Universidad Nacional Autónoma de México

Posgrado en Ciencia e Ingeniería de la Computación

Arquitectura de inteligencia artificial para sistemas multiagente colaborativos

T E S I S

que para obtener el grado de:

Maestro en Ingeniería
(Computación)

P R E S E N T A

Emmanuel Hernández Hernández

Director: Dr. Jesús Savage Carmona

Ciudad Universitaria, México D.F., 2008

Dedicatoria

A la U.N.A.M., mi alma mater y mi casa.
A mi esposa Nadxelle
A mis padres Rafael, Gloria y mis hermanos Alberto y Eunice.

Agradecimientos

Quiero agradecer a todos aquellos que me ayudaron a elaborar este trabajo, pero en particular. . .

A la Universidad Nacional Autónoma de México, por el privilegio de ser parte de ella.

A CONACYT por el apoyo económico recibido durante mis estudios.

A Javier Jiménez amigo y compañero del laboratorio de Bio-Robótica.

Al Dr. Jesús Savage Carmona, mi director de tesis y tutor desde la licenciatura. Agradezco la confianza que ha depositado en mi y la estima en que me tiene.

Índice general

Índice de figuras

Índice de cuadros

Resumen

Cada día se encuentran en nuestro entorno diario una gran cantidad de aparatos electrónicos que empiezan a tener agentes dentro de su funcionamiento cotidiano. Hornos de microondas que perciben y toman decisiones con base en la información de estado interno, externo y temporizadores. Hornos de gas que autorregulan su temperatura, monitorean fugas de gas, perciben y reaccionan ante estímulos externos; refrigeradores que están al tanto de la cantidad y tipo de artículos que mantienen en refrigeración [Wei99]. Si bien dichos agentes mantienen en su mayoría un comportamiento reactivo, poco a poco empiezan a tener rasgos más característicos de agentes inteligentes o también llamados agentes autónomos. En la rama automotriz y aeronáutica, es común que se tengan muchos sistemas pequeños que perciben y reaccionan de forma local; ahora cada vez existe mas colaboración entre dichos sistemas para poder afrontar problemas mas complicados como frenar bajo condiciones de nieve o lluvia pesada, o poder trasladar con éxito en las pistas de un aeropuerto un avión entre puertas de despegue diferentes ante todo el movimiento constante intrínseco de la operación diaria. Problemas complicados en el área de robótica implican la coordinación de diferentes robots en un entorno para cumplir una meta común [Yan06].

Con respecto a los mundos virtuales, incluyendo dentro de estos a los videojuegos, es posible encontrar a un agente inmerso en un mundo dinámico distribuido, que necesita, en la gran mayoría de las ocasiones, colaborar con otros agentes inmersos en el mismo mundo. [Can06]. La colaboración de agentes virtuales en los años recientes, ha permitido

experimentar diferentes situaciones y retos novedosos a las personas que usan videojuegos convirtiéndose en una de las técnicas mas solicitadas por compañías expertas en el ramo en la actualidad. [Cor07]. El uso de agentes inteligentes en mundos virtuales para realizar trabajo colaborativo en ocasiones implica el uso de tecnología existente de videojuegos para su buen desempeño, como lo indica [Ado01]. El gran impulso que han tenido los videojuegos en la actualidad, ha permitido el uso de tecnología en situaciones en donde la visualización de mundos virtuales de gran calidad es importante, como puede apreciarse en el trabajo de [Jac03] durante las olimpiadas Atenas 2004. En el campo de videojuegos, han existido muchos enfoques que buscan aprovechar las metodologías propias de la inteligencia artificial para poder generar arquitecturas de software reutilizables que hagan uso de bajos recursos computacionales (15 % de tiempo de procesador) [Ork05]. La gran mayoría utiliza máquinas de estado finito, sin embargo, esta metodología evita que podamos encontrar comportamientos emergentes propios de los agentes como se definen en la inteligencia artificial moderna; y al mismo tiempo nos aleja del aprendizaje no supervisado [Ork07]. Una mejor metodología debe de poder aprovechar los conceptos de agentes y aprendizaje de forma combinada como se propone en [Ork06].

Por otro lado, también es importante denotar la valía de contar con una arquitectura multiagentes funcional. Dentro de la evaluación del estado del arte realizada, no se encontraron muchas implementaciones de la metodología elegida, o al menos que fueran libres o utilizables. Por lo tanto, la implementación aquí presentada es importante, pues le da valor extra al sistema desarrollado. Además, el código será liberado para que la gente aporte en ideas, mejoras y optimizaciones al mismo.

El objetivo que se busca es desarrollar un conjunto de librerías que permitan incorporar, en sistemas gráficos tridimensionales, los mecanismos necesarios para contar con múltiples agentes organizados para colaborar entre si y con ello resolver problemas diversos. Dentro de los problemas a los que se buscarán proponer soluciones, se encuentran la búsqueda de objetos que cambian de posición de forma dinámica y el combate entre distintos grupos de agentes. La diferencia principal entre esta propuesta y otras existentes ([MD99] y [Che03]), radica en las consideraciones propias para ambientes tridimensionales interactivos; es decir, contarán con pocos recursos computacionales dedicados debido a que compartirán un único procesador, aun cuando puede tener múltiples núcleos, para realizar a la par los cálculos tradicionalmente más costosos, todo ello sin perder el rendimiento del apartado gráfico. También se incluirá un sistema social que ayude durante el proceso de toma de decisiones, considerando colaboradores y antagonistas en distintas jerarquías configurables.

Las librerías tienen como motivación el trabajo de [Ork06], que ha sido mencionado como la mejor aproximación a inteligencia artificial basada en agentes implementada de forma efectiva en un videojuego [Mil06]. Por lo tanto, se tomará como base su trabajo y se incorporarán nuevos enfoques como los que se proponen en [Nil98], [LD07] y [Fin92]. Otro enfoque que se incorporará, es el uso de programación genética para algunos comportamientos reactivos ya establecidos; [Ork06] deja la puerta abierta a dicha variante debido a la propuesta que hace.

Organización de la tesis

La tesis está dividida en cinco capítulos:

- El **capítulo 1** ofrece un panorama de los sistemas multiagentes, sus aplicaciones y los temas tratados a lo largo de este trabajo.
- En el **capítulo 2** se hace una revisión del estado del arte de las técnicas usadas en la generación de sistemas multiagentes. Posteriormente se revisa uno de ellos, el más exitoso en videojuegos hasta el momento: *C4*. Finaliza con la vertiente utilizada en el trabajo.
- El **capítulo 3** explica en detalle el funcionamiento de la arquitectura, puesto que es necesario comprender su funcionamiento para una posterior implementación.
- El contenido del **capítulo 4** describe la elaboración de la arquitectura de inicio a fin, la implementación de cada fase, su encapsulamiento para un modelo orientado a objetos y ejecución *multihilos*.
- En el **capítulo 5** se discuten las pruebas realizadas al sistema, sus aciertos y errores.

Finalmente se incluyen las conclusiones y trabajo futuro con respecto del trabajo, así como dos apéndices. Primero (apéndice A) muestra el programa completo de *CLIPS* utilizado. El segundo, apéndice B, muestra los detalles del experimento realizado para evolucionar el comportamiento de pelea para los agentes usando funciones definidas automáticamente.

Notación

A continuación se presenta la notación utilizada a lo largo de este trabajo. En el cuadro se presentan las abreviaturas, acrónimos y señalizaciones que aparecen a lo largo de los capítulos.

AABB	Caja envolvente alineada con los ejes cartesianos, *Axis Aligned Bounding Box*.
ADF	Funciones definidas automáticamente. *Automatical Defined Functions*
AI	Inteligencia Artificial, *Artificial Intelligence*.
ALU	Unidad Aritmética Lógica, *Arithmetic Logic Unit*.
AOBB	Caja envolvente orientada con los ejes cartesianos, *Axis Oriented Bounding Box*.
API	Interfaz de programación de aplicaciones, *Application Program Interface*.
BDI	Creencia-Deseo-Intención, *Belief-Desire-Intention*.
BS	Esfera Envolvente, *Bounding Sphere*.
BSM	Máquinas de estado del comportamiento, *Behavioural State Machines*.
C4	*Creature Smarts*.
CG	Cómputo Gráfico.
CLIPS	*C Language Integrated Production System*.
CPU	Unidad Central de Proceso *Central Processor Unit*.
CORBA	*Common Object Request Broker Architecture*.
DAI	Inteligencia Artificial Distribuida, *Distribuited Artificial Inteligence*.

DC	Cómputo Distribuido, *Distribuited Computing*.
FOV	Campo de visión, *Field of View*.
FEAR	*First Encounter Assault Reccon*.
F	Cuadros por segundo, *Frames per second*.
FPS	Juego de disparos de primer persona, *First Person Shooter*.
GUI	Interfaz gráfica de usuario*Graphic User Interface*.
HLSL	*High Level Shader Language*.
HTTP	*Hypertext Transfer Protocol*.
HTML	*HyperText Markup Language*.
KQML	*Knowledge Query and Manipulation Language*.
LOH	Distancia de audición, *Lenght of Hearing*.
LOV	Distancia de visión, *Lenght of View*.
MAS	Sistemas Multi Agentes, *Multi Agent Systems*.
MD3	Formato de modelos animados.
MUSS	Sistema de múltiples soldados, *Multiple Soldier System*.
OAA	Arquitectura Abierta para Agentes, *Open Agent Architecture*.
OLE	*Object Linking and Embedding*.
PG	Programación genética.
PK3	Archivo comprimido en formato *ZIP*.
RAM	Memoria de Acceso Aleatorio *Random Access Memory*
RV	Realidad Virtual.
SGI	Sistema Gráfico Interactivo.
SQL	*Structured Query Language*.
STRIPS	*Stanford Research Institute Problem Solver*.
UCS	Unidad de Control de Secuencia.
UDP	*User Datagram Protocol*.
XML	*Extensible Markup Language*.

Capítulo 1

Introducción

Como resultado de los recientes avances tecnológicos en materia de cómputo, hoy en día se tienen sistemas poderosos formados por *CPUs* con gran densidad de componentes. El equipo de súper cómputo de hace unos años (memoria RAM del orden de *gigabytes* y discos duros de gran capacidad) es ahora el *hardware* común de las computadores personales.

Tales avances han producido un vertiginoso desarrollo de *software*, antes difícil de implementar debido a las limitaciones técnicas. Entre las aplicaciones que demandan mayor consumo de recursos, se encuentran aquellas referentes al entretenimiento, el cómputo científico y procesos de automatización. La industria del entretenimiento es de las que más ha propiciado el desarrollo tecnológico. Tan sólo basta con mirar hacia la creciente y ya enorme industria de las gráficas por computadora. En ella pueden considerarse a productos como juegos de video y utilerías de postproducción utilizadas en televisión y cine: *game engines*, recorridos virtuales, diseño asistido por computadora, aplicaciones de procesamiento de imágenes y video, entre otras. En la misma industria, en el área de la ciencia e investigación, destacan las aplicaciones empleadas para hacer procesamiento de señales, cálculo numérico, simulaciones, aplicaciones médicas, de ingeniería y militares, sólo por nombrar algunas.

Para el caso particular del estudio aquí presentado, el ámbito que más importa

es el de los sistemas multiagentes *MAS* con un enfoque claro a la implementación de la arquitectura.

1.1. Agente

No existe una definición universalmente aceptada de agente, sin embargo, existe como consenso que cuentan con autonomía como parte central; otras características adicionales como el aprendizaje pueden presentarse o no en diferentes contextos del uso de agentes. Apoyándose en la definición de [Woo95] decimos:

> **Definición 1** *Un agente es un sistema de cómputo que se encuentra situado en un ambiente; el agente es capaz de realizar acciones de forma autónoma en el ambiente para cumplir con sus objetivos de diseño.*

De esta forma, un agente se encuentra sumergido en un ambiente sobre el cual no tiene control, pero si puede influenciarlo para lograr el objetivo que persigue y la forma de influenciar el ambiente, se realiza por medio de efectores para realizar acciones. Las acciones pueden verse como el repertorio de posibilidades con el que cuenta el agente para lograr realizar la meta para la cual ha sido construido. No todas las acciones son posibles de realizarse todo el tiempo, debido a que puede ser posible que haya que cumplir con ciertas precondiciones para su ejecución; por ejemplo, un agente que compra acciones en la bolsa de valores, puede realizar la acción de comprar, sin embargo, es necesario tener primero el dinero suficiente para realizar dicha compra [Wei99]. El problema principal al cual se enfrenta un agente para cumplir con su objetivo, consiste en decidir cual acción realizar para las condiciones en las cuales se encuentra el ambiente.

Básicamente un agente es un objeto activo con la habilidad de percibir, razonar y actuar; se supone que el agente tiene conocimiento, así como también mecanismos para manipularlo e inferir a partir de él. Un agente también cuenta con la capacidad de comunicarse, es decir, enviar o recibir mensajes.

1.2. Ambientes

El medio donde los agentes se desenvuelven es de gran importancia para el diseño de una arquitectura correcta. [Rus95] sugiere la siguiente clasificación de ambientes por sus propiedades:

Accesibles e Inaccesibles . Un ambiente accesible es cuando el agente puede obtener información completa, precisa y actualizada del estado del ambiente; por lo tanto un ambiente inaccesible no cumple con alguna o algunas de las características mencionadas.

Determinista y no determinista. Un ambiente determinista tiene una única reacción a una acción, es decir, las consecuencias de una acción son conocidas previamente. En un ambiente no determinista no es posible saber con seguridad cual será la consecuencia a una acción determinada.

Episódico y no episódico. En un ambiente episódico, el desempeño de un agente es dependiente del número discreto de episodios existente sin tener repercusiones directas entre las decisiones que se realicen en cada episodio.

Estático y dinámico. Un ambiente estático no presenta cambios a menos que haya interacción por parte del agente. En un ambiente dinámico, existen cambios que no son realizados de forma necesaria por el agente.

Discreto y continuo. Un ambiente es discreto si existe un número finito de acciones y percepciones. Un ambiente continuo, cuenta con una cantidad infinita ya sea de acciones o percepciones.

Otra forma de clasificar las características del ambiente está especificado en el siguiente cuadro (Cuadro 1.1):

Propiedad	**Definición**
Conocido	El ambiente es conocido por el agente
Predecible	Las situaciones en el ambiente pueden ser predecidas
Controlable	El agente puede modificar el ambiente
Histórico	El estado del ambiente depende de la historia del mismo o solo del instante actual
Teolológico	Los elementos del ambiente tienen un propósito
Tiempo real	El ambiente cambia mientras el agente delibera

Cuadro 1.1: Características de ambientes respecto a los agentes

1.3. Agentes inteligentes

No todos los agentes que tenemos en nuestro alrededor son agentes inteligentes. El problema de determinar si un agente es inteligente o no puede verse como determinar que es la inteligencia, lo cual es complicado. En este trabajo, se considerara que un agente es inteligente si es capaz de tener un comportamiento autónomo flexible para cumplir los objetivos que tenga propuestos [Woo95]. Las características del concepto de flexibilidad que se buscan son:

Reactivo. Son agentes que perciben su ambiente y reaccionan ante estímulos en el momento adecuado para cumplir con sus objetivos.

Pro activos. Son agentes que toman la iniciativa, es decir, no esperan a un estímulo externo para realizar acciones que le lleven a cumplir con sus objetivos.

Sociabilidad. Son agentes capaces de comunicarse con otros agentes para satisfacer sus objetivos, que para este trabajo es de suma importancia.

1.4. Arquitecturas de agentes

1.4.1. Arquitecturas abstractas

De forma abstracta, el ambiente donde un agente se desenvuelve puede caracterizarse como un conjunto de estados:

$$S = \{s_1, s_2, \ldots\}$$

En cualquier instante, podemos suponer que el agente se encuentra en uno de dichos estados. De la misma manera, podemos suponer que las acciones que es capaz de realizar pueden ser representadas como un conjunto A de acciones:

$$A = \{a_1, a_2, \ldots\}$$

De lo anterior, podemos decir que un agente puede verse como una función del tipo:

$$Accion : S^* \rightarrow A$$

Que asocia secuencias de estados del ambiente a acciones. El agente asocia la acción necesaria a realizar con el estado actual mediante experiencias proporcionadas. Estas secuencias son representadas como una secuencia de estados del ambiente.

1.4.2. Arquitecturas para Agentes Inteligentes

Agentes basados en lógica

La creación de agentes inteligentes supone que el estado del ambiente puede ser representado mediante alguna representación simbólica. El estado interno de los agentes puede ser representado como una base de datos que contiene fórmulas tradicionales de cálculo de predicados, estos predicados son un conjunto de *creencias* del estado del ambiente. En esta arquitectura, el proceso de toma de decisiones es modelado por un conjunto de reglas de deducción. Cada decisión modifica el ambiente por lo tanto se agregan nuevos predicados para representar el estado actual. El problema con este tipo de representaciones tiene se debe a que es muy difícil manejar información que cambia con el tiempo; también es difícil enfrentar problemas de tipo procedural.

Arquitecturas reactivas

Esta arquitectura tiene tres principios en los cuales se basa:

- Evitar representaciones simbólicas; la toma de decisiones se realizará con base en la manipulación sintáctica de dichas representaciones.
- La idea de comportamiento inteligente y racional esta asociado al ambiente donde el agente esta inmerso; es un producto de su interacción con él.
- El comportamiento inteligente emerge de la interacción de más comportamientos simples.

Un ejemplo de este tipo de arquitectura es la desarrollada por *Rodney Brooks* la cual llamó *arquitectura de subdivisión de módulos* [Ron86]. Existen dos características principales de esta arquitectura en particular:

Conjunto de comportamientos para cumplir tareas. La implementación de *Brooks* se realiza mediante un conjunto de máquinas de estado que corresponden a un conjunto de reglas del tipo:$situacion \rightarrow accion$

Ejecución de múltiples reglas a la vez. En su implementación, *Brooks* propone que se establezca un sistema de jerarquías en la que los comportamientos se organizan en capas. Las capas mas bajas de la jerarquía inhabilitan a las capas mas altas, siendo las capas mas altas las de mayor prioridad.

La función de decisión *acción* se realiza mediante un conjunto de comportamientos y además se apoya en una relación de inhibición; en este caso, un comportamiento es una pareja (c, a) donde $c \subseteq P$ y P es un conjunto de preceptos llamados condiciones y $a \in A$ es una acción. Un comportamiento (c, a) se ejecutará cuando se encuentre en un estado $s \in S$ si y sólo si $s \in C$. Llamaremos $Com = \{(c, a) | c \subseteq P \; y \; a \in A\}$ el conjunto de reglas de comportamiento. Asociado a su conjunto de comportamientos, existe un subconjunto de reglas $R \subseteq Com$ que son una relación binaria que describe la inhibición en el conjunto de comportamientos: $\prec \subseteq R \times R$. La notación es del tipo: $b_1 \prec b_2$ si $(b_1, b_2) \in \prec$, lo cual significaría que b_1 es de menor jerarquía que b_2.

De las ventajas de esta metodología se tiene: *simplicidad, economía, poder llevar la secuencia en la que los eventos se realizan, robustez contra errores y elegancia en el diseño.* Por otro lado, existen problemas al usar este tipo de arquitecturas: es necesario contar con suficiente información del ambiente, los agentes sólo pueden hacer uso de información local, es difícil incluir aprendizaje con base en la experiencia; también es complicado de definir el comportamiento exacto ante una situación ya que las respuestas son del tipo emergente y, por lo mismo, no hay una metodología para poder generar la respuesta exacta ante una situación; finalmente, incrementar el número de capas en la arquitectura del agente hace más difícil diseñar su comportamiento.

Arquitecturas creencia-deseo-intención

Esta arquitectura se basa en los principios filosóficos del *razonamiento práctico*, es decir, el proceso de decidir, momento a momento, que acción realizar para llevarnos a nuestras metas. Esta arquitectura es conocida cambien como arquitectura *BDI* por su nombre en inglés *Belief-Desire-Intention*.

El razonamiento práctico incluye dos procesos importantes: decidir cuáles *metas* queremos realizar y *cómo* vamos a lograr dichas metas. Al primer proceso le llamamos *deliberación* y al segundo razonamiento de *medios y fines*.

En este caso, las intenciones siguen los siguientes comportamientos en el razonamiento práctico:

- Conducen el razonamiento de medios y fines, porque involucra decidir como lograr una meta.
- Restringen las deliberaciones futuras.
- Son persistentes.
- Influyen en las creencias sobre las cuales el razonamiento práctico *futuro* es creado.

Un problema en el diseño de agentes por razonamiento práctico es conseguir un buen balance entre las diferentes tareas a realizar. En ocasiones es necesario que el agente elimine algunas intenciones, ya sea porque no son posibles de alcanzar, se han alcanzado o ya no son requeridas debido al cumplimiento de otra acción. También es posible que el agente llegue a un punto donde necesita detenerse para *reconsiderar* las intenciones que tiene cargadas en el sistema. Reconsiderar puede ser un proceso costoso computacionalmente hablando. Agentes que no reconsideren lo suficiente pueden realizar tareas que ya no son necesarias, y por el contrario, agentes que reconsideran demasiado no cuentan con el tiempo suficiente para poder ejecutar las tareas para las cuales fueron diseñados. En el trabajo de [KD91] se menciona el estudio ante dos tipos diferentes de agentes: agentes *seguros y cautelosos*; de sus resultados se obtuvieron experimentos referentes al factor γ la velocidad de cambio del ambiente:

- γ bajo. *El ambiente* no *cambia rápidamente*. Los agentes seguros tienen mejor desempeño respecto a los cautelosos, debido a que los segundos pasan demasiado tiempo reconsiderando las acciones que deben de realizar.
- γ alto. *El ambiente cambia rápidamente*. Los agentes cautelosos tienen mejor desempeño respecto a los seguros, debido a que los segundos no pueden percibir condiciones propias del entorno que les ocasionan que los hechos que tienen registrados han cambiado.

Existen siete componentes clave en el diseño de agentes *BDI*:

- Establecer el *conjunto de creencias* acerca del ambiente donde se encuentra el agente
- Crear una *función creencia-revisión*, que procesa las percepciones de entrada, las creencias actuales del agente y con base en ello, genera nuevas creencias
- *Función generadora de opciones*, determina las opciones disponibles basándose en las creencias disponibles acerca del ambiente y sus intensiones actuales.

- *Conjunto de opciones actuales*, que representan las posibilidades de acciones a ejecutar por parte del agente.
- *Función de filtrado*. Representa el proceso de deliberación de intenciones con base en sus creencias, deseos e intensiones.
- *Conjunto de intenciones*. Representan la meta actual del agente.
- *Función de selección de acciones*. Determina la acción a realizar con base en sus intenciones.

La figura 1.1 muestra un diagrama general de la implementación de esta arquitectura.

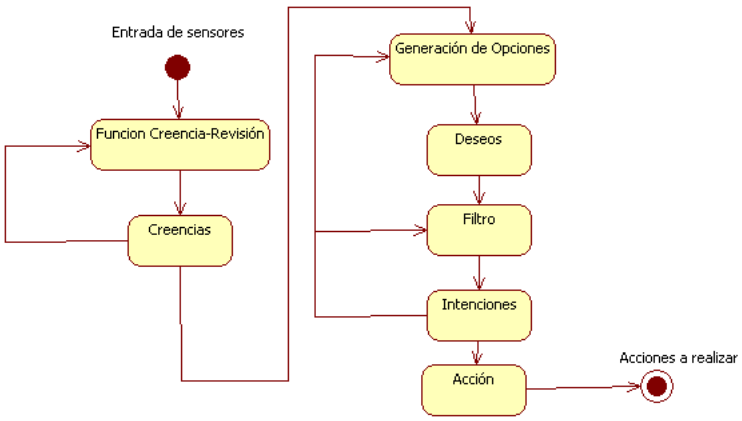


Figura 1.1: Diagrama general de una arquitectura creencia-deseo-intención

El estado de un agente en cualquier momento, queda descrito por una terna de elementos (B, D, I), donde $B \in$ *Creencias*, $D \in$ *Deseos* y $I \in$ *Intenciones*.

La función *creencia-revisión* Fcr es del tipo:

$Fcr : \varphi(B) \times P \rightarrow \varphi(B)$

La función *generación de opciones* Fgo es del tipo:

$Fgo : \varphi(B) \times \varphi(I) \rightarrow \varphi(D)$

El proceso de deliberación, la *Función de filtro* FF, es representada por:

$FF : \varphi(B) \times \varphi(D) \times \varphi(I) \rightarrow \varphi(I)$

Siendo esta última de las funciones más complicadas debido a que tiene que cumplir con tres funciones diferentes: la primera es que debe *descartar* cualquier intención que no sea posible alcanzar o que el costo de alcanzarla es mayor al beneficio recibido. La segunda es *conservar* las intenciones que aun son necesarias de alcanzar y de las cuales se espera obtener un beneficio positivo. La tercera consiste en *adoptar* nuevas intenciones, ya sea para lograr las intenciones actuales o explorar nuevas posibilidades.

La función de ejecución *FE* es de la forma:

$Fcr : \varphi(I) \times P \rightarrow A$

Finalmente, las arquitecturas *creencia-deseo-intención* son similares al proceso de razonamiento que se lleva día a día para cumplir con nuestros objetivos en la vida cotidiana. Debido a la familiaridad de este esquema de trabajo, puede parecer simple su utilización en casos prácticos; sin embargo, el problema principal radica en implementar la funciones necesarias descritas anteriormente.

Arquitecturas multicapa

Cuando se cuentan con los comportamientos reactivos y pro activos de un agente, es natural pensar en descomponer dichos comportamientos en subsistemas para poder cumplir con el requerimiento. Los subsistemas generados llevarán ciertas reglas de unión y jerarquía para interactuar entre cada una de las *capas* o sistemas.

En general, es posible identificar dos tipos de arquitecturas basadas en capas:

- *Arreglo horizontal.* Las capas del sistema reciben información directamente de los sensores de entrada. De esta manera, cada capa produce resultados a la salida que sugieren que acción realizar para la situación en la que se encuentra. La figura 1.2 muestra un ejemplo de configuración de arreglo horizontal.
- *Arreglo vertical.* En este esquema, tanto las acciones de salida como los sensores de entrada tienen contacto con, cuando mas, una capa del sistema. La figura 1.3 muestra un ejemplo de configuración de arreglo vertical.

El problema de utilizar una arquitectura horizontal es poder decidir cual de todas las sugerencias de acciones es la apta para la situación en la cual se encuentra el agente en el momento, dicho elemento se le conoce como *mediador*. Diseñar un *mediador* puede ser complicado porque implica considerar todas las interacciones entre las capas. Para un

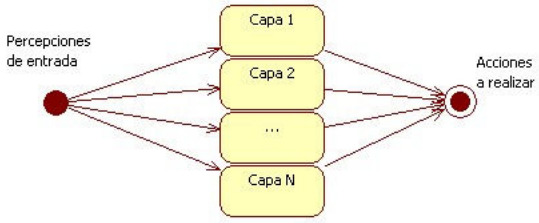


Figura 1.2: Arquitectura multicapa horizontal

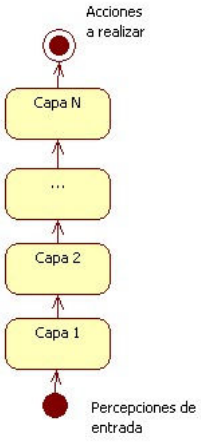


Figura 1.3: Arquitectura multicapa vertical

problema con n capas y m posibles acciones, tenemos un conjunto de posibilidades de orden n^m, lo cual ocasiona un cuello de botella al tener que procesar una gran cantidad de acciones posibles antes de tomar una decisión. Las arquitecturas verticales pueden ser subdivididas en arquitecturas de *una pasada* (Figura 1.3) y de *dos pasadas* (Figura 1.4) . En las arquitecturas de una sola pasada, el control pasa de forma secuencial entre cada capa, hasta que la capa final genera la acción de salida resultante. En las arquitecturas de dos pasadas, la información llega hasta la capa superior y luego el control pasa a las capas inferiores.

En la actualidad las arquitecturas multicapa son de lo más popular. La división de funciones en diferentes módulos, es natural llevarla a un esquema de capas. Su principal problema es que carecen de claridad conceptual y semántica, así como también es complicado establecer las relaciones entre las distintas capas.

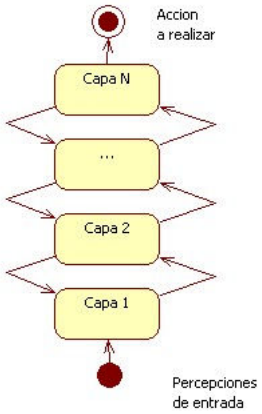


Figura 1.4: Arquitectura multicapa vertical de dos pasadas

1.5. Sistemas multiagente

Existen situaciones en las cuales un sólo agente necesita interactuar con otros agentes para poder resolver problemas complicados. Cuando en un ambiente existen múltiples agentes (MAS - por sus siglas en inglés Multi Agent Systems), pueden llegar a ser muy numerosos y es mejor manejarlos como sociedades de agentes. Cuando un agente necesita intercambiar información con otro, es necesario establecer ciertas reglas, a las que llamaremos *protocolo* para poder realizar una comunicación efectiva. En el diseño de un protocolo de comunicación es necesario considerar los siguientes tipos de mensajes:

- Proponer una acción
- Aceptar una acción
- Rechazar una acción
- Cancelar una acción
- Estar en desacuerdo de ejecutar una acción

Es posible considerar otros tipos de mensajes, sin embargo, están sujetos a la implementación específica para resolver el problema para el cual fue diseñado.

1.5.1. Características de los ambientes multiagentes

Las características principales de los ambientes multiagentes son:

- Cuentan con su propia infraestructura al especificar su protocolo de comunicación e interacción.
- Son típicamente abiertos y no tienen diseño centralizado
- Contienen agentes que son autónomos y distribuidos, además, pueden ser individualistas o cooperativos

También, dependiendo del tipo de ambiente, se presentan las siguientes características en el diseño (Cuadro 1.2):

1.5.2. Comunicación entre agentes

Coordinación

Los agentes se comunican para lograr mejores resultados propios o mejores resultados en común. La comunicación permite a los agentes coordinar sus acciones y su comportamiento, generando comportamientos coherentes en sociedad.

La coordinación es una propiedad de un sistema de agentes realizando una actividad en un ambiente compartido. Llamaremos *cooperación* a la coordinación de agentes que no son antagonistas; la *negociación* es la coordinación entre agentes que compiten entre si o agentes individualistas. Para lograr que dos agentes puedan cooperar correctamente, cada uno de ellos debe tener el modelo del otro agente, para así poder anticipar la acción del otro. La coherencia se refiere a que tan bien un sistema se comporta como una sola unidad.

La figura 1.5 muestra la forma en la que se encuentra organizada la coordinación entre agentes:

Significado

Existen tres aspectos en los estudios de comunicación:

- *Sintaxis*. Cómo están estructurados los símbolos en la comunicación.

Propiedad	Elementos a considerar
Autonomía	Arquitectura interna Protocolo de interacción Plataforma Lenguaje a utilizar
Infraestructura de comunicación	Memoria compartida o envío de mensajes Conectados o sin conexión Punto a punto, multicast o broadcast Síncrono o asíncrono
Directorio de servicios	Directorio *amarillo* o *blanco*
Protocolo de mensajes	KQML, HTTP, HTML, OLE, CORBA
Servicios para mediar	Por transacciones u ontológicos
Servicios de seguridad	*Timestamps* o Autentificación
Soporte de operaciones	Almacenamiento, redundancia recuperación, conteo

Cuadro 1.2: Características de sistemas multiagentes

- *Semántica*. Lo que los símbolos significan
- *Pragmatismo*. Cómo son interpretados los símbolos.

El significado es la combinación de la semántica y el pragmatismo. Los agentes se comunican para entender y ser entendidos como se menciona en [Sin97]:

- *Descriptivo o prescriptivo*. Algunos mensajes describen fenómenos, otros describen comportamientos. La mayoría de los lenguajes están diseñados para intercambiar actividades y comportamientos.
- *Personal o convencional*. Un mensaje puede ser interpretado de forma particular por un agente; sin embargo, entre mas agentes puedan compartir la misma interpretación, será más fácil la comunicación.

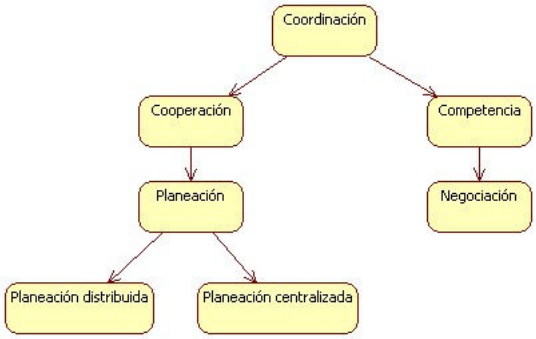


Figura 1.5: Taxonomía de la coordinación entre agentes

- *Objetivo o subjetivo*. La información proporcionada al agente debe de ser objetiva, de lo contrario es posible que sea imposible interpretarla.
- *Hablante, oyente o sociedad*. La elaboración del mensaje debe de basarse en quien va a escucharlo, ya sea un solo agente o un grupo de ellos.
- *Semántico o pragmático*. Considera como los agentes deben de usar la comunicación.
- *Contextualización*. Algunos mensajes no pueden ser entendidos por si solos, es posible que sea necesario apoyarse en la historia del ambiente.
- *Cobertura*. El lenguaje utilizado debe de ser capaz de cubrir cualquier mensaje que se desee enviar.
- *Identidad*. El mensaje a enviar debe de contener a quien o quienes va destinado.
- *Cardinalidad*. Un mensaje enviado hacia un solo agente puede ser entendido diferente si se envía a múltiples agentes.

Tipos de mensajes

Existen dos tipos básicos de mensajes: *hechos y preguntas*. Cada agente, ya sea activo o pasivo, debe tener la habilidad de aceptar información. En su forma más sencilla, esta información se le hace llegar al agente en forma de un hecho. Un agente debe de ser capaz de contestar preguntas, primero teniendo la capacidad de recibirlas y después la capacidad de enviar la respuesta en forma de un hecho. Desde el punto de vista de envío y recepción de mensajes,

decimos que un agente es pasivo si espera preguntas para contestar, de la misma manera, decimos que un agente es activo si realiza preguntas. El cuadro 1.3 muestra las capacidades de los agentes desde el punto de vista de envío recepción.

Característica Agente	**simple**	**pasivo**	**Activo**	**ProActivo**
Recibe hechos	•	•	•	•
Recibe preguntas		•		•
Envía hechos		•	•	•
Envía preguntas			•	•

Cuadro 1.3: Capaciades de agentes

Niveles de comunicación

Los niveles de comunicación son especificados en diferentes capas. Las capas más bajas corresponden al medio de interconexión; las capas de nivel intermedio corresponden al formato y/o la sintaxis de la información transmitida; finalmente en la capa superior se encuentra el significado de la información y el tipo de mensaje.

Existen comunicaciones *binarias*, es decir un agente con un agente y *n-arias* que corresponden a comunicación de un agente con muchos agentes.

En general, un mensaje dentro un protocolo de comunicación debe de tener los siguientes cinco elementos:

- Fuente
- Destino o destinos
- Lenguaje utilizado
- Funciones de codificación y decodificación
- Acciones a ser realizadas por él o los receptores

KQML

KQML son las siglas de *Knowledge Query and Manipulation Language*, que quiere decir, *Lenguaje de solicitud y manipulación de conocimiento*. Este lenguaje, fue diseñado buscando ser

universal incluyendo dentro del cuerpo del mensaje, toda la información necesaria para poderlo entender. El protocolo tiene la siguiente estructura general:

(KQML-formato
:fuente <word>
:destino <word>
:lenguaje <word>
:ontología <word>
:contenido <datos>
...)

Como puede observarse los datos que son necesarios para formar un mensaje usando KQML, son independientes del dominio donde se encuentren. La semántica del mensaje se forma por los campos: *contenido, lenguaje y ontología*. De esta manera KQML es un encapsulador de la información a ser enviada; se han registrado algunas aplicaciones que hacen uso de KQML encapsulando *Prolog,Lisp y Sql*.

Desde el punto de vista de KQML, los agentes pueden ser vistos como clientes o servidores. Su comunicación puede ser síncrona o asíncrona (Figura 1.6).

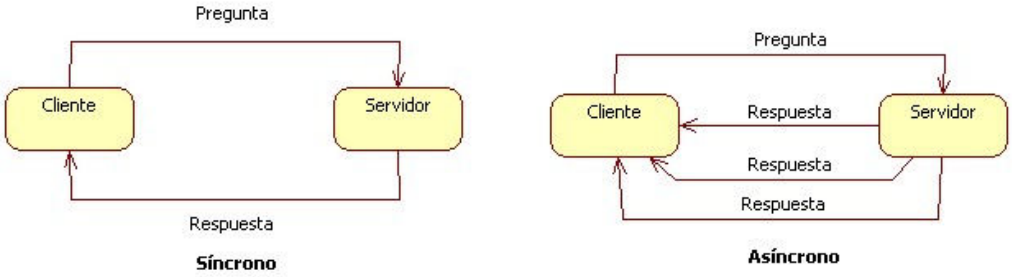


Figura 1.6: Comunicación síncrona y asíncrona usando KQML

1.5.3. Sociedades de agentes

Un grupo de agentes puede formar una sociedad en la cual llevan acabo diferentes papeles. El grupo define los papeles y los papeles definen los compromisos o responsabilidades asociadas. Cuando un agente se une a un grupo, puede tener uno o mas papeles a desempeñar.

Los grupos definen el contexto social en el cual los agentes interactuan. Este tipo de agrupamiento involucra abstracciones de la sociología y de la teoría de modelos de sociedades.

Los compromisos sociales son los compromisos que un agente tiene con otro agente; estos compromisos pueden verse como maneras flexibles de restringir el comportamiento de un agente autónomo.

Se dice que un grupo de agentes forman un equipo cooperativo cuando:

- Todos los agentes comparten una meta común.
- Cada agente requiere realizar sus propias tareas para lograr la meta común al grupo o aun subgrupo.
- Cada agente acepta una petición para lograr su cometido.

1.6. Algoritmos genéticos y programación evolutiva

1.6.1. Algoritmos genéticos

Las primeras muestras aparecen en artículos publicados por Holland en la Universidad de Michigan durante los años sesentas ([Coe03]).

Es una estrategia de búsqueda que está inspirada en los mecanismos de adaptación de las entidades biológicas y es utilizada generalmente en problemas de optimización. En esta técnica se cuentan con un conjunto, llamado población, de posibles soluciones a las que se les denomina individuos. Dichos individuos tienen cierto desempeño en la solución del problema que se desee resolver; la idea es proponer nuevas soluciones (nuevos individuos) a partir del desempeño mostrado por las soluciones actuales. Para poder generar nuevos individuos se hace uso de dos operadores genéticos denominados cruza y mutación; estos operadores genéticos actúan sobre la estructura genética de dos individuos (cruza) o de uno solo (mutación) con la esperanza de obtener un individuo con mejores resultados de desempeño. Los individuos que forman parte de la nueva población son resultado de las operaciones realizadas entre los individuos de las poblaciones anteriores. Existen diferentes métodos estocásticos para seleccionar sobre que individuos se deben de operar tanto cruza como mutación. En general los algoritmos genéticos se utilizan en problemas de optimización de funciones de minimización o maximización [Per08] [Fog00].

1.6.2. Estrategias Evolutivas

El primer estudio que se tiene noticia es de Bienert, Rechenberg y Schwefel, en la Universidad Técnica de Berlín en 1965 [Per08].

Una estrategia evolutiva, es un procedimiento estocástico de adaptación paramétrica con un paso adaptativo. Podemos encontrarla en dos esquemas: el $(1 + 1)$ y el $(\lambda \pm \mu)$. En el primer caso, se hace evolucionar un solo individuo haciendo uso de sólo un operador genético, la mutación. Como tal no hay selección porque solo hay un individuo y solo habrá reemplazo si el descendiente es mas apto que el progenitor, se puede garantizar que en un número infinito de intentos se puede encontrar al individuo mas óptimo. En el caso de $(\lambda \pm \mu)$, λ indica el tamaño de la población y μ el de la descendencia; el símbolo $\pm$ se utiliza para indicar que existe reemplazo de individuos por inclusión o por inserción. Existe un mecanismo similar al que tienen los genéticos para descartar a los individuos no factibles. Al igual que los algoritmos genéticos, existe también los operadores de cruza y mutación.

Para poder utilizar una estrategia evolutiva se tiene que tener mucho conocimiento del problema a resolver, por lo que son mejores solucionadores de problemas que un algoritmo genético, ya que el algoritmo genético es de propósito más general. El criterio de reemplazo es de tipo determinista a diferencia de los algoritmos genéticos que es estocástico; el tamaño de la descendencia está fijo en las estrategias evolutivas y todos los individuos pueden estar sujetos a mutación; mientras que en los algoritmos genéticos, la descendencia está fija a su taza de cruce y la mutación es de baja probabilidad [Coe03] [Fog00].

1.6.3. Programación Evolutiva

La primera mención de esta técnica la tiene Lawrence Fogel en San Diego California en los 1960s. Este método se ideó para poder hacer evolucionar soluciones a problemas, sobretodo, en el área de predicción. Básicamente, las posibles soluciones para un problema en específico, tienen la forma de máquinas de estado finitas, las cuales se mutan aleatoriamente y se conservan las mejores. [Per08]

Los algoritmos genéticos hacen énfasis en simular algunos mecanismos que se aplican a los sistemas genéticos naturales, a diferencia de la programación evolutiva, que hace énfasis en la relación de comportamiento entre padres y su descendencia. En la programación evolutiva, la selección se hace de forma probabilística, dando con esto la oportunidad de que soluciones no muy buenas sobrevivan y formen parte de la siguiente generación. A diferencia de los algoritmos genéticos, no se hace un esfuerzo para que se procesen la mayor

cantidad de esquemas ni se tiene tan restringida la operación de mutación; además, tienen una representación más natural del problema y los operadores genéticos son sensibles al contexto y más cerrados [Coe03] [Fog00].

1.6.4. Programación Genética: Funciones definidas automáticamente

Lawrence J. Fogel et al. concibieron el uso de la evolución simulada en la solución de problemas (sobre todo de predicción). Su técnica, denominada "Programación Evolutiva"(*PG*) consistía básicamente en hacer evolucionar autómatas de estados finitos, los cuales eran expuestos a una serie de símbolos de entrada (el ambiente), y se esperaba que, eventualmente, serían capaces de predecir las secuencias futuras de símbolos que recibirían. *Fogel* utilizó una función de "pago"que indicaba que tan bueno era un cierto autómata para predecir un símbolo, y usó un operador modelado en la mutación para efectuar cambios en las transiciones y en los estados de los autómatas que tenderían a hacerlos más aptos para predecir secuencias de símbolos [Coe03].

Las funciones definidas automáticamente o *ADF* -Automatically Defined Functions- por sus siglas en inglés. En los problemas donde se usan *ADF*, cada individuo de la población tiene una representación de tipo arbórea, donde cada nodo es una función que recibe una cantidad variable de parámetros. Es en dicha función donde se usan los parámetros recibidos para generar el cuerpo del programa. Las ramificaciones que se tengan en el árbol denotan condicionales. La figura 1.7 ilustra la forma del árbol que representa a un individuo [Koz94].

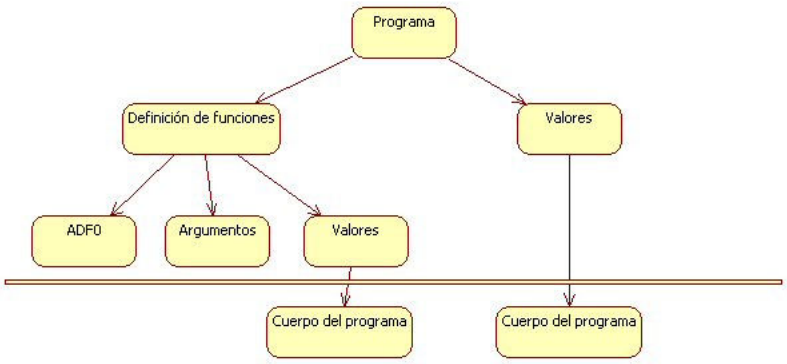


Figura 1.7: Forma de un individuo

De esta forma, podemos generar funciones que a su vez usan funciones como

parámetros de entrada. Con algunas consideraciones adicionales, es posible también generar condicionales dentro de las funciones y con ello poder escribir programas que se generan de forma automática. Dentro de las ventajas del uso de *ADFs* tenemos:

- *Minimización de código*. El uso de funciones minimiza el código que se genera
- *Reutilización de código*. Es posible reutilizar código para problemas similares.
- *Encapsulamiento*. Al identificar una función *de buen desempeño* puede utilizarse tantas veces como sea necesario para mejorar el desempeño total del individuo.

1.7. Sistemas Expertos y *CLIPS*

Un Sistema Experto (SE) es un conjunto de programas de computo ligado a una base de conocimientos adquirida generalmente a través de varios expertos humanos dentro de un dominio especifico (la nube) que incluye un conjunto de algoritmos, reglas o técnicas de razonamiento a través de los cuales se pueden hacer inferencias para la solución de problemas o que permiten dar recomendaciones para el análisis, el diagnóstico, apoyar a técnicas de enseñanza y/o en general recomendar, actuar y explicar sus acciones en actividades en las cuales se requiere generalmente del saber de expertos humanos dentro de una nube especifica [Ser07]. Una herramienta para la construcción de sistemas expertos es *CLIPS*.

CLIPS, *C Language Integrated Production System*, fue creado en 1985 por la *NASA* en el Centro Espacial *Johnson*. Desde antes de 1984 la *NASA* ha utilizado diferentes sistemas expertos algunos basados en *LISP*, sin embargo, no era posible reutilizar el código debido a las diferentes arquitecturas en las cuales se ejecutaban. La *NASA* decidió que lo mejor sería utilizar un sistema experto que estuviera basado en algún lenguaje mucho mas difundido entre distintas plataformas como lo es lenguaje *C*; por lo que comenzó a buscar proveedores que se dieran a la tarea de crear herramientas para dicho propósito. Después de varias deliberaciones, *NASA* llegó a la conclusión de que la mejor decisión sería realizar su propio sistema con base en lenguaje *C* lo cual dio origen a la primera versión de *CLIPS* [Gar08b].

En su estado actual *CLIPS* se mantiene de forma independiente por la comunidad de software libre y es ampliamente difundido para crear sistemas basados en reglas.

Capítulo 2

Estado del arte

El capítulo 1 ofrece los antecedentes y aspectos relacionados con el objetivo de este trabajo con la finalidad de adentrarse en las diferentes aplicaciones en las que pueden emplearse sistemas multiagentes. Como ha sido mencionado con anterioridad, el objetivo es crear una arquitectura multiagentes que sea usada en problemas de mundos virtuales usando gráficas tridimensionales. En específico se busca que sea útil para futuras aplicaciones en videojuegos.

Los antecedentes propios de los *MAS* tiene que ver con la historia de los sistemas de inteligencia artificial distribuidos *DAI* y el cómputo distribuido *DC*.

En la mitad de la década de los 70's, los investigadores en *DAI* y *DC* crearon la teoría básica, las arquitecturas, los experimentos que mostraban cómo la interacción y la división de tareas podían ser aplicadas en la resolución de tareas. Los experimentos que realizaron demostraron que un comportamiento racional inteligente no es atribuido a computadoras por separado, sino que emerge de la interacción de entidades con comportamientos simples [Men08].

Existen arquitecturas multiagentes que han sido creadas para poder resolver pro-

blemas generales como las descritas por [MD99] y [Che03]; sin embargo, también existen propuestas como las de [BR01] y [Ork05] que solventan muy particulares problemáticas como la incorporación de su tecnología en mundos virtuales tridimensionales.

Una problemática de las arquitecturas de software en general, consiste en la pérdida de desempeño al utilizar metodologías que tratan de hacerlas generales y viceversa, perder generalidad al establecer métodos para aplicarse en una situación particular. Esta problemática también se ve reflejada en los *MAS* [WM98], sin embargo, considerar sólo un subconjunto del universo de sistemas, en este caso los mundos virtuales tridimensionales, en el diseño de una arquitectura, permitirá mejorar la funcionalidad que se pierde al incluir sistemas del total del universo existente. Este enfoque permitiría cumplir de forma parcial con la propuesta de [VS95] al estandarizar los mecanismos y protocolos de interacción.

2.1. Bosquejo histórico

Dentro de los primeros esfuerzos por estandarizar los *MAS* podemos encontrar el trabajo de [VS95] perteneciente al *Object Manager Group*. Su propuesta consistía en la elaboración de un modelo de referencia que funcionara como guía para hacer uso de la tecnología de agentes en otros sistemas. En este esquema, existen dos entidades que se relacionan entre si dentro del ambiente: agentes y agencias. Los agentes pueden ser componentes de software o programas y las agencias lugares. Los agentes y las agencias colaboran entre si haciendo uso de patrones y políticas de interacción; la forma de distinguir agentes consiste en sus capacidades, tipo de interacción y movilidad. Por su parte, las agencias permiten la ejecución concurrente de múltiples agentes, movilidad y seguridad.

Otro modelo que se propuso para estandarizar la tecnología para *MAS* fue *Foundation for Intelligent Physical Agents*. Su metodología consistió en crear una serie de especificaciones para poder crear *MAS*. Este enfoque puede considerarse la especificación mínima de un marco de referencia para agentes en un ambiente abierto. El marco de referencia consiste en un modelo del ambiente y una plataforma para los agentes. La plataforma incluye especificaciones de control (*Agent Management Language*) y comunicación (*Agent Communication Language*) para agentes [Men08].

El modelo propuesto en [BJ97] llamado el *Knowledge-able Agent-oriented System* describe las implementaciones de los agentes desde un agente genérico, hasta agentes bien definidos como buscadores y mediadores. Incluye la descripción de la comunicación entre agentes mediante mensajes usando políticas de conversación.

2.2. Arquitecturas Multiagentes

2.2.1. Open Agent Architecture

Una propuesta muy aceptada es la realizada por [MD99]: *Open Agent Architecture (OAA)*. Esta arquitectura esta diseñada principalmente para sistemas abiertos y de colaboración entre agentes. Los agentes colaboran para cumplir con sus objetivos de diseño y en principio pueden estar escritos en cualquier lenguaje de programación como *C/C++, Java o Php*.

El principio de funcionamiento de *OAA* está basado en agentes pertenecientes a una comunidad que trabajan en conjunto para cumplir con metas, por lo que se hace énfasis en los mecanismos de comunicación y recuperación de errores durante su ejecución.

Dentro de *OAA* siempre existe un agente especial llamado *Facilitator* quien es el responsable de los mecanismos de cooperación y coordinación. Es posible que en un sistema existan múltiples *Facilitator*. Cuando un agente es creado en el sistema, debe de ser registrado en un *Facilitator* donde reporta las tareas que es capaz de realizar. De esta manera, cualquier agente puede solicitar algún servicio a la comunidad de agentes existente. Para hacerlo, es necesario establecer comunicación con algún *Facilitator* y mandar su solicitud. El *Facilitator* recibe la petición y determina cuales de los agentes que tiene registrados pueden cumplir con el requerimiento en su totalidad o parcialmente, los agentes elegidos reciben una tarea a realizar y son llamados entonces *Agentes de servicio*.

Los *agentes de servicio* realizan las tareas asignadas y envían sus resultados de regreso al *Facilitator* quien las recolectará y enviará el resultado final al solicitante original. La *Figura 2.1* muestra como el Agente A se comunica con un *Facilitator*, el cual a su vez envía tareas a dos agentes mas (agentes B y C); una vez que los agentes terminan con la asignación regresan los resultados al *Facilitator* y el *Facilitator* envía el resultado al Agente A.

En el caso descrito anteriormente, el Agente A no tiene porque saber cuantos agentes tuvieron que intervenir para cumplir con su requerimiento, estableciendo con esto el concepto de *delegación transparente*. En caso de que exista mas de un *Facilitator*, es posible que cada uno de ellos envíe una posible solución y el agente que solicitó el servicio pueda elegir cual es la mejor de todas.

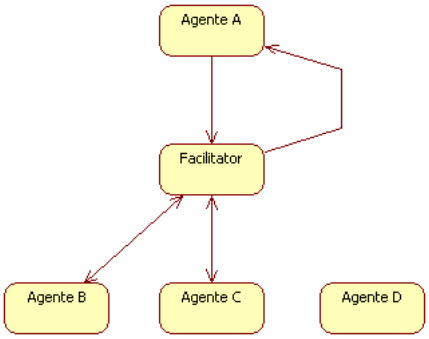


Figura 2.1: Diagrama de interacción entre agentes en OAA

2.2.2. JACK

A diferencia de *OAA*, [Ltd06] *JACK Intelligent Agent architecture* se diseñó para trabajar con sistemas cerrados. Entiéndase por sistema cerrado, al sistema en el cual los componentes han sido creados para trabajar entre sí de forma específica.

JACK fue diseñado por la empresa *Agent Oriented Software* pensando en la necesidad de tener un acercamiento más fuerte con las aplicaciones que usen los beneficios de los *MAS*. Usar *JACK* involucra diseñar agentes para trabajar explícitamente unos con otros. De forma muy similar a *OAA*, cuando un agente necesita de algún servicio, envía una petición a los agentes para conocer su ubicación y el tipo de tareas que son capaces de resolver; bajo este esquema, se considera que todos los agentes están disponibles todo el tiempo.

Un sistema que haga uso de *JACK* permanece inactivo hasta que se proporcionan metas o acciones a realizar. Una vez que una meta o una acción es recibida, se realizará un plan para cumplirla; sin embargo, es posible que debido al estado del ambiente y a las acciones de los agentes en el ambiente, algún plan no pueda ser cumplido en su totalidad y por lo mismo la meta no se alcanzaría. En esta situación, se seguirían probando planes secundarios hasta agostarse las posibilidades y con ello determinar que la meta que se está planteando no es alcanzable. Un plan para *JACK* consiste en realizar acciones, mandar mensajes y generar hechos de tal forma que cumplan con el objetivo que se persigue.

Aun cuando *JACK* está pensado para hacer que agentes colaboren entre sí, no tiene soporte directo para ello. La capacidad de trabajar en equipo, así como el control de fallas, está dada por otro módulo llamado *JACK Teams*. Este módulo le proporciona a *JACK* las

herramientas necesarias para crear un entorno de colaboración agrupando agentes en equipos de trabajo. La *Figura 2.2* muestra como se realiza la interacción entre los agentes usando *JACK Teams* donde, básicamente, los agentes se agrupan en equipos, que a su vez pueden estar formados por más equipos, los cuales interaccionan unos con otros para alcanzar sus objetivos.

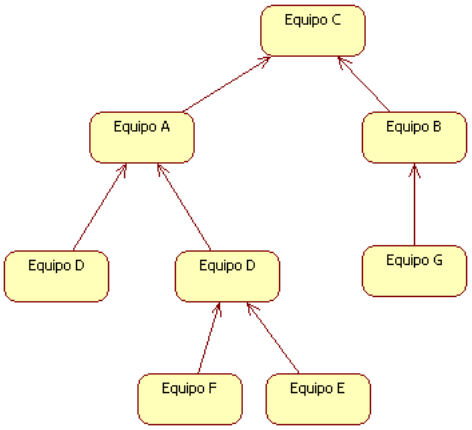


Figura 2.2: Diagrama de interacción entre agentes en JACK

Los agentes dentro de *JACK* son autónomos, proactivos y reactivos; sin embargo, carecen de estructura social y situación dentro del grupo; la incorporación de un agente de *JACK* en un ambiente y la comunicación con otros agentes es quien proporciona capacidades sociales mas no son atributos explícitos.

2.2.3. Creature Smarts: C4

Siguiendo la filosofía de los caracteres sintéticos, *Creature Smarts (C4)* [BR01] propone una arquitectura que busca proporcionar una representación más robusta de las acciones de un agente en un mundo virtual. Los objetivos que busca cubrir *C4* son:

- *Robustez*. Tomar decisiones haciendo uso de conocimiento imperfecto del mundo.
- *Reactividad*. Reaccionar de forma adecuada, a cambios en el ambiente.
- *Adaptabilidad*. La capacidad de aprender por su experiencia en el mundo.

- *Honestidad*. Posee suficiente integridad perceptual para ser sorprendido cuando las cosas no se comportan como han sido planeadas.
- *Expresividad*. Los agentes tienen personalidad, pueden expresar sorpresa y permanecer con la expresión que tenían anteriormente.
- *Sensibilidad*. Mostrar sentido común, no importando su personalidad.
- *Escalabilidad*. Tener la capacidad de incluir múltiples creaturas con características similares.

La arquitectura interna de *C4* se muestra en la *Figura 2.3*. En general, el mundo contiene una lista de creaturas y una lista de objetos; además, se apoya en un *blackboard (pizarrón)* para comunicar los eventos relacionados con el mundo. Respecto a las operaciones mediante red y dibujo, cuenta con mecanismos de sincronización para llevar acabo esto.

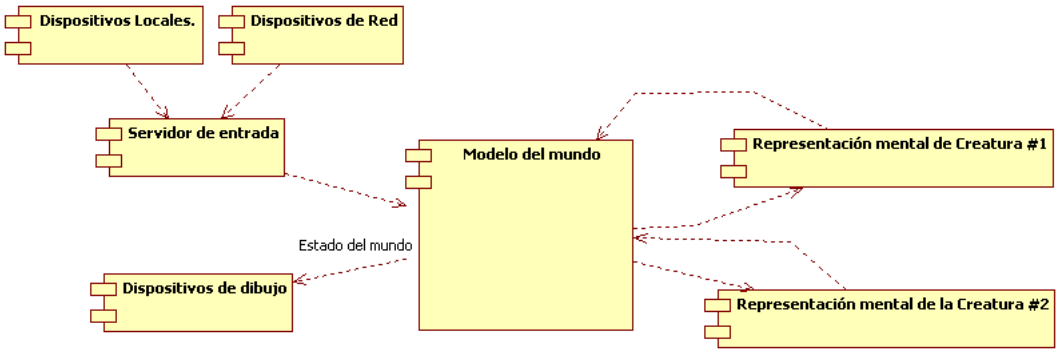


Figura 2.3: Arquitectura interna de *C4*

Los eventos en el mundo son registrados mediante estructuras genéricas llamados *DataRecords*, los cuales pueden representar eventos de tipo acústico, visual, simbólico, etc. Al final de cada ciclo de ejecución del mundo, el *DataRecord* asociado a cada agente es actualizado. La información actualizada es procesada por el mundo en el siguiente ciclo de ejecución.

El cerebro de cada creatura es agrupado en colecciones de sistemas discretos, para comunicarse unos con otros, hacen uso de un *blackboard*, de tal manera que cada parte del cerebro puede leer o escribir en *"slots"* del *blackboard*.

El esquema de *C4* establece que la ejecución de los módulos que forman al cerebro debe de realizarse de forma serial. La forma de clasificar los módulos del cerebro es la siguiente:

- Representación del mundo
- Selección de acciones
- Navegación
- Control de movimientos

Representación del mundo. Es necesario que cada agente cuente con una representación del mundo para así establecer creencias y deseos. A través de un *sistema sensorial* se percibe el estado del mundo, el cual, si se considera como único punto de percepción dentro de cada representación del cerebro y se evita cuestiones como poder ver a través de las paredes, proporciona *honestidad sensorial*. Una vez que obtiene una percepción de un hecho, es necesario almacenarlo en la memoria del agente; el proceso de almacenamiento puede seguir algún esquema de control de transferencia a memoria inmediata, corto plazo o largo plazo.

Selección de acciones. Usando los datos de lectura del *sistema sensorial*, es necesario decidir que acciones deben de realizarse; para ello puede apoyarse de diferentes técnicas de selección. En general, la ejecución de una acción depende de un conjunto de condiciones previas que deben de cumplirse, y su duración puede ser establecida mediante criterios de duración o cumplimiento de alguna condición. Dado el estado del mundo y el estado interno, es posible que una o mas acciones tengan que ejecutarse, algunas de ellas pudiendo ser mutuamente excluyentes. *C4* propone la utilización de dos listas de acciones: en la primera lista se almacenarán las acciones que no son comunes y en la segunda las tareas que se ejecutan con regularidad; primero se ejecutarán todas las acciones de la lista de *no comunes* hasta que ésta se encuentre vacía, para así continuar con la otra.

Navegación. El sistema de navegación extrae información de una acción que va a ser ejecutada para descomponerla, de ser necesario, en la ejecución de acciones de movimientos más simples.

Control de movimientos. Consiste, si la implementación de la arquitectura lo permite, en agregar cambios en las animaciones de los personajes de tal forma que no tengan la

apariencia de ser siempre la misma y con ello lograr tener una mejor apariencia de vida artificial. El tipo de cambios que *C4* sugiere son: gestos simples, exageración de los movimientos, orientación de extremidades.

2.2.4. F.E.A.R. AI

F.E.A.R. (First Encounter Assault Reccon) es un videojuego del tipo FPS *(First Person Shooter)* que fue publicado en 2005, el cual ha sido reconocido como un parte aguas en el diseño de inteligencia artificial para videojuegos [Mil06]. En esta arquitectura [Ork05], un agente está formado por un *blackboard*, memoria de trabajo, un conjunto de subsistemas y un conjunto de sensores muy al estilo de *C4* que fue mencionado en la sección anterior. Los sensores detectan cambios en el mundo y lo registran en memoria. El planeador hace uso de las lecturas de los sensores para decidir que acciones debe de realizar y comunicarle sus decisiones a subsistemas inferiores para su ejecución. Dentro de los subsistemas considerados se encuentran apuntar, navegación, animación y uso de armas.

A diferencia de *C4*, *F.E.A.R. AI engine* utiliza un planeador en tiempo real como mecanismo de toma de decisiones. Cuando se han detectado cambios significativos se realiza una replaneación de acciones, aunque tiene la limitante que sólo puede tener una meta activa a la vez. La comunicación entre el planeador de acciones y los demás subsistemas se realiza mediante un *blackboard*, dicho *blackboard* es accedido con una frecuencia *constante* establecida previamente.

Dentro de las características importantes a considerar en esta arquitectura tenemos:

- Procesamiento distribuido
- Uso de memoria cache
- Conocimiento centralizado
- Recolección de basura
- Planeación *ligera*

Procesamiento distribuido. En el momento de evaluar la posibilidad de ejecutar una acción, es necesario verificar una serie de precondiciones asociadas a dicha acción; sin embargo, la verificación de algunas de las precondiciones puede ser un proceso que exige *muchos* recursos computacionales, como lo son las operaciones de planeación de rutas y verificación

de intersección de rayos. El problema de contar con precondiciones de evaluación *costosas* radica en el tiempo de procesador que es necesario utilizar, debido a que es muy importante mantener el *Framerate* (número de cuadros por segundo) que tiene una aplicación gráfica de alto desempeño como lo es un videojuego, generalmente 30 *Fps (Frames per second)* es un número aceptable, pero los estándares actuales exigen 60 *Fps*. Distribuir el cómputo de dichos cálculos en más de un cuadro de las operaciones de dibujo es una característica de esta arquitectura.

Uso de memoria cache. Las lecturas de los sensores son almacenadas en un espacio de memoria donde son clasificadas según su procedencia y el tipo de información que contienen.

Conocimiento centralizado. Al utilizar un medio de almacenaje centralizado para el conocimiento del agente, se proporciona la oportunidad de organizar la información para su óptima utilización por parte de otros subsistemas.

Recolección de basura. Durante el proceso de generación de conocimiento y percepción del mundo es generada mucha información que necesita ser manipulada por diversos subsistemas; sin embargo, existe la posibilidad que parte de ese conocimiento no sea utilizado. El proceso de recolección de basura elimina el conocimiento inválido del sistema.

Planeación ligera. El proceso de planeación está estructurado de tal forma que se realicen la menor cantidad de cálculos distribuidos en mas de un cuadro. Dentro de las consideraciones que se realizan se encuentra el apoyarse en una tabla *hash* para evaluar sólo las acciones que llevan a cumplir con la meta buscada, evitando con ello considerar el conjunto completo de combinaciones de acciones dentro de la búsqueda. El tipo de representación del estado del mundo es centrada en el agente, es decir, cada agente sólo tiene información de su propio estado ignorando el estado de los demás.

La novedad que presenta esta arquitectura respecto a otras, es el uso de A^* para realizar la planeación de acciones, bajo este esquema, los nodos representan acciones y las aristas representan precondiciones. Se busca el camino que tenga el menor costo para llegar al nodo que representa la meta que se busca alcanzar. Es posible que durante la búsqueda algunos nodos se desactiven debido a que alguna de las precondiciones no ha sido cumplida; sin embargo, se seleccionará la ruta que establezca la menor cantidad de cálculos implicados en la precondición. Incluir pesos en las precondiciones permite establecer acciones preferidas para cumplir con la meta que se busca, lo cual le agrega gran potencial a esta arquitectura.

2.3. Técnica elegida para el presente trabajo

Para la realización de este trabajo, se decidió implementar la técnica de *Jeff Orkin (F.E.A.R. AI)* realizando algunas consideraciones. Según los resultados expuestos en los proyectos donde ha sido utilizado *F.E.A.R. AI*, es una arquitectura robusta para fines de planeación de acciones en un *MAS*. Además, ha probado ser utilizada de forma comercial exitosamente al estar embebida en un ambiente tridimensional que realiza *render* en tiempo real.

Durante el tiempo en que se investigó, se encontraron implementaciones tanto cerradas [Ltd06] como abiertas [MD99] para usar *MAS*; sin embargo, al ser tan generales [Men08] no consideran los problemas más importantes que se presentan al formar parte de un sistema más complicado como un *game engine*:

- Tiempo de procesamiento seriamente limitado
- Planeación *ligera*
- Múltiples caminos en el cumplimiento de metas
- Conocimiento centralizado
- Memoria restringida
- Generación de múltiples comportamientos para igual número de agentes

Tiempo de procesamiento seriamente limitado. El módulo de *AI* se le proporciona, en promedio, un 15 % [Ork05] de tiempo de procesamiento de la aplicación. Es importante considerar dentro de la arquitectura que las operaciones costosas como planeación de rutas y lectura de sensores no pueden disponer en determinado instante de todo el tiempo de procesador necesario para finalizar su cálculo. A diferencia del trabajo de [Ork05] se propone seguir el esquema de *Microthreads* [Sim01].

Planeación ligera. Algunos cálculos como planeación de rutas y análisis de la escena (buscar enemigos, lugares donde cubrirse, etc.), deben de evitarse hasta donde sea posible. Para ello se propone apoyarse en *Microthreads* y además reutilizar parte de los resultados obtenidos por el sistema sensorial.

Múltiples caminos en el cumplimiento de metas. Cuando existe mas de una serie de opciones para cumplir con la meta que se busca, es necesario considerar la que tenga el mejor valor siguiendo alguna heurística. Se proporcionarán dos metodologías diferentes para buscar cumplir con las metas: *clips* y A^*.

Conocimiento centralizado. Cada uno de los agentes sólo tendrá información que es posible adquirir por sí mismo evitando considerar datos pertenecientes a otros agentes. Se incorporará el uso de un dispositivo de radio que permitirá a los agentes enviar y recibir información de un grupo previamente designado.

Memoria restringida. La cantidad de memoria asociada y la duración de su veracidad debe de ser restringida debido a la incorporación de múltiples entidades dentro del sistema. La restricción de memoria se planteará haciendo uso de restricciones simples como límites sensoriales, tiempo máximo de validez de información y sustitución de hechos en memoria compartida.

Generación de múltiples comportamientos para igual número de agentes. Un problema común en la incorporación de alguna tecnología de *AI* en un videojuego, consiste en generar el comportamiento para cientos de agentes ([Ork05], [CE04]). Compartir comportamientos entre agentes es algo común, sin embargo, las acciones que puede realizar cada uno de ellos, las rutas que pueden seguir como rutina diaria, el uso de artículos y las capacidades físicas, los hacen diferentes. Se utilizará *scripting* externo y embebido para la configuración de comportamientos.

Hay diferentes módulos que deben ser construídos para obtener la arquitectura deseada. Haciendo una enumeración, pueden considerarse los siguientes:

1. Sistema gráfico para el mundo virtual
2. Sistema de control de modelos animados 3D
3. Sistema de colisiones
4. Sistema de agentes

Un videojuego provee de todas las etapas comentadas y algunas más, como el sistema de sonido, *GUI*, reglas y efectos especiales, etc., que no se considerarán. Las primeras dos la abarcan parte del sistema gráfico, mientras que la segunda y tercera es parte del sistema lógico.

Capítulo 3

Diseño de la arquitectura

Una vez descrito el estado del arte de las arquitecturas multiagentes consideradas en el capítulo anterior, en el presente se ofrece una descripción del tipo de sistema que se construirá y la arquitectura diseñada para cubrir con las necesidades propuestas.

3.1. Descripción del sistema objetivo

El sistema objetivo consta de un mundo virtual formado por un área acotada por un piso y dividida por múltiples paredes contenidas en un *skybox*[1]. Las paredes definirán los límites del mundo así como también crearán cuartos y regiones de distintos tamaños.

El mundo virtual contendrá múltiples agentes, representados mediante *avatares*[2], organizados en equipos para cumplir con objetivos comunes. El objetivo que se plantea

[1] Volumen, generalmente una caja, de dimensiones superiores a los objetos de una escena cuyas normales se encuentran orientadas al centro de la escena. Se utiliza en ambientes tridimensionales para incrementar el nivel de inmersión de una escena al mostrar un horizonte bien definido.

[2] Un avatar es la representación (visual, auditiva, táctil, etc.) de una entidad virtual.

cubrir mediante la colaboración de los agentes es el combate entre distintos equipos, para ello se apoyarán de por lo menos dos tipos diferentes de armas. Cada agente deberá tener su propio sistema sensorial para poder percibir el mundo.

Otros objetos que también deberán haber en el mundo virtual, son aquellos que les permitirán a los agentes recuperar la salud perdida durante el combate, recuperar municiones y cajas para poder ser utilizadas para cubrirse de ataques enemigos.

Para cubrir con las necesidades descritas anteriormente, la arquitectura se dividirá en dos secciones, la primera corresponde al apartado gráfico y la segunda al apartado del *MAS*.

3.2. Sistema gráfico

El apartado gráfico tiene una gran importancia porque éste limita los alcances y el tiempo máximo de procesamiento que el *MAS* tendrá; por lo que es importante contar con una buena organización del mismo. El apartado gráfico se dividirá en los siguientes módulos:

- Organización de la escena
- Carga de modelos
- Control de cámara

3.2.1. Organización de la escena

Organizar una escena mediante una estructura arbórea o *scene graph*[3] permite realizar operaciones computacionalmente costosas, como prueba de intersección y control de visibilidad de objetos, de forma eficiente y sistematizada [HD94].

El *scene graph* asociado a este trabajo se encuentra ilustrado en la siguiente figura (*Figura 3.1*).

Con el fin de cumplir con los objetivos de la sección 3.1, en el *scene graph* del sistema se considera una entidad, a la que se le llamará *nivel*, que contendrá los elementos necesarios para definir el área donde los *avatar* interactuarán: piso, paredes, objetos para restaurar salud, municiones, etc. Sólo es posible tener un *nivel* por escena.

[3] Un *scene graph* es una técnica para organizar los objetos gráficos y lógicos de una escena para hacer un manejo más eficiente de los mismos.

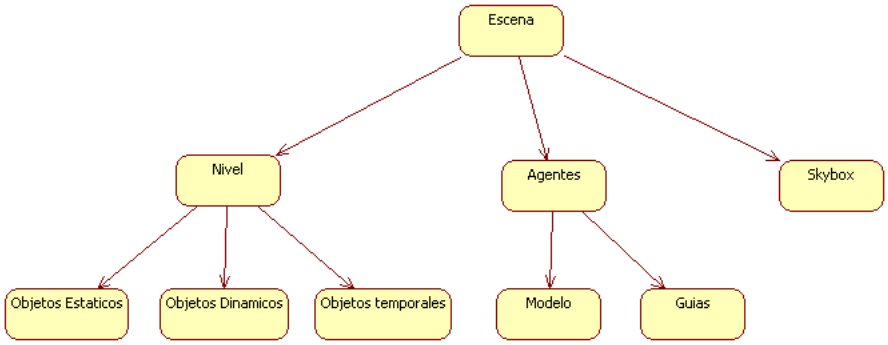


Figura 3.1: Estructura del *scene graph*

Si bien es necesario usar un *scene graph* para la sección de visualización, también es importante organizar la *información lógica* de la escena. La *información lógica* a la que se hace referencia consiste en *metadata*[4] asociada a modelos tridimensionales que les dota de propiedades especiales; la siguiente lista menciona ejemplos de *metadata* asociada a modelos:

- Control de visibilidad
- Factor de penetrabilidad
- Frecuencia de refresco
- Modificación de propiedades como salud, municiones, incrementar rango visión o memoria
- Velocidad de movimiento
- Volúmenes de colisión

De hecho, el control de la *información lógica* de un ambiente puede llegar a ser más complicado que el diseño propio del ambiente [Joh01]. Para organizar mejor la información de un *nivel*, se propone la división en tres categorías: *objetos estáticos, dinámicos y temporales*. Cada categoría en si misma es un contenedor que sigue sus propias reglas para operación.

[4]Información adicional a la información [Wik08]

Objetos estáticos. Corresponde a los objetos que son inamovibles en el ambiente; por ejemplo: paredes, pisos, techos, fuentes, estatuas, etc. Durante el *ciclo de actualización*[5] no modifican sus propiedades y sólo se consideran para operaciones de *culling*[6].

Objetos dinámicos. En este conjunto se incluyen los objetos que cambian su posición en el ambiente y siempre se encuentran en el sistema; como por ejemplo, los volúmenes de colisión asociados a los *avatar* de los agentes, así como también los objetos que modifican las propiedades de salud y el número de municiones. Durante el *ciclo de actualización*, se actualizan sus propiedades como posición y visibilidad; además de ser considerados para operaciones de *culling* en el sistema.

Objetos temporales. Los objetos de este conjunto son removidos, con el tiempo, del ciclo de simulación del sistema y no siempre tienen una representación gráfica asociada. Durante el *ciclo de actualización*, se actualizan sus propiedades como posición, tamaño, valor[7].

Una vez que los objetos son categorizados, puede realizarse optimizaciones para cada caso particular.

Optimizaciones

Optimización para objetos estáticos. Estos objetos regularmente son asociados para realizar cálculos de intersección de volúmenes en la sección física de la arquitectura. Una técnica muy difundida para cumplir con esta meta es el uso de *octrees* [WA92]. Los *octrees* permiten disminuir el numero de cálculos al biparticionar el espacio donde se encuentran los objetos de una escena gráfica, respecto a los ejes *x*,*y*,*z* y después organizando la información en árboles donde cada nodo tiene potencialmente 8 nodos hijo. La *Figura 3.2* muestra como se organiza un *octree* y la *Figura 3.3* un ejemplo de división para un *nivel* de este trabajo.

Como puede observarse en la *Figura 3.3*, al aplicar *octrees* en un *nivel* sólo basta con tres niveles para poder disminuir la cantidad de cálculos significativamente. El nivel 3 del *octree* sólo contiene 10 geometrías, lo cual disminuye la cantidad de cálculos si se compara con las 141 que consta el *nivel* del ejemplo.

Optimización para objetos temporales. Al igual que con los objetos estáticos, se utiliza una organización *octree* para estos objetos. Sólo que ahora el árbol generado se replantea

[5] El ciclo de actualización corresponde al intervalo, dentro del ciclo de vida de una aplicación gráfica, donde los objetos actualizan sus propiedades antes de dibujar el siguiente cuadro de animación.

[6] Culling. Procedimiento de eliminación de caras, superficies y objetos ocultos [WA92]

[7] Entendiendo como valor algún parámetro asociado que es modificado con el tiempo como la intensidad de un sonido, la velocidad de un objeto, etc.

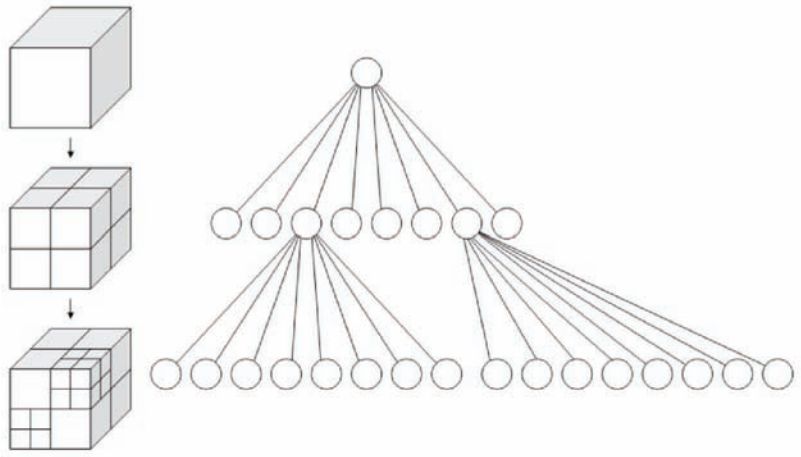


Figura 3.2: Organización de una escena tridimensional con octrees

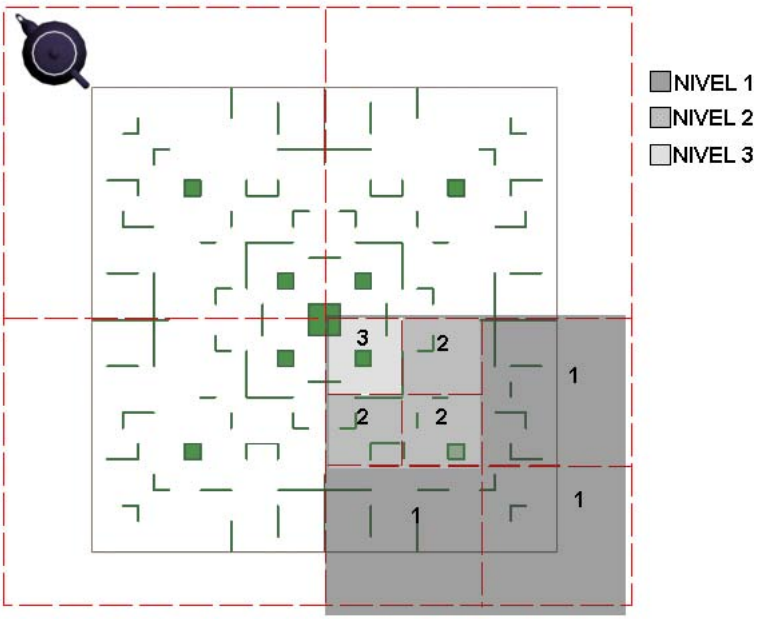


Figura 3.3: Organización de una *nivel* con octrees

durante el *ciclo de actualización* debido a que los objetos potencialmente cambian de posición. Además de considerar la organización dinámica del objeto dentro del *octree*, se generan distintos contenedores que clasifican los objetos en subconjuntos y con ellos se reduce el

número de búsquedas.

3.2.2. Carga de modelos

Modelos estáticos

Los modelos que se necesitan visualizar en el mundo virtual consideran paredes, piso, cajas, objetos pequeños como maletines médicos, municiones, etc. Cada archivo no sólo contiene información de los vértices y caras que forman a la geometría, sino también materiales y texturas asociadas. Existen numerosos formatos de archivo que cubren con los requisitos; sin embargo, es necesario considerar dos requisitos adicionales:

- Compatibilidad con *DirectX 9.0*
- Documentación suficiente para análisis y preprocesamiento de geometrías

Estas dos restricciones permitirán crear un *nivel* en un modelador profesional y posteriormente analizarlo mediante un programa de computadora para generar *metadata* asociada por geometría (volúmenes de colisión) y por *nivel* (malla de navegación del lugar) .

Se consideraron los archivos en formato *3Ds, Obj y X*, pero el elegido fueron el formato *X*[8] debido a que cubre con ambas características adicionales mencionadas.

Modelos animados

Como se mencionó en la sección 3.1, considerar *avatars* en un mundo virtual implica contar con modelos tridimensionales que contengan múltiples animaciones para representar acciones. Las acciones que se buscan corresponden a combate entre agentes, por lo que es necesario que cuenten con las siguientes animaciones:

- Caminar
- Correr
- Brincar
- Agacharse

[8] Formato nativo de DirectX

- Atacar

Al buscar soporte de archivos animados por *Internet*, se encontraron suficientes librerías que permiten cargar animaciones para personajes diversos tanto comerciales como de software libre: [FX08], [Cal08], [Gra08]; sin embargo, ninguno de los formatos soportados por las librerías contiene personajes animados con las características que se buscan. Se podrían instalar *plug-ins* en modeladores profesionales como *3D Studio Max*, *Maya*, *Lightwave*, *Blender* y con ello modelar los personajes necesarios, lo cual llevaría mucho tiempo. Se optó por buscar algún formato 3D comercial que tuviera soporte de múltiples animaciones y que incluyera todas las mencionadas con anterioridad. Los formatos *UTX*[9], *Md2*[10] *y Md3*[11] cumplen con la lista de animaciones buscada. El formato elegido es el *Md3* debido a que cuenta con la mejor documentación en línea, además de tener animación de piernas y torso independiente. El cuadro 3.1 muestra las animaciones contenidas en un archivo *Md3*.

Animación	**Piernas**	**Torso**	**Animación**	**Piernas**	**Torso**
Gesto de victoria		•	Vuelta	•	
Disparar		•	Brincar	•	
Torso agachado		•	Caer	•	
Torso de pie		•	Brincar hacia atrás	•	
Torso recto		•	De pie inactivo	•	
Nadar	•		Agachado inactivo	•	
Caminar agachado	•		Morir #1	•	•
Caminar	•		Morir #2	•	•
Correr	•		Morir #3	•	•
Caminar de espaldas	•				

Cuadro 3.1: Animaciones contenidas en un archivo *Md3*

3.2.3. Control de cámara

Es necesario contar con un modelo de cámara virtual que permita visualizar el mundo desde cualquier posición. Debido a la naturaleza de los movimientos de los agentes, debe de poderse realizar una libre navegación por el mundo. Para ello, se optó por hacer uso de *hardware* adicional; un *joystick* compatible con *DirectX* modelo *Xbox 360*.

[9] UTX. Formato de archivos soportado por *Unreal Engine (Epic Games)*
[10] Md2. Formato de archivos soportado por *Id Tech 2 (Id Software)*
[11] Md2. Formato de archivos soportado por *Id Tech 3 (Id Software)*

El *joystick* del *Xbox 360* cubre muy bien las necesidades de navegación debido a su diseño, que desde un principio, está dirigido al control de objetos en espacios tridimensionales; además, cuenta con múltiples botones que pueden ser útiles para ser asociados a diferentes funciones en el mundo virtual. La *Figura 3.4*[12] muestra la distribución de botones y *joysticks* de un control de *Xbox 360*.

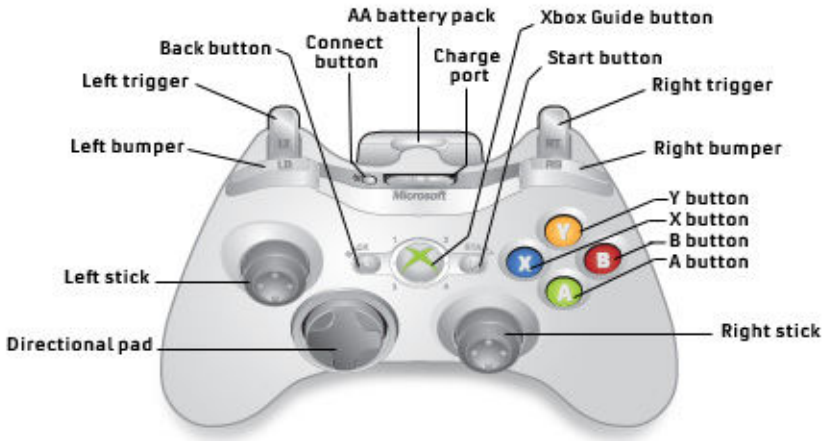


Figura 3.4: Control para *Xbox 360*

Una vez elegido el dispositivo de control, se seleccionará el modelo de cámara a implementar. Para fines de este trabajo, no se considerará realizar navegación automática[13] por la escena, ni la realización de cámaras del tipo *follower*[14], por lo que el modelo de cámara de [Dan01] *(FPS*[15]*)* será más que suficiente.

3.3. Sistema multiagentes

Una vez establecidas las restricciones propias del sistema gráfico, es posible proponer un modelo de agente que cubra las necesidades propuestas. Para ello se propone la arquitectura de agente de la *Figura 3.5*. En general, la arquitectura considera que el agente cuenta con un *sistema sensorial*, un conjunto de efectores, un mapa básico del mundo, un conjunto de procedimientos para resolver problemas, un módulo de comunicación para hablar

[12] Imagen obtenida en [Cor08]
[13] Cámara controlada mediante *scripting* previamente determinado
[14] Cámara seguidora de objetos
[15] Juego de disparos en primera persona, *First Person Shooter*

con otros agentes, memoria, *razonamiento* y control de su estado interno. A continuación se describirán los módulos que forman a la arquitectura.

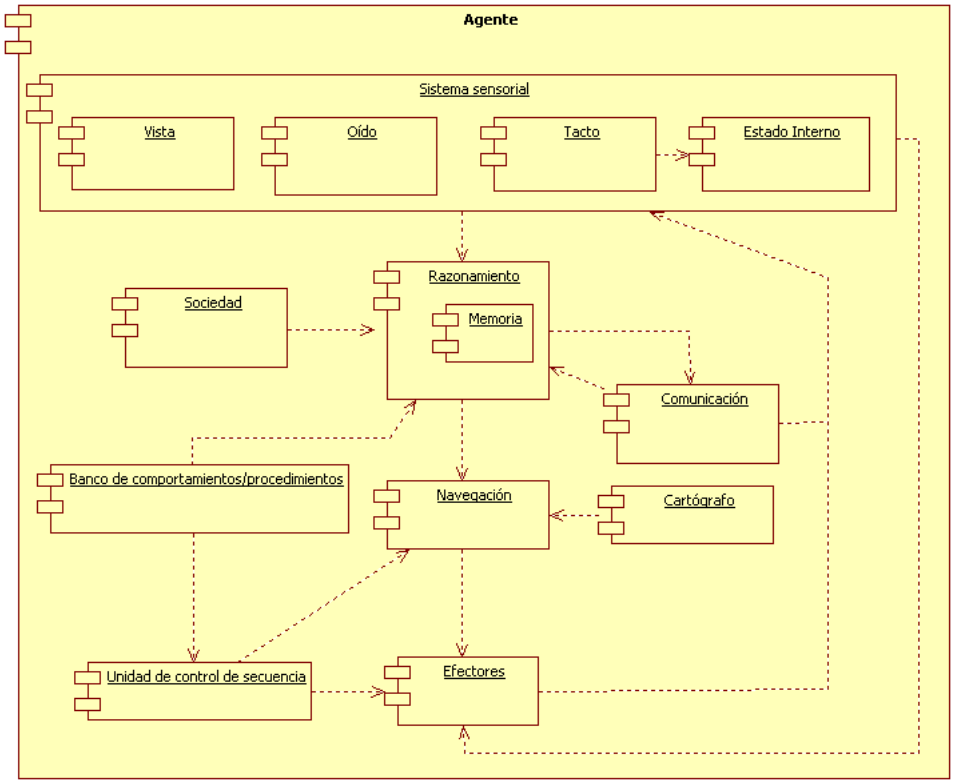


Figura 3.5: Arquitectura interna de un agente

3.3.1. Sistema sensorial

El *sistema sensorial* es de suma importancia debido a que mediante éste se percibe el estado interno y el mundo [Sve95]. En el sistema que se plantea, se consideran entidades virtuales, por lo que es posible crear sentidos para los agentes que les permitan obtener información del mundo de manera privilegiada y diferente; es decir, crear sentidos adicionales a los sentidos humanos, como alguno que permita saber cuando alguien está en problemas. No obstante, crear sentidos diferentes a los humanos puede crear una falta ilusión de inteligencia [Ork05].

Los humanos contamos con 5 sentidos: vista, oído, gusto, olfato y tacto, de los cuales, no todos son de interés en este trabajo. El sentido del gusto implicaría que los agentes pudieran comer y probar objetos en la escena, y el sentido del olfato implicaría que los objetos contaran con una propiedad adicional que es el aroma, el cual puede ser percibido a distancia previamente especificada. Por lo anterior, el gusto y el olfato no fueron considerados dentro de este trabajo.

Percepción externa

Los sentidos de percepción externa que se considerarán en este trabajo son:

Vista. Los agentes necesitan percibir los objetos mediante observación, de tal forma que sea posible distinguir otros agentes, objetos en el *nivel* como municiones, lugares estratégicos donde protegerse de ataques y paredes cercanas para evitar colisionar con ellas. Los objetos con representación gráfica visible se considerarán para este sentido.

Oído. La percepción de *"sonidos"* permitirá al agente saber, sin necesidad de establecer contacto visual, que existen *perturbaciones*[16] cerca de donde se encuentra. Hacer uso de este sentido implica que algunos objetos en el mundo virtual producirán *sonidos virtuales*. Las fuentes auditivas que se considerarán son agentes desplazándose y armas de alcance lejano siendo utilizadas.

Tacto. Mediante este sentido, el agente tendrá una respuesta natural ante el contacto con otros cuerpos rígidos. Evitará traspasar objetos como paredes, cajas y otros agentes. Y de la misma manera, podrá tomar objetos en el mundo y ser dañado por ataques de otros agentes. En este caso, se considerarán representaciones gráficas visibles (paredes, cajas,etc.) e invisibles (sonidos, disparos, etc.).

Percepción interna

Básicamente, la percepción interna que se busca tener, consiste en acceder a las siguientes propiedades:

Salud. Cada agente debe de ser capaz de saber la cantidad de salud que tiene actualmente.

Municiones. Conocimiento de la cantidad de municiones restantes en el arma de lejano alcance.

[16]Se considerará perturbación a la existencia de sonidos producidos por alguna entidad virtual

Orientación. Orientación actual del agente en el mundo virtual.

Posición. Posición actual del agente en el mundo virtual.

En general, las lecturas obtenidas por el *sistema sensorial* establecen para el agente el estado actual del mundo. Haciendo uso de sus valores y con las metas propias del agente, debe de ser posible decidir que acción realizar ante cualquier situación.

3.3.2. Sociedad

Este módulo le permite al agente saber cual es su posición dentro de la sociedad de agentes que forman al mundo virtual. Considerar un sistema en el cual coexistan múltiples agentes que colaboran en equipo implica una estructura de equipos simples; sin embargo, se optó por una organización similar a la de [Ltd06]. La *Figura 3.6* muestra un ejemplo de la organización adoptada.

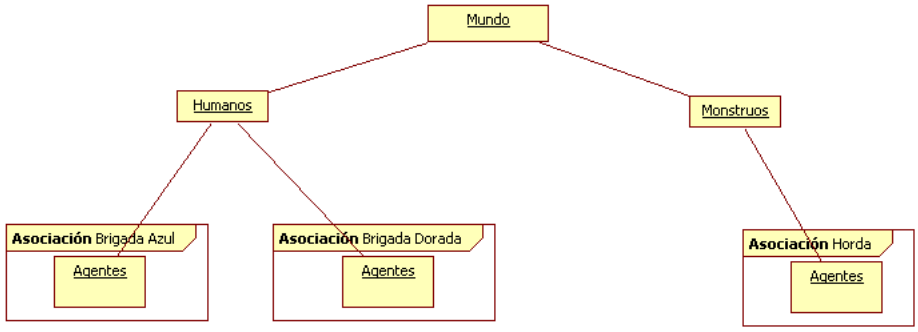


Figura 3.6: Organización social del sistema

Al igual que en [Ltd06], se cuenta con una estructura que agrupa agentes, a la cual se le denominó *asociación*. Cada *asociación* recibe un nombre que la distingue de otras y puede formar parte, a su vez, de otra *asociación*.

Si bien tanto en [Ltd06] como en [MD99] la organización de los agentes busca poder establecer grupos de ayuda donde cualquier agente puede solicitar servicios; en este caso, las *asociaciones* buscan establecer un *orden* para poder determinar quien es *amigo*, quien *enemigo* y de quien se deben de aceptar órdenes.

Del ejemplo de la *Figura 3.6* se observan cinco asociaciones que pertenecen al mundo virtual: *brigada azul, brigada dorada, horda, humanos y monstruos*. La *asociación* de

humanos está formada por dos *asociaciones*: *brigada azul y brigada dorada*; mientras que las demás *asociaciones* siguen una organización directa más simple.

Dentro de cada *asociación* existen miembros que tienen jerarquías internas determinadas por un valor numérico; entre más grande es el número, mayor es su jerarquía. La *Figura 3.7* muestra un ejemplo de la organización interna de una *asociación*.

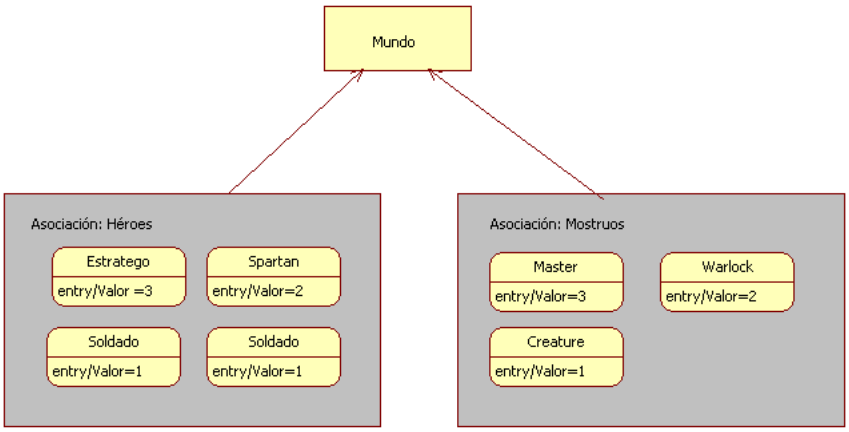


Figura 3.7: Organización interna de una *asociación*

Como puede observarse, la *asociación* de *héroes* está formada por cuatro miembros que, en orden de jerarquía, son: *estratego (valor 3), spartan (valor 2) y soldado (valor 1)*. Si un *estratego* da órdenes a un *spartan* o a un *soldado*, estos deben de obedecerle, caso contrario, si un *soldado* diera órdenes a otro *soldado* o a un *spartan* dicha orden sería ignorada.

3.3.3. Razonamiento

Se considerará al módulo de razonamiento como el componente que realiza la planeación de acciones, que el agente llevará acabo para cumplir con los objetivos que persigue. De la *descripción del sistema objetivo*, se sabe que habrá múltiples agentes en el mundo virtual que estarán organizados en equipos y su meta es que un equipo elimine a otro. Por lo que es necesario establecer objetivos para cumplir con la meta propuesta, el siguiente cuadro *(Cuadro 3.2)* muestra los objetivos generales de un agente.

Dada la lista de objetivos que se han planteado, es necesario que cada agente sea capaz de realizar diversas acciones que al combinarse entre sí, le lleve a cumplir con los

Objetivo
Apoyar amigo
Encontrar enemigo
Matar enemigo
Sobrevivir

Cuadro 3.2: Cuadro de objetivos generales de un agente

objetivos. El cuadro 3.3 muestra una lista de acciones que cada agente puede realizar.

Acción	Acción
Atacar lejano	Atacar cercano
Atacar desde una cubierta	Buscar objetos para recuperar salud
Buscar una cubierta	Buscar municiones
Buscar un enemigo	Dirigirse a una posición
Reportar un enemigo a la vista	Solicitar ayuda
Huir	Atender llamado de auxilio

Cuadro 3.3: Acciones que puede realizar un agente

Una vez que se cuenta con los objetivos y con las acciones disponibles, es posible establecer una dependencia entre éstos. La manera de representar la dependencia entre objetivos y acciones que se utilizará es similar a la de *STRIPS*[17], donde se tienen precondiciones y postcondiciones asociadas a una acción. El cuadro 3.5 muestra las acciones que llevan a cumplir las metas propuestas.

Objetivo	**Acciones**
Apoyar amigo	Atender llamado de auxilio
Encontrar enemigo	Buscar enemigo
Matar enemigo	Ataque cercano
	Ataque encubierto
	Ataque lejano
Sobrevivir	Huir
	Buscar amigo

Cuadro 3.4: Cuadro para el objetivo *Sobrevivir*

[17] *STRIPS (Stanford Research Institute Problem Solver)* [Nil98]

Al igual que en [Ork05], es posible que exista más de una forma de cumplir con un objetivo; como puede observase en el cuadro 3.4 donde existen tres acciones que pueden cumplir la meta *matar enemigo*. En determinado momento, debe ser posible decidir entre varias opciones de acciones, ya que en este caso, sólo una acción puede ser llevada acabo a la vez. Se plantea un sistema de jerarquías para marcar el orden de *preferencia* entre un grupo de acciones.

De la misma forma en que se tiene un sistema jerárquico para decidir entre distintas acciones, es necesario establecer un sistema de jerarquías para definir, en un momento dado, cual objetivo tiene más importancia. Un sistema de capas y subcapas cumple muy bien con la estructura jerárquica buscada para objetivos y acciones, en donde las capas superiores tienen mayor importancia que las inferiores. El *cuadro 3.5* muestra la estructura de capas propuesta.

<table>
<tr><td colspan="2">Capa 4
Sobrevivir</td></tr>
<tr><td rowspan="3">Capa 3
Atacar enemigo</td><td>Subcapa 3
Atacar desde cubierta</td></tr>
<tr><td>Subcapa 2
Atacar lejano</td></tr>
<tr><td>Subcapa 1
Atacar cercano</td></tr>
<tr><td colspan="2">Capa 2
Apoyar amigo</td></tr>
<tr><td colspan="2">Capa 1
Encontrar enemigo</td></tr>
</table>

Cuadro 3.5: Cuadro de objetivos generales de un agente

La estructura mostrada en el *cuadro 3.5* indica que la meta más importante a cumplir es *sobrevivir*. Esto quiere decir que se prefiere buscar la supervivencia sobre *atacar a un enemigo*. En una situación de combate, si las condiciones no favorecen la supervivencia del agente, éste buscará la forma de sobrevivir. De forma muy similar, si el agente no está atacando a un enemigo ni necesita sobrevivir a un ataque, puede buscar *apoyar a un*

amigo en combate o *buscar un enemigo*. Ahora bien si se encuentra en condición de combatir, puede realizarlo mediante tres acciones diferentes: *atacar lejano, cercano o encubierto*. El agente preferirá atacar de cerca a un enemigo, después desde una posición donde no sea dañado fácilmente (encubierto) y finalmente un ataque lejano.

Para poder crear la estructura de capas descrita, es necesario contar con un árbitro el cual decidirá cual de los comportamientos que se activan debe de ejecutarse. La *Figura 3.8* muestra como el árbitro toma los comportamientos activados y elige uno de ellos.

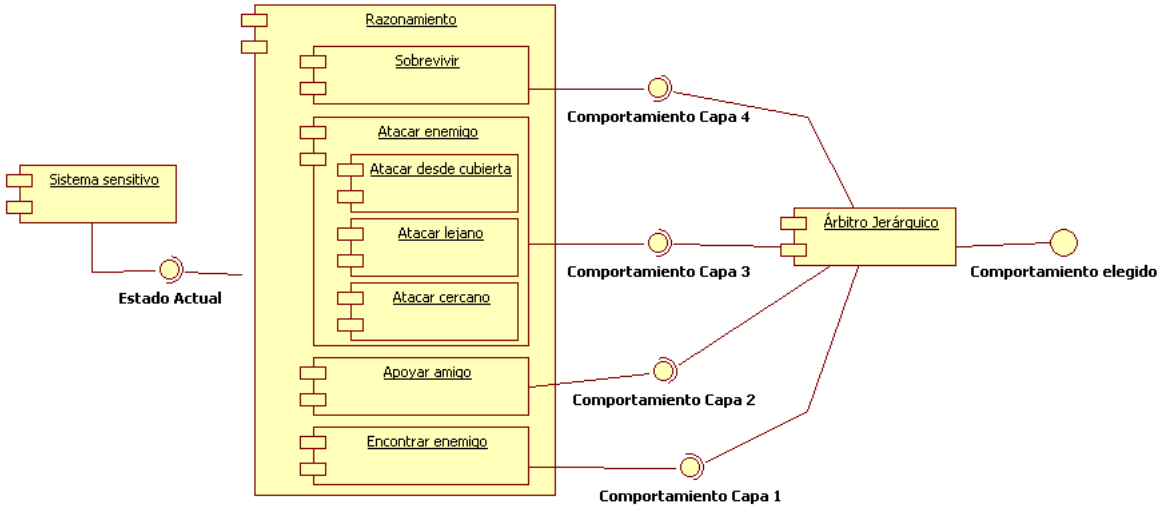


Figura 3.8: Objetivos generales

Las acciones, al seguir un esquema similar a *STRIPS*, necesitan cubrir una serie de precondiciones para poder ser activadas. Como se mencionó en la sección 3.3.1, las lecturas obtenidas por el *sistema sensorial* establecen el estado actual del mundo para un agente; subconjuntos de los hechos obtenidos del estado del mundo serán las precondiciones para cada acción. El *cuadro 3.6* muestra la lista de acciones disponibles junto con sus precondiciones y postcondiciones asociadas.

Acción	Precondiciones	Postcondiciones
Atacar lejano	Enemigo a la vista Municiones suficientes	
Atacar cercano	Enemigo a la vista Enemigo cerca	
Atacar desde cubierto	Enemigo a la vista Municiones suficientes Cubierta cercana	
Buscar cubierta	Enemigo a la vista	
Buscar enemigo	NO enemigo a la vista	
Reportar enemigo a la vista	Enemigo a la vista	
Huir	Peligro NO municiones suficientes NO *asociación* cercana	Solicitar ayuda
	Peligro NO salud suficiente NO *asociación* cercana	Solicitar ayuda
Atender llamado de auxilio	Seguro Municiones suficientes Solicitud de ayuda	
Buscar amigo	Seguro NO municiones suficientes	Solicitar posición
	Seguro NO salud suficiente	Solicitar posición

Cuadro 3.6: Cuadro Acción-Precondición

Para hacer mas claro la dependencia entre acciones se pueden realizar diagramas de dependencia de hechos y acciones para cada objetivo. Las *Figuras 3.9, 3.10, 3.13* y *3.12* muestran los diagramas de dependencias para cada uno de los objetivos planteados.

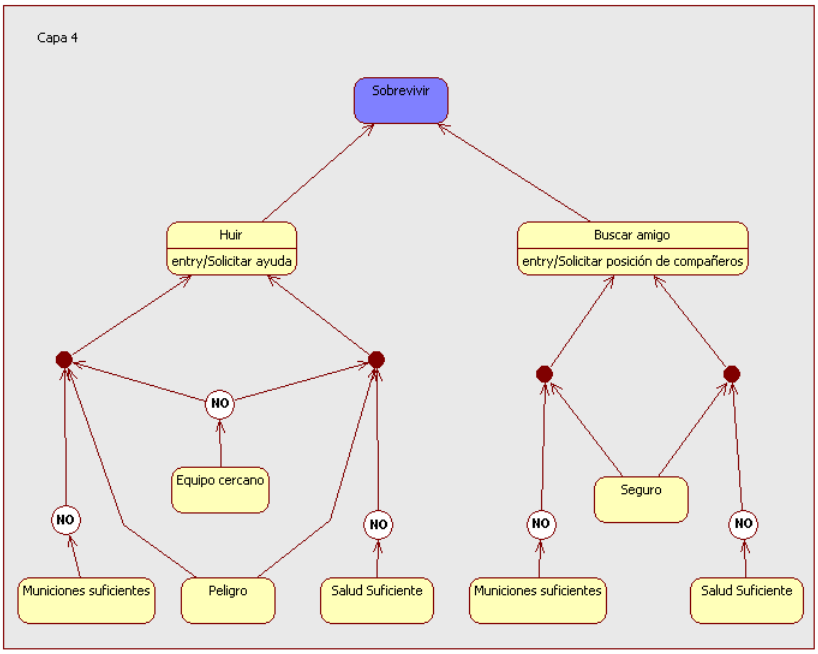


Figura 3.9: Dependencia de acciones y hechos para el objetivo *Sobrevivir*

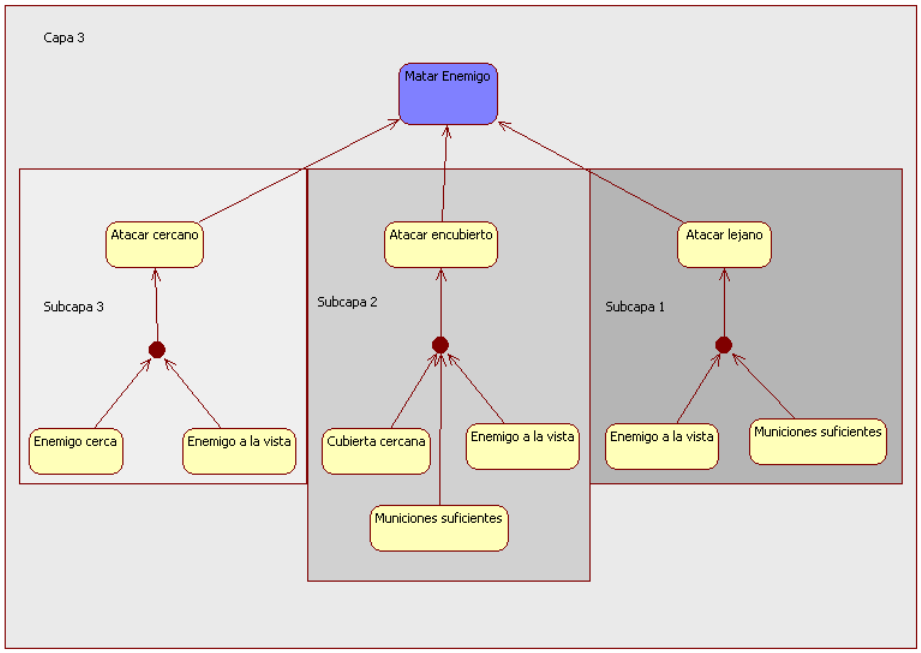


Figura 3.10: Dependencia de acciones y hechos para el objetivo *Matar enemigo*

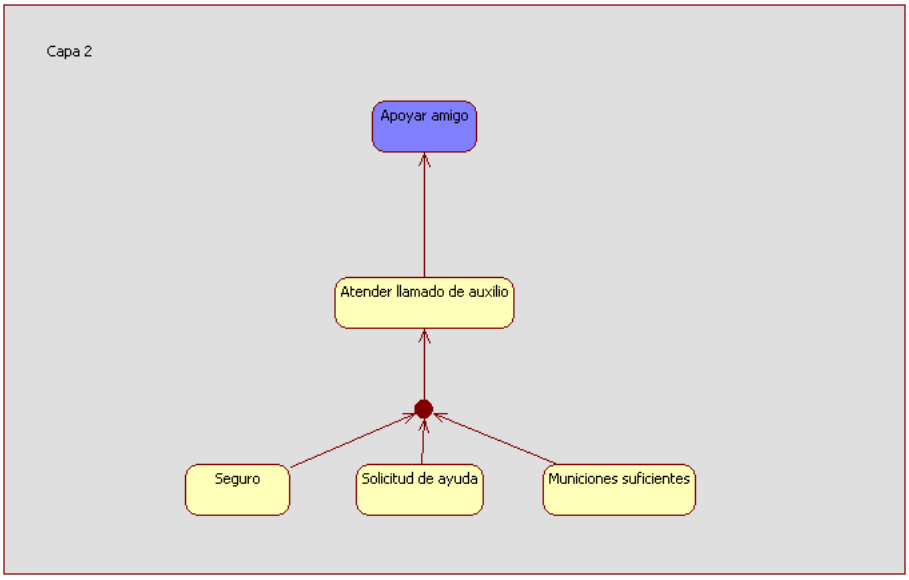


Figura 3.11: Dependencia de acciones y hechos para el objetivo *Apoyar amigo*

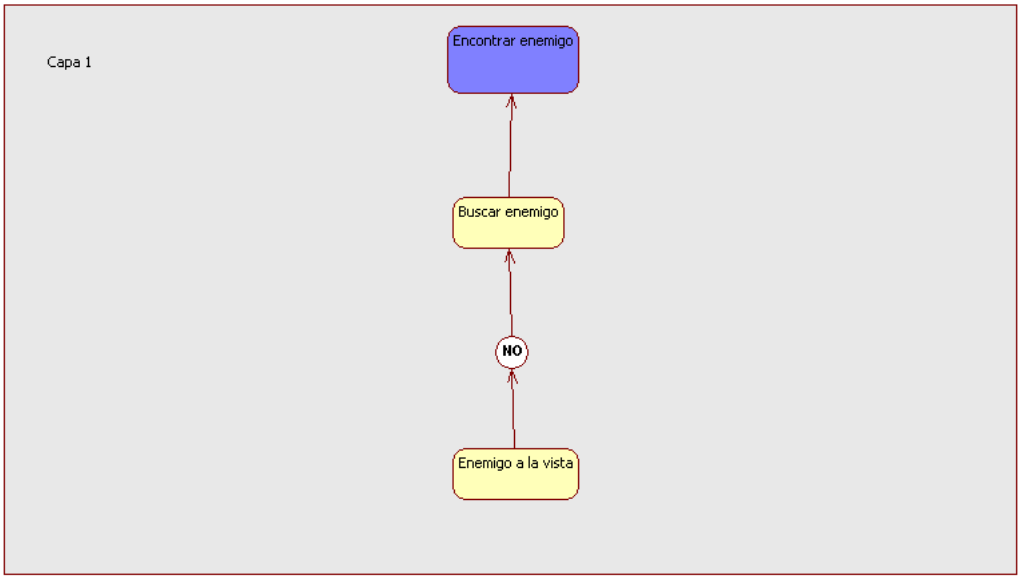


Figura 3.12: Dependencia de acciones y hechos para el objetivo *Buscar enemigo*

Haciendo uso del estado del mundo y con los objetivos propios del agente, debe de ser posible decidir la serie de acciones que deben realizarse para cumplir con los objetivos ante cualquier situación.

3.3.4. Memoria

La información obtenida por el *sistema sensorial* es procesada y almacenada en la *memoria* para que después sea utilizada por el módulo de *razonamiento*.

Cada elemento almacenado en memoria necesita ser clasificado, por lo que a cada objeto se le anteponen las siguientes propiedades adicionales:

- *Tipo*. Identificador de la memoria
- *Valor*. De ser necesario, almacena cantidades.
- *Momento de realización*. Instante en el cual fue generado el elemento
- *Tiempo de validez*. Duración máxima de la validez del elemento.

Con estas propiedades, es posible acceder rápidamente a los elementos, así como también descartar la información que haya superado la duración máxima en el sistema o actualizar elementos con información reciente.

3.3.5. Comunicación

Este modulo permite establecer contacto entre los diferentes agentes que pertenecen a una *asociación*. Está compuesto por dos canales de comunicación: transmisión y recepción de mensajes. El canal de recepción es común para los integrantes de la brigada, mientras que cada agente cuenta con su canal privado de transmisión de información.

Para establecer comunicación entre los agentes no basta con la descripción de canales de transmisión, también juega un papel muy importante el lenguaje a utilizar para transmitir la información. Existen muchas opciones para establecer comunicación entre agentes [Sin97] [Wei99], sin embargo se optó por utilizar *KQLM*[18] como el formato para empaquetar la información por su simplicidad, además de cubrir con los alcances de este trabajo [MJ95]. Para el contenido de los mensajes se seleccionó *XML* por las mismas razones.

[18]KQML. *Knowledge Query and Manipulation Language*[Fin92]

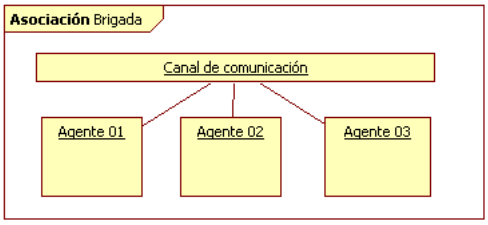


Figura 3.13: Estructura del módulo de comunicación

3.3.6. Banco de comportamientos/procedimientos

Para los propósitos de este trabajo, se le llamará *comportamiento* o *procedimiento* a la ejecución secuencial de acciones *simples* que trabajan entre sí para alcanzar un fin común. Se le considera una acción *simple* a cualquier acción que no puede ser descompuesta en elementos más sencillos. Ejemplo de acciones simples son: avanzar, retroceder, vuelta a la derecha, vuelta a la izquierda, disparar, etc.

Dentro del conjunto de acciones posibles que ejecuta un agente, es posible que existan algunas acciones *"complicadas"* que se construyan al llamar secuencialmente a acciones *simples*; como por ejemplo: evadir un obstáculo desconocido, buscar un objeto en un área, etc.

El banco de *comportamientos* le proporciona al agente un conjunto de solucionadores de problemas que son considerados por el *módulo de razonamiento* para alcanzar los objetivos del agente. Dentro de las acciones mencionadas en la sección 3.3.3 se identifica por lo menos una acción que puede ser considerada *comportamiento*: *atacar aun enemigo*. *Atacar a un enemigo* puede ser implementada de forma tan simple como apuntar y disparar o puede ser toda una secuencia de movimientos para esquivar ataques, apuntar y disparar cuando sea pertinente. De la misma manera, buscar puede ser una secuencia de movimientos que le permitan a los agentes escudriñar un área de forma lineal o seguir un patrón de conducta más complicado.

3.3.7. Unidad de control de secuencia

Este módulo es utilizado para llevar el control de la ejecución de un *comportamiento* previamente elegido por el *módulo de razonamiento*.

Las partes que constituyen a la *unidad de control de secuencia (UCS)* son:

- Cargador
- Registros dinámicos de almacenamiento
- Memoria de instrucciones
- Pila de control
- *ALU*[19]
- Módulo de enlace con memoria de razonamiento
- Módulo de enlace con acciones del agente
- Apuntador de programa

El *cargador* tiene como función inicializar la *UCS* y cargar un comportamiento. Los *registros dinámicos de almacenamiento* permiten almacenar los resultados de operaciones parciales redundantes obtenidos por el *ALU*. La memoria de instrucciones contiene la secuencia de acciones y lectura de sensores que el *comportamiento* tiene definido. La *pila de control* permite realizar saltos condicionales y a subrutina dentro de la secuencia del *comportamiento*. La *ALU* realiza operaciones aritméticas especificadas en el cuerpo del *comportamiento*. El *módulo de enlace con memoria de razonamiento* permite a la *UCS* acceder a las lecturas realizadas por el *sistema sensorial*. El *módulo de enlace con acciones del agente* invoca a ejecutar acciones simples. Y finalmente, el *apuntador de programa* lleva registro de la siguiente acción o lectura de sensores a realizar.

Como puede observarse, las partes que constituyen la *UCS* son muy similares a las correspondientes a un *CPU* de una computadora; sin embargo tiene dos grandes diferencias: los módulos de enlace con memoria de razonamiento y acciones del agente; y que desde su diseño se consideró el uso de *microthreads*[Sim01] para su funcionamiento. Por lo tanto, puede cambiar de *comportamientos* rápidamente, trabajar en múltiples llamados de *ciclo de actualización* e interrumpir su ejecución en cualquier momento.

3.3.8. Navegación

El propósito de este módulo es dotarle al agente de los algoritmos necesarios para poder calcular la rutas, realizar movimientos *absolutos* o *relativos* tanto para traslación como para rotación, así como también llevar el control de posición y orientación en el mundo virtual.

[19] Unidad Aritmética Lógica

Cálculo de rutas

Con apoyo de la información proporcionada por el *cartógrafo*, se calcula la ruta que un agente deberá tomar para llegar de un lugar a otro.

Tradicionalmente, el cálculo de rutas es una de las operaciones computacionalmente más costosas, por lo que se hará uso de *microthreads* en su funcionamiento.

Movimientos

Absolutos. Corresponde a los movimientos que hacen uso del origen absoluto de referencias. Ejemplo de movimientos de este tipo son: dirigirse al punto *(4.0,5.0)* u orientarse 30° respecto el eje *X*.

Relativos. Corresponde a los movimientos que hacen uso de un origen relativo de referencias. Generalmente el origen relativo de referencias es la posición y orientación actual del agente. Los movimientos que los agentes pueden realizar de este tipo son: avanzar, retroceder, dar vuelta a la izquierda, dar vuelta a la derecha, etc.

Control de posición y orientación

Esta sección del módulo lleva un registro de la posición y orientación actual del agente.

3.3.9. Cartógrafo

El cartógrafo contiene un mapa que puede ser tanto *topológico* como *lógico-topológico* del *nivel* donde se encuentre el agente.

Mapa topológico

Contiene una descripción de las regiones a las cuales puede llegar un agente dentro del mundo virtual. Su representación generalmente es una una malla con nodos conectados entre sí. La *Figura 3.14* muestra un ejemplo de mapa topológico.

En este trabajo, los nodos dentro de la malla corresponden a posiciones específicas del mundo virtual y la conectividad representa la posibilidad de llegar de un nodo a otro sin encontrar obstáculos conocidos, como por ejemplo, las paredes.

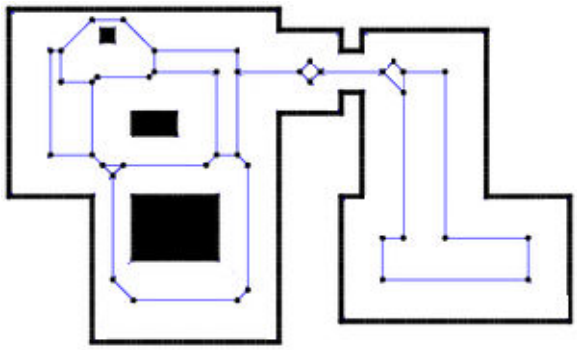


Figura 3.14: Ejemplo de mapa topológico

Si bien el cartógrafo no considera obstáculos dinámicos, es el punto de partida para la elaboración de cálculo de una ruta.

Capítulo 4

Descripción del sistema desarrollado

Habiendo planteado la arquitectura deseada, en este capítulo se abordará el asunto específico de su implementación en computadora, los módulos desarrollados y las aplicaciones auxiliares empleadas para alcanzar el objetivo.

En cuanto al trabajo realizado, se trata de un conjunto de bibliotecas de programación que permiten incorporar múltiples agentes para colaborar entre sí en un mundo virtual tridimensional. En específico, se han diseñado considerando emplearse en futuras aplicaciones de computo gráfico *(CG)* o de realidad virtual *(RV)*. Debido a la pretensión de utilidad en otras aplicaciones de *CG* o *RV* , la construcción ha sido realizada desde la funciones mas básicas, tratando de hacer el menor uso posible de código ajeno para proveer una fácil y amplia integración con otros sistemas en desarrollo.

De la misma forma en la que fue dividida la arquitectura en el capítulo 3, este capítulo está estructurado en dos módulos:

- *El sistema gráfico*

- *El sistema multiagentes*

Cada módulo tiene sus propias consideraciones de desarrollo, pero en general, al unirse ambos tienen la estructura mostrada por el diagrama de bloques de la *Figura 4.1*. Si bien no todos los partes del sistema están mostradas en la figura, si se ejemplifica de forma simple su forma de interactuar entre sí. El nombre de la aplicación construida es *Multiple Soldier System (MUSS)*.

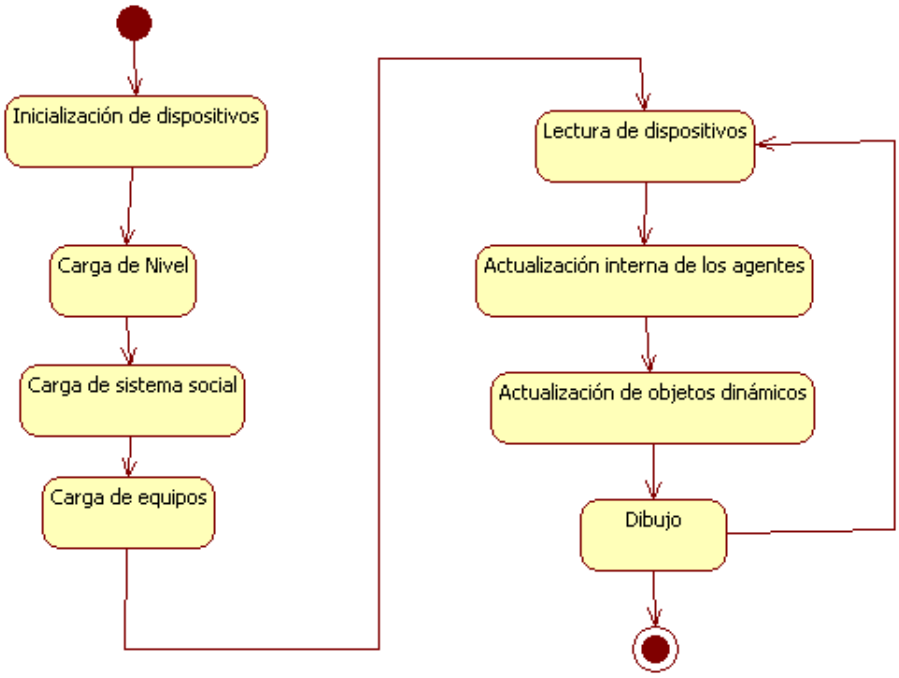


Figura 4.1: Diagrama de bloques de uso de la arquitectura

Como se puede observar, ambos módulos son utilizados en la forma tradicional en la que un *sistema gráfico interactivo (SGI)* opera. Es decir, está considerado un módulo de configuración y carga, un módulo de adquisición de información, un módulo de actualización y finalmente, el módulo de dibujo. Usar esta estructura permite su incorporación en otros *SGI*.

El lenguaje de programación utilizado durante todo el desarrollo del sistema es C/C++ debido a su buen desempeño, sin embargo, es posible hacer implementaciones futu-

ras en otros lenguajes como *C#* o *Java*.

Para llevar un mejor control de la configuración de los módulos de *software*, se decidió hacer uso de un lenguaje de tipo *script*. Existen muchas opciones disponibles para incorporar *scripting* en aplicaciones como *perl*,*python*, *ruby* e incluso *java*; pero se decidió utilizar *lua* [Lua08] debido a los pocos recursos que necesita para funcionar, su simplicidad y facilidad para incorporarse a aplicaciones escritas en lenguaje *C*.

Respecto al *API* de dibujo que se utilizará, se seleccionó *DirectX 9.0* debido a que presenta mejor desempeño que *OpenGL* en computadoras sin tarjetas de video con aceleración.

4.1. Sistema Gráfico

El sistema gráfico se dividirá en los siguientes módulos:

- *Render*[1]
- Modelos y organización de la escena
- Control de movimientos

4.1.1. *Render*

El módulo de *render* se planteó mediante un esquema de clases que heredan de una clase abstracta a la que se llamó *Dx9_3DObject* que tiene el método virtual *render* (dibujar). Esta consideración permite a diversas clases definir de forma independiente su forma de dibujarse y además permite organizar los objetos en una forma simple mediante contenedores. Se crearon diversas clases para poder definir los objetos que heredarán de la clase virtual *Dx9_3DObject*. La *Figura 4.2* muestra la descripción de las mismas.

De la *Figura 4.2*, podemos ver que existen diferentes objetos gráficos como esferas, cilindros, prismas, conos y modelos; cada uno de ellos cuenta con su propia forma de configurarse, actualizarse y dibujarse. Es importante mencionar que hace falta una clase más en el diagrama de la figura, la clase que hace falta es *Muss_Bot* la cual se analizará en la sección correspondiente a la implementación del agente.

[1] En el área de *CG*, *render* está asociado a las operaciones de dibujo sobre un dispositivo

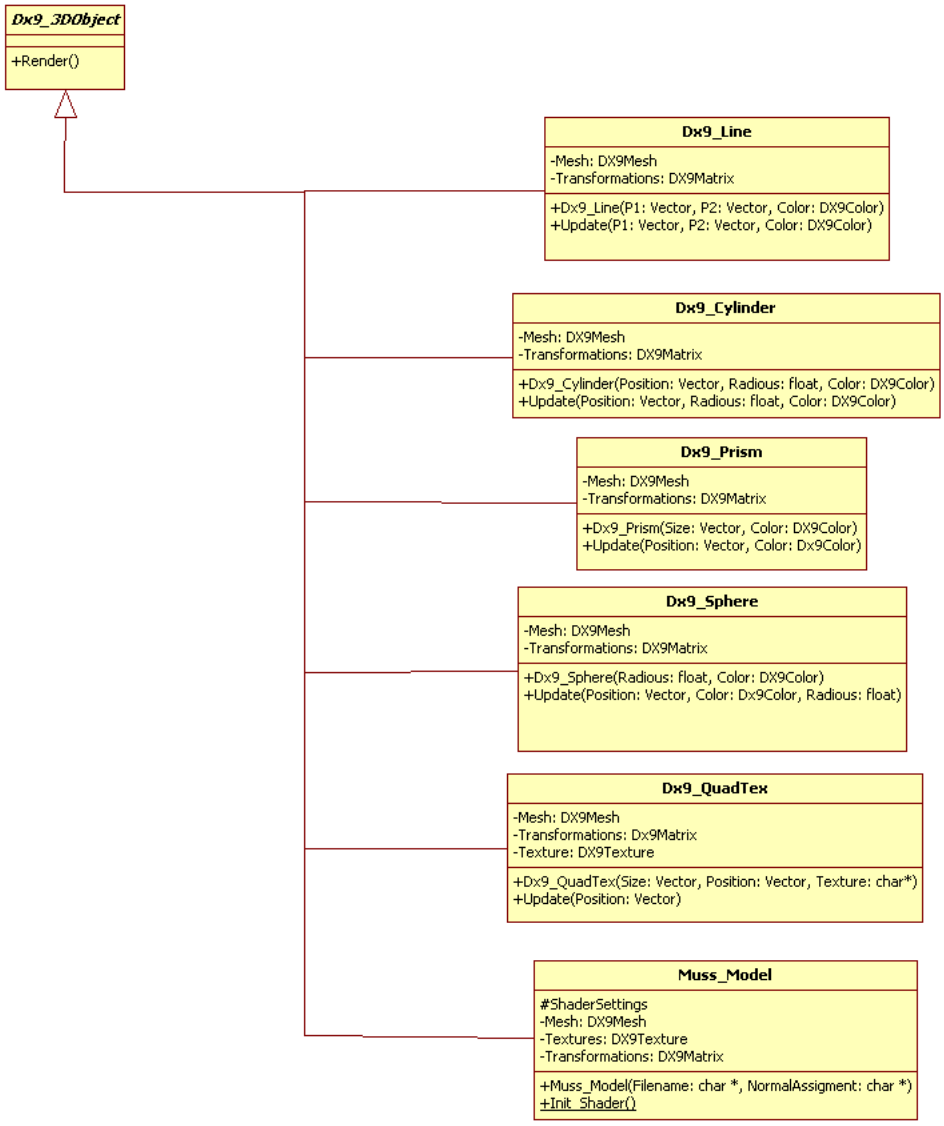


Figura 4.2: Organización de las clases dibujables en el sistema

Shaders

Con la finalidad de mejorar tanto el desempeño durante el *render* así como la apariencia de las gráficas, se incorporó un *shader*[2] *programable* dentro de los métodos asociados

[2] *Shader*. También conocido como *sombreador*, es un procedimiento mediante el cual son manipulados los vértices y los *pixels* de un modelo dentro del pipeline de render en un sistema gráfico

a dibujar. Los *shaders* que se incorporaron son:

- *Phong*[3]
- *Bump mapping*[4]
- *Skybox*

Debido a que se decidió utilizar *DirectX 9.0* como *API* de render, el lenguaje de programación de *shaders* que se utilizó fue *HLSL*[5].

4.1.2. Modelos y organización de la escena

Como se mencionó en el *capítulo 3*, es necesario considerar dos tipos de modelos para utilizarse en el sistema: modelos estáticos en formato *X* y modelos animados en formato *md3*. Los modelos estáticos consisten en paredes, piso, techo, fuentes, estatuas, etc.; los modelos animados pueden ser los objetos que se encuentren en el ambiente virtual que permiten recuperar energía y municiones, así como también los propios *avatar* de los agentes. La combinación de todos estos modelos en posiciones distintas forma lo que se le denominó un *nivel*.

Entonces, para poder definir un nivel es necesario establecer los modelos y posiciones de todos los objetos del mundo virtual. Para lograr que la definición de los objetos sea simple, se adoptó el uso de *lua* para generar la configuración. El *script* de *lua* a su vez tendrá llamadas a mas *scripts* para permitir un intercambio más sencillo de configuraciones. El *código 4.1* muestra un ejemplo del uso de *lua* para crear la configuración de un nivel.

Código 4.1: *Script* de configuración

```
1 --++++++++++++++++++++++++++++++++++++
2 --+Archivo de configuración de Nivel01+
3 --++++++++++++++++++++++++++++++++++++
4 Levels={"Level01"}
5 Size=# Levels;
6 -- Archivo de configuración de modelos estáticos
7 Level01={Name="Level01", File="Scripts\\Level01.lua"}
8 -- Archivo de configuración de los agentes
9 BotsTeamFile = "Scripts\\Bots01.lua"
```

[3]Modelo de sombreado que permite interpolar normales dentro de un polígono en una escena gráfica

[4]Bump mapping. Técnica dentro del modelo de sombreado de *phong* que permite sustituir normales durante el proceso de rasterización

[5]High Level Shading Language

```
10  -- Archivo de configuración de las tareas de los agentes
11  TaskSettingsFilename ="Scripts\\TaskSettings.lua"
12  -- Archivo de configuración del sistema social del sistema
13  SocietyFilename = "Scripts\\Society.lua"
14  -- Archivo de configuración de los objetos del mundo virtual
15  ItemsFilename    = "Scripts\\Items.lua"
16  -- Archivo de configuración de las cubiertas del mundo virtual
17  CoverFilename    = "Scripts\\Cover.lua"
```

Del código mostrado, es posible identificar los múltiples archivos de configuración que forman la configuración del *nivel*: *modelos estáticos*, *agentes*, *tareas de agentes*, *sistema social*, *objetos* y *cubiertas*.

Al existir múltiples archivos de configuración deben de existir también múltiples cargadores para poder interpretar la información de cada archivo. Aun cuando existen diferentes configuraciones, todos los modelos, a excepción de los agentes, pueden ser representados dentro del sistema por un objeto del tipo *Muss_Model* (*Figura 4.2*). Su diferencia será en el manejo interno y que se agruparán en tres diferentes contenedores de modelos: *estáticos*, *dinámicos* y *temporales* (las diferencias entre cada tipo de objeto está especificada en la sección 3.2.1 del capítulo 3). La *Figura 4.3* muestra el proceso de carga y manejo interno de los modelos en el sistema.

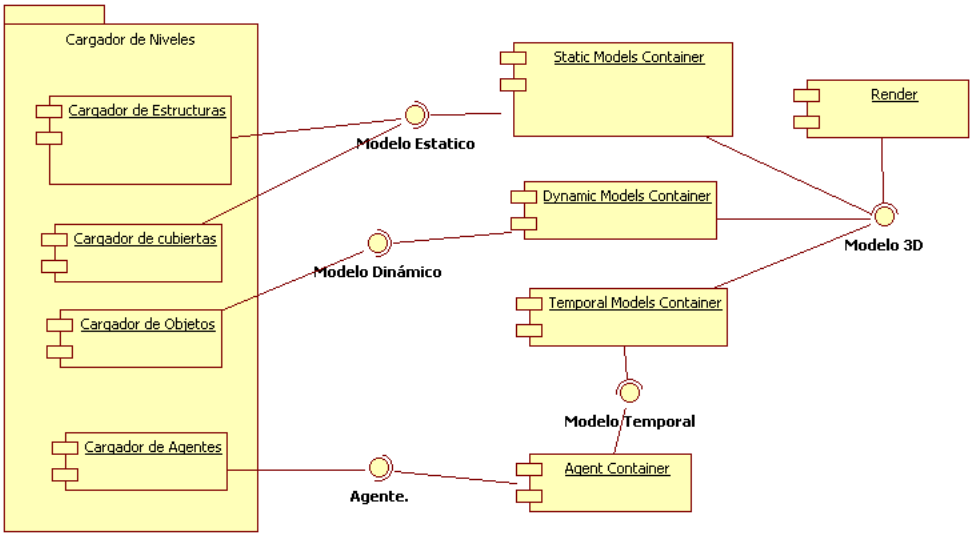


Figura 4.3: Proceso de carga y manejo de los modelos en el sistema

Como puede observarse, los modelos son organizados por contenedor según su categoría. Los modelos temporales son generados por los agentes durante su ejecución en el sistema, por lo que no son cargados durante el proceso de inicialización. El proceso de *render* extrae los modelos de cada contenedor para dibujarlos según su propia definición.

Modelos en formato *md3*

Como se indicó en la sección 3.2.2 del capítulo 3, el formato elegido es *md3*; sin embargo, los modelos *md3* se encuentran contenidos dentro de un archivo *pk3*[6]. El archivo *pk3* es un archivo contenedor, es decir, internamente agrupa diferentes archivos. Las partes que contiene un archivo *pk3* se ilustran en la *Figura 4.4*.

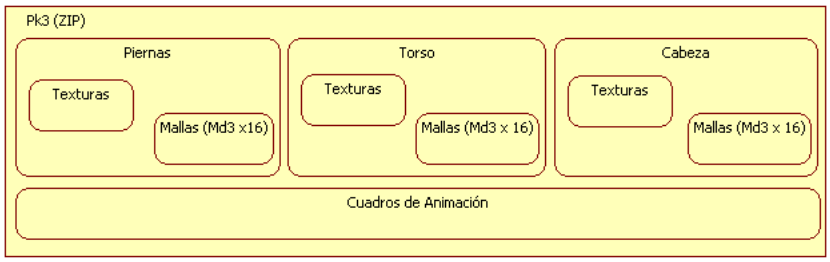


Figura 4.4: Estructura interna de un archivo en formato *Pk3*

Cada archivo *pk3* está asociado al modelo tridimensional de un humanoide y contiene, por lo menos, tres archivos *md3* (piernas, torso y cabeza), las texturas correspondientes a cada modelo e información cuadro a cuadro de las animaciones. El listado completo de las animaciones está mostrado en el cuadro 3.1 del capítulo 3. La *Figura 4.5* muestra un ejemplo del tipo de modelos contenidos en un archivo *pk3*.

Dentro del sistema, los archivos pk3 son cargados siguiendo la estructura de la clase *Pk3_Model* (*Figura 4.6*).

Si bien la clase *Pk3_Model* no hereda de *DX9_3DObject* para incluirse dentro del proceso de dibujo lo hace mediante un contenedor. Éste forma parte de la implementación del agente como se verá en las siguientes secciones.

[6]Un archivo *Pk3* consiste en un archivo en formato *zip* con la extensión renombrada

Figura 4.5: Ejemplo de modelos en formato *Pk3*

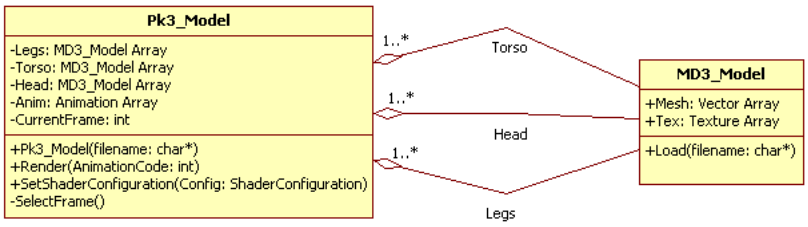
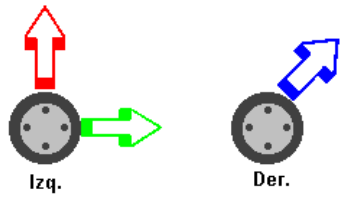


Figura 4.6: Clase *Pk3_Model*

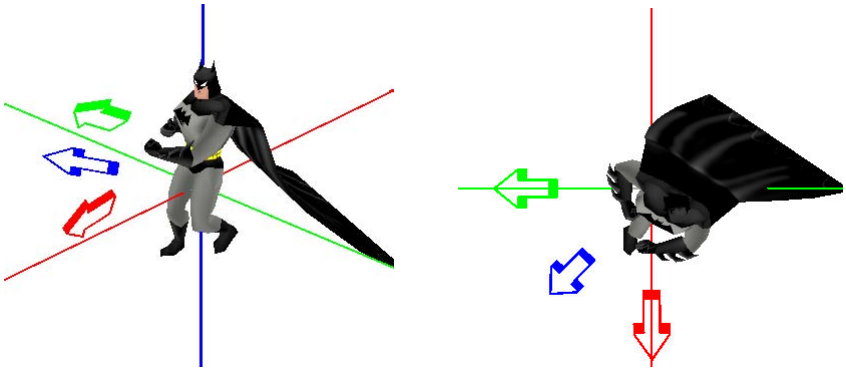
4.1.3. Control de movimientos

El control de movimientos de los agentes se diseñó para simular que el control proviene del uso de dos *joysticks*, a los que se les hará referencia como *joystick izquierdo (JI)* y *joystick derecho (JD)* (*Figura 4.7(a)*). Cada *joystick* tiene asociado un movimiento diferente del agente, *JI* controla los movimientos para avanzar, retroceder, desplazamiento lateral a la izquierda y desplazamiento lateral a la derecha; *JD* tiene asociado cambiar la orientación del agente. Las *Figuras 4.7(b)* y *4.7(c)* muestran el uso del control de movimientos.

Como se puede observar de las figuras anteriores, los movimientos de los *agentes* en el mundo virtual están asociados respecto a su propio eje de referencias. Cuando *JI* es presionado hacia arriba o abajo indica un movimiento de avanzar o retroceder respecto a la orientación actual del agente; de forma similar *JI* también indica un movimiento de



(a) *Joystick izquierdo* controla movimientos, *joystick derecho* orientaciones



(b) Asociación de movimientos en 3D (c) Asociación de movimientos en 2D

Figura 4.7: Asignación de movimientos a *joysticks izquierdo (JI)* y *joystick derecho (JD)*

desplazarse a la derecha o izquierda al presionarse en dichas direcciones. El cambio de orientación del agente al presionar *JD* está ilustrado en la *Figura 4.8*.

Dividir el control de movimientos de los agentes en dos secciones: movimiento y orientación, permite controlar más eficientemente su navegación en el mundo virtual como [Her00] lo describe.

Los cálculos correspondientes al movimiento descrito con anterioridad son:

$$\overline{V_n}(x, y, z) = (\overline{JD}.x, \overline{JD}.y, 0)$$

$$\overline{B_n} = \overline{V_n} \times \overline{U}$$

$$\overline{P_n} = \overline{P_a} + \overline{V_n} \cdot \overline{JI}.y + \overline{B_n} \cdot \overline{JI}.x$$

Donde:

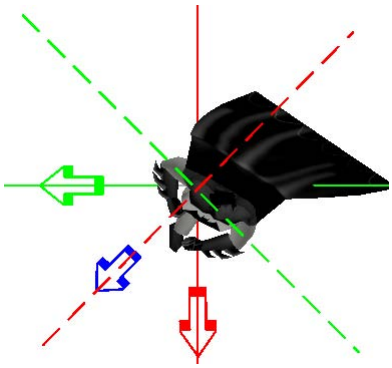


Figura 4.8: Cambio de orientación asociado a *JD*

$\overline{JI}$ vector unitario asociado al *joystick izquierdo*, el cual tiene componentes *(x,y)*
$\overline{JD}$ vector unitario asociado al *joystick derecho*, el cual tiene componentes *(x,y)*
$\overline{U}$ vector unitario que indica hacia donde se encuentra *arriba* en la orientación actual del agente
$\overline{B_n}$ vector unitario *binormal*[7] a la dirección de observación y a la dirección hacia arriba
$\overline{V_n}$ vector que describe la orientación nueva del agente
$\overline{P_a}$ vector que contiene la posición actual del agente
$\overline{P_n}$ vector que contiene la posición nueva del agente

4.2. Sistema Multiagentes

Haciendo alusión al modelo descrito en la sección 3.3 del capítulo 3, se describirá la implementación de cada uno de los módulos que forman a la arquitectura interna del agente. (*Figura 3.5*).

4.2.1. Sistema sensorial

Dentro de la implementación del sistema sensorial se considerará la percepción *externa* e *interna* por separado. Cada sentido necesitará acceder a los objetos que se encuentren en la escena: estáticos, dinámicos y temporales. La implementación particular de cada sentido determinará cuáles y cómo deberán ser utilizados. Las *figuras 4.9* y *4.10* muestran la implementación del sistema sensorial.

El sistema sensorial sigue una estructura de trabajo que permite interrumpir en

[7] Un vector binormal es un vector ortogonal a dos vectores

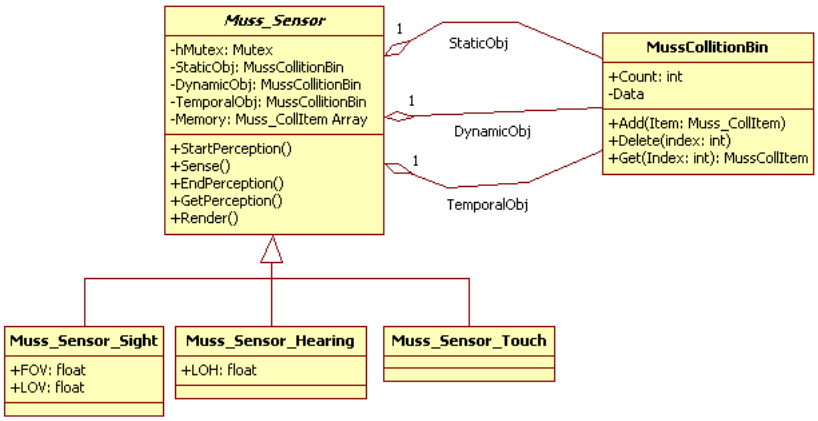


Figura 4.9: Estructura sensitiva de un agente

cualquier momento su ejecución *(microthreads)* [Sim01], eso permite realizar las operaciones de sensado limitando el uso del procesador. La *figura 4.11* muestra los pasos necesarios para llevarlo acabo.

Cada uno de los parámetros de configuración del aparato sensorial es configurable por cada agente de forma independiente usando *lua*.

La percepción externa en este trabajo incluye los sentidos de la *vista*, el *oído* y el *tacto*. Dentro de su implementación es necesario utilizar los modelos tridimensionales que aparecen en el mundo virtual. A cada modelo se le asociará una o mas geometrías que lo envolverán. Las geometrías a utilizar son: *axis aligned bounding box*[8] *(AABB)*, *axis oriented bounding box*[9] (AOBB), *bounding sphere*[10] *(BS)* y *cápsula*[11]. El cuadro 4.1 muestra la asociación de geometrías utilizadas por el sistema sensorial. La *figura 4.12* muestra la clase *Muss_CollItem* para almacenar los datos asociados a las geometrías de intersección.

[8] *axis aligned bounding box*,caja envolvente alineada con los ejes
[9] axis oriented bounding box, caja envolvente orientada con los ejes
[10] *bounding sphere*, esfera envolvente
[11] cilindro de radio *R* con semiesferas de radio *R* en los extremos

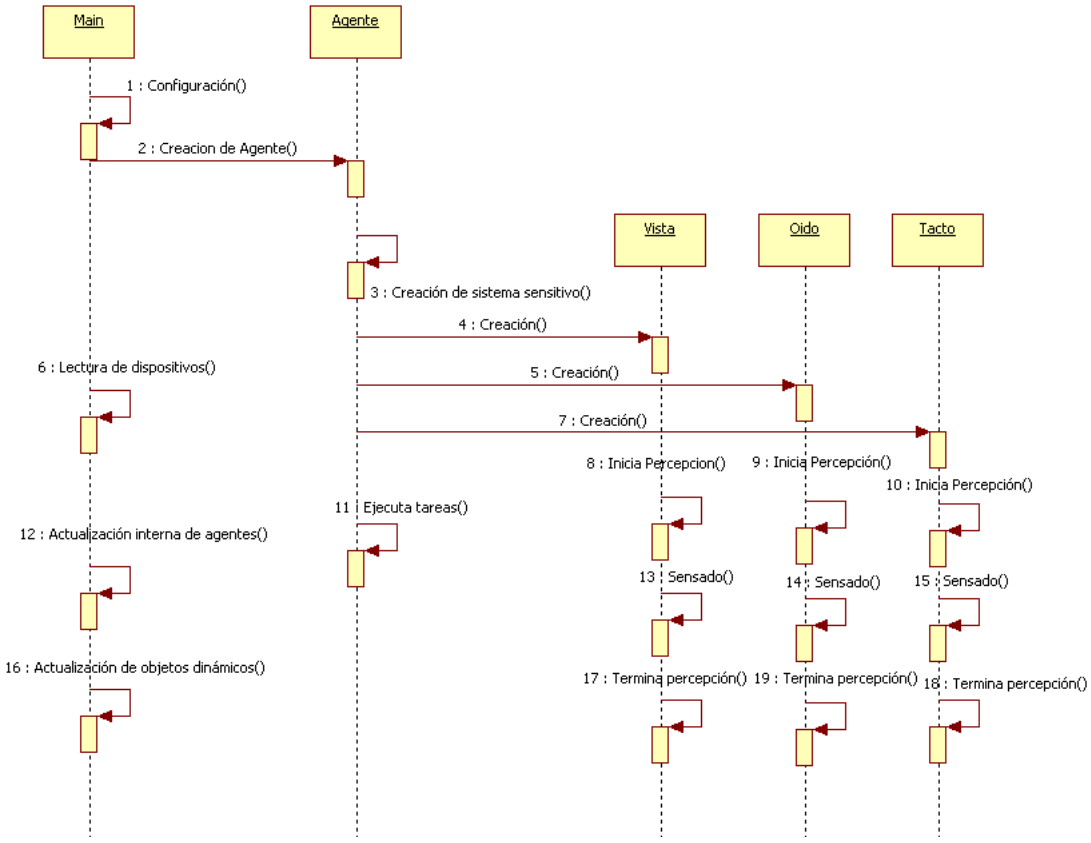


Figura 4.10: Secuencia de ejecución del aparato sensorial

Objeto	**Geometría asociada**
Pared	*Axis Aligned Bounding Box*
Estatua	*Axis Aligned Bounding Box*
Avatar	*Capsula*
	Bounding sphere
Disparo	*Bounding sphere*
Sonido virtual	*Bounding sphere*

Cuadro 4.1: Asociación de geometrías

Una vez que los objetos del mundo virtual tienen asignado una geometría, es posible realizar pruebas de intersección geométrica entre ellas para simular cada uno de los

Figura 4.11: Secuencia de ejecución de un sentido

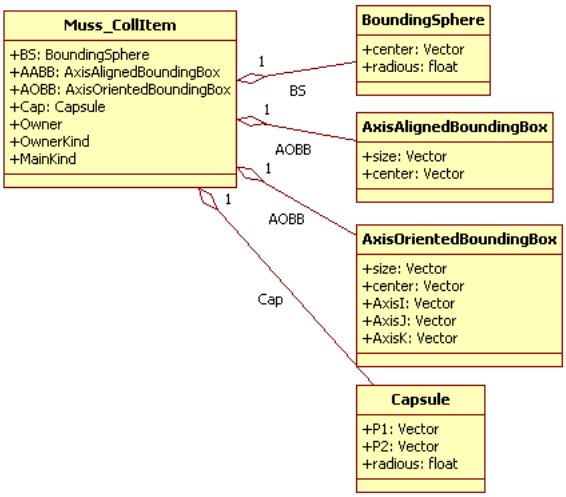


Figura 4.12: Estructura para almacenar los datos asociados a intersección

sentidos que se buscan implementar.

Vista

El sentido de vista considera que cada agente cuenta con un *campo de visión (FOV)* y un *alcance máximo de visión (LOV)* (*figura 4.13*). Cualquier objeto que entre en dicho volumen se considerará visible siempre y cuando no haya algún otro objeto que se interponga.

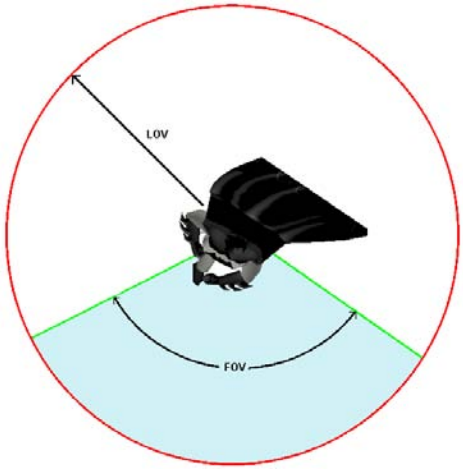


Figura 4.13: Campo de visión de un agente

Además de considerar los límites físicos propios, cada agente tiene un límite de memoria visual, es decir, sólo es capaz de recordar un número previamente especificado de objetos. El *código 4.2* muestra el proceso sensorial.

Código 4.2: Proceso sensorial para visión

```
1  /*Proceso de visión*/
2  for(int i=0;i<TotalObjetosDinamicos;i++)
3  {
4     if( distancia(Objetos[i].Posicion,
5                   Agente.Posicion)     < Agente.LOV )
6     {
7        Obj=Objetos[i].Posicion - Agente.Posicion ;
8        if( angulo_entre_vectores(Obj,
9                                  Agente.View) < Agente.FOV)
10          {
11             if( ExistePared(Objetos[i].Posicion,
12                             Agente,
13                             ObjectosEstáticos) ==false)
14             {
```

```
15                      AgregaMemoriaVisual(Objetos[i]);
16                  }
17              }
18      }
19  }
```

Los objetos dinámicos del sistema son la totalidad de agentes, mientras que las paredes, fuentes, estatuas y demás objetos que nunca cambian de posición son objetos estáticos.

Como parte del proceso de visión, la función `ExistePared` determina el objeto estático a menor distancia en la dirección a la que apunta el agente (*figura 4.14*).

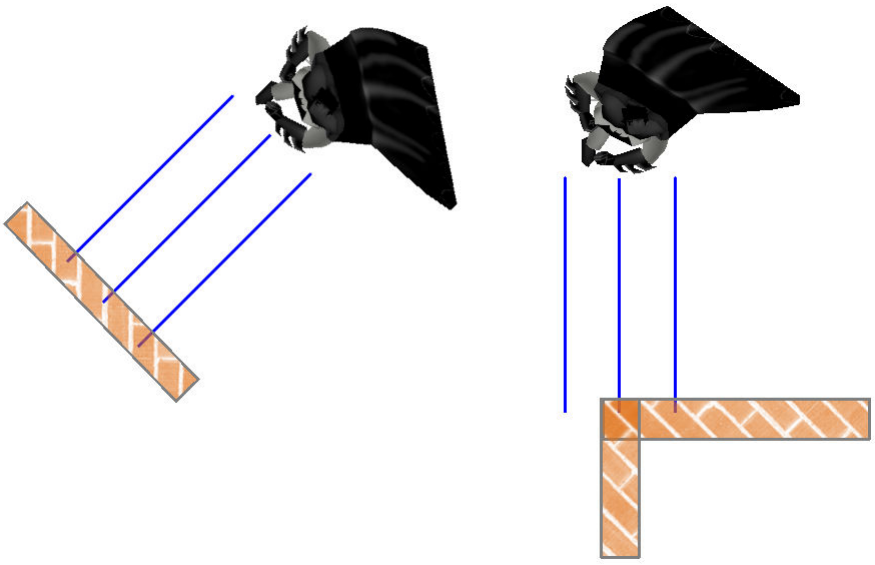


Figura 4.14: Detección de objetos cercanos

Oído

El sentido del oído considera que cada agente cuenta con un *alcance máximo de audición (LOH)* (*figura 4.15*). Cualquier *sonido* que entre en dicho volumen se considerará audible.

Figura 4.15: Campo auditivo de un agente

Además de considerar los límites físicos propios, cada agente tiene un límite de memoria auditiva, es decir, sólo es capaz de recordar un número previamente especificado de sonidos. El *código 4.3* muestra el proceso sensorial.

Código 4.3: Proceso sensorial para audio

```
1  /* Proceso de audición */
2  for(int i=0;i<TotalObjetosTemporales;i++)
3  {
4      if( Objetos[i].Tipo == Tipo_Sonido )
5      {
6          if( distancia(Objetos[i].Posicion,
7                        Agente.Posicion)     < Agente.LOH )
8          {
9              AgregaMemoriaAuditiva(Objetos[i]);
10         }
11     }
12 }
```

Algunos objetos dentro del sistema producen *sonidos virtuales* los cuales son almacenados en el contenedor de *objetos temporales*. Los *sonidos virtuales* tienen forma esférica,

una posición que no cambia y una intensidad que disminuye con el tiempo. El *cuadro 4.2* muestra la lista actual de entidades que generan *sonidos virtuales*. La estructura mostrada en la *figura 4.16* muestra el almacenamiento de objetos temporales y la *figura 4.17* muestra el sonido que genera un agente al caminar.

Generador de sonido	**Geometría asociada**	Parámetros iniciales
Agente caminando	Esfera	Radio=100, Factor de atenuación =1.0 Frecuencia de disminución = 50Hz
Agente agachado caminando	Esfera	Radio=50, Factor de atenuación =1.0 Frecuencia de disminución = 50Hz
Disparo	Esfera	Radio=200, Factor de atenuación =1.0 Frecuencia de disminución = 50Hz

Cuadro 4.2: Objetos generadores de sonidos virtuales

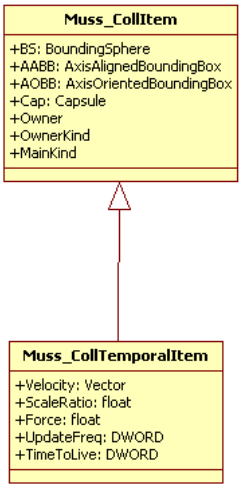


Figura 4.16: Estructura de almacenamiento de objetos temporales

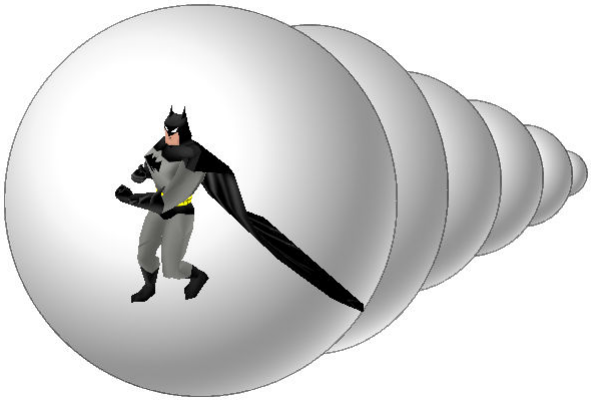


Figura 4.17: Sonido producido por un agente

Tacto

El sentido del tacto considera que ninguna entidad sólida puede atravesar otra; por ejemplo: ningún agente puede atravesar objetos, los disparos no atraviesan paredes ni tampoco cajas. Cada entidad cuenta con una geometría de colisión como se especifica en el cuadro 4.1 (*figura 4.18*). El *código 4.4* muestra el proceso sensorial.

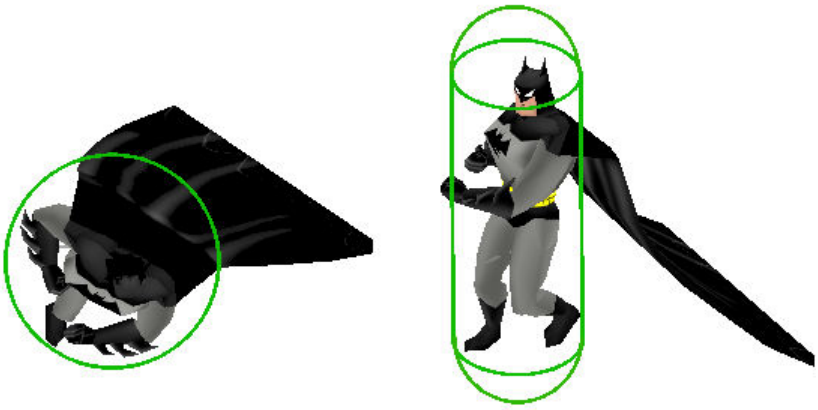


Figura 4.18: Geometría de colisión de un agente

Código 4.4: Proceso sensorial para tacto

```
1 /* Proceso de sistema táctil */
2
3 DoneStatic=false; i=0;
```

```
4  ObjetosStaticos  = ObjetosEstáticos.Raiz;
5  do
6  {
7     if( Intersección(ObjetosStaticos[i],
8                      Agente.Capsula) == true )
9     {
10       if( ObjetosStaticos[i].Contenedor== false)
11       {
12           DoneStatic = true;
13       }
14       else
15       {
16           i=0;
17           ObjetosStaticos=ObjetosStaticos[i].Raiz;
18       }
19    }
20 }while(!DoneStatic&&i<ObjetosStaticos[i].Count);
21
22
23 if(DoneStatic == false)
24 {
25     DoneTemp = false;
26     for(int i=0;i<TotalObjetosTemporales&&DoneTemp==false;i++)
27     {
28        if( Objetos[i].Tipo == Tipo_Disparo )
29        {
30          if( Intersección(Objetos[i],
31                           Agente.Capsula) == true )
32          {
33              DoneTemp=true;
34          }
35        }
36     }
37
38     if(DoneTemp == false)
39     {
40        DoneDynamic=false;
41
42        for(int i=0;i<TotalObjetosDinamicos&&DoneDynamic==false;i++)
43        {
44           if( Intersección(Objetos[i],
45                            Agente.Capsula) == true )
46           {
47               DoneDynamic=true;
48           }
49        }
50     }
51 }
```

Percepción interna

La *salud* y *municiones* forman parte de la estructura de datos propia del agente; mientras que la *posición* y *orientación* son parte del sistema de navegación, el cual se explicará en una sección posterior. La *figura 4.19* muestra la estructura de la clase agente, *Muss_Agent*.

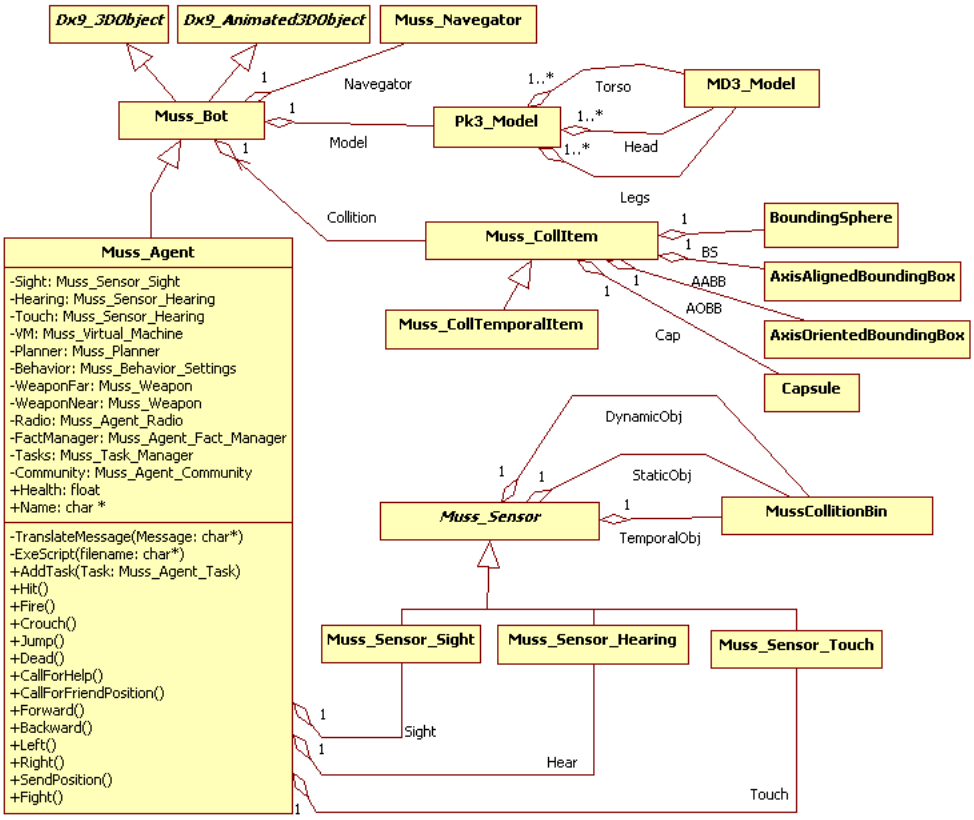


Figura 4.19: Estructura de la clase *Muss_Agent*

4.2.2. Sociedad

La estructura social, como se mencionó en el capítulo 3 sección 3.3.2, tiene una forma arbórea de manera que un agente puede pertenecer a un grupo y dicho grupo estar contenido en otro. La clase asociada se encuentra en la *figura 4.20*.

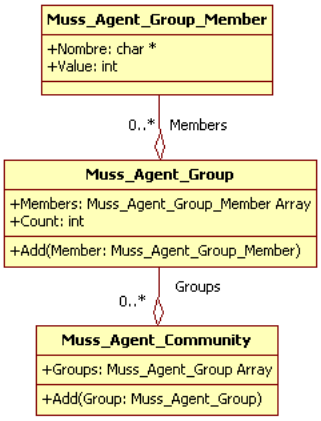


Figura 4.20: Estructura de la clase *Muss_Agent_Group*

Para poder definir los datos que llenarán la estructura, se hará uso de un *script* de *lua*. El *código 4.5* muestra un ejemplo de *script* de configuración. Como puede observarse, existen valores diferentes para cada uno de las posiciones sociales que tiene un mismo grupo, aunque pueden tomar el mismo valor. Cuando existen más categorías dentro del árbol, los nodos más cercanos al nodo raíz tienen mayor jerarquía que los nodos hoja.

Código 4.5: *Script* de configuración de estructura social

```
1  --[[Agents Society Model]]--
2  Root="Brigade";
3
4  Brigade={ "Heroes","Monsters",  Heroes=1, Monsters=1};
5  BrigadeSize = #Brigade;
6
7  Heroes={"Stratego","Spartan","Soldier",   Stratego=3, Spartan=2, Soldier=1};
8  HeroesSize= #Heroes;
9
10 Monsters={"Master","Warlock","Creature", Master=3, Warlock=2, Creature=1};
11 MonstersSize= #Monsters;
```

4.2.3. Razonamiento

El módulo de razonamiento se apoya en programar el sistema de reglas establecido en el capítulo 3 sección 3.3.3. Para ello se incorporará *CLIPS* [Gar08a] de forma embebida dentro del sistema. De esta forma, cada agente puede hacer uso de su propia instancia de *CLIPS* de forma independiente.

Cada *script* de *CLIPS* necesita ser alimentado por un conjunto de hechos que permitan ejecutar el sistema de reglas, por lo que a cada agente se le ha dotado de un manejador de hechos, así como diversas *fábricas* de los mismos. Cada fábrica está encargada de generar el hecho que le corresponde haciendo uso de los datos provistos por el *sistema sensorial*. Las clases mostradas en la *figura 4.21* cubren con las funciones descritas.

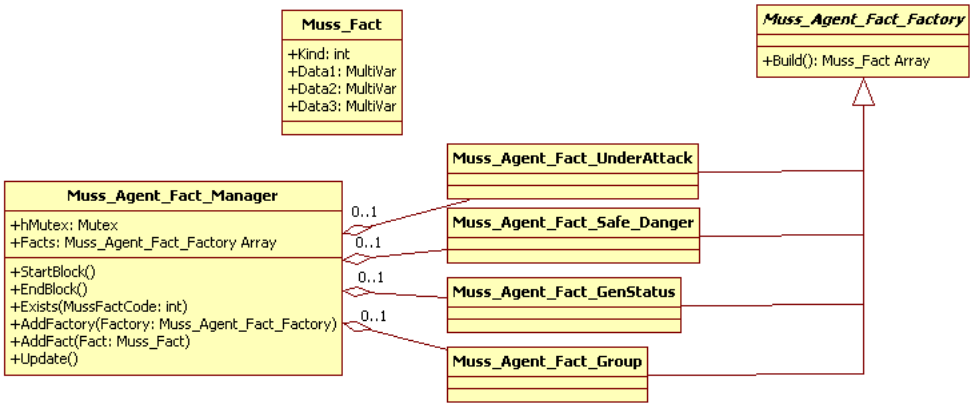


Figura 4.21: Clases involucradas en el proceso de generación de hechos

En el sistema, cada agente cuenta con su propio dispositivo manejador de hechos (clase *Muss_Agent_Fact_Manager*) y diversas clases cuya labor es fabricar hechos (clases *Muss_Agent_Fact_Safe_Danger*, *Muss_Agent_Fact_GenStatus*, *Muss_Agent_Fact_Group* y *Muss_Agent_Fact_UnderAttack*) periódicamente. El manejador de hechos almacena la información generada por cada fábrica, la cual se utilizará como parámetros de entrada para *CLIPS*.

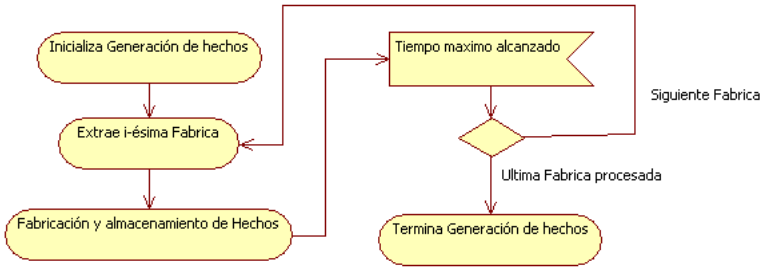


Figura 4.22: Proceso de generación de hechos

Las fábricas de hechos pueden hacer uso o no del sistema sensorial, por lo que inferir un hecho puede implicar también algún cálculo que demande mucho tiempo, es por

ello que el proceso de actualización de hechos sigue el esquema de *microthreads* para poder asignarle un tiempo máximo de procesamiento a cada ejecución.

El *cuadro 4.3* muestra la generación de hechos por fábricas.

Fabrica	**Hechos**
Muss_Agent_Fact_Safe_Danger	Peligro
	Seguro
	Enemigo a la vista
Muss_Agent_Fact_GenStatus	Municiones suficientes
	Salud suficiente
Muss_Agent_Fact_Group	Equipo cercano
Muss_Agent_Fact_UnderAttack	Bajo ataque enemigo

Cuadro 4.3: Generación de hechos por fábrica

Aun cuando existen fábricas de hechos, algunos de ellos es posible que no sean creados de esta manera; como por ejemplo el hecho *llamada de auxilio*, el cual es recibido por el sistema de comunicaciones y agregado directamente al manejador de hechos.

Cada agente puede utilizar su propio *script* en *CLIPS*, sin embargo, para este trabajo todos los agentes cuentan con el mismo programa. El código del *script* se encuentra en el Apéndice A.

4.2.4. Memoria

La memoria del agente se encuentra implementada como un árbol que divide el tipo de información a almacenar por categoría (tipo) y después es ordenada desde la más reciente a la más antigua. Una parte importante de la memoria se encuentra contenida dentro del proceso de razonamiento y el formato de almacenamiento corresponde al de un *hecho* anteponiendo las características indicadas en la sección 3.3.4 del capítulo 3. La *figura 4.23* muestra la estructura de almacenamiento de memoria y la *figura 4.24* muestra la organización de la memoria dentro del proceso de almacenamiento.

Existe otra sección de la memoria que se encuentra dentro del módulo del *banco de comportamientos/procedimientos*.

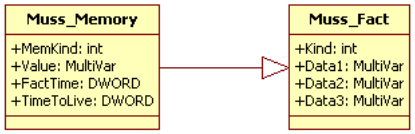


Figura 4.23: Estructura de almacenamiento de memoria

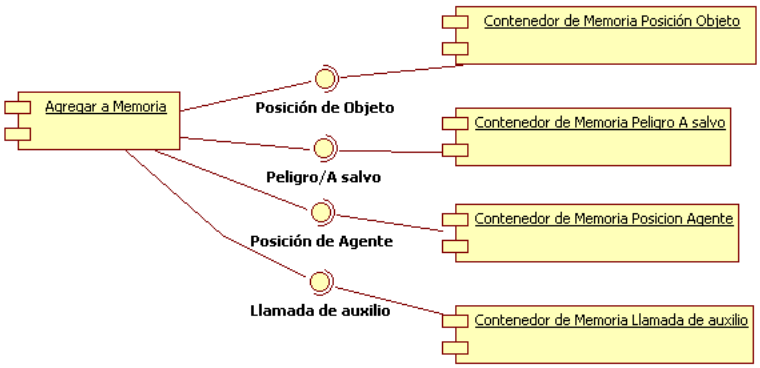


Figura 4.24: Proceso de almacenamiento de memoria

4.2.5. Comunicación

La implementación de este módulo se realizó usando una configuración de *sockets UDP multicast*, la cual se muestra en la *Figura 4.25*.

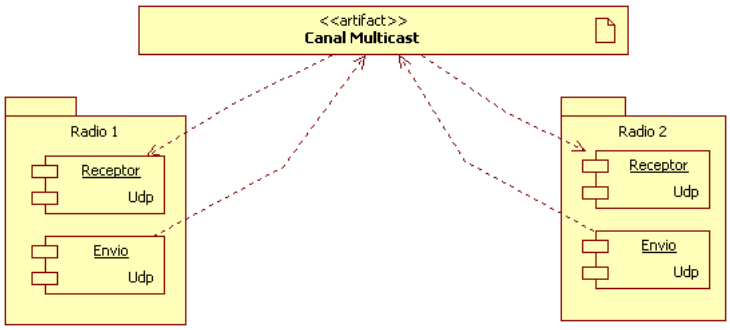


Figura 4.25: Estructura del módulo de comunicación

El canal de comunicación común del *multicast* permite establecer contacto entre todos los clientes que se conecten a él; por lo que al usar este esquema, es posible que

cada *asociación* pueda intercambiar mensajes de forma exclusiva entre sus miembros. Los destinatarios de los mensajes pueden ser uno o mas agentes miembros del equipo, lo cual se especifica en el formato del mensaje.

El envío de mensajes se encuentra en formato *KQML* mientras que el contenido se encuentra en *XML*. El siguiente código es un ejemplo del tipo de mensajes que se envían:

```
<msg>
    <from>Batman</from>
    <to>All</to>
    <context>Info</context>
    <content>
    <help  pos_x=40.0 pos_z=400.0 />
    </content>
</msg>
```

Para poder incorporar a cada agente esta metodología de comunicación, se construyó la clase *Muss_Agent_Radio* que permite realizar el mecanismo descrito anteriormente. La *Figura 4.26* muestra la estructura que tiene la implementación descrita para este módulo.

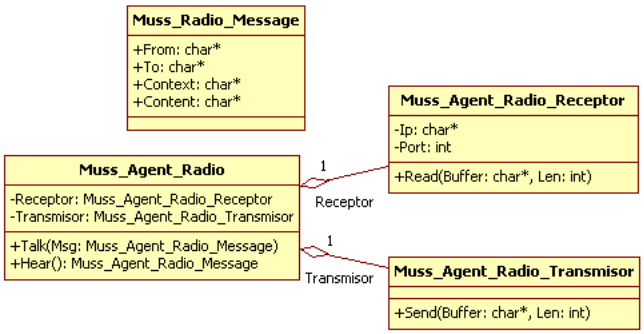


Figura 4.26: Estructura de la implementación del módulo de comunicación

4.2.6. Banco de comportamientos/procedimientos

En muchas ocasiones los comportamientos que un agente necesita realizar, aun cuando sean factibles, son difíciles de establecer de forma manual. Uno de los comportamientos necesarios que no son simples de poder planear es el combate (*atacar un enemigo*).

En principio, un agente es capaz de percibir y de ejecutar acciones de distintas maneras, además de que el entorno cambia según la situación en la que se encuentre. Es posible que un agente que ejecuta acciones haciendo uso de un algoritmo fijo no pueda funcionar bien en otras situaciones y se tendría que verificar la validez del algoritmo original en el nuevo entorno, hacer modificaciones y de ser necesario, replantear el algoritmo completamente.

Dentro de este módulo, se pretende generar *automáticamente* comportamientos de combate para agentes. Cada agente tiene una cantidad bien establecida de *acciones simples* como: caminar a la izquierda, caminar a la derecha, caminar hacia enfrente, caminar hacia atrás, etc; además de las lecturas propias del *sistema sensorial*. El cuadro 4.4 muestra la lista de *acciones simples* que todo agente puede ejecutar.

Acción simple	**Acción simple**
avanzar	retroceder
moverse a la izquierda	moverse a la derecha
saltar	agacharse
disparar	golpear

Cuadro 4.4: *Acciones simples* que puede ejecutar un agente

El procedimiento que se utilizará, consiste en generar un programa mediante programación genética con *ADF*[12] (sección 1.6.4). Las *ADF* que se utilizarán se les dará una representación arbórea donde en cada nodo se ejecutará una función generada automáticamente, una acción básica o alguna condicional; la *figura 4.27* muestra un ejemplo de la forma de las *ADF* que se utilizarán. Las entidades consideradas en los *ADF* son:

Funciones. Reciben un número constante de parámetros y regresan un valor; además pueden ser llamadas en el cuerpo del programa de forma individual y también pueden ser parte de la evaluación de una expresión matemática en un condicional o como parámetro de entrada de otra función

Argumentos. Sólo pueden aparecer en el cuerpo de una expresión matemática para evaluación en un condicional o como parámetros de entrada de alguna función, que a su vez puede ser una expresión utilizando las mismas.

Acciones. Sólo pueden ser llamadas de forma individual y no pueden ser llamadas dentro de alguna expresión de evaluación.

[12] ADF. Funciones que se generan automáticamente, *Automatically Defined Functions*

Condicional. Está formada por una expresión de evaluación y dos ramas: de resultar verdadera la expresión de evaluación, entendiendo por verdadero distinto de cero, se ejecutara el código correspondiente a la rama izquierda y al ser falsa se ejecutará el código correspondiente a la rama derecha.

Constantes. Valores que pertenecen a los números reales en un intervalo fijo.

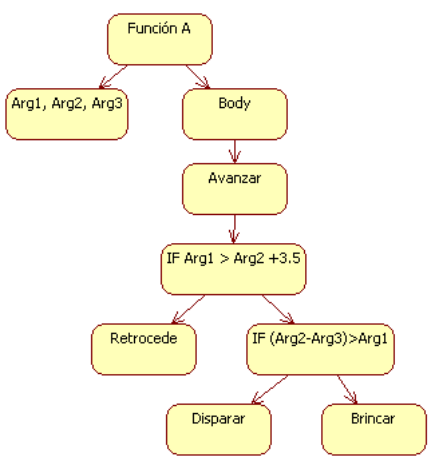
La *figura 4.27* muestra el uso de los elementos descritos dentro de la estructura de un *ADF* así como su representación dentro del cromosoma. La función llamada *Función A* tiene una estructura con argumentos constantes y el cuerpo será generado por el proceso evolutivo del algoritmo. Aun cuando en la figura sólo se muestra la estructura de la función *Función A*, puede haber más funciones definidas de forma muy similar. Las funciones generadas por el proceso del *ADF* pueden ser llamadas en el cuerpo de *Main* exclusivamente, de esta forma se evitan posibles ciclos infinitos. Las *figuras 4.27(c)* y *4.27(d)* representan a los cromosomas equivalentes de la *Función A* y *Función Principal* respectivamente. Como puede observarse, las operaciones aritméticas siguen notación prefija; y además se hace uso de marcadores (*ELSE y ElseIF*) para delimitar el alcance del operador condicional *IF*. Durante el proceso de generación de individuos se agrega la restricción del control de aridad para evitar crear individuos no válidos. La representación de un individuo tiene un cromosoma resultante formado por la unión de cuatro cromosomas similares a los mostrados en la *figuras 4.27(c)* y *4.27(d)* por lo que su longitud es variable.

Como se describió anteriormente, en cada nodo del árbol generado se ejecutará: una función generada automáticamente, una acción básica o alguna condicional. El objetivo que se buscará es poder vencer cada situación de combate con la mayor salud posible antes de que se terminen las municiones. Para ello se establecerán un pequeño banco de situaciones a las cuales se enfrentará cada individuo generado por el programa evolutivo.

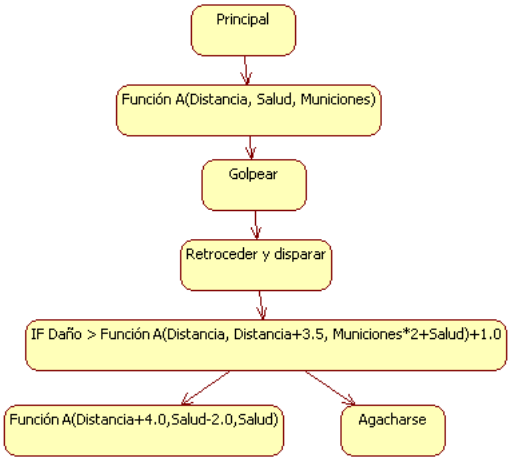
El banco de pruebas está formado por un agente que tiene que cumplir la tarea de sobrevivir a ataques proporcionados por otro agente. El agente al cual se le generará el comportamiento se le llamara *jugador* y al agente que atacará se le llamará *Enemigo*. El *jugador* tendrá a su disposición el conjunto completo de *acciones simples* mientras que *enemigo* sólo podrá *apuntar* y *disparar* además, sus movimientos estarán dados por una curva pseudoaleatoria (curva de *Lissajous*[13]) como la mostrada en la *figura 4.28*.

Respecto a las funciones de evaluación utilizadas, se probaron dos funciones de evaluación distintas. En la primera función de evaluación propuesta (*función A*), los agentes lograban *aprender* la secuencia que seguía su oponente y con ello esquivaban todos los

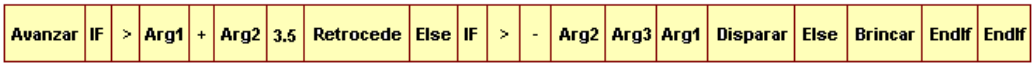
[13] $(x, y) = (A * sin(a * t + \delta), B * sin(b * t)); A = 1, B = 2$



(a) Función A



(b) Función Principal



(c) Cromosoma equivalente Función A

(d) Cromosoma equivalente Función Principal

Figura 4.27: Ejemplo de una *ADF*

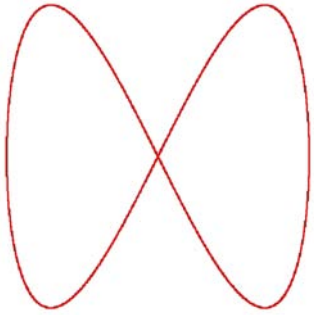


Figura 4.28: Trayectoria de *Enemigo*

ataques. La *función B* genera movimientos erráticos que permiten atacar y además evadir ataques del enemigo. Las funciones de evaluación mencionadas se muestran en los *Códigos 4.6 y 4.7*.

Código 4.6: Función A

```
1  /*FUNCION A*/
2      if(!ActionCount||Movements<5)
3          return 10000.0;
4      if(!EvalCrit->KCount[0]->pFunc->Count())
5          return 6000.0;
6      if(g_DamageCreated+g_DamageTaken==0.0||g_DamageCreated==0.0)
7          return 5000.0;
8      else if(g_DamageCreated==g_DamageTaken)
9          res=4000.0-g_DamageCreated;
10     else if(g_DamageCreated<g_DamageTaken)
11     {
12         if(((float)g_DamageCreated/(float)g_DamageTaken)>0.2f)
13           res=6500.0;
14         else
15           res=3000.0-(g_DamageTaken+1.0)/(g_DamageCreated+1.0);
16     }
17     else
18         res=2000.0-(g_DamageTaken+1.0)/(g_DamageCreated+1.0);
```

La función A penaliza fuertemente a los individuos que no se mueven y que tampoco usan acciones. La siguiente penalización abarca a los individuos que recibieron mas daño que el que generaron. Los valores inferiores a 2000 corresponden a individuos que en *proporción* pueden cometer más daño que el daño recibido.

Código 4.7: Función B

```
1  /*FUNCION B*/
2      if(!ActionCount||Movements<5)
3          return 10000.0;
4      if(!EvalCrit->KCount[0]->pFunc->Count())
5          return 6000.0;
6      if(g_DamageCreated+g_DamageTaken==0.0||g_DamageCreated==0.0)
7          return 5000.0;
8      else if(g_DamageCreated==g_DamageTaken)
9          res=4000.0-g_DamageCreated;
10     else if(g_DamageCreated<g_DamageTaken)
11     {
12         if(((float)g_DamageCreated/(float)g_DamageTaken)>0.2f)
13           res=6500.0;
14         else
15           res=3000.0-g_DamageTaken;
16     }
17     else
18         res=2000.0-g_DamageCreated;
```

La *función B* penaliza fuertemente a los individuos que no se mueven y que tampoco usan acciones. La siguiente penalización abarca a los individuos que recibieron mas daño que el que generaron. Los valores inferiores a 2000 corresponden a individuos que *en unidades* pueden cometer más daño que el daño recibido.

Aun cuando se obtienen lo que parecería excelentes resultados con la *función A*, se seleccionaron los comportamientos creados por la *función B* debido a que la secuencia para esquivar disparos de manera perfecta solo podría funcionar ante la situación del experimento. El comportamiento errático y agresivo de la *función B* permite tener mejor desempeño ante más situaciones.

El siguiente es un ejemplo del código generado mediante *ADF* (*código 4.8*):

Código 4.8: Ejemplo de código generado mediante *ADF*

```
1  Eval:1970.000
2  MaxSize:128
3  FuncCount:3
4  Body, Size: 128
5  400,200,100,600,601,200,100,101,201,102,602,100,101,102,200,101,102,101,
6  102,300,301,777,302,303,777,313
7  Func#0, Size: 73
8  312,400,204,102,561,400,204,102,201,540,540,303,310,777,310,777,303,307,
9  400,203,204,202,202,514,204,514,101,589,200,595,549,204,201,102,203,533,
10 102,514,777,400,201,100,100,777,777,311,311,400,203,101,547,777,777,400,
11 202,200,100,203,102,201,101,204,102,100,101,777,777,313,777,777,777,310,302
12 Func#1, Size: 24
13 308,307,310,301,310,306,310,310,305,400,200,100,200,203,100,571,102,777,
14 303,777,310,310,313,310
15 Func#2, Size: 5
16 307,310,310,307,306
```

Como puede observarse, el programa se encuentra codificado haciendo uso de números enteros. Cada entero está asociado a diferentes categorías de elementos del cuerpo del programa; por ejemplo: los enteros en el rango de 300 a 311 indican el código de una acción, los códigos 200 a 204 corresponden a operadores aritméticos, etc. El *cuadro 4.5* muestra las equivalencias de los código utilizados.

Una vez que los programas han sido creados es necesario convertirlos de la interpretación arbórea codificada en la cual se encuentran, a un formato que sea posible ejecutar de forma más eficiente por la *unidad de control de secuencia*. La *figura 4.29* ilustra el procedimiento de conversión.

En el *Apéndice B* se muestran los detalles del experimento creado para la evolución de los programas usando programación evolutiva.

Elemento	Rango de códigos
Argumentos	100 – 103
Operadores aritméticos	200 – 204
Acciones	300 – 311
Condicional	400, 777
Constantes	500 – 599
Funciones	600 – 602

Cuadro 4.5: Significado de los códigos de los cromosomas

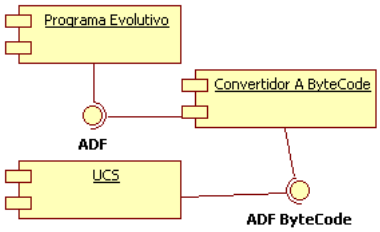


Figura 4.29: Secuencia de conversión de código

Comportamientos adicionales

No todos los procedimientos del sistema se han creado mediante el uso de *ADFs*; el comportamiento *huir* y el comportamiento *buscar enemigo*, por ejemplo, se realizaron mediante la codificación directa de una secuencia de acciones.

Huir

Este comportamiento aleja al agente de los enemigos que se encuentren cerca de él. Su implementación consiste en generar un vector que apunte en dirección contraria a los agentes enemigos que se encuentren cerca, a éste vector se le llamará *vector de escape* ($\overline{V_s}$)[Mil06]. El cálculo del *vector de escape* es el siguiente:

$$\overline{V_s} = \sum_e \left(\frac{\overline{V_a} - \overline{V_e}}{((\overline{V_a} - \overline{V_e}) \cdot (\overline{V_a} - \overline{V_e}))^{\frac{3}{4}}} \right)$$

Donde:

V_a es la posición del agente

V_e es la posición de enemigo

Básicamente, se calculan un vector unitario en dirección contraria a cada enemigo,

dicho vector unitario después es dividido entre la distancia que existe entre el agente y el enemigo; con esto los enemigos más cercanos influirán con mayor fuerza al *vector de escape*.

Buscar enemigo

La implementación de la búsqueda se realizará usando un algoritmo de *wandering* (*vagar*) [Mil06]. El algoritmo considera que los agentes pueden controlar de forma independiente su orientación y su desplazamiento. En la primera etapa, se selecciona aleatoriamente un ángulo de desvío dentro de un rango previamente definido. La siguiente etapa consiste en desviar la dirección hacia donde observa el agente con el ángulo calculado anteriormente. El tercero y último paso consiste en avanzar en la dirección elegida.

Para el caso particular de este trabajo, el algoritmo de *wandering* es muy simple de implementar porque los agentes cuentan con el control independiente de orientación y movimiento. Sin embargo, es necesario modificar el algoritmo para acoplarse correctamente. Las modificaciones son dos: la primera consiste en verificar, antes de avanzar, que no se encuentre algún obstáculo enfrente y en dado caso seleccionar un ángulo de corrección en dirección contraria al obstáculo; la segunda modificación consiste en realizar lecturas del *sistema sensorial* para determinar si se ha encontrado algún enemigo.

4.2.7. Unidad de control de secuencia *UCS*

La estructura interna de la *UCS* se ilustra en la siguiente *figura 4.30*

Como puede observarse, la *UCS* cuenta con un núcleo principal (*Muss_VMCore_Main*) donde es depositado el código del programa que es utilizado; además cuenta con otros tres núcleos (*Muss_VMCore_1*, *Muss_VMCore_2*, *Muss_VMCore_3*) donde son depositados subprogramas que son utilizados durante la ejecución del programa principal. Cada núcleo cuenta con espacio para cargar código (*Muss_VMData*), un *stack* (*Muss_VMStack*) y almacenamiento para datos (*Muss_VMRegisters*). Todos los núcleos pueden acceder a un *ALU* para realizar las operaciones que necesiten, también pueden invocar la ejecución de acciones de forma externa (*Action_Poll*) y además obtener datos de las lecturas del sistema sensorial (*Param_Extractor*).

Las clases mostradas en la *figura 4.31* crean la estructura de la *UCS* descrita.

Cada uno de los métodos *Execute* de las clases presentadas tienen asignados un límite de tiempo de procesamiento, por lo que su ejecución no es demandante de recursos de procesador.

Existen dos motivos por los cuales la *UCS* tiene núcleos adicionales al núcleo principal. El primero consiste en acoplar los programas generados por el *ADF* de forma

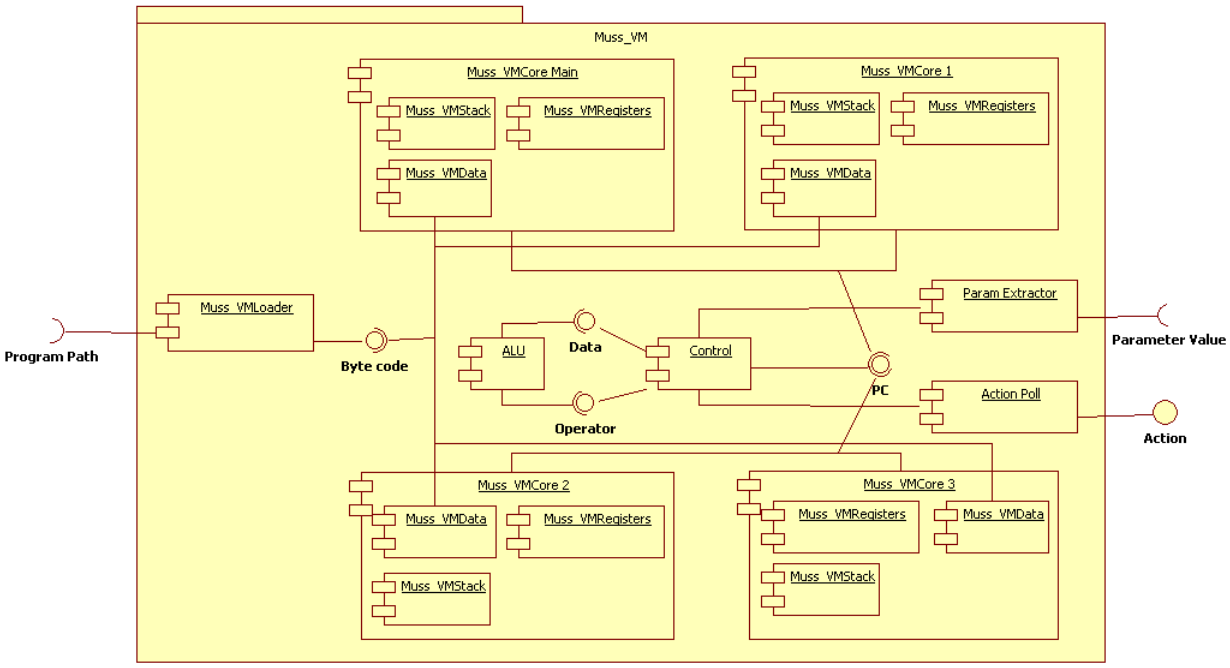


Figura 4.30: Estructura interna de la *UCS*

natural al sistema, ya que los programas de la *ADF* tienen tres funciones secundarias y una principal. El segundo motivo consiste en dar la posibilidad de una segunda versión del *UCS* que mantenga en ejecución un bloque de código de forma continua, sobretodo si se considera una ejecución *multicore* en *CPUs* reales y con ello utilizar el *hardware* que las nuevas generaciones de consolas de videojuegos tienen.

4.2.8. Navegación

El módulo de navegación, como se mencionó en el capítulo 3, está dividido en tres partes: *control de movimientos*, *control de posición y orientación*, y *cálculo de rutas*.

Para el *control de movimientos* y *control de posición y orientación* se creó una clase que permitiera realizar los cálculos correspondientes al movimiento y además llevar el control de los datos actuales de posición y orientación. La *figura 4.32* muestra las estructuras que almacenan la información requerida y los métodos para realizar movimientos.

Los cálculos correspondientes a los movimientos de los agentes son los mismos a los mostrados en el apartado de *Control de movimientos* (sección 4.1.3) de este capítulo.

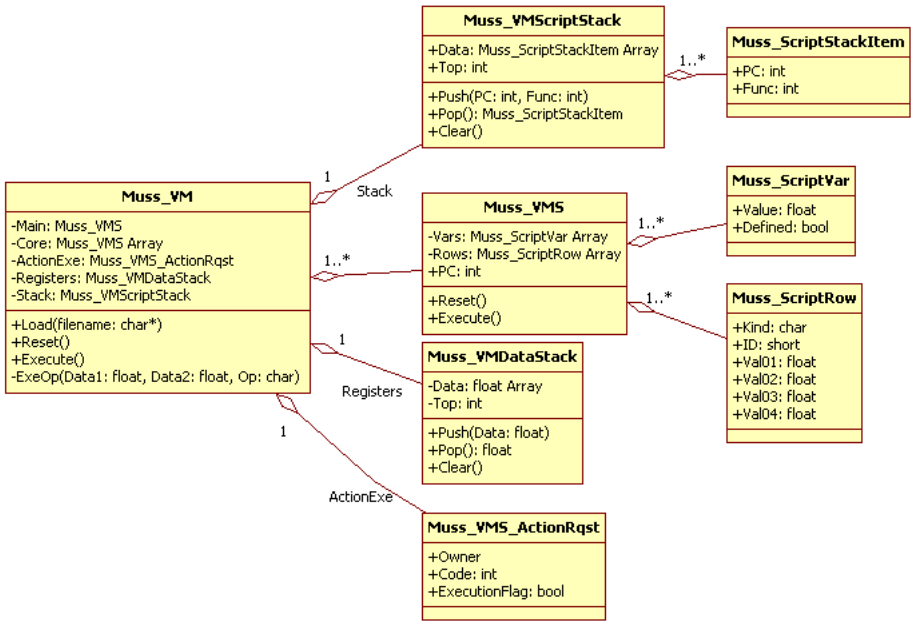


Figura 4.31: Clases para construcción de la *UCS*

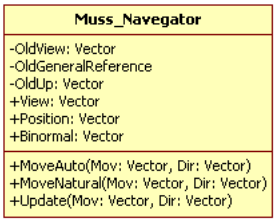


Figura 4.32: Clases para el control parcial de navegación

Para el cálculo de rutas es necesario contar con la información del *nivel* donde se encuentra el agente (*cartógrafo*); con esta información y haciendo uso del algoritmo A^* es como se realiza la navegación.

4.2.9. Cartógrafo

Desde las especificaciones del sistema gráfico, se tiene que los modelos del mundo virtual están constituidos por modelos tridimensionales en formato *X* y el cartógrafo debe contener un mapa topológico de la escena. Es por ello que es necesario construir una

aplicación que procese las geometrías contenidas en el modelo en formato *X* para generar, tanto los volúmenes de colisión asociados a las geometrías, como un mapa topológico. El procesamiento del archivo *X* se realizará *fuera de línea*, es decir, antes de la ejecución del mundo virtual, con unos cuantos parámetros de entrada: nombre del archivo a procesar y la densidad de análisis de la malla en dirección *X* (*N*) y en dirección *Z* (*M*).

Los pasos que sigue el programa analizador de geometrías son:

1. Extracción una a una de las geometrías
2. Obtención del *AABB* y *Cápsula*
3. Organización del árbol de volúmenes de colisión
4. Obtención del nodo raíz del árbol para tener las dimensiones de la escena
5. División de la escena en nodos pertenecientes a una malla de densidad *NxM*
6. Para cada nodo de la malla, verificar si está contenido dentro de algún *AABB*
7. Almacenamiento de resultados del árbol de colisiones y de los nodos de malla

Como puede verse, el programa básicamente divide la escena en *NxM* nodos y luego uno a uno verifica si está dentro o fuera de alguno objeto. Los nodos que se encuentren fuera de cualquier otro objeto de la escena se considerarán nodos libres y los demás serán nodos ocupados; esto formará la topología de la escena. La *figura 4.33* muestra una escena con obstáculos que ha sido dividida en nodos, los nodos libres corresponden al cruce entre las líneas y los nodos ocupados están marcados dentro de los objetos contenidos en la escena.

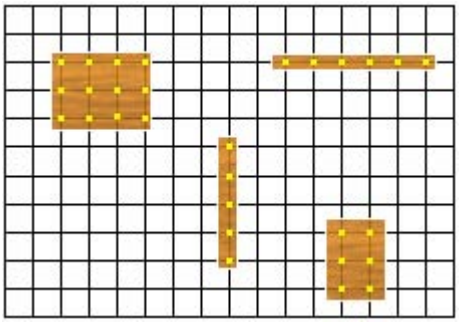


Figura 4.33: Creación de la topología de una escena

La conectividad entre nodos se realiza mediante reglas simples. Un nodo está conectado siempre con los nodos que se encuentran a distancia unitaria de él (*figura 4.34*). De la *figura 4.34* podemos ver que la conectividad del *nodo E* es sólo con los nodos *B*, *D*, *F* y *H*.

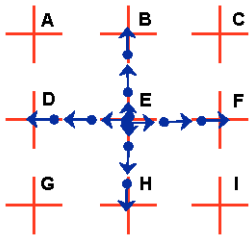


Figura 4.34: Conectividad entre nodos

La *figura 4.35* muestra los puntos calculados por el procedimiento descrito; las pequeñas cruces en el suelo son su marca.

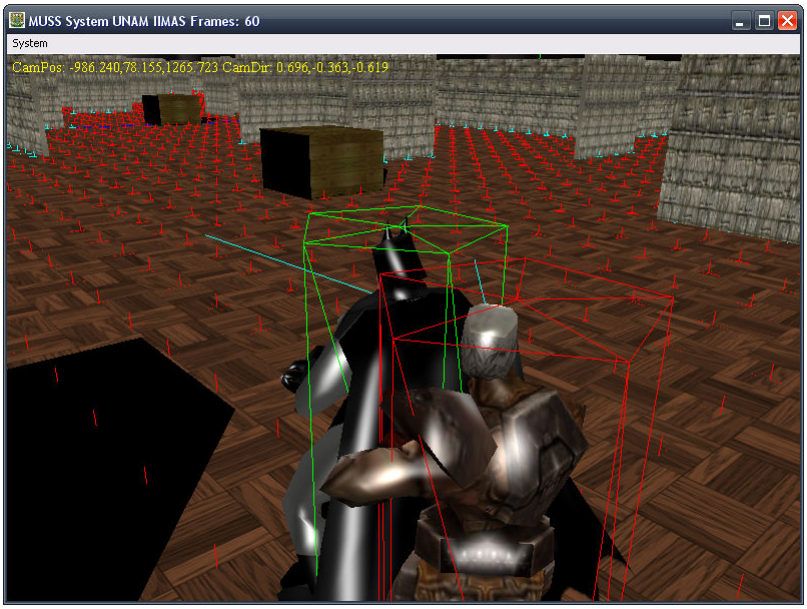


Figura 4.35: Topología de un nivel

Una vez que se cuenta con la topología, es posible realizar cálculos de rutas mediante el algoritmo A^*. El cómputo asociado al cálculo de rutas tradicionalmente es demandante de recursos, por ello es necesario considerar que su realización puede llevarse mucho tiem-

po. La *figura 4.36* muestra las estructuras y clases necesarias para realizar la planeación de rutas para un agente.

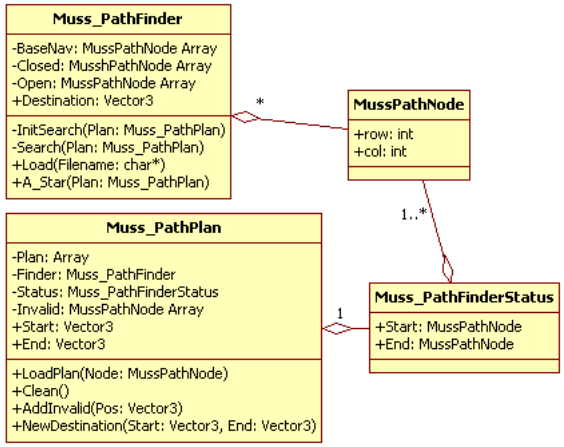


Figura 4.36: Clases para la creación de rutas

Para evitar que la ejecución del algoritmo se lleve demasiado tiempo se realizó una implementación de *microthreads* para dividir los cálculos en múltiples cuadros de la ejecución del mundo virtual. El *microthread* considera las operaciones de verificación de nodos abiertos y nodos cerrados como las partes clave para realizar los cálculos rápidamente.

Capítulo 5

Pruebas de sistema

Debido a que el sistema elaborado está compuesto por diversos módulos, las pruebas realizadas se dividieron en las siguientes secciones:

1. **Sistema sensorial**
2. **Comportamientos / Procedimientos**
3. **Razonamiento**
4. **Comportamiento general**

Para realizar algunas de las pruebas listadas es necesario conocer datos del estado interno del agente, para ello se elaboró un pequeño módulo adicional que se incorporó a cada agente el cual permite ver cual es el estado actual. A este módulo se le llamo *BrainViewer* el cual funge como una pizarra en la cual se puede saber el estado en el cual se encuentra cada agente. La información mostrada en la pizarra se explicará en cada sección de prueba.

Para todas las pruebas descritas se construyó un *nivel* que consta de 170 objetos estáticos diferentes, 20 obstáculos desconocidos y una malla de 40000 nodos para el cartógrafo.

5.1. Sistema sensorial

Las pruebas realizadas en esta fase son relativamente simples y corresponderán a verificar el correcto funcionamiento de los *sensores externos* (vista, oído, tacto) e *internos* (salud y municiones).

5.1.1. Vista

Las pruebas se pueden catalogar en:

1. Detección de agentes sólo dentro del *FOV* y *LOV*.
2. Objetos detrás de paredes no deben de ser visibles.
3. Ejecución por múltiples agentes

Detección de agentes sólo dentro del *FOV* y *LOV*

Se verificará que los agentes sólo puedan percibir objetos que se encuentren dentro de su *FOV* y *LOV* establecidos.

En la *figura 5.1* es posible ver que el *agente A* tiene dentro de su *LOV* (marcado como un octágono) y de su *FOV* (marcado como 2 líneas) al *agente B*. La *figura 5.2* muestra un acercamiento de la misma escena donde es posible ver el *BrainViewer* marcado con un rectángulo punteado y con una flecha punteada la indicación de que hay un agente a la vista.

La *figura 5.3* permite ver que el *Agente B* está fuera del *LOV* del *Agente A* pero si está dentro del *FOV*. El *BrainViewer* del *Agente A* no reporta al *Agente B* como visible.

La *figura 5.4* permite ver que el *Agente B* está fuera del *FOV* del *Agente A* pero si está dentro del *LOV*. La *figura 5.5* muestra un acercamiento de la misma escena donde es posible que el *BrainViewer* del *Agente A* no reporte al *Agente B* como visible.

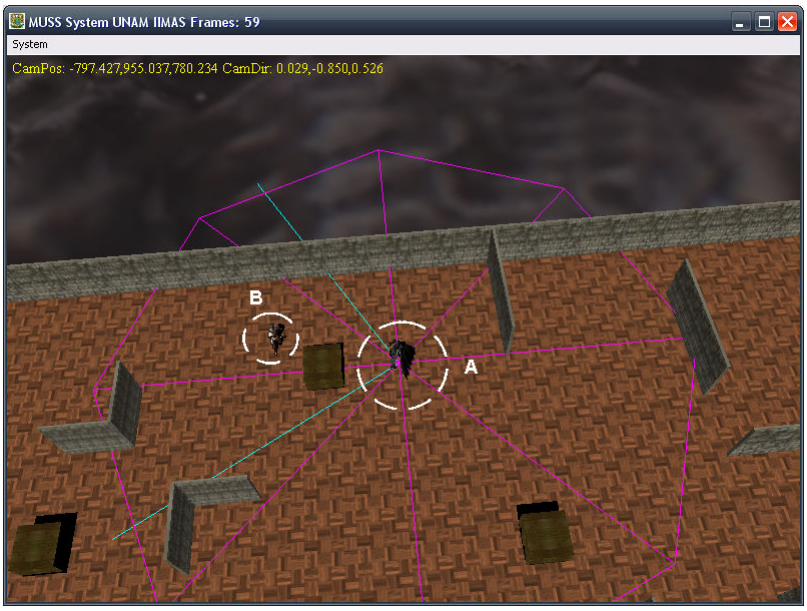


Figura 5.1: *El Agente B* está dentro del *FOV* y *LOV* del *Agente A*

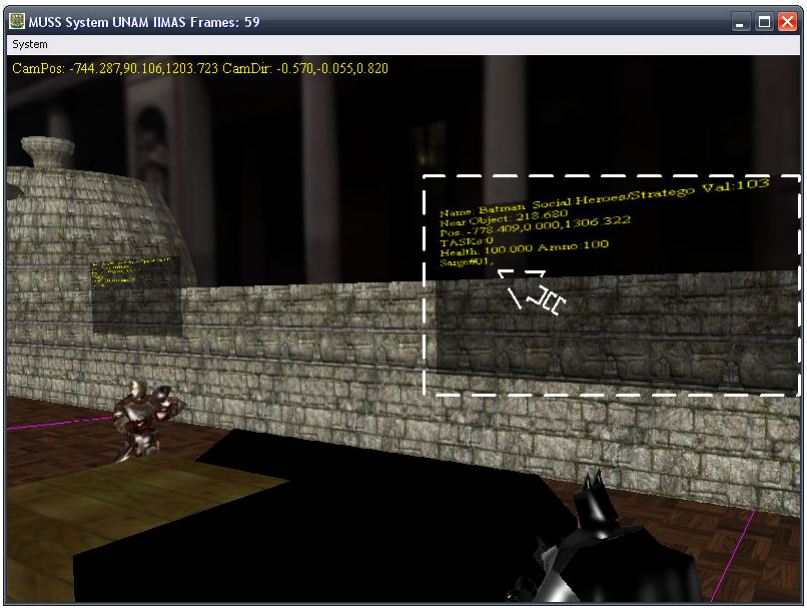


Figura 5.2: *BrainViewer* permite ver que el *Agente A* puede ver al *Agente B*

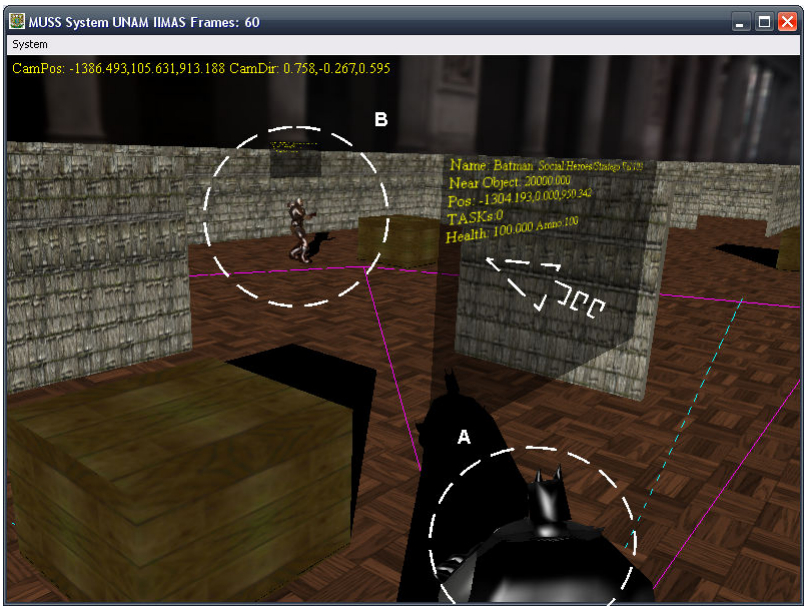


Figura 5.3: *Agente B* no se encuentra visible para el *Agente A*

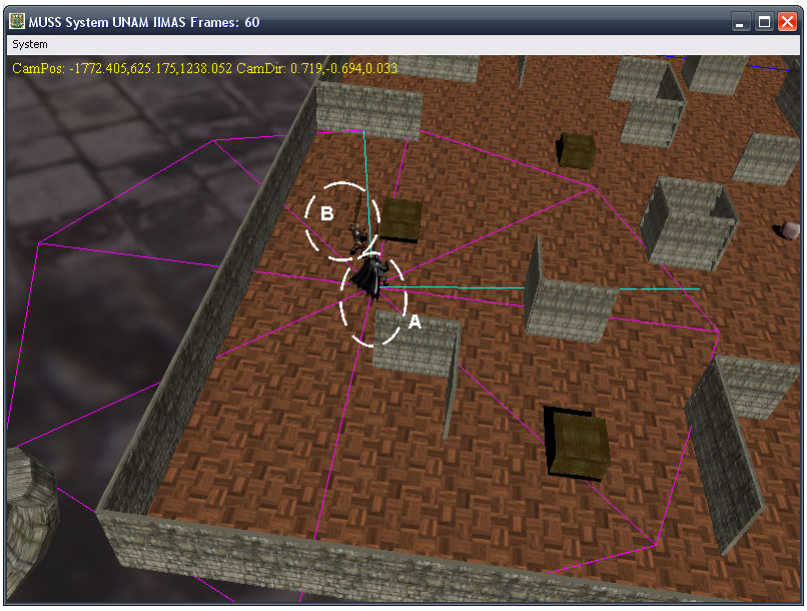


Figura 5.4: *El Agente B* está dentro del *LOV* y *FOV* del *Agente A*

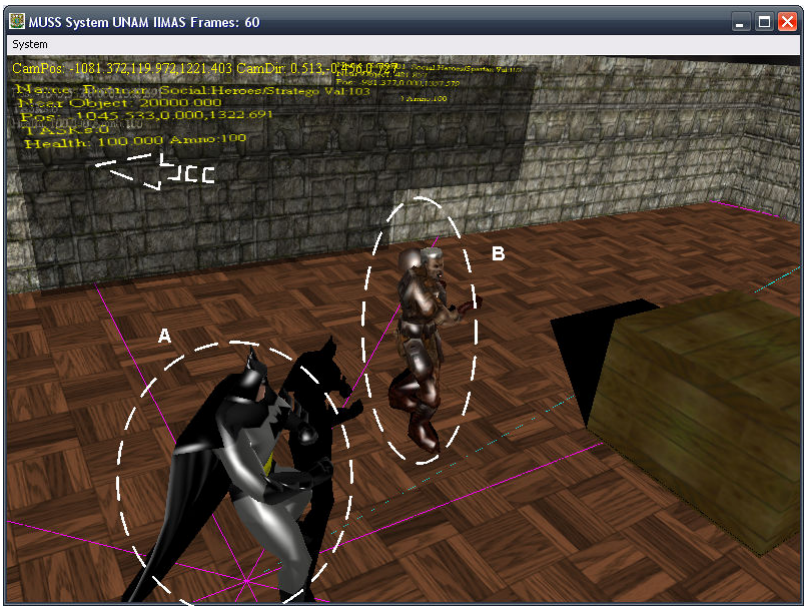


Figura 5.5: *Agente B* no se encuentra visible para el *Agente A*

Objetos detrás de paredes no deben de ser visibles

Se verificará que los agentes sólo puedan percibir objetos que se encuentren dentro de su *FOV* y *LOV* establecidos y además no haya alguna pared que se interponga.

En la *figura 5.6* es posible ver que el *agente A* tiene dentro de su *LOV* (marcado como un octágono) y de su *FOV* (marcado como 2 líneas) al *agente B* pero se encuentra la *pared C* en medio, de tal forma, que no es posible que el *Agente A* pueda ver al *Agente B*. La *figura 5.7* muestra un acercamiento de la misma escena donde es posible ver el *BrainViewer* indicando que no hay ningún agente a la vista.

En la *figura 5.8* es posible ver que el *agente A* tiene dentro de su *LOV* (marcado como un octágono) y de su *FOV* (marcado como 2 líneas) al *agente B* pero se encuentra la *caja C* en medio. La *caja C* no oculta completamente al *Agente B* por lo que es posible que el *Agente A* pueda ver al *Agente B*. La *figura 5.9* muestra un acercamiento de la misma escena donde es posible ver el *BrainViewer* indicando que el *Agente A* puede ver al *Agente B*.

Figura 5.6: *El Agente B* está dentro del *FOV* y *LOV* del *Agente A* y además hay una pared en medio

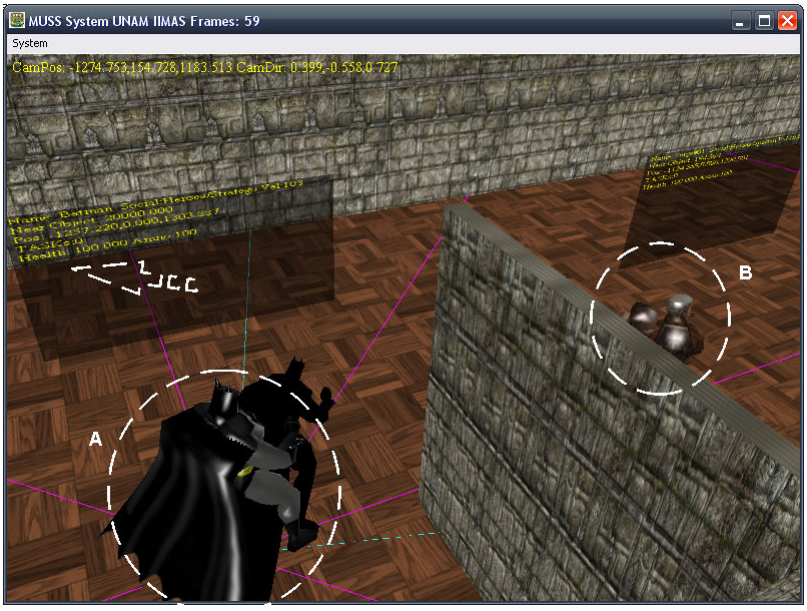


Figura 5.7: *Agente B* no se encuentra visible para el *Agente A*

Figura 5.8: *El Agente B* esta dentro del *LOV* y *FOV* del *Agente A* y ademas hay una caja en medio

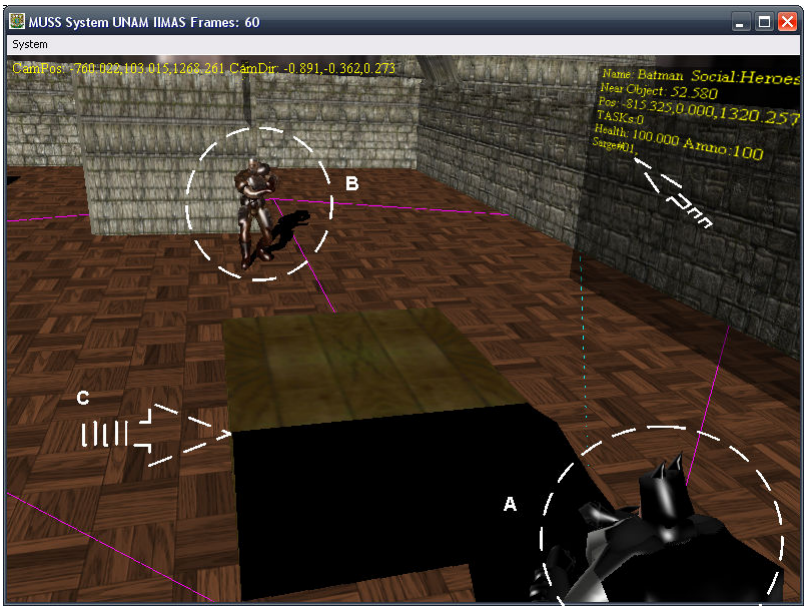


Figura 5.9: *Agente B* se encuentra visible para el *Agente A*

Ejecución por múltiples agentes

Se verificará que el sentido de la vista tenga una implementación que permita su ejecución por múltiples agentes dentro del sistema sin comprometer el rendimiento del apartado gráfico.

Para esta prueba se incluirán veintidós agentes en el sistema y se les reunirá para que se encuentren cercanos unos a otros, ademas habrá en el mundo ciento cincuenta obstáculos estáticos. Para verificar su correcto funcionamiento se registró los cuadros por segundo *FPS* que el sistema presenta durante la ejecución de la implementación de este sentido. La *figura 5.10* muestra la gráfica del desempeño del sistema de visión para veintidós agentes.

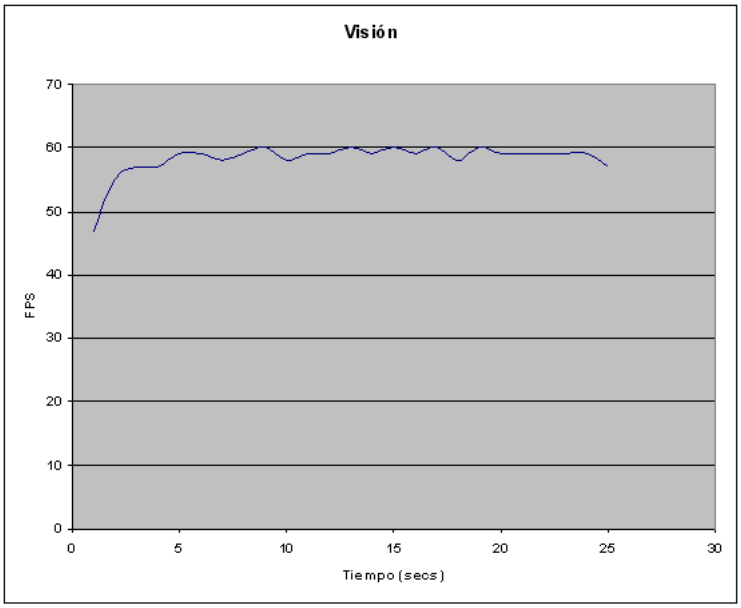


Figura 5.10: Desempeño del sentido de la vista

Como puede observarse en la gráfica, aun cuando al principio existen un bajo conteo de cuadros debido a la inicialización de la aplicación, el sistema de visión es estable ya que durante la ejecución del sistema, los *FPS* se encuentran muy cercanos a 60Hz, el cual es el valor máximo posible debido a las restricción física del monitor donde se realizó la prueba. La *figura 5.11* y la *figura 5.12* muestran la congregación de los veintidós agentes mencionados.

Figura 5.11: Múltiples agentes observándose unos a otros

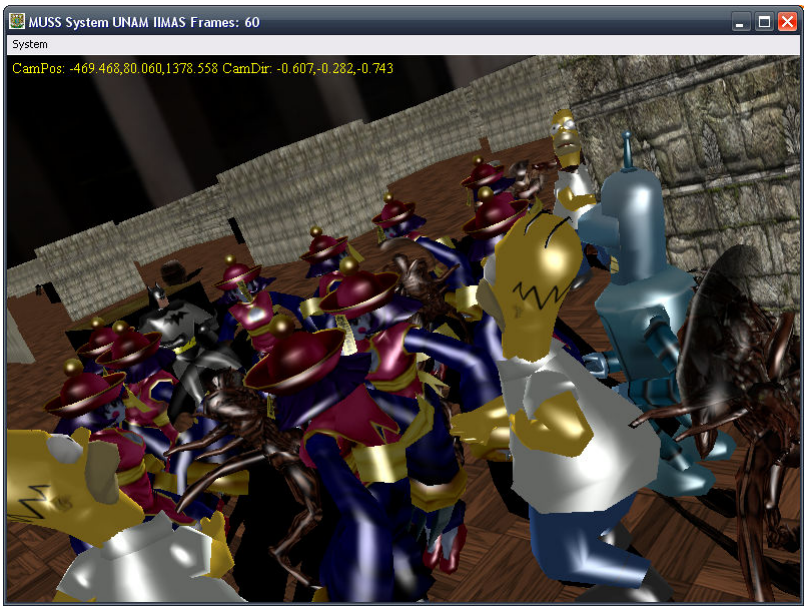


Figura 5.12: Múltiples agentes observándose unos a otros (acercamiento)

5.1.2. Oído

Las pruebas se pueden catalogar en:

1. Detección de agentes sólo dentro del *LOH*.
2. Ejecución por múltiples agentes.

Detección de agentes solo dentro del *LOH*

Se verificará que los agentes sólo puedan percibir objetos que se encuentren dentro de su *LOH* establecidos.

En la *figura 5.13* es posible ver que el *agente A* tiene dentro de su *LOH* (marcado como una esfera) al *agente B*. Para que el *sentido del oído* funcione, es necesario contar con *sonidos virtuales*. Los agentes al caminar producen *sonidos* que se representan como esferas que disminuyen de tamaño con el tiempo.

En la *figura 5.14* el *Agente B* comienza a dirigirse hacia el *Agente A* produciendo con su movimiento *sonidos*. La *figura 5.15* muestra un acercamiento de la misma escena donde es posible ver el *BrainViewer* con una flecha punteada la indicación de que hay un agente detectado por audición.

La *figura 5.16* permite ver que el *Agente B* está fuera del *LOH*. Cuando comienza el movimiento el *BrainViewer* del *Agente A* no reporta al *Agente B* como detectado por audición (*figura 5.17*).

Ejecución por múltiples agentes

Se verificará que el sentido del oído tenga una implementación que permita su ejecución por múltiples agentes dentro del sistema sin comprometer el rendimiento del apartado gráfico.

Para esta prueba se incluirán veintidós agentes en el sistema y se les reunirá para que se encuentren cercanos unos a otros. Una vez reunidos se moverán todos a la vez lo cual ocasionará que los *sonidos virtuales* sean percibidos por todos los agentes involucrados. Para verificar su correcto funcionamiento se registró los cuadros por segundo *FPS* que el sistema presenta durante la ejecución de la implementación de este sentido. La *figura 5.18* muestra la gráfica del desempeño del sistema de audición para veintidós agentes.

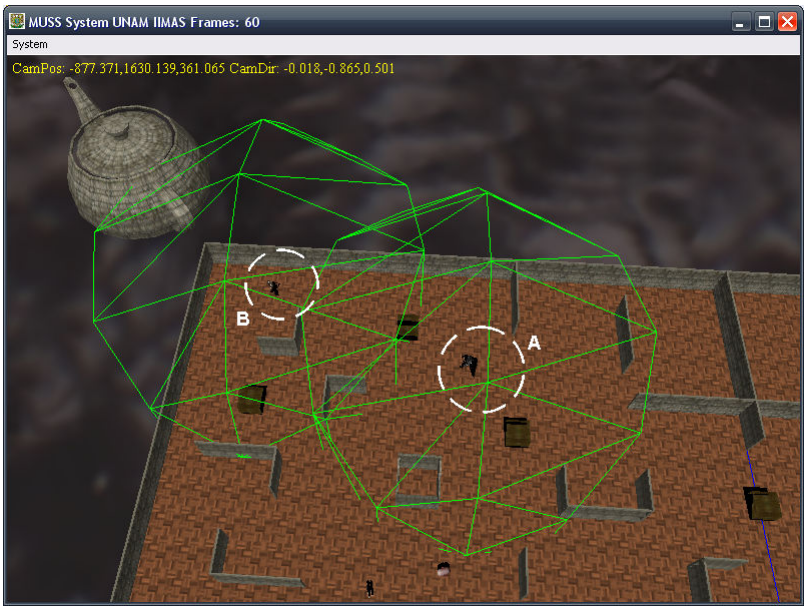


Figura 5.13: El *Agente B* se encuentra dentro del *LOH* del *Agente A*

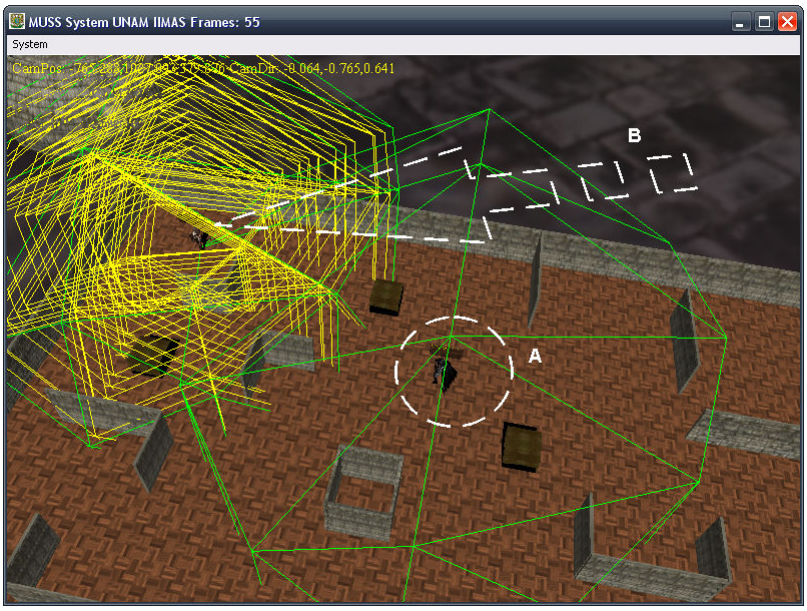


Figura 5.14: Movimiento del *Agente B* hacia el *Agente A*

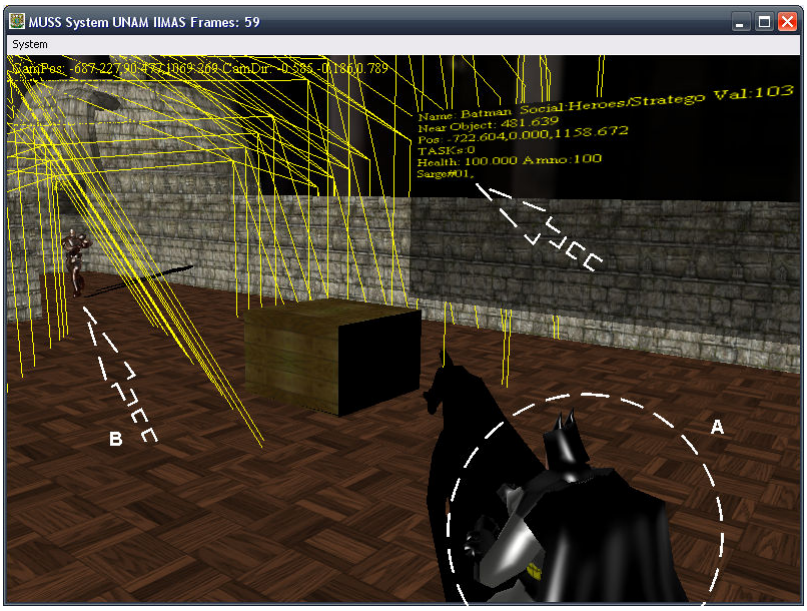


Figura 5.15: *Agente A* detecta por audición al *Agente B*

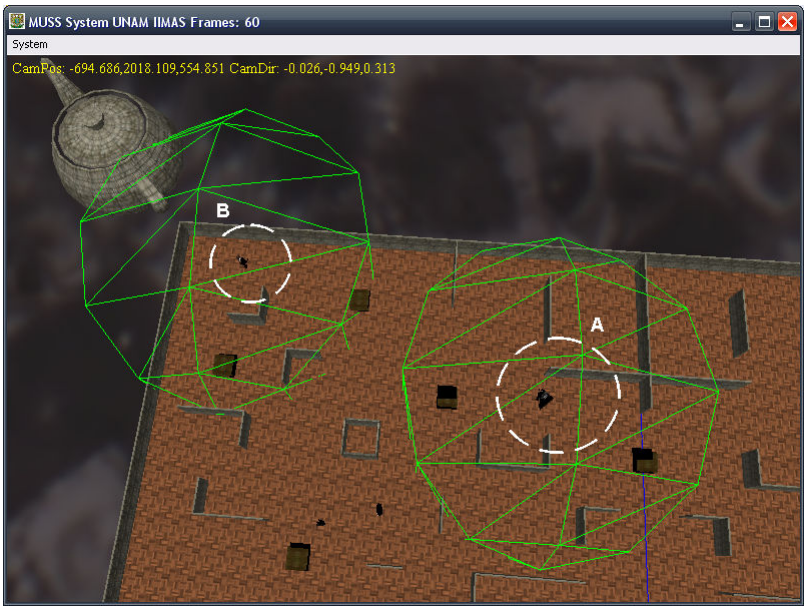


Figura 5.16: El *Agente B* no se encuentra dentro del *LOH* del *Agente A*

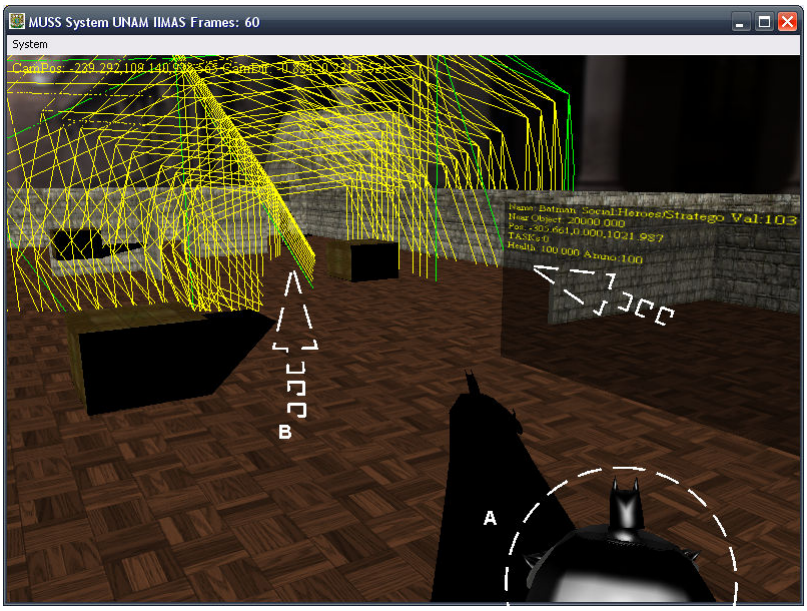


Figura 5.17: *Agente A* no detecta por audición al *Agente B*

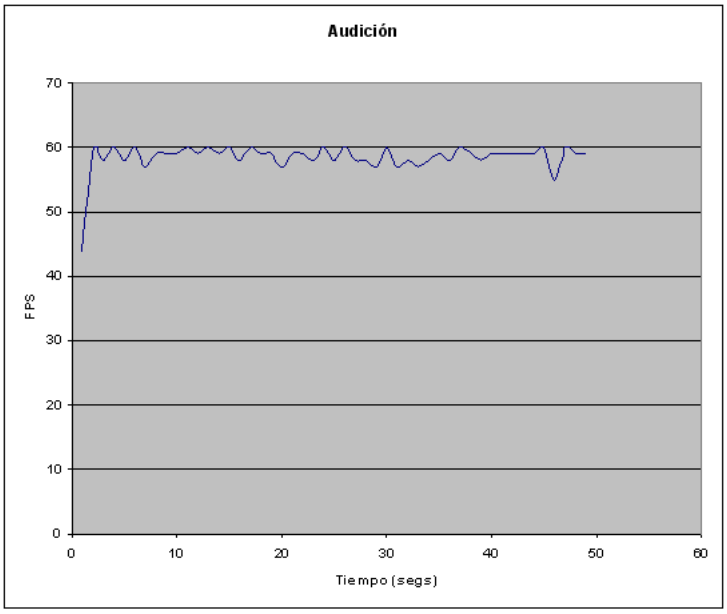


Figura 5.18: Desempeño del sentido del oído

Como puede observarse en la gráfica, aun cuando al principio existen un bajo conteo de cuadros debido a la inicialización de la aplicación, el sistema de audición es estable ya que durante la ejecución del sistema, los *FPS* se encuentran muy cercanos a 60Hz el cual es el valor máximo posible debido a las restricción física del monitor donde se realizó la prueba.

5.1.3. Tacto

Las pruebas de este sentido consistirán en verificar que los agentes no pueden transpasar objetos.

En la *figura 5.19* se muestra al *Agente A* y al *Agente B* en sus posiciones iniciales. Se enviará un comando que solicite al *Agente B* dirigirse hacia el *Agente A*; si bien el planeador evita los obstáculos conocidos, como las paredes, para llevar a un agente de un punto a otro, no toma en cuenta obstáculos desconocidos como las cajas que son usadas para cubrirse. El sentido del tacto evitará que el *Agente B* atraviese cualquier obstáculo desconocido en su ruta hacia el *Agente A*.

Figura 5.19: Posición inicial para prueba de sentido del tacto

La *figura 5.20* se creó con la superposición de siete imágenes que muestran los

movimientos del *Agente B* al dirigirse hacia el *Agente A*. El *Agente B* intenta moverse directamente hacia el punto #6, sin embargo encuentra un obstáculo desconocido en el punto #2 por lo que choca contra él y lo bordea en los puntos #3 y #4. Al final intenta llegar a la misma posición donde se encuentra el *Agente A* pero tiene que detenerse antes porque no es posible atravesarlo (punto #7).

Figura 5.20: Prueba del sentido del tacto en siete pasos

En al siguiente prueba, se verificará que el sentido del tacto tenga una implementación que permita su ejecución por múltiples agentes dentro del sistema sin comprometer el rendimiento del apartado gráfico.

Para esta prueba se incluirán veintidós agentes en el sistema y se les solicitará que se reúnan en un solo punto. Esto ocasionará que sólo uno de ellos logre llegar al punto solicitado y los demás intentarán acercarse ocasionando colisiones de unos agentes con otros. Para verificar su correcto funcionamiento se registró los cuadros por segundo *FPS* que el sistema presenta durante la ejecución de la implementación de este sentido. La *figura 5.21* muestra la gráfica del desempeño del sistema de audición para veintidós agentes.

Como puede observarse en la gráfica, aun cuando al principio existen un bajo conteo de cuadros debido a la inicialización de la aplicación, el sentido del tacto tiene unos cuantos problemas de estabilidad sobretodo cuando existen múltiples objetos cerca. En la

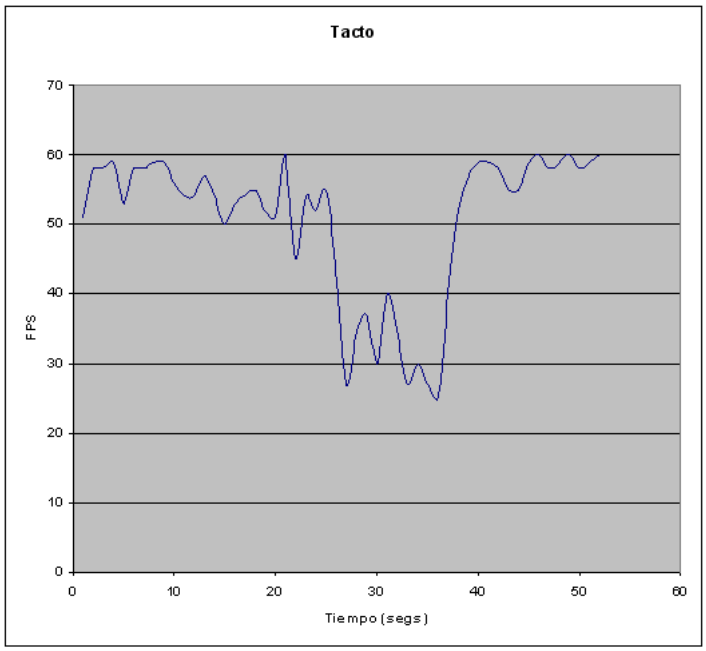


Figura 5.21: Desempeño del sentido del tacto

zona donde se realizó el conteo mas bajo, los valores se encuentran muy cercanos a 30Hz, se recuerda que el valor mínimo considerado para gráficas en tiempo real es 19Hz.

5.1.4. Salud y municiones

Esta prueba es muy simple, consiste en efectuar daño a un agente y verificar que el daño es registrado. De la misma forma se disparará un arma y se verificará que se lleve el conteo de municiones correcto.

En la *figura 5.22* se aprecia el estado inicial del *Agente A* y del *Agente B*. El *Agente A* tiene 100 unidades de salud y el *Agente B* tiene 100 municiones. Las municiones y la salud se muestran contenidas en el *BrainViewer* de cada agente; cada *BrainViewer* se encuentra marcado con un rectángulo punteado y la salud y municiones con una flecha.

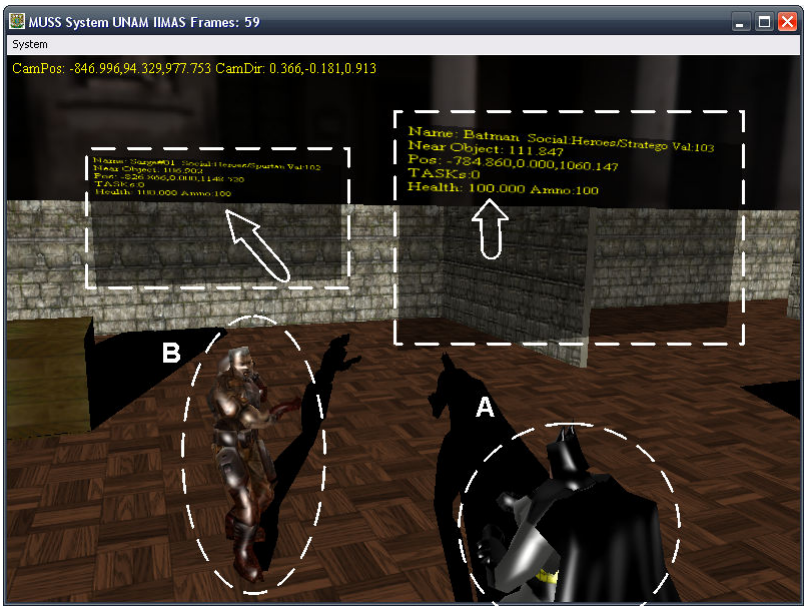


Figura 5.22: Estado inicial para salud y municiones del *Agente A* y *B* respectivamente

En la *figura 5.23* se aprecia como después de un disparo (letra *E*) del *Agente B* hacia el *Agente A* son modificados los valores salud y municiones. Las municiones que tiene el *Agente B* han disminuido a 91 (letra *D*) mientras que la salud del *Agente A* ha disminuido a 93 unidades (letra *C*).

5.1.5. Desempeño del sistema sensorial completo

En las pasadas pruebas se realizaron pruebas de desempeño del los sentidos de forma separada. En esta prueba se medirá el desempeño del uso simultáneo de todos los

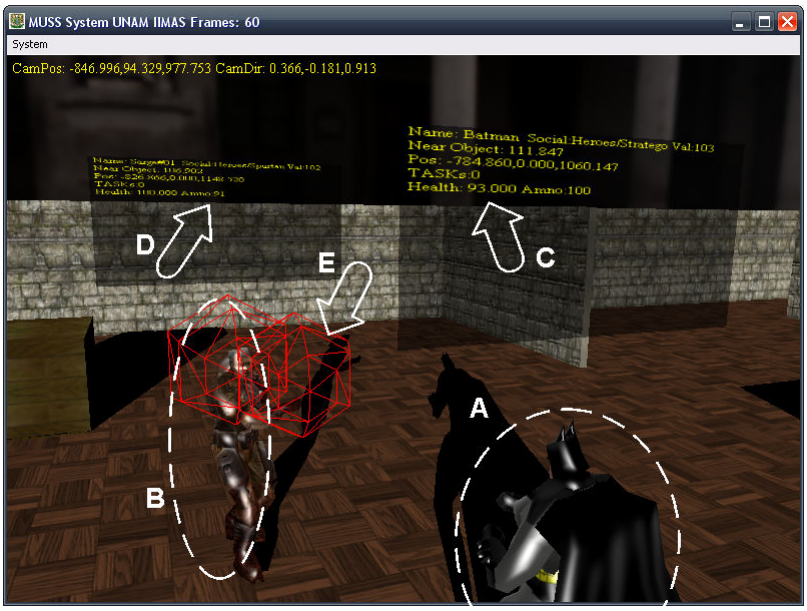


Figura 5.23: Salud del *Agente A* y municiones del *Agente B* después de un ataque

sentidos.

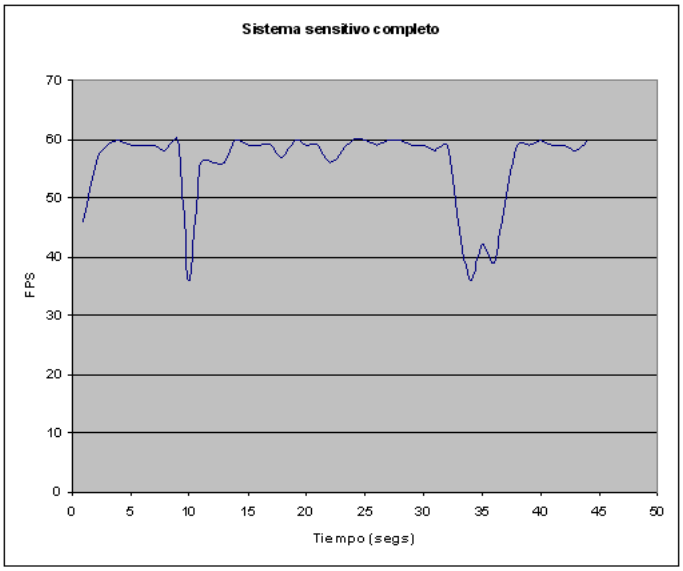


Figura 5.24: Desempeño del sistema sensorial completo

La prueba consistirá en llevar a un punto en común a veintidós agentes, los cuales usarán tanto el sistema sensorial externo como el interno para evitar obstáculos desconocidos, percibir los sonidos que todos hagan al caminar y ver a todos los agentes que se encuentren dentro de sus límites. La *figura 5.24* muestra el desempeño obtenido por el sistema sensorial completo.

Como puede observarse en la gráfica, el sistema en general tiene un comportamiento estable al presentar un *FPS* cercano a 60Hz. En la misma gráfica existen dos puntos en los cuales disminuyó de forma significativa el *FPS*. En estos dos puntos existió una congregación de múltiples agentes en un punto muy cercano. Aun en los puntos donde el *FPS* disminuyó significativamente, el *FPS* fue superior a 30Hz lo cual permite saber que el sistema sensorial puede tener un desempeño bueno dentro del sistema gráfico.

5.2. Comportamientos / Procedimientos

Las pruebas realizadas en esta fase corresponden a verificar el correcto funcionamiento de los siguientes comportamientos:

1. Atacar a un enemigo.
2. Huir.
3. Buscar enemigo.
4. *Path planning*[1]

5.2.1. Atacar a un enemigo

El comportamiento fue generado mediante programación evolutiva con *ADF* usando la situación de prueba descrita en la sección 4.2.6 del capítulo 4, por lo que está prueba consistirá en verificar que el comportamiento de pelea funciona correctamente.

Se colocaron dos agentes en el mundo virtual muy cercanos entre sí y se invocó el comportamiento de pelea para ambos. Cada agente tiene 100 unidades de salud y 100 municiones. Los disparos efectuados por los agentes tienen valores configurables y para esta prueba los disparos de armas de largo alcance disminuyen 5 unidades de salud y los

[1] Planeador de rutas

disparos de armas de corto alcance disminuyen 2 unidades de salud. La *figura 5.25* muestra la posición inicial para la prueba de combate entre dos agentes, el *Agente A* y el *Agente B*; el *Agente A* tiene dentro de su campo de visión al *Agente B*.

Figura 5.25: Posición inicial para prueba de comportamiento de combate entre dos agentes

Una vez que el combate comienza, la secuencia de instrucciones que tiene cada agente les indica como moverse y además cuando disparar. En la *figura 5.26* es posible ver como el *Agente A* esquiva el ataque del *Agente B* y además ataca, aunque sin exactitud al disparar. En la *figura 5.27* ambos agentes han corregido su objetivo en la mira y se atacan frente a frente disparando y esquivando. El combate se desarrolla hasta que ambos agentes mueren, la *figura 5.28* muestra a ambos agentes cayendo muertos.

El combate uno contra uno se puede considerar exitoso, aunque por instantes los agentes comienzan a realizar movimientos que les ponen en situaciones no convenientes como acercarse demasiado al enemigo.

En una segunda prueba se colocaron doce agentes en el mundo virtual muy cercanos entre sí y se invocó el comportamiento de pelea para todos. Al igual que en la prueba anterior, cada agente tiene 100 unidades de salud y 100 municiones. Los disparos de armas de largo alcance disminuyen 5 unidades de salud y los disparos de armas de corto alcance disminuyen 2 unidades de salud. La *figura 5.29* muestra la posición inicial para la prueba de

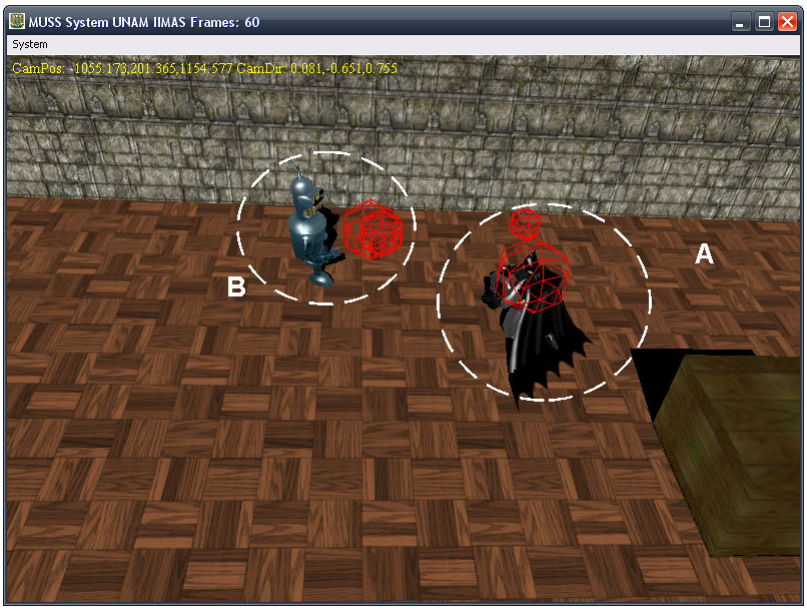


Figura 5.26: *Agente A* esquivando ataque de *Agente B*

Figura 5.27: *Agente A* en combate frente a frente con el *Agente B*

Figura 5.28: *Agente A* y *Agente B* mueren combatiendo

combate.

Los doce agentes están organizados en tres asociaciones diferentes de tal forma que hay tres bandos combatiendo en esta prueba. La primera asociación está formada por los agentes #1, #2, #3 y #4. La segunda asociación la forman los agentes #5, #7, #9 y #12. La tercera y última asociación está formada por los agentes #6, #8, #10 y #11. La *figura 5.30* muestra a los doce agentes combatiendo.

Una vez que ha comenzado el combate, las asociaciones comienzan a atacarse entre sí, lo cual ocasiona que algunos de los agentes caigan en combate. La *figura 5.31* muestra encerrados en círculos a los agentes caídos en combate. Al finalizar el combate algunos agentes sobreviven a la pelea *figura 5.32*.

El combate entre múltiples agentes se considera que funcionó correctamente, sin embargo existen algunos movimientos que pierden coherencia dentro de la secuencia que se ejecuta.

Figura 5.29: Posición inicial de prueba de combate entre doce agentes

Figura 5.30: Combate de prueba entre doce agentes

Figura 5.31: Agentes muertos en el combate de prueba entre los doce agentes

Figura 5.32: Agentes sobrevivientes al combate

5.2.2. Huir

En esta prueba se verificará que el comportamiento *huir* funcione correctamente. Se colocarán varios agentes en el mundo virtual de tal forma que se formen dos *asociaciones* antagonistas.

La *figura 5.33* muestra la configuración mencionada. La primera *asociación* sólo esta formada por el agente #1, la segunda *asociación* esta formada por los agentes #2, #3, #4 y #5.

La *figura 5.34* muestra la superposición de tres momentos diferentes durante la ejecución del comportamiento *huir*. Como puede observarse, los agentes se alejan de sus enemigos mas cercanos en dirección de las flechas marcadas en la figura. El agente #5 no se aleja de su enemigo el agente #1, debido a que a espaldas suyas se encuentra una caja que evita el movimiento, aun así se desliza por la orilla de la caja buscando alejarse.

Dados los resultados anteriores, se considera que el comportamiento funciona correctamente.

Figura 5.33: Configuración inicial para prueba *Huir*

Figura 5.34: Dirección de desplazamiento durante la prueba

5.2.3. Buscar enemigo

La prueba para este comportamiento es simple. Se colocará un agente en el mundo virtual y se le asignará el comportamiento de *buscar enemigo*, en dicho momento el agente deberá desplazarse por el mundo virtual buscando algún agente enemigo. La *figura 5.35* y la *figura 5.36* muestra la realización de esta prueba en dos ocasiones diferentes.

La prueba #1 muestra el rastreo que hace el agente durante el proceso de búsqueda donde cada número indica el proceso de la búsqueda desde su posición inicial (paso #1) hasta el último paso (paso #12). De la misma forma, la prueba #2 muestra la ejecución de la búsqueda en otra sección del nivel desde el paso #1 al paso #17. Como puede observarse, durante la búsqueda, se realizan movimientos que alteran la dirección de observación del agente permitiéndole cubrir cada vez un área más grande del *nivel* además de que al mismo tiempo sale de áreas encerradas.

Debido a los resultados mostrados se puede decir que el procedimiento de búsqueda si bien no es eficiente debido a su naturaleza aleatoria de control de movimiento, si permite encontrar eventualmente a un agente enemigo dentro del mundo virtual.

Figura 5.35: Prueba #1 *Buscar enemigo*

Figura 5.36: Prueba #2 *Buscar enemigo*

5.2.4. *Path planning*

Si bien el *path planning* no es un *comportamiento* como tal desde el punto de vista del *MUSS*, se incorporará en esta sección.

La prueba consistirá en colocar en el mundo virtual a veintidós agentes organizados en cuatro *asociaciones* diferentes. Cada asociación se encontrará en una esquina diferente del *nivel*. Después se les solicitará a todos llegar a un mismo punto del *nivel* y para ello necesitará cada agente invocar a su propio planeador de rutas. La *figura 5.37* ilustra las posiciones de las cuatro brigadas (*A,B,C y D*) y el punto destino (*E*).

La *figura 5.38* está formada por la superposición de diez momentos diferentes de la ejecución de la prueba; si bien es difícil seguir a cada agente en la imagen, se ilustra el movimiento (marcado con flechas) de cada asociación al punto destino.

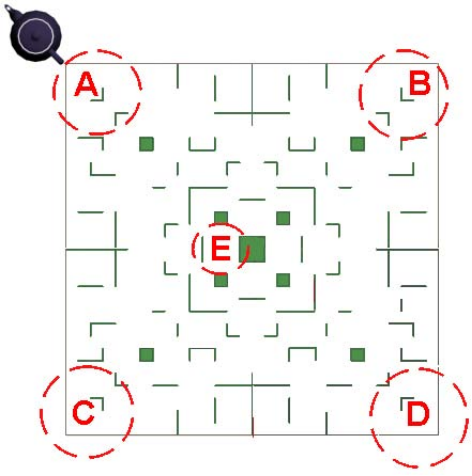


Figura 5.37: Posiciones iniciales de las asociaciones de agentes para la prueba del *Path Planning*

Como puede observarse en la *figura 5.39*, los cálculos concernientes al *path planning* no constituyen una carga pesada al sistema en general ya que aun se tienen valores cercanos a 60Hz. Las disminuciones de *FPS* durante la ejecución se deben principalmente al uso del *sistema sensorial* como se analizó en las pruebas anteriores.

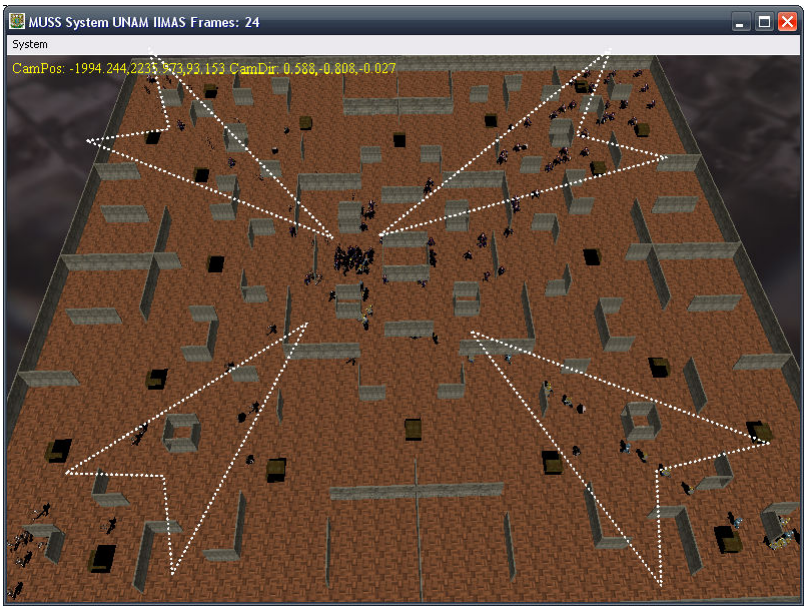


Figura 5.38: Movimiento de las asociaciones hacia el punto destino

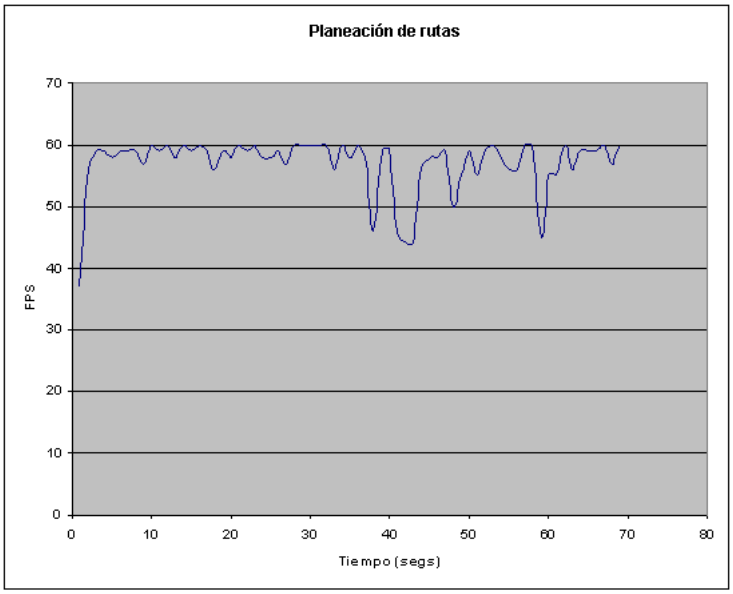


Figura 5.39: Desempeño de la planeación de rutas

5.3. Razonamiento

En estas pruebas se verificará que la secuencia de acciones y comportamientos generados por el módulo de razonamiento sean correctos. Para este fin se crearon ocho bancos de pruebas en donde se proporcionarán estímulos sensoriales y se obtendrán las acciones que se deberán efectuar. Los primeros cuatro bancos de pruebas tienen como objetivo verificar la respuesta de los cuatro objetivos generales y los cuatro posteriores verificarán el comportamiento en conjunto.

El razonamiento se encuentra implementado haciendo uso de *CLIPS*. El programa hecho en *CLIPS* necesita que se le provea de hechos que son extraídos del mundo por el sistema sensorial. El cuadro 5.1 muestra los hechos que se utilizarán dentro de la prueba.

Hecho	**Hecho**	**Hecho**	**Hecho**
Enemigo a la vista	Municiones suficientes	Llamada de ayuda	Cubierta cercana
Salud suficiente	Enemigo cercano	Equipo cercano	Peligro

Cuadro 5.1: Hechos utilizados para prueba de razonamiento

5.3.1. Sobrevivir

Los hechos utilizados en esta prueba y sus resultados se muestran en los cuadros 5.2, 5.3, 5.4 y 5.5. En este apartado es importante que el agente se encuentre en alguna situación adversa para comprobar que le es conveniente huir en ese instante. El código de *CLIPS* correspondiente a esta sección del razonamiento se muestra en *Código Fuente A.1*.

Hechos	Razonamiento por **CLIPS**
Enemigo a la vista	REPORT ENEMY ON SIGHT
NO Municiones suficientes	REQUEST HELP AT POS 01
NO Salud suficiente	GOAL: Flee
Peligro	

Cuadro 5.2: Hechos utilizados para banco de pruebas *sobrevivir* #1

Hechos	Razonamiento por **CLIPS**
Peligro	REPORT ENEMY ON SIGHT
NO Municiones suficientes	REQUEST HELP AT POS 01
NO Salud suficiente	GOAL: Flee

Cuadro 5.3: Hechos utilizados para banco de pruebas *sobrevivir* #2

Hechos	Razonamiento por **CLIPS**
Peligro	REQUEST HELP AT POS 02
NO Municiones suficientes	GOAL: Flee
Salud suficiente	

Cuadro 5.4: Hechos utilizados para banco de pruebas *sobrevivir* #3

Hechos	Razonamiento por **CLIPS**
Salud suficiente	REQUEST FRIEND POS 1
NO Municiones suficientes	GOAL: Seek-friend

Cuadro 5.5: Hechos utilizados para banco de pruebas *sobrevivir* #4

Código Fuente 5.1: Objetivo *Sobrevivir*

```
1  ;;;-----------------------------------------------
2  ;;;--SURVIVE----
3  ;;;-----------------------------------------------
4
5  ;;;**********
6  ;;;***FLEE***
7  ;;;**********
8  (defrule survive-rule-flee
9        (goal (action survive))
10    ?f<-(action flee)
11 =>
12    (printout t "GOAL: Flee" crlf)
13    (retract ?f)
14 )
15
16 (defrule survive-rule-seek-friend
17    (goal (action survive))
18    ?f<-(action seek-friend)
19 =>
20    (printout t "GOAL: Seek-friend" crlf)
21    (retract ?f)
22 )
23
24 ;;;****************
```

```
25 ;;;***FLEE RULES***
26 ;;;****************
27 (defrule Flee-rule-01
28    (declare (salience 4010))
29    (fact-danger)
30    (not (fact-health-ok))
31    (not (fact-team-near))
32    (not (layer-activated))
33 =>
34    (assert (action flee))
35    (assert (layer-activated))
36    (printout t "REQUEST HELP AT POS 01" crlf)
37 )
38
39 (defrule Flee-rule-02
40    (declare (salience 4010))
41    (fact-danger)
42    (not (fact-amno-ok))
43    (not (fact-team-near))
44    (not (layer-activated))
45 =>
46    (assert (action flee))
47    (assert (layer-activated))
48    (printout t "REQUEST HELP AT POS 02" crlf)
49 )
50
51 (defrule Flee-rule-03
52    (declare (salience 4020))
53    (not (fact-danger))
54    (not (fact-amno-ok))
55    (not (layer-activated))
56 =>
57    (assert (action seek-friend))
58    (assert (layer-activated))
59    (printout t "REQUEST FRIEND POS 1" crlf)
60 )
61
62 (defrule Flee-rule-04
63    (declare (salience 4021))
64    (not (fact-danger))
65    (not (fact-health-ok))
66    (not (layer-activated))
67 =>
68    (assert (action seek-friend))
69    (assert (layer-activated))
70    (printout t "REQUEST FRIEND POS 2" crlf)
71 )
```

De los cuadros mostrados es posible ver que ante una situación adversa, como no tener salud suficiente, municiones suficientes, percibir un peligro o ver algún enemigo a la vista, la reacción del agente es, si hay un enemigo a la vista, reportar enemigo, después

solicitar que alguien del equipo le ayude en la posición en la que se encuentra y finalmente huir. Cuando el agente no está en peligro pero tampoco está en condiciones de pelear, busca la ayuda de un amigo (cuadro 5.5). En ningún momento el agente intentó atacar al enemigo, ni buscar con quien combatir debido a que no estaba en una situación favorable, por lo que este razonamiento funciona correctamente.

5.3.2. Atacar enemigo

Los hechos utilizados en esta prueba y sus resultados se muestran en los cuadros 5.6, 5.7, 5.8 y 5.9. En este apartado es importante que el agente se encuentre en condiciones favorables para combatir, es decir, con municiones y salud, para comprobar que le es conveniente atacar. El código de *CLIPS* correspondiente a esta sección del razonamiento se muestra en *Código Fuente 5.2*.

Hechos	Razonamiento por **CLIPS**
Enemigo a la vista	REPORT ENEMY ON SIGHT
Municiones suficientes	GOAL: ATTACK FAR
Salud suficiente	
Peligro	

Cuadro 5.6: Hechos utilizados para banco de pruebas *atacar enemigo* #1

Hechos	Razonamiento por **CLIPS**
Enemigo a la vista	REPORT ENEMY ON SIGHT
Municiones suficientes	GOAL: ATTACK NEAR
Salud suficiente	
Enemigo cerca	
Peligro	

Cuadro 5.7: Hechos utilizados para banco de pruebas *atacar enemigo* #2

Hechos	Razonamiento por **CLIPS**
Enemigo a la vista	REPORT ENEMY ON SIGHT
Municiones suficientes	GOAL: ATTACK NEAR
Salud suficiente	
Enemigo cerca	
Cubierta cercana	
Peligro	

Cuadro 5.8: Hechos utilizados para banco de pruebas *atacar enemigo* #3

Hechos	Razonamiento por **CLIPS**
Enemigo a la vista	REPORT ENEMY ON SIGHT
Municiones suficientes	GOAL: ATTACK COVER
Salud suficiente	
Cubierta cercana	
Peligro	

Cuadro 5.9: Hechos utilizados para banco de pruebas *atacar enemigo* #4

Hechos	Razonamiento por **CLIPS**
Enemigo a la vista	REPORT ENEMY ON SIGHT
NO Municiones suficientes	GOAL: ATTACK NEAR
Salud suficiente	
Enemigo cerca	
Peligro	

Cuadro 5.10: Hechos utilizados para banco de pruebas *atacar enemigo* #5

Código Fuente 5.2: Objetivo *Atacar enemigo*

```
72 ;;;-----------------------------------------------------
73 ;;;---Attack----
74 ;;;-----------------------------------------------------
75
76
77 ;;;************
78 ;;;***Attack***
79 ;;;************
80 (defrule attack-rule-attack-near
81       (goal (action attack))
82    ?f<-(action attack-near)
83 =>
```

```
84      (printout t "GOAL: ATTACK_NEAR" crlf)
85      (retract ?f)
86  )
87
88  (defrule attack-rule-attack-cover
89          (goal (action attack))
90      ?f<-(action attack-cover)
91  =>
92      (printout t "GOAL: ATTACK_COVER" crlf)
93      (retract ?f)
94  )
95
96  (defrule attack-rule-attack-far
97          (goal (action attack))
98      ?f<-(action attack-far)
99  =>
100     (printout t "GOAL: ATTACK_FAR" crlf)
101     (retract ?f)
102 )
103
104 ;;;******************
105 ;;;*** Attack RULES***
106 ;;;******************
107 (defrule Attack-rule-near
108     (declare (salience 3030))
109     (fact-enemy-near)
110     (fact-enemy-on-sight)
111     (not (attack-method-decided))
112     (not (layer-activated))
113 =>
114     (assert (action attack-near))
115     (assert (attack-method-decided))
116     (assert (layer-activated))
117 )
118
119 (defrule Attack-rule-from-cover
120     (declare (salience 3020))
121     (fact-enemy-on-sight)
122     (fact-cover)
123     (fact-amno-ok)
124     (not (attack-method-decided))
125     (not (layer-activated))
126 =>
127     (assert (action attack-cover))
128     (assert (attack-method-decided))
129     (assert (layer-activated))
130 )
131
132 (defrule Attack-rule-far
133     (declare (salience 3010))
134     (fact-enemy-on-sight)
135     (fact-amno-ok)
```

```
136     (not (attack-method-decided))
137     (not (layer-activated))
138 =>
139     (assert (action attack-far))
140     (assert (attack-method-decided))
141     (assert (layer-activated))
142 )
```

De los cuadros mostrados es posible ver que ante una situación favorable para el combate (salud y municiones suficientes), al ver a un enemigo, la reacción del agente es reportar enemigo a la vista y después atacar de tres diferentes maneras, una más favorable que otra según las circunstancias. Por ejemplo, al tener un enemigo cercano prefiere atacarlo de cerca que salir buscando un lugar donde cubrirse o disparar desde lejos (cuadros 5.7, 5.8 y 5.10). Cuando el enemigo no está cerca y hay donde cubrirse, prefiere atacarlo desde la cubierta que atacarlo al descubierto (cuadro 5.9). Por último, si no hay una circunstancia a la cual tomarle ventaja (estar cerca del enemigo o atacarle desde cubierto) decide atacarle desde lejos. En ningún momento el agente intentó salir huyendo, tampoco ignorar al enemigo por lo que este razonamiento funciona correctamente.

5.3.3. Apoyar amigo

Los hechos utilizados en esta prueba y sus resultados se muestran en los cuadros 5.11 y 5.12. En este apartado es importante que el agente tenga un llamado de ayuda registrado, no haber enemigos cerca y estar en condición de ayudar, es decir, con municiones y salud. El código de *CLIPS* correspondiente a esta sección del razonamiento se muestra en *Código Fuente 5.3*.

Hechos	Razonamiento por **CLIPS**
Llamada de ayuda	GOAL: Backup
Municiones suficientes	
Salud suficiente	

Cuadro 5.11: Hechos utilizados para banco de pruebas *apoyar amigo* #1

Hechos	Razonamiento por **CLIPS**
Llamada de ayuda	GOAL: Backup
Municiones suficientes	
Salud suficiente	
Equipo cercano	
Cubierta cercana	

Cuadro 5.12: Hechos utilizados para banco de pruebas *apoyar amigo* #2

Código Fuente 5.3: Objetivo *Apoyar amigo*

```
143 ;;;-----------------------------------------------
144 ;;;---Backup----
145 ;;;-----------------------------------------------
146
147 ;;;************
148 ;;;***BACKUP***
149 ;;;************
150 (defrule backup-rule-help-call
151         (goal (action backup))
152    ?f<-(action help-call)
153 =>
154    (printout t "GOAL: Backup" crlf)
155    (retract ?f)
156 )
157
158 ;;;******************
159 ;;;***BACKUP RULES***
160 ;;;******************
161 (defrule backup-rule-01
162    (declare (salience 2010))
163    (not (fact-danger))
164    (fact-help-call)
165    (fact-amno-ok)
166    (not (layer-activated))
167 =>
168    (assert (action help-call))
169    (assert (layer-activated))
170 )
```

De los cuadros mostrados es posible ver que ante una situación favorable para ayudar a alguien (salud y municiones suficientes) y no haber enemigo a la vista, la reacción del agente al recibir una petición de ayuda, es ayudarle a un compañero de su misma *asociación*. En ningún momento el agente intentó salir huyendo, ni pelear con algún enemigo, ni tampoco ignorar la petición de ayuda de un amigo, por lo que este razonamiento funciona correctamente.

5.3.4. Buscar enemigo

Los hechos utilizados en esta prueba y sus resultados se muestran en los cuadros 5.13 y 5.14. En este apartado es importante que el agente esté en buenas condiciones, es decir, con municiones y salud suficientes y sin algún enemigo a la vista. El código de *CLIPS* correspondiente a esta sección del razonamiento se muestra en *Código Fuente 5.4*.

Hechos	Razonamiento por **CLIPS**
Salud suficiente	GOAL: Seek Enemy
Municiones suficientes	

Cuadro 5.13: Hechos utilizados para banco de pruebas *buscar enemigo* #1

Hechos	Razonamiento por **CLIPS**
Salud suficiente	GOAL: Seek Enemy
Municiones suficientes	
Peligro	

Cuadro 5.14: Hechos utilizados para banco de pruebas *buscar enemigo* #2

Código Fuente 5.4: Objetivo *Buscar enemigo*

```
171 ;;;------------------------------------------------
172 ;;;---Find----
173 ;;;------------------------------------------------
174
175 ;;;**********
176 ;;;***Find***
177 ;;;**********
178 (defrule find-rule-seek-enemy
179       (goal (action find))
180    ?f<-(action seek-enemy)
181 =>
182    (printout t "GOAL:  Seek Enemy" crlf)
183    (retract ?f)
184 )
185
186 ;;;******************
187 ;;;***FIND RULES***
188 ;;;******************
189 (defrule find-rule-01
190    (declare (salience 1010))
191    (not (fact-enemy-on-sight))
192    (not (layer-activated))
193 =>
194    (assert (action seek-enemy))
```

```
195     (assert (layer-activated))
196 )
```

De los cuadros mostrados es posible ver que ante una situación en la cual no hay algún enemigo con quien combatir, ni tampoco hay solicitudes de ayuda, pero hay suficientes municiones y salud, la reacción del agente es buscar algún enemigo. En ningún momento el agente realizó algún comportamiento extraño como intentar salir huyendo o buscar algún amigo, por lo que este razonamiento funciona correctamente.

5.3.5. Situaciones concurrentes

En esta prueba se busca tener diferentes situaciones en las cuales se cumplen mas de una meta pero sólo debe de activarse la que tiene mayor prioridad. Las configuraciones de hechos se muestran en los cuadros 5.15 , 5.16 y 5.17.

Hechos	Razonamiento por **CLIPS**
Salud suficiente	REPORT ENEMY ON SIGHT
Municiones suficientes	GOAL: ATTACK NEAR
Enemigo a la vista	
Enemigo cerca	
Cubierta cercana	
Peligro	
Equipo cercano	
Llamada de ayuda	

Cuadro 5.15: Hechos utilizados para banco de pruebas *situaciones concurrentes* #1

Hechos	Razonamiento por **CLIPS**
Salud suficiente	GOAL: Seek Enemy
Municiones suficientes	
Cubierta cercana	
Peligro	
Equipo cercano	
Llamada de ayuda	

Cuadro 5.16: Hechos utilizados para banco de pruebas *situaciones concurrentes* #2

Hechos	Razonamiento por **CLIPS**
Salud suficiente	GOAL: Backup
Municiones suficientes	
Cubierta cercana	
Equipo cercano	
Llamada de ayuda	

Cuadro 5.17: Hechos utilizados para banco de pruebas *situaciones concurrentes* #3

En todas las situaciones de prueba, los hechos pueden activar mas de un objetivo a la vez; sin embargo, debido a la estructura de capas que tiene el sistema de razonamiento sólo una de ellas es activada: la capa superior. En el cuadro 5.15 se colocaron todos los hechos posibles registrados y de todos los comportamientos el elegido es atacar desde una cubierta, lo cual es correcto. En el cuadro 5.16 existe posibilidad de ayudar a alguien, pero hay peligro por lo que se decide buscar al enemigo, lo cual es correcto. Si en la situación actual existe la posibilidad de ayudar a alguien y no hay peligros cercanos se decide atender la petición de ayuda (cuadro 5.17).

Dadas las situaciones planteadas a través de los bancos de pruebas descritos se puede decir que la planeación de comportamientos funciona correctamente.

5.4. Comportamiento general

Una vez habiendo realizado pruebas por separado de los módulos más importantes del sistema, en esta prueba se evaluarán la interacción de todos módulos en conjunto mediante la ejecución de una situación de prueba.

La situación de prueba consiste en crear cuatro *asociaciones* de agentes en puntos separados del *nivel* y permitirles ejecutar su módulo de razonamiento. El total de los agentes es veintidós. Todos los agentes comenzarán con 100 unidades de salud y con la misma configuración de arma con 100 municiones. La *figura 5.40* muestra la configuración inicial de la prueba.

En un principio todos los agentes comenzaron a buscar a un enemigo dentro del nivel, después de unos cuantos segundos algunos de los agentes alcanzaron a percibir que habían otros agentes cercanos, sin embargo, tuvo que pasar un poco mas de tiempo hasta que uno de ellos estableció contacto visual. En el momento en el que se estableció contacto

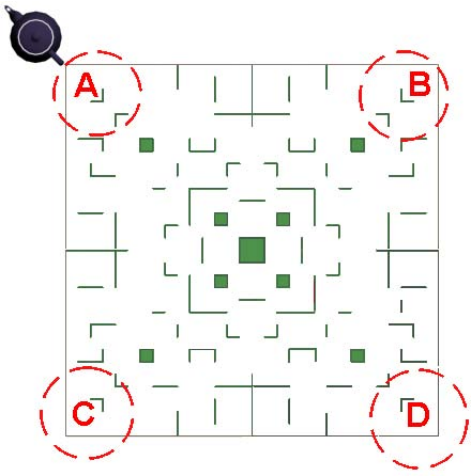


Figura 5.40: Posición inicial de la situación de prueba general

visual, se levantó un aviso a toda su *asociación*, ocasionando que todos se dirigieran al punto en cuestión. Durante el trayecto hacia el punto algunos de los compañeros fueron vistos por otros agentes generándose mas reportes y congregándose dos *asociaciones* en una zona del *nivel*.

Al establecer contacto visual con enemigos, comenzó el combate. Los agentes que se encontraban más cerca del centro, murieron debido al ataque de tanto agentes enemigos como de disparos de agentes de su propia *asociación*. Antes de morir los agentes siguieron combatiendo sin intentar huir, esto fue un efecto no esperado ya que lo deseable sería que al estar con la posibilidad de morir, se alejaran de la zona de combate; sin embargo continuaron hasta que el último quedo solo e intento huir. En general, los últimos agentes sobrevivientes permanecieron buscándose entre si largo tiempo hasta que sólo un agente quedó en pie.

Otra situación que se presentó fue que durante la pelea algunos agentes se separaron de sus compañeros y al verse con poca salud intentaron huir, pero la maniobra no fue del todo efectiva ya que seguían los disparos. La situación de encontrar a un enemigo, llamar a compañeros reportando enemigo a la vista y después pelear se repitió constantemente entre cada encuentro.

El comportamiento agresivo cambió con los agentes sobrevivientes debido a que la cantidad de municiones restante era poca, por lo que comenzaban a buscar apoyo de otros compañeros. Algunos de ellos eran los últimos de su asociación por lo que seguían buscando.

El combate lejano, si bien les permite esquivar disparos y vencer al enemigo, ocasionaba en gasto excesivo de municiones por lo que se terminaban rápidamente. Además de que debido a la secuencia predefinida de movimientos, a veces los enemigos salían de la mira y comenzaban a buscar sin encontrar al mismo agente con el que se encontraban peleando.

Durante su recorrido por el nivel algunos agentes pudieron encontrar objetos para recuperar su salud y también municiones. Si dichos agentes estaban con cantidades bajas de los mismos al tomar los objetos recuperaban su conducta agresiva.

Repitiendo el experimento en mas de una ocasión se obtuvieron resultados similares. Las principales diferencias consistían en el tiempo para encontrar un enemigo, lugar del encuentro y como se dividían las *asociaciones* ante reportes separados de combates en diferentes partes del nivel.

Dados los resultados obtenidos en esta prueba, el sistema se comportó correctamente, permitiendo a los agentes colaborar entre sí para vencer a los demás agentes en combate.

Conclusiones y trabajo futuro

En esta tesis se ha planteado una arquitectura para sistemas multiagentes colaborativos, haciendo énfasis en su incorporación dentro de un sistema con despliegue gráfico tridimensional.

Por medio de las clases especificadas y construidas, las consideraciones de los diversos módulos que forman a cada agente, así como la interacción entre los mismos, se concluye que el objetivo de crear una arquitectura para sistemas multiagentes colaborativos ha sido cumplida. Esto se logró considerando la arquitectura propuesta por Robert Burke [BR01] y por Jeff Orkin [Ork05].

Dentro de la implementación realizada sólo se utilizaron tres librerías desarrolladas por otras personas: *Kunzip.dll* para descompactar archivos en formato *zip*. *CLIPS* para la sección de razonamiento y finalmente en el apartado gráfico se tiene *DirectX*. Por lo que esta arquitectura puede ser llevada a otras plataformas, como sistemas basados en *Unix*, al sustituir los módulos que hacen uso de las librerías mencionadas por sus equivalentes.

Debido a la naturaleza *multihilos* y *microthreads* del sistema, se pudieron tener buenos resultados de rendimiento en su utilización dentro del apartado gráfico, como fue mostrado en las pruebas. Esto permite incrementar el tiempo de procesador que el apartado gráfico utiliza o aumentar el número de módulos en el *MAS*.

Otra característica es que la implementación se realizó usando lenguaje *C/C++*

por lo que incorporarse en otro sistema gráfico distinto al creado en este trabajo es posible, debido a que es el lenguaje más popular para realizar aplicaciones gráficas por su buen desempeño.

Utilizar *LUA* como lenguaje de *scripting* permite tener configuraciones flexibles de los parámetros del programa sin necesidad de recompilar código.

En cuestión de resultados entregados existe aun trabajo por hacer. En el capítulo 5 se han mostrado las evaluaciones realizadas al sistema. En ellas se observa que, a pesar de obtener buenos resultados, aun es necesario continuar con el desarrollo y actualización de algunos de los módulos diseñados. Si bien sólo se plantean unas cuantas acciones y comportamientos, se tiene la posibilidad de agregar más y con ello enriquecer el repertorio disponible.

La creación de comportamientos usando programación genética obtuvo resultados aceptables, sin embargo, una forma distinta de evaluar a los individuos usando el sistema desarrollado dentro de la función de evaluación permitirá crear nuevos comportamientos que obtengan mejores resultados. El procedimiento de búsqueda, si bien es funcional, es poco eficiente, por lo que una implementación del mismo usando programación genética podría lograr mejores resultados.

Cómo contribuciones del trabajo de tesis, considero que están:

- La creación de una biblioteca para animación de personas en formato *Md3* tanto en versión *OpenGL* como en versión *DirectX*.
- Una clase que permite incluir *CLIPS* de manera embebida dentro de cualquier aplicación hecha en *C++*.
- Una clase que permite utilizar A^* para planear rutas con miles de nodos que usa pocos recursos computacionales.
- Una implementación de *ADF* y un convertidor a lenguaje intermedio de las mismas.
- Las clases utilizadas para la incorporación de la arquitectura en otros sistemas gráficos.

En resumen, de acuerdo con los resultados obtenidos y el trabajo realizado, se concluye que se cumplieron los objetivos aunque es necesario trabajar con mayor énfasis para mejorar el desempeño y la flexibilidad de los módulos utilizados. Es de aquí de donde se parte para determinar el trabajo futuro.

1. Respecto a la *UCS*, dentro de su diseño se consideró la posibilidad de poder cargar hasta tres programas adicionales al principal, por lo que es posible crear el mecanismo dentro de la arquitectura que le permita intercambiar secciones del programa entre agentes buscando resolver algún problema, abriendo con ésto una gama completa de posibilidades.
2. Incorporar nuevas técnicas para desarrollar comportamientos
3. Incrementar el sistema sensorial al agregar el olfato
4. Cambiar el módulo de razonamiento de *CLIPS* a A^*
5. Utilizar modelos en un formato de animación por huesos de tal forma que sea posible controlar mejor la animación de los *avatares* de los agentes.

Finalmente, para darle mayor continuidad, habría que vincularlo con alguna tarea real de robótica, utilizando o completando los módulos requeridos.

Apéndice A

Sistema de razonamiento en *CLIPS*

En el capítulo 4 se indicó que cada agente utilizaría un *script* programado en *CLIPS* pero no se proporcionó el programa. El siguiente bloque de código corresponde al *script* que todos los agentes utilizan:

Código Fuente A.1: Objetivo *Sobrevivir*

```
1  ;;;**************************
2  ;;;* AGENT LIVE PLAN Ver 2.0*
3  ;;;**************************
4
5  ;;;***GOAL***
6  (deftemplate goal
7     (slot action
8     (allowed-symbols attack find survive backup)
9           (default ?NONE)
10    )
11 )
12 ;;;-----------------------------------------------
13 ;;;--SURVIVE-----
14 ;;;-----------------------------------------------
15 ;;;***********
```

```
16 ;;;***FLEE***
17 ;;;**********
18 (defrule survive-rule-flee
19       (goal (action survive))
20    ?f<-(action flee)
21 =>
22    (printout t "GOAL: Flee" crlf)
23    (retract ?f)
24 )
25
26 (defrule survive-rule-seek-friend
27    (goal (action survive))
28    ?f<-(action seek-friend)
29 =>
30    (printout t "GOAL: Seek-friend" crlf)
31    (retract ?f)
32 )
33 ;;;*****************
34 ;;;***FLEE RULES***
35 ;;;*****************
36 (defrule Flee-rule-01
37    (declare (salience 4010))
38    (fact-danger)
39    (not (fact-health-ok))
40    (not (fact-team-near))
41    (not (layer-activated))
42 =>
43    (assert (action flee))
44    (assert (layer-activated))
45    (printout t "REQUEST HELP AT POS 01" crlf)
46 )
47
48 (defrule Flee-rule-02
49    (declare (salience 4010))
50    (fact-danger)
51    (not (fact-amno-ok))
52    (not (fact-team-near))
53    (not (layer-activated))
54 =>
55    (assert (action flee))
56    (assert (layer-activated))
57    (printout t "REQUEST HELP AT POS 02" crlf)
58 )
59
60 (defrule Flee-rule-03
61    (declare (salience 4020))
62    (not (fact-danger))
63    (not (fact-amno-ok))
64    (not (layer-activated))
65 =>
66    (assert (action seek-friend))
67    (assert (layer-activated))
```

```
68      (printout t "REQUEST FRIEND POS 1" crlf)
69  )
70
71  (defrule Flee-rule-04
72      (declare (salience 4021))
73      (not (fact-danger))
74      (not (fact-health-ok))
75      (not (layer-activated))
76  =>
77      (assert (action seek-friend))
78      (assert (layer-activated))
79      (printout t "REQUEST FRIEND POS 2" crlf)
80  )
81  ;;;-----------------------------------------------
82  ;;;---Attack----
83  ;;;-----------------------------------------------
84  ;;;************
85  ;;;***Attack***
86  ;;;************
87  (defrule attack-rule-attack-near
88          (goal (action attack))
89      ?f<-(action attack-near)
90  =>
91      (printout t "GOAL: ATTACK NEAR" crlf)
92      (retract ?f)
93  )
94
95  (defrule attack-rule-attack-cover
96          (goal (action attack))
97      ?f<-(action attack-cover)
98  =>
99      (printout t "GOAL: ATTACK COVER" crlf)
100     (retract ?f)
101 )
102
103 (defrule attack-rule-attack-far
104         (goal (action attack))
105     ?f<-(action attack-far)
106 =>
107     (printout t "GOAL: ATTACK FAR" crlf)
108     (retract ?f)
109 )
110 ;;;******************
111 ;;;***Attack RULES***
112 ;;;******************
113 (defrule Attack-rule-near
114     (declare (salience 3030))
115     (fact-enemy-near)
116     (fact-enemy-on-sight)
117     (not (attack-method-decided))
118     (not (layer-activated))
119 =>
```

```
120     (assert (action attack-near))
121     (assert (attack-method-decided))
122     (assert (layer-activated))
123  )
124
125  (defrule Attack-rule-from-cover
126     (declare (salience 3020))
127     (fact-enemy-on-sight)
128     (fact-cover)
129     (fact-amno-ok)
130     (not (attack-method-decided))
131     (not (layer-activated))
132  =>
133     (assert (action attack-cover))
134     (assert (attack-method-decided))
135     (assert (layer-activated))
136  )
137
138  (defrule Attack-rule-far
139     (declare (salience 3010))
140     (fact-enemy-on-sight)
141     (fact-amno-ok)
142     (not (attack-method-decided))
143     (not (layer-activated))
144  =>
145     (assert (action attack-far))
146     (assert (attack-method-decided))
147     (assert (layer-activated))
148  )
149  ;;;-----------------------------------------------------
150  ;;;---Backup----
151  ;;;-----------------------------------------------------
152  ;;;************
153  ;;;***BACKUP***
154  ;;;************
155  (defrule backup-rule-help-call
156          (goal (action backup))
157     ?f<-(action help-call)
158  =>
159     (printout t "GOAL: Backup" crlf)
160     (retract ?f)
161  )
162  ;;;******************
163  ;;;***BACKUP RULES***
164  ;;;******************
165  (defrule backup-rule-01
166     (declare (salience 2010))
167     (not (fact-danger))
168     (fact-help-call)
169     (fact-amno-ok)
170     (not (layer-activated))
171  =>
```

```
172     (assert (action help-call))
173     (assert (layer-activated))
174 )
175 ;;;-----------------------------------------------
176 ;;;---Find----
177 ;;;-----------------------------------------------
178 ;;;**********
179 ;;;***Find***
180 ;;;**********
181 (defrule find-rule-seek-enemy
182         (goal (action find))
183     ?f<-(action seek-enemy)
184 =>
185     (printout t "GOAL:  Seek Enemy" crlf)
186     (retract ?f)
187 )
188 ;;;******************
189 ;;;***FIND RULES***
190 ;;;******************
191 (defrule find-rule-01
192     (declare (salience 1010))
193     (not (fact-enemy-on-sight))
194     (not (layer-activated))
195 =>
196     (assert (action seek-enemy))
197     (assert (layer-activated))
198 )
199 ;;;-----------------------------------------------
200 ;;;---Side Rules----
201 ;;;-----------------------------------------------
202 (defrule side-rule-02
203     (declare (salience 6000))
204     (fact-enemy-on-sight)
205 =>
206     (printout t "REPORT ENEMY ON SIGHT" crlf)
207 )
208 ;###############################################################
209 ;;;++++++++++++++++++
210 ;;;+ Initial Facts +
211 ;;;++++++++++++++++++
212 (defrule startup ""
213   (declare (salience 9000))
214   =>
215   (assert (goal (action survive)))
216   (assert (goal (action attack)))
217   (assert (goal (action backup)))
218   (assert (goal (action find)))
219 )
```

Apéndice B

Evolución de comportamientos mediante programación genética con *ADF*

A continuación se presentan los detalles del experimento de generación de comportamiento de combate.

B.1. Condiciones del experimento

Las condiciones de los experimentos son:

- Representación de tipo entera
- Máximo tamaño del cromosoma: 128
- 200 generaciones máximo

- Numero de individuos por generación: 100
- Mecanismo de selección: torneo de tamaño 2
- Probabilidad de mutación: 0.15
- Máximo crecimiento en mutación: 16
- Probabilidad de cruza: 0.9
- Elitismo: 0.15
- Operadores a considerar: suma, resta, multiplicación, mayor que, menor que
- Los argumentos básicos que proporcionan para la función principal son: distancia del enemigo, salud actual y municiones.
- Rango de constantes: [-10,10]
- Número de funciones generadas automáticamente: 3
- Número máximo de municiones: 100

Los movimientos del agente consideran que siempre se está viendo de frente a un enemigo; es decir, *avanzar* implica avanzar hacia el enemigo, *retroceder* se refiere a alejarse del enemigo, etc.

B.2. Función de evaluación

Se probaron dos funciones de evaluación distintas. En la primera función de evaluación propuesta (*función A*), los agentes lograban *aprender* la secuencia que seguía su oponente y con ello esquivaban todos los ataques. La *función B* genera movimientos erráticos que permiten atacar y además evadir ataques del enemigo. Las funciones de evaluación mencionadas se muestran en los *Códigos B.1 y B.2*.

Código B.1: Función A

```
1  /*FUNCION A*/
2      if(!ActionCount||Movements<5)
3          return 10000.0;
4      if(!EvalCrit->KCount[0]->pFunc->Count())
5          return 6000.0;
6      if(g_DamageCreated+g_DamageTaken==0.0||g_DamageCreated==0.0)
```

```
7         return 5000.0;
8     else if(g_DamageCreated==g_DamageTaken)
9         res=4000.0-g_DamageCreated;
10    else if(g_DamageCreated<g_DamageTaken)
11    {
12        if(((float)g_DamageCreated/(float)g_DamageTaken)>0.2f)
13          res=6500.0;
14        else
15          res=3000.0-(g_DamageTaken+1.0)/(g_DamageCreated+1.0);
16    }
17    else
18        res=2000.0-(g_DamageTaken+1.0)/(g_DamageCreated+1.0);
```

La función A penaliza fuertemente a los individuos que no se mueven y que tampoco usan acciones. La siguiente penalización abarca a los individuos que recibieron mas daño que el que generaron. Los valores inferiores a 2000 corresponden a individuos que en *proporción* pueden cometer más daño que el daño recibido.

Código B.2: Función B

```
1   /*FUNCION B*/
2     if(!ActionCount||Movements<5)
3         return 10000.0;
4     if(!EvalCrit->KCount[0]->pFunc->Count())
5         return 6000.0;
6     if(g_DamageCreated+g_DamageTaken==0.0||g_DamageCreated==0.0)
7         return 5000.0;
8     else if(g_DamageCreated==g_DamageTaken)
9         res=4000.0-g_DamageCreated;
10    else if(g_DamageCreated<g_DamageTaken)
11    {
12        if(((float)g_DamageCreated/(float)g_DamageTaken)>0.2f)
13          res=6500.0;
14        else
15          res=3000.0-g_DamageTaken;
16    }
17    else
18        res=2000.0-g_DamageCreated;
```

La *función B* penaliza fuertemente a los individuos que no se mueven y que tampoco usan acciones. La siguiente penalización abarca a los individuos que recibieron mas daño que el que generaron. Los valores inferiores a 2000 corresponden a individuos que *en unidades* pueden cometer más daño que el daño recibido.

Aun cuando se obtienen lo que parecería excelentes resultados con la *función A*, se seleccionaron los comportamientos creados por la *función B* debido a que la secuencia para esquivar disparos de manera perfecta solo podría funcionar ante la situación del experimen-

to. El comportamiento errático y agresivo de la *función B* permite tener mejor desempeño ante más situaciones.

B.3. Resultados

Se ejecutó el programa en varias ocasiones y se seleccionó un caso representativo para mostrar resultados y hacer análisis de los mismos. La *figura B.1* muestra los resultados en 200 generaciones.

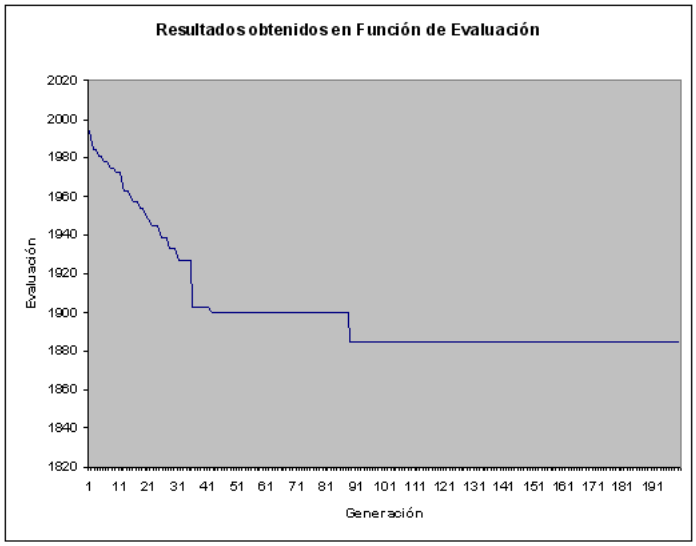


Figura B.1: Resultados obtenidos en 200 generaciones

Observando la *figura B.1*, es posible apreciar que se encontraron individuos que lograban pasar la prueba de sobrevivir a los ataques efectuados por *Enemigo* y conforme las generaciones de ADF se incrementaban lo hacían de forma más eficiente debido al proceso natural de la programación genética; sin embargo, se presentó convergencia prematura principalmente debido a el límite de municiones.

Del comportamiento de los individuos, se generan individuos que atacan a *Enemigo* constantemente acercándose y alejándose esquivando algunos disparos pero al final siempre realizan más daño que el recibido. Los movimientos son erráticos pero se puede apreciar los ciclos repetitivos que se crean al usar las funciones generadas en automático, lo cual puede ser de gran utilidad si se busca que haya oportunidad de poder vencerlos.

Bibliografía

[Ado01] Rogelio Adobbati. Gamebots: A 3D Virtual World Test-Bed for Multi-agent research. *FIPA Workshop*, 2001.

[BJ97] Benoit P. Bradshaw J.M., Dutfield S. Kaos: Toward an industrial-strength open agent architecture. *Software Agents*, 1997.

[BR01] Isla Damian Burke Robert. Creature Smarts: The Art and Architecture of a Virtual Brain. In *In Proceedings of the Game Developers Conference*, pages 147–166, San Jose, California USA, 2001. GDC 2001.

[Cal08] Cal3D. *Library (Computing). Cal3D - 3d character animation library*. Sitio de internet en línea. Disponible en `https://gna.org/projects/cal3d/`. Accedido el 10 de enero de, 2008.

[Can06] Julio Cano. Developing Ambient Intelligence. *Proceedings of the First International Conference on Ambient Intelligence Developments*, 2006.

[CE04] Donnart Jean-Yves Chiva Emmanuel, Devade Julien. *Motivational Graphs: A New Architecture for Complex Behavior Simulation*. Charles River Media, Estados Unidos, 2004.

[Che03] Christopher Cheong. A comparison of jack intelligent agents and the open agent architecture. 2003.

[Coe03] Carlos Coello. *Apuntes de Introducción a la computación evolutiva*. IPN, México, 2003.

[Cor07] Kynapse. Kynogon Corp. State of the art. Sitio de internet en línea. Disponible en `http://www.kynogon.com/`. Accedido el 2 de diciembre, 2007.

[Cor08] Microsoft Corp. XBOX 360 Controller (Imagen). Sitio de internet en línea. Disponible en `http://www.xbox.com/en-ZA/support/xbox360/howto/peripherals/howto-peripherals-wiredcontroller.htm`. Accedido el 16 de marzo de, 2008.

[Dan01] Treglia II Dante. *Camera control Techniques*. Charles River Media, Estados Unidos, 2001.

[Fin92] Tim Finin. An overview of QKML: A Knowledge Query and Manipulation Language. Technical report, U. of Maryland CS Department, Maryland, USA, Febrero de 1992.

[Fog00] David B. Fogel. *Evolutionary computation Vol. 1. Basic Algorithms and operations*. Institute of Physics Publishing, USA, 2000.

[FX08] EMotion FX. *Library (Computing)*. *Mystic Game Development*. Sitio de internet en línea. Disponible en `http://www.mysticgd.com/site2007/`. Accedido el 10 de enero de, 2008.

[Gar08a] Riley Gary. CLIPS. Sitio de internet en línea. Disponible en `http://clipsrules.sourceforge.net/`. Accedido el 11 de mayo de, 2008.

[Gar08b] Riley Gary. Historia de CLIPS. Sitio de internet en línea. Disponible en `http://clipsrules.sourceforge.net/WhatIsCLIPS.html#History`. Accedido el 03 de junio de, 2008.

[Gra08] Granny. *Library (Computing)*. *Granny 3D animation library*. Sitio de internet en línea. Disponible en `http://www.radgametools.com/granny.html`. Accedido el 10 de enero de, 2008.

[HD94] Baker M. Pauline Hearn Donald. *Computer Graphics 2nd Ed*. Prentice Hall, Estados Unidos, 1994.

[Her00] Emmanuel Hernández. Interfaz grafica para controlar y simular robots móviles usando realidad virtual. *UNAM. Facultad de Ingeniería*, Febrero 2000.

[Jac03] Jeffrey Jacobson. Using ÇaveUT"to Build Immersive Displays With the Unreal Tournament Engine and a PC Cluster. *Second International Workshop on Infrastructure of Agents*, June 2003.

[Joh01] Ratcliff John W. *Sphere Trees for Fast Visibility Culling, Ray Tracing and Range Searching*. Charles River Media, Estados Unidos, 2001.

[KD91] Georgeff M. Kinny D. Commitment and Effectiveness of situated agents. *Twelfth International Joint Conference on Artificial Intelligence*, 1991.

[Koz94] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, USA, 1994.

[LD07] Fox Maria Long Derek. Planning Domain Definition Language. Sitio de internet en linea. http://planning.cis.strath.ac.uk/competition/ Accedido el 2 de diciembre de, 2007.

[Ltd06] Agent Oriented Software Pty. Ltd. *JACK Intelligent Agents Agent Manual*. Agent Oriented Software Pty. Ltd, Estados Unidos, 2006.

[Lua08] Lua. The programming language. Sitio de internet en línea. Disponible en `http://www.lua.org/`. Accedido el 10 de marzo de, 2008.

[MD99] Moran Douglas B. Martin David L., Cheyer Adam. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 1999.

[Men08] Roberto A. Flores Mendez. Towards a Standardization of Multi-Agent System Frameworks. Sitio de internet en línea. Disponible en `http://www.acm.org/crossroads/xrds5-4/multiagent.html`. Accedido el 3 de marzo de, 2008.

[Mil06] Ian Millington. *Artificial Intelligence for Games*. Morgan Kauffmann, USA, 2006.

[MJ95] Finin Tim Mayfield James, Labrou Yannis. Evaluation of kqml as an agent communication language. Junio 1995.

[Nil98] N. J Nilsson. STRIPS Planning Systems. In *Artificial Intelligence: A New Synthesis*, pages 373–400, San Francisco California, Agosto de 1998.

[Ork05] Jeff Orkin. Agent Architecture Considerations for Real Time Planning in Games. *AIIDE*, 2005.

[Ork06] Jeff Orkin. Three States and a Plan: The AI of F.E.A.R. *Game Developers Conference*, 2006.

[Ork07] Jeff Orkin. Learning Plan Networks in Conversational Video Games. Master Thesis, MIT, USA, 2007.

[Per08] Anselmo Perez. Introducción a la programación Evolutiva. Sitio de internet en linea. `http://surf.de.uu.net/encore`. Accedido el 8 de marzo de, 2008.

[Ron86] Brooks Roney. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, (2):14–23, 1986.

[Rus95] Norving P. Russell, S. *Artificial Inteligence: A modern approach*. Prentince Hall, Estados Unidos, 1995.

[Ser07] Marcellin Jacques Sergio. *Notas del Curso:* Construcción de Sistemas Expertos. Posgrado en Ciencias e Ingeniería de la Computación, México. UNAM, 2007.

[Sim01] Carter Simon. *Managing AI with Microthreads*. Charles River Media, Estados Unidos, 2001.

[Sin97] Munindar P. Singh. Considerations on Agent Communication. *FIPA Workshop*, 1997.

[Sve95] Loncaric Sven. Virtual reality foundations. Junio 1995.

[VS95] O'Connor P. Virdhagriswaran S., Osisek D. Standardizing agent technology. *ACM StandardView*, Mayo 1995.

[WA92] Watt Mark Watt Alan. *Advanced Animation and Rendering Techniques Theory and Practice*. Addsion-Wesley, Estados Unidos, 1992.

[Wei99] Gerhard Weiss. *Multi Agent Systems. A modern approach to Distributed Artificial Intelligence.* MIT Press, Massachusetts, USA, 1999.

[Wik08] Wikipedia. *Library (Computing). Wikipedia, the free encyclopedia*. Sitio de internet en línea. Disponible en `http://en.wikipedia.org/wiki/Metadata`. Accedido el 4 de mayo de, 2008.

[WM98] Jennings N.R. Wooldridge M. Pitfalls of agent-oriented development. *Proceedings of the Second International Conference on Autonomous Agents (Agents '98)*, Mayo 1998.

[Woo95] Jennings N.R. Wooldridge, M. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 1995.

[Yan06] Huang Yanwen. Multiagent cooperation based entertainment robot. In *ACM. Robotica Vol. 24*, pages 643–648, USA, Septiembre de 2006. ACM Press.