estados OpenGL, arrays de vértices, OpenGL Shader Language y mostrar listas como parte del núcleo de la escena. Esto hace de OSG uno de las herramientas disponibles de mayor rendimiento de gráficos. También soporta fácil personalización del proceso de dibujo, como la aplicación continua de mallas de Nivel de Detalle (CLOD) en la parte superior de la escena gráfica.

Productividad

El desarrollo de aplicaciones se concentra en el contenido y la forma en que ese contenido está controlado, en vez del código de bajo nivel.

La combinación de la experiencia adquirida en gráficos de escena con modernos métodos de ingeniería de software como patrones de diseño, junto con una gran cantidad de comentarios en los inicios del ciclo de desarrollo, ha hecho posible diseñar una librería que sea limpia y extensible. De este modo, los usuarios han podido adoptar OSG para integrar con sus propias aplicaciones.

Base de datos cargadoras

Para leer y escribir bases de datos, la base de datos de la biblioteca (osgDB) incluye soporte para una amplia variedad de formatos de bases de datos ampliable a través de un mecanismo dinámico de plugins.

COLLADA, LightWave (. Lwo), (. Obj), OpenFlight (. FLT), TerraPage (. Txp), Gráficos de carbono (. Geo), 3D Studio MAX (. 3ds), Peformer (. Pfb), AutoCAD? (. dxf), Quake modelos (. md2). Direct X (. X), e Inventor Ascii 2.0 (. Iv) / VRML 1.0 (. WRL), Taller de diseño (. Dw) y AC3D (. Ac) y los nativos .OSG de formato ASCII.

Los cargadores de imagen incluyen: .Rgb, .Gif, .Jpg, .Png, .Tiff, .Pic, .Bmp, .Dds, .Tga y QuickTime (bajo OSX).

Node Kits

La escena gráfica también tiene una serie de Node Kits, que son bibliotecas independientes que pueden ser compiladas en sus aplicaciones o cargadas en tiempo de ejecución para añadir soporte a sistemas de partículas (osgParticle), anti-alias de texto de alta calidad (osgText), framework de efectos especiales (osgFX), framework de sombras (osgShadow), controles interactivos (osgManipulator) y la simulación visual centrada en efectos (osgSim).

Portabilidad

El núcleo gráfico ha sido diseñado para tener un mínimo de dependencia en cualquier plataforma específica, que requiere poco más que la norma C + + y OpenGL. Esto ha permitido a OSG ser rápidamente portado a una amplia gama de plataformas - originalmente desarrollada en IRIX, luego portada a Linux, a Windows, FreeBSD, Mac OSX, Solaris, HP-UX, AIX e incluso PlayStation2.

Es completamente independiente del sistema de ventanas, que facilita que los usuarios puedan añadir su propia ventana de librerías y aplicaciones en la parte superior.

Escalabilidad

OSG se ejecuta desde portátiles hasta computadores multi-core, multi-GPU y sistemas de cluster. Esto es posible porque soporta múltiples contextos gráficos OpenGL, ha sido diseñado para hacer caché de datos locales y hacer uso de él casi en su totalidad como sólo lectura. Esto permite funcionar en múltiples CPU's que están vinculados a varios subsistemas de gráficos. Soporta múltiples gráficos y contexto multi-threading a través de osgViewer.

Anexo G. Configuración Visual Studio 2005

G.1 Cómo ejecutar proyectos de OSG

1. Crear un proyecto nuevo de **aplicación de consola Win32**.

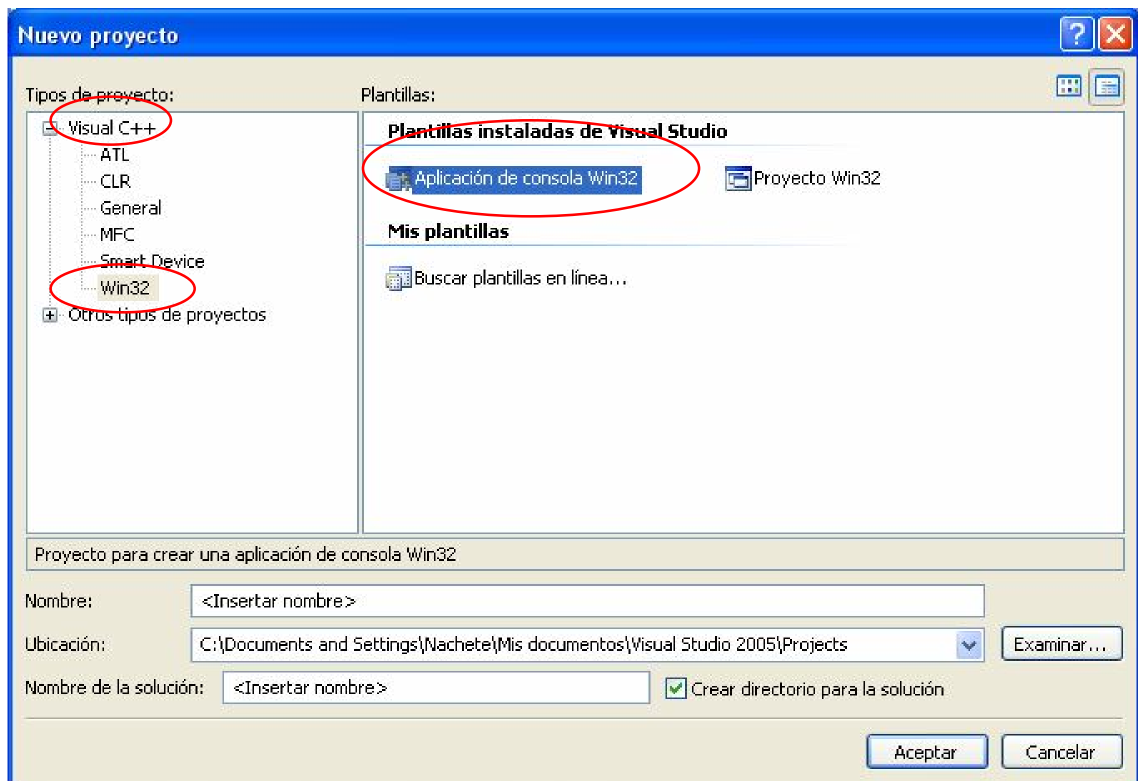


Figura G.1 Crear un proyecto nuevo de aplicación de consola Win32

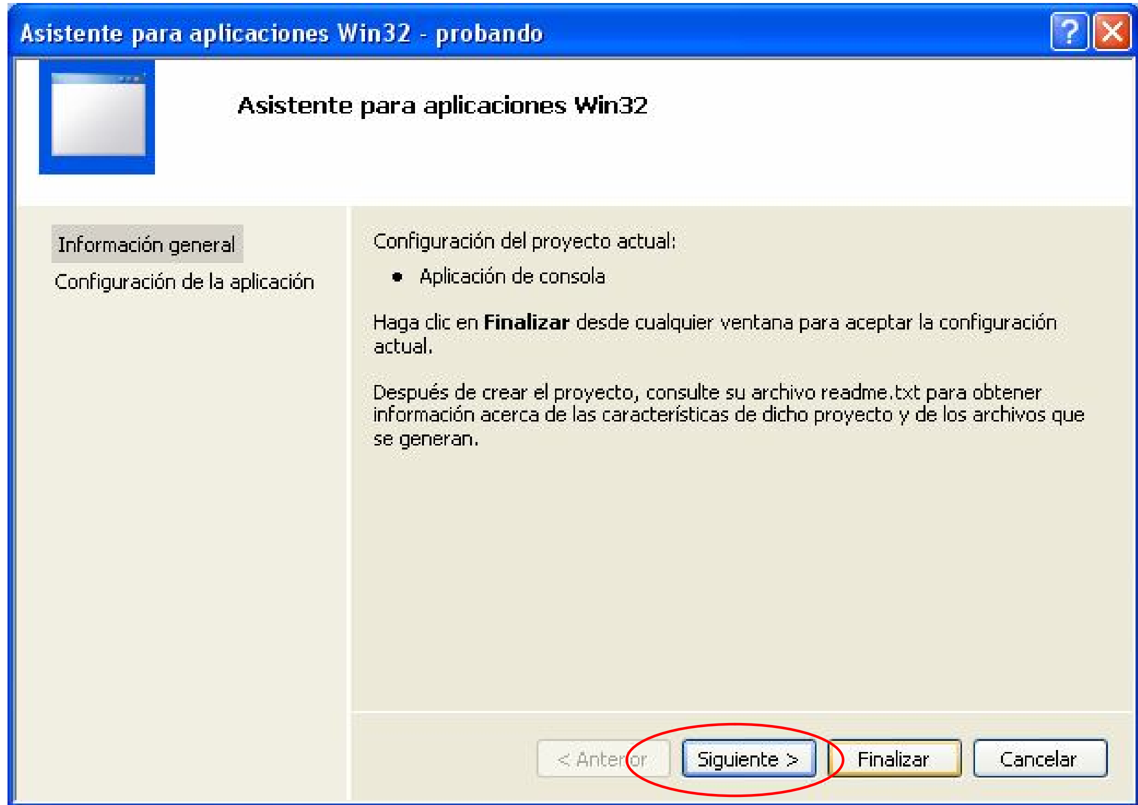


Figura G.2 Asistente para aplicaciones Win32

2. Marcamos la opción adicional **Proyecto vacío**

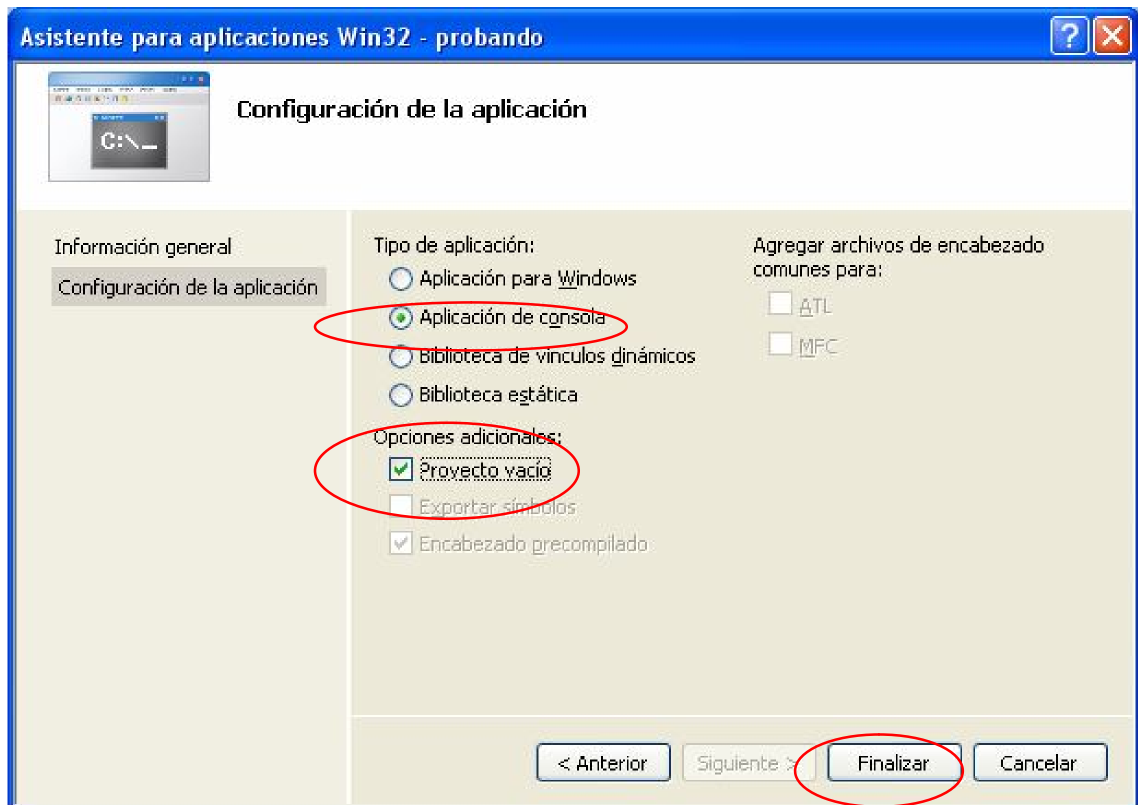


Figura G.3 Marcamos la opción adicional “Proyecto vacío”

3. Buscar el código fuente y añadirlo al proyecto en la carpeta **Archivos de código fuente**.

4. La configuración de la solución se hará siempre en **Release** y no en **Debug**.

5. Añadir al proyecto las siguientes librerías en **Propiedades del proyecto→**

   →Vinculador→entrada→dependencias adicionales

Librerías: **glut32.lib glu32.lib opengl32.lib osg.lib osgdb.lib osgga.lib osgsim.lib osgviewer.lib osgutil.lib** (deben estar separadas por un espacio).

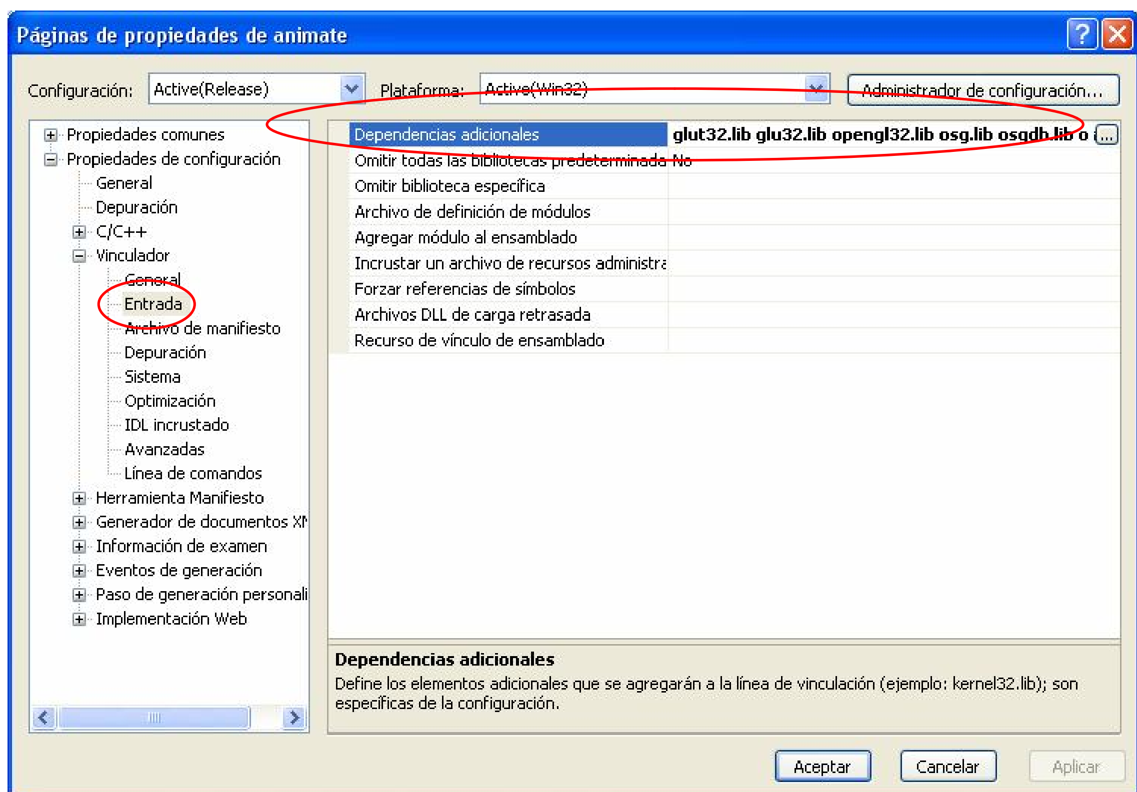


Figura G.4 Añadir al proyecto las siguientes librerías

6. Ir a **Herramientas→Opciones→Proyectos y soluciones→Directorios de VC++:**

Archivos ejecutables: Establecemos la ruta del directorio **bin** del OpenSceneGraph

D:\Nacho\OpenSceneGraph\bin

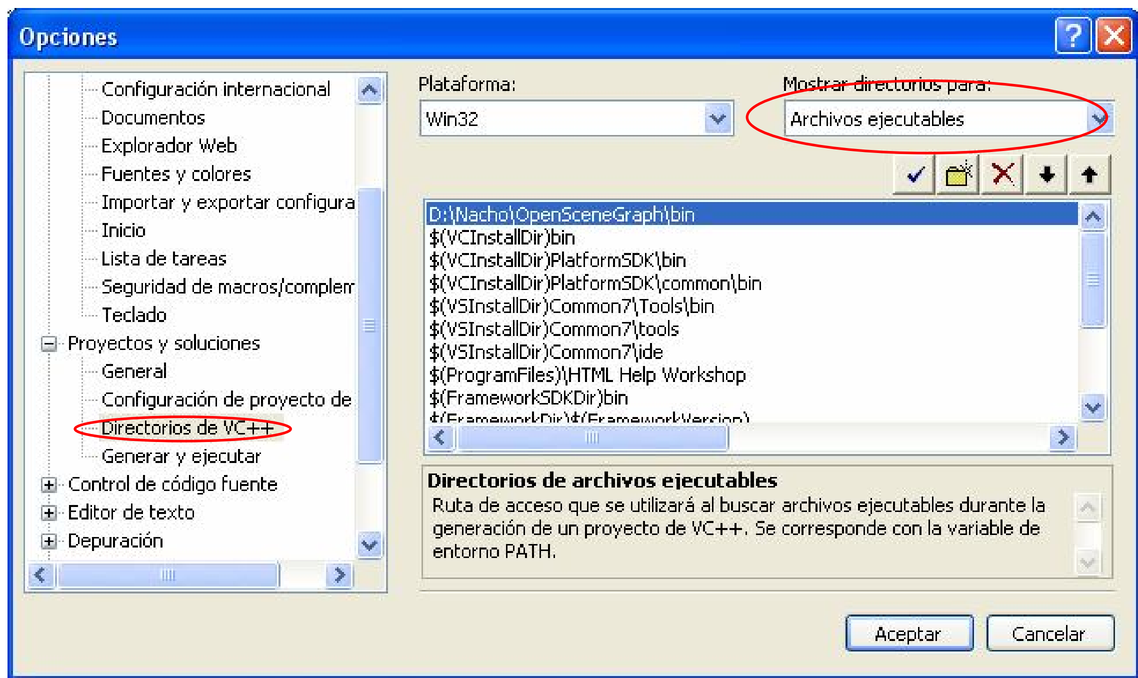


Figura G.5 Establecer la ruta del directorio bin

Archivos de inclusión: Establecemos las rutas del directorio **include** del OpenSceneGraph

D:\Nacho\OpenSceneGraph\osgr\OpenSceneGraph\include

D:\Nacho\OpenSceneGraph\include

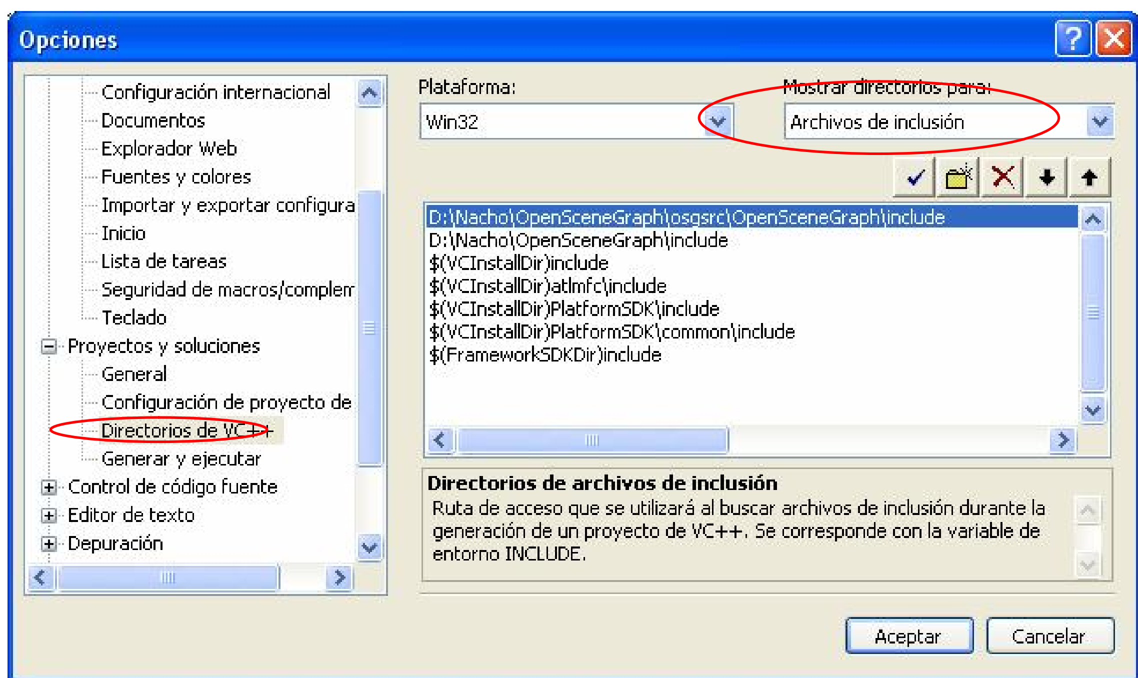


Figura G.6 Establecer la ruta del directorio include

Archivos de biblioteca: Establecemos la ruta del directorio **lib** del OpenSceneGraph

D:\Nacho\OpenSceneGraph\lib

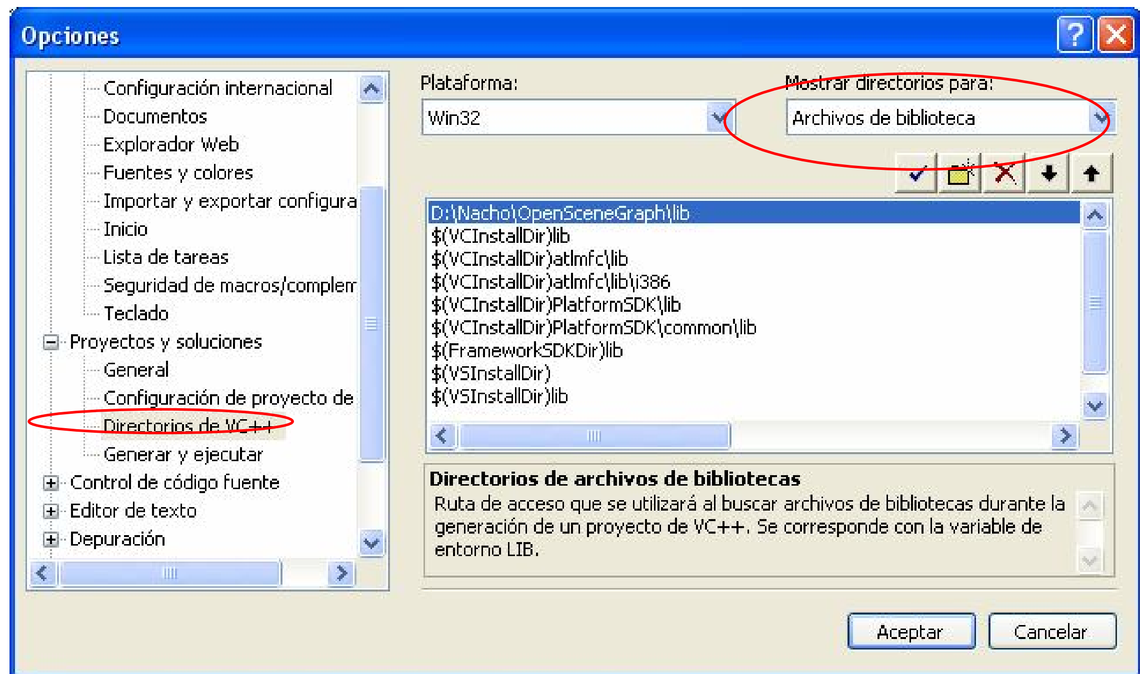


Figura G.7 Establecer la ruta del directorio lib

7. En el directorio del proyecto creado añadir en el directorio de **Release** las siguientes **librerías dll** (todas las que se encuentran en el directorio **bin** del OSG)**:**

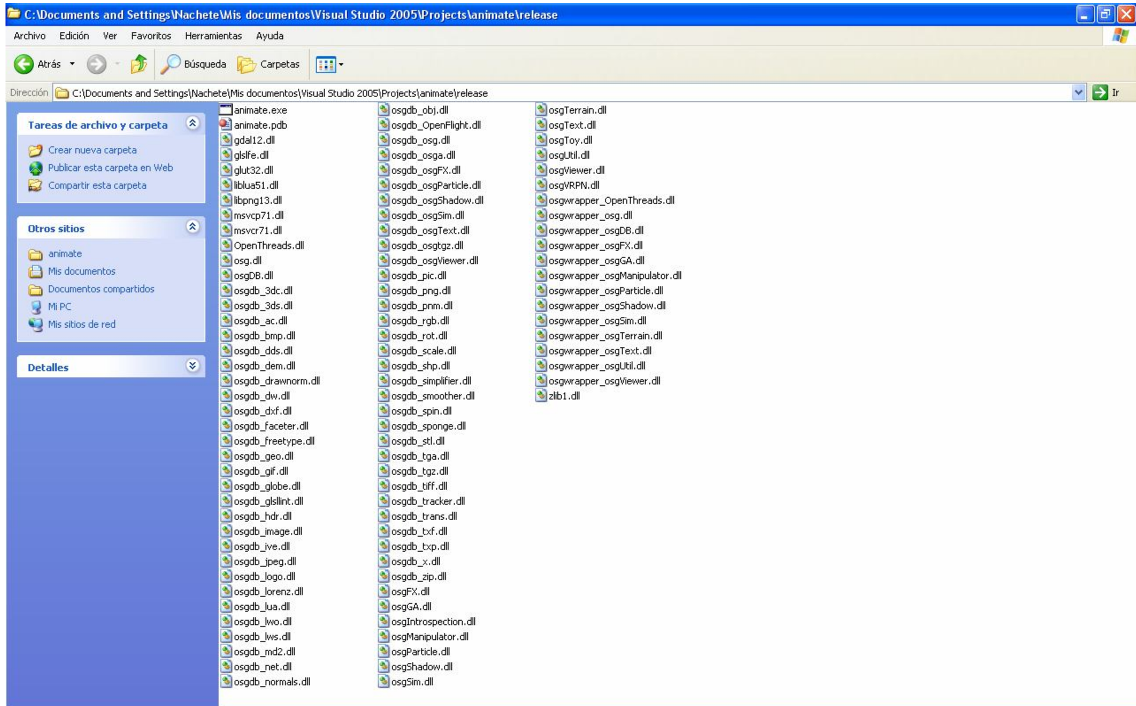


Figura G.8 Añadir en el directorio Release todas las librerías que se encuentran en el directorio bin

8. Para generar la solución ir a **Generar->Generar solución**

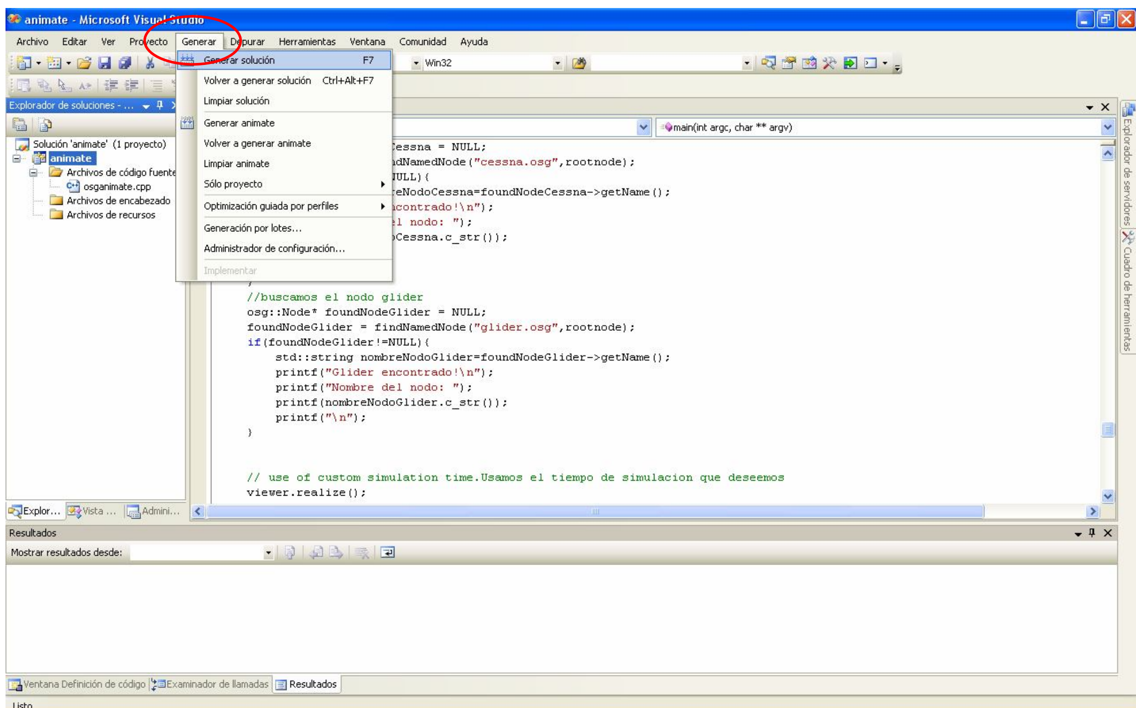


Figura G.9 Generar la solución

9. Para ejecutar el código pinchar en "**Iniciar depuración**".

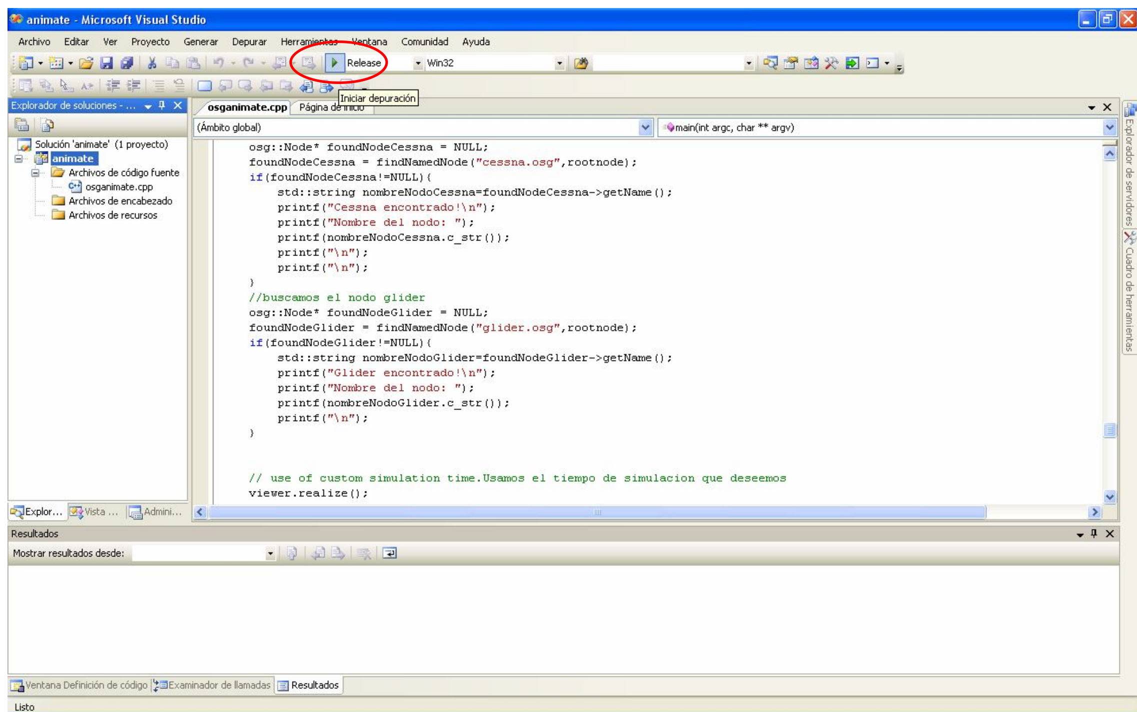


Figura G.10 Para ejecutar el código pinchar en “Iniciar depuración”

Anexo H. Librerías Log4J

Log4j es una biblioteca open source desarrollada en Java por la Apache Software Foundation que permite a los desarrolladores elegir la salida y el nivel de granularidad de los mensajes o “logs” en tiempo de ejecución y no en tiempo de compilación como es comúnmente realizado. La configuración de salida y granularidad de los mensajes es realizada en tiempo de ejecución mediante el uso de archivos de configuración externos.

Por defecto Log4J tiene 6 niveles de prioridad para los mensajes (debug, info, warn, error, fatal, trace). Además existen otros dos niveles extras (all y off):

H.1 Niveles de prioridad

Ordenados de mayor a menor:

FATAL: se utiliza para mensajes críticos del sistema, generalmente después de guardar el mensaje el programa terminará.

ERROR: se utiliza en mensajes de error de la aplicación que se desea guardar, estos eventos afectan al programa pero lo dejan seguir funcionando, como por ejemplo que algún parámetro de configuración no es correcto y se carga el parámetro por defecto.

WARN: se utiliza para mensajes de alerta sobre eventos que se desea mantener constancia, pero que no afectan el correcto funcionamiento del programa.

INFO: se utiliza para mensajes similares al modo "verbose" en otras aplicaciones.

DEBUG: se utiliza para escribir mensajes de depuración, este log no debe estar activado cuando la aplicación se encuentre en producción.

TRACE: se utiliza para mostrar mensajes con un mayor nivel de detalle que debug.

ALL: este es el nivel más bajo posible, habilita todos los logs.

OFF: este es el nivel más alto posible, deshabilita todos los logs.

H.2 Configuración

En Log4J los mensajes son enviados a una (o varias) salida de destino, lo que se denomina un *appender*.

Existen varios appenders disponibles y configurados, aunque también se pueden crear y configurar appenders propios para una aplicación concreta.

Típicamente la salida de los mensajes es redirigida a un fichero de texto *.log* (*FileAppender*, *RollingFileAppender*), a un servidor remoto donde almacenar registros (*SocketAppender*), a una dirección de correo electrónico (*SMTPAppender*), e incluso en una base de datos (*JDBCAppender*).La salida a la consola puede utilizarse (*ConsoleAppender*) pero se perdería gran parte de la utilidad de Log4J.

Mediante un layout puede darse formato de presentación a los mensajes.Permite presentar el mensaje con el formato necesario para almacenarlo simplemente en un archivo de texto *.log* (*SimpleLayout* y *PatternLayout*), en una tabla HTML (*HTMLLayout*), o en un archivo XML (*XMLLayout*).

Además es posible añadir información extra al mensaje, como la fecha en que se generó, la clase que lo generó, el nivel que posee...

La API es totalmente configurable, ya que se realiza mediante un archivo en formato XML o en formato Java Properties (clave=valor), generalmente llamado *log4j.properties*.

Los mensajes que se procesarán en nuestro caso serán los que sean de prioridad info, o superior, tal y como se indica en el fichero de configuración

```
### Root logger ###

log4j.rootLogger=info,archivo.
```

H.3 Uso

Para utilizar log4j se han de importar sus clases necesarias en el código.

A continuación hay que definir una variable estática del tipo *org.apache.log4j.Logger* con el nombre de la clase que va a escribir en el registro.

Finalmente se configura el objeto *Logger*, mediante alguna de las siguientes opciones:

- Configuración básica, invocando el método o*rg.apache.log4j.BasicConfigurator.configure* que configura el registro como un *ConsoleAppender* y un *PatternLayout* también predefinido.

- Configuración de la API del Log4J también en el mismo código.

- Leer la configuración del fichero *log4j.properties* (a ubicar en el directorio “src” del proyecto eclipse), en el que estará definido el nivel mínimo que debe poseer la traza para ser almacenada en el registro, el/los appenders a utilizar, y sus correspondientes layouts. Por defecto, al instanciar un Logger, Log4J busca en la raíz del classpath de la aplicación un fichero llamado *log4j.properties* que contiene la configuración.

Parte 5. Bibliografía

TOMA DE DECISIONES

Benford, S. D., y Fahlén, L. E., "A Spatial Model of Interaction in Large Virtual Environments". Procedente de la tercera conferencia europea sobre Computer Supported Cooperative Work (ECSCW'93). Milán. Italia. Kluwer Academic Publishers, pp.109-124, 1993.

Benford, S. y col. "User Embodiment in Collaborative Virtual Environments". Procedente de la conferencia ACM sobre Human Factors in Computing Systems (CHI'95). Denver. Colorado, EEUU, pp.242-249, 1995.

Chen, C.T., Extensions of the TOPSIS for group decision-making under fuzzy environment, Fuzzy Sets and Systems 114 (2000) 1-9.

Chim, Y. C. , Kassim, A. A., Ibrahim, Y., Character recognition using statistical moments. Image and Vision Computing 17 (1999) 299–307.

Dourish, P., Bellotti, V., "Awareness and Coordination in Shared Workspaces", Procedente de la cuarta conferencia ACM sobre Computer Supported Cooperative Work (CSCW). Toronto. Ontario. Canada, pp.107-114, 1992.

Greenhalgh, C.; "Large Scale Collaborative Virtual Environments", Tesis Doctoral. Universidad de Nottingham, 1997.

Herrero, P., De Antonio, A.; Diseño de un Modelo de Percepción para agentes virtuales inteligentes basado en el sistema de percepción de los seres humanos. pp 11-24. 2003.

Herrero, P., De Antonio, A.; A Human Based Perception Model for Cooperative Intelligent Virtual Agents. Procedente de la Décima Conferencia Internacional sobre Sistemas de Información Cooperativos (CoopIS 2002). Irving. California. USA, pp. 195-212, 2002.

Hwang, C.L., Yoon, K., Multiple Attributes Decision Making Methods and Application, Springer-Verlag, Berlin Heidelberg, 1981.

Jiang, Q., Chen, C.H., A multi-dimensional fuzzy decision support strategy. Decision Support Systems 38 (2005) 591– 598.

Kaufmann, A., Gupta, M.M., Introduction to Fuzzy Arithmetic: Theory and Applications, Van Nostrand Reinhold, New York, 1985.

Pajares, G., de la Cruz, J.M., Visión por Computador: Imágenes digitales y aplicaciones, RA-MA, 2008.

Pajares, G. de la Cruz, J.M., Ejercicios resueltos de visión por Computador, RA-MA, 2007.

Pajares, G., Ruz, J. J., Lanillos, P., Guijarro, M., de la Cruz, J. M., Santos, M., Generación de trayectorias y toma de decisiones para UAVs. Revista Iberoamericana de de Automática e Informática Industrial Vol. 5, Núm. 1, Enero 2008, pp. 83-92.

Ribeiro, R.A., Fuzzy multiple attribute decision making: A review and new preference elicitation techniques. Fuzzy Sets and Systems 78 (1996) 155-181.

Ríos, S., Bielza, C., Mateos, A., Fundamentos de los Sistemas de Ayuda a la Decisión, RA-MA, 2002.

Tellaeche, A., BurgosArtizzu, X. P., Pajares, G., Ribeiro, A., Fernandez-Quintanilla, C., A new vision-based approach to differential spraying in precision agriculture. Computers and Electronics in Agriculture. 60 (2008) 144-155.

Wallenius, J., Dyer, J.S., Fishburn, P.C., Steuer, R., Zionts, S., Deb, K., Multiplw Criteria Decision Making, Multiattribute Utility Theory: Recent Accomplishments and What Lies Ahead. SAL&HSE Graduate School Seminar: September 6, 2007.

Wang. W., Fenton, N., Risk and Confidence Analysis for Fuzzy Multicriteria Decision Making. (disponible on-line. http://www.dcs.qmul.ac.uk/~norman/papers/Wang_Fenton_Risk_and_Confidence.pdf)

TEORÍA DE AGENTES Y JADE

Aranda, G. A., Seminario de Jade, Curso de Programación de Agentes, Sevilla, 23 de Mayo de 2005.

Bellifemine,F., Caire,G., Poggi,A., Rimassa,G., JADE A White Paper,September 2003

Botía, J.A., Introducción a los agentes software-MAS,DIIC. Universidad de Murcia.

Caire, G., JADE TUTORIAL: JADE PROGRAMMING FOR BEGINNERS, CSELT. 2003

Fernández, C., Gómez, J., Pavón, J., Desarrollo de Agentes software.
Introducción a la tecnología de agentes, Depto. de Sistemas Informáticos y Programación. UCM 2003.
Disponible en http://grasia.fdi.ucm.es Junio 2008

FIPA-OS V2.1.0 Distribution Notes. Pagina Oficial de FIPAOS.
Disponible en www.fipa.org Junio 2008

Franklin, S., Graesser, A., Is It an Agent, or Just a Program?: A Taxonomy for Autonomous Agents, Institute for Intelligent Systems, University of Memphis, Memphis, TN 38152, USA.

Garamendi, J. F., Agentes Inteligentes: JADE, Programa de Doctorado: Informática y Modelización Matemática, URJC, Abril, 2004.

García-Montoro, C., Análisis y clasificación de lenguajes de programación orientados a agentes (capítulo 8), Departamento de Sistemas Informáticos y Computación, Universidad de Valencia, Mayo, 2007.

Group of the Object Management. Agent Technology Green Paper
OMG Document agent/00-09-01 Version 1.0 September 1, 2000.

Pavón, J., Agentes inteligentes, Comunicación entre agentes,

Depto. de Sistemas Informáticos y Programación, UCM 2004.
Disponible en http://grasia.fdi.ucm.es Junio 2008

Vaucher, Jean y Ncho, Ambroise. "JADE Tutorial and Primer". Dep. D'informatique. Université de Montréal. September 2003.
Disponible en: http://www.iro.umontreal.ca/~vaucher/Agents/Jade/JadePrimer.html Junio 2008

Wu, Chao-Lin, Multi-Agent System JADE - Java Agent DEvelopment framework, May 13th, 2004.

Zambonelli, F., Jennings, N. R., Wooldridge,M., Developing Multiagent Systems: The Gaia Methodology, October 3, 2003.

Ejemplos de código:

La Plataforma JADE para desarrollo de Agentes
http://www.geocities.com/paolaneriortiz/jade.htm Junio 2008

Foros de debate:

Club Developers
http://www.clubdevelopers.com/blogs/index.php?blog=15&cat=50 Junio 2008

Otros sitios de interés:

Java Agent DEvelopment Framework
http://jade.tilab.com/ Junio 2008

JADE v3.6 API
http://jade.tilab.com/doc/api/index.html Junio 2008

Suau, P., Tutorial Jade
http://www.dccia.ua.es/~pablo/tutorial_agentes/index.html Junio 2008

JAVA3D

Documentación:

De Tapia, C., Del Corral, M. Chehayeb, A., Java 3D
Disponible en http://gsii.usal.es/~corchado/igrafica/descargas/temas/Tema13.pdf Junio 2008.

Molina, A. I., Proyecto fin de carrera: "Sistema de diseño de Entornos Virtuales de edificios domotizados con Java 3D" Univ. Castilla La-Mancha. Junio 2002. pp. 86-95
Disponible en
http://chico.inf-cr.uclm.es/mortega/docencia/PFC/2002_ana_isabel/MEMORIAPFC.pdf Junio 2008.

Selman, D., Java 3D Programming
Disponible en
http://www.mat.uniroma2.it/~picard/SMC/didattica/materiali_did/Java/Java_3D/Java_3D_Programming.pdf Junio 2008.

Sun Microsystems. Java 3D 1.5.1 API Documentation
Disponible en http://download.java.net/media/java3d/javadoc/1.5.1/index.html Junio 2008.

Sun Microsystems. Java 3D 1.3 API Specification Guide.
Disponible en
http://java.sun.com/products/javamedia/3D/forDevelopers/J3D_1_3_API/j3dguide/index.html
Junio 2008.

Sun Microsystems. Java 3D API Tutorial.
Disponible en http://java.sun.com/developer/onlineTraining/java3d/index.html Junio 2008.

Sun Microsystem. Java 3D API Tutorial. Traducción de Juan Antonio Palos
Disponible en http://programacion.com/java/tutorial/3d Junio 2008.

Java 3D 1.3.2 API Documentation (include information for utilities).
Disponible en http://download.java.net/media/java3d/javadoc/1.3.2/index.html Junio 2008.

Java 3D FAQ.
Disponible en http://wiki.java.net/bin/view/Javadesktop/Java3DFAQ Junio 2008.

Ejemplos de código fuente

Java 3D examples.
http://www.java2s.com/Code/Java/3D/Catalog3D.htm Junio 2008.

Java 3D examples.
http://www.java-tips.org/ Junio 2008.

Foros de debate

gamedev.net. Discussion Forums. Java Development
http://www.gamedev.net/community/forums/topic.asp?topic_id=207095 Junio 2008.

Sun Developer Network. Multimedia and Imaging APIs - Java 3D.
http://forum.java.sun.com/forum.jspa?forumID=21 Junio 2008.

java.net Forums. Java Desktop Technologies. Java 3D.
http://forums.java.net/jive/forum.jspa?forumID=70 Junio 2008.

Otros sitios de interés

The Java 3D Home Page
http://java.sun.com/products/java-media/3D/ Junio 2008.

The Java 3D Graphics Community
http://www.j3d.org/ Junio 2008.

The j3d.org Code Repository
http://code.j3d.org/ Junio 2008.

SOCKETS

Conexión con sockets entre un programa Java y uno C
http://www.chuidiang.com/clinux/sockets/sockets_simp.php
Febrero 2008

C++ Sockets Library
http://www.alhem.net/Sockets/index_spanish.html
Febrero 2008

Definición arquitectura cliente-servidor
http://www.monografias.com/trabajos24/arquitectura-cliente-servidor/arquitectura-cliente-servidor.shtml
Febrero 2008

Java / C++ Socket Class (Keith Vertanen ´s research)
http://www.keithv.com/project/socket.html
Febrero 2008-06-22

Java Sockets
http://www.alhem.net/jsockets/index.html
Febrero 2008

López Hernández, F., JNI: Java Native Interface, Octubre, 2007.

Socket de Internet
http://es.wikipedia.org/wiki/Socket_de_Internet#Librer.C3.ADas_de_sockets
Febrero 2008

SolarSockets C++
http://www.solarsockets.solar-opensource.com/index.php/Portada
Febrero 2008

Winsock Programmer's FAQ Section 6.1: Basic Example Programs
http://tangentsoft.net/wskfaq/examples/basics/
Febrero 2008

ENTORNO 3D

Leriche Vázquez, R., Notas sobre OpenSceneGraph,Universidad Nacional Autónoma de Mexico, Departamento de Realidad Virtual, 22 de Mayo 2007.

OpenSceneGraph
http://www.openscenegraph.org/projects/osg/wiki/Support/Tutorials
Febrero 2008

Tutorials Blender
http://www.blender.org/education-help/tutorials/
Junio 2008

Introducción al 3D StudioMax, **Centro de Tecnología Informática,**Universidad de Navarra