Parte III

Aportaciones de la Tesis

Capítulo 4

Marco de Simulación del Comportamiento

En este capítulo se presenta y describe un Marco General para la Simulación del Comportamiento de Agentes Virtuales Autónomos, el cual permite reproducir acciones características del comportamiento humano mediante la animación autónoma de actores virtuales en entornos 3D.

Las siguientes secciones introducen la nueva arquitectura, describiendo los detalles de implementación y los componentes que la forman. También se hace una presentación del ambiente virtual de prueba elegido para evaluar el sistema de animación desarrollado, al tiempo que se describen todos los elementos que integraran el mundo 3D.

4.1 Marco General de Simulación

En el capítulo anterior se han discutido brevemente algunos marcos de simulación del comportamiento para AVAs. En ellos destaca la integración de todas las técnicas que permiten la animación de los actores virtuales; pues, es esa la finalidad última de un marco de simulación, proporcionar una estructura que facilite el diseño de simulaciones de entorno virtuales inteligentes.

A continuación se presenta una nueva arquitectura para el desarrollo y simulación de AVAs en entornos 3D; mostrada en la Figura 4.1, y a la cual llamaremos MaGeS.

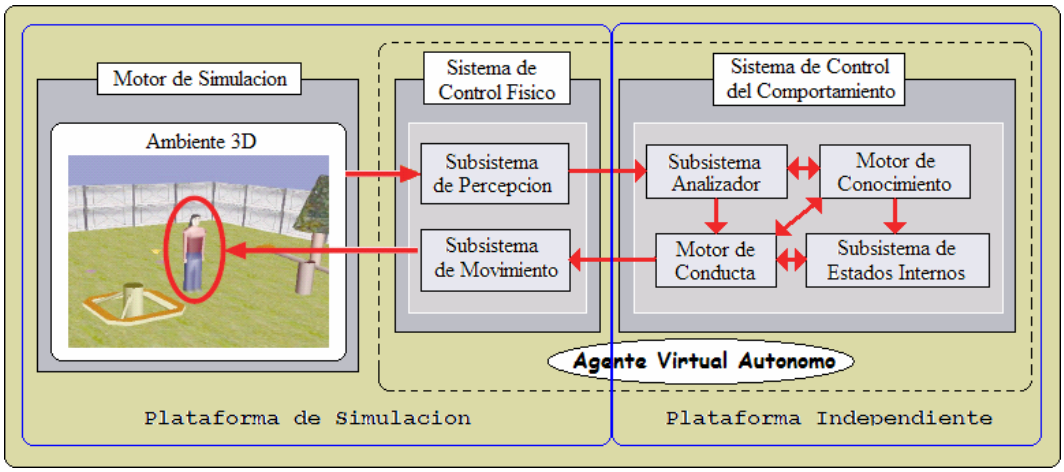


Figura 4.1. Marco General de Simulación para AVAs (MaGeS)

MaGeS esta formado por dos sistemas principales; el Motor de Simulación, y el Sistema Agente Virtual Autónomo (el cual es el sistema de animación del actor virtual, propiamente dicho).

El Motor de Simulación es el encargado de controlar el ciclo de ejecución de la animación, así como del dibujado del mundo virtual 3D y los elementos físicos que lo componen.

El Sistema Agente Virtual Autónomo incorpora todos los elementos que permiten la animación de los actores virtuales sin intervención externa. Se divide en dos sistemas que separan las actividades físicas del agente, de las cognitivas. El Sistema de Control Físico (SCF) es el encargado de llevar a efecto los movimientos requeridos por el AVA, así como el de recolectar información del entorno, según las capacidades de percepción que incorpore. El Sistema de Control del Comportamiento (SCC) correspondería a la actividad "cerebral" del AVA, emulando mecanismos como la memoria y las emociones, así como el procesamiento de la información que maneja (tanto externa como interna) para la toma de decisiones sobre las acciones a seguir en cada instante de tiempo. Por consiguiente, mediante el Subsistema de Percepción del SCF, el AVA es capaz de identificar los elementos que componen el entorno gráfico (objetos estáticos, objetos inteligentes, otros AVAs, sonidos), pudiendo reconocer los cambios que se producen a su alrededor. La información percibida es enviada al SCC, donde es utilizada para definir las acciones a seguir, las cuales son traducidas a ordenes motoras y enviadas al SCF, para ser ejecutadas por el Subsistema de Movimiento; quedando de esta forma completado el ciclo de interacción del

AVA con su entorno, y el cual se repite para cada unidad de tiempo que dura la simulación.

La principal diferencia entre MaGeS y arquitecturas presentadas por otros autores en trabajos precedentes, es la forma como la información es organizada y procesada para la toma de decisiones, en especial en el Sistema de Control del Comportamiento, el cual es discutido en detalle en el siguiente capítulo.

Otro aspecto resaltante de MaGeS es la separación que hace entre el SCF y el SCC. Aun cuando no resulte novedosa la idea de independizar los procesos orientados a las actividades físicas de las cognitivas [THAL01], por aquello de mantener separado lo relacionado a la mente del cuerpo; la arquitectura del AVA dentro de MaGeS aporta un grado más de independencia que la introducida en [THAL01].

Separar los procesos relacionados a la actividad mental de la física, no solo es congruente con lo que sucede en la naturaleza, sino que además, desde un punto de vista de programación, permite trabajar ambos módulos simultáneamente, enfocándose en los requerimientos propios de cada sistema. Por ejemplo, el SCC decide las acciones a llevar a cabo, como puede ser la de caminar hacia un punto determinado, trazando el camino a seguir. El SCC da la orden al SCF de dar un paso en una dirección, sin preocuparse de los aspectos gráficos a considerar para la correspondiente acción. Es el SCF el que se encarga de ajustar todos los ángulos de las extremidades del modelo geométrico del agente para realizar la animación del caminar manteniendo una transición natural entre la postura original y la nueva. Así mismo, el SCF incorpora todas las rutinas necesarias para el movimiento y transiciones entre movimientos del AVA, sin preocuparse de en qué momento serán utilizadas.

El sistema presentado por J-S. Monzani y colegas [THAL01] (descrito brevemente en la Sección 3.1.3.3) esta divido en dos componentes, uno de bajo nivel llamado ACE y otro de alto nivel llamado IVA (Figura 3.11), dedicados a la simulación física y de comportamiento, respectivamente; de allí la designación de bajo y alto nivel, referidas a las actividades a las que están asociadas. El IVA de [THAL01] corresponde al SCC en MaGeS, mientras que el ACE engloba tanto al Motor de Simulación como al SCF. Por otro lado, mientras que en MaGes cada AVA es responsable, no solo de su sistema de toma de decisiones, sino también de la actualización y mantenimiento de su apartado gráfico; el ACE [THAL01] controla tanto el ciclo de simulación como el manejo del apartado gráfico de todos los agentes mediante el Manejador de Tareas, permitiéndoles caminar, ejecutar secuencias de animación, coordinar su interacción con objetos, etc.

Comenzaremos por describir las herramientas de programación utilizadas, para luego pasar a los detalles de implementación de la arquitectura MaGeS, poniendo especial interés en el Motor de Simulación.

4.1.1 Lenguajes y Ambientes de Programación

Todo desarrollo de software requiere una cuidadosa elección de las herramientas de programación seleccionadas para lograr, no solo el mejor desempeño posible de la aplicación, sino también para cumplir los requerimientos de un buen diseño, fácil desarrollo, y cómodo mantenimiento.

En ese sentido, las aplicaciones que incluyen animación por ordenador no son la excepción. A continuación se describen las herramientas de software empleadas para el desarrollo del sistema aquí presentado.

OpenGL es una interfase de programación de aplicaciones (API) estándar, desarrollada por Silicon Graphics en 1992, y en la cual se colocó especial interés en su portabilidad, posibilidades de expansión, y por supuesto, su rendimiento. Desde sus inicios se ha consolidado como la librería por excelencia para desarrollar aplicaciones 2D y 3D con independencia de la plataforma, proporcionando varios cientos de funciones que dan acceso a todas las características del hardware gráfico. Sin embargo, para hacer su uso mas cómodo es complementada con la librería GLU (OpenGL Utility Library) la cual provee funciones de alto nivel, ofreciendo características que van desde simples funciones OpenGL encubiertas, hasta complejos componentes que soportan sofisticadas técnicas de dibujado. Sus características incluyen:

- Escalado de Imágenes
- Dibujado de objetos 3D incluyendo esferas, cilindros y discos.
- Generación automática de mapeado a niveles, partiendo de una simple imagen.
- Soporte de superficies curvas a través de NURBS.
- Soporte para teselado de polígonos no-convexos.
- Transformaciones de propósito especial y matrices.

Por otro lado, si bien la portabilidad de OpenGL permite que el código fuente de una aplicación pueda ser el mismo para una maquina Sparc corriendo Linux, que para un PC corriendo Windows y usando un acelerador gráfico, esas dos plataformas utilizan sistemas de ventanas diferentes (X-Windows y MS-Windows, respectivamente); y aunque podrían utilizarse sus

respectivos manejadores de ventanas, esto haría perder la portabilidad del código.

GLUT (OpenGL Utility Toolkit) es un API específicamente diseñado para cubrir la necesidad de una interfase de programación independiente del sistema de ventana para programas OpenGL. Algunas de sus funcionalidades son las siguientes:

- Múltiples ventanas para dibujado de OpenGL.
- Manejador de eventos de retorno de procesos.
- Sofisticados dispositivos de entrada.
- Rutina "controladora" y temporizadores.
- Un simple menú cascada desplegable.
- Rutinas para generar varios objetos en alambre o sólidos.
- Soporte para bitmap y molde para fuentes.
- Funciones para el manejo de ventanas

GLUT simplifica la implementación de programas que usan dibujado con OpenGL, requiriendo muy pocas rutinas para mostrar escenas gráficas. Además, GLUT es un sistema de ventanas independiente, ya que la librería ha sido implementada para las plataformas mas usadas; por lo que la portabilidad del código generado se mantiene entre plataformas, sin necesidad de cambios.

C++ es un lenguaje de programación de alto nivel orientado a objetos de propósito general, el cual incorpora una variedad de características que facilitan una programación elegante y modular. Además, su alto rendimiento y versatilidad ha contribuido a que su uso incluya la creación de otros lenguajes de programación de alto nivel como JAVA, y el desarrollo de sistemas operativos como Linux; pudiendo encontrar soporte para la mayoría de las plataformas. Es por ello que este lenguaje es seleccionado para la implementación de MaGeS, utilizando Microsoft Visual C++ v6, como entorno de programación. Es importante destacar que se utilizó el modo consola de MS Visual C++, y el código escrito en C++ estándar, para mantener su portabilidad.

Prolog es un lenguaje de programación de alto nivel inspirado en el paradigma de la programación declarativa relacional (basada en fragmentos de la lógica de primer orden; fundamentalmente cláusulas de Horn), y su nombre se deriva de PROgramación LOGica (comentado en la Sección 1.1.2). Ampliamente usado en el campo de la Inteligencia Artificial (IA), en particular en el desarrollo de aplicaciones de Sistemas Expertos, Procesamiento de Lenguaje Natural, Base de Datos Deductivas, y Programación Distribuida y Multiagente, entre otras. La implementación de

muchas de las técnicas en IA con Prolog, usualmente requieren menos código y tiempo de desarrollo, por lo que su uso resulta de gran utilidad en este campo. Como entorno de desarrollo para la programación en Prolog se empleó AMZI! Prolog v6.0, ya que al haber sido implementado en lenguaje C, ofrece total compatibilidad en la comunicación con aplicaciones C externas; además, poniendo a un lado sus cualidades para aplicaciones en IA, también permite compilar los programas, generando código binario que es ejecutado por una maquina virtual prolog; esto facilita la ejecución de código independiente del entorno de desarrollo (siendo prolog un lenguaje interpretado por naturaleza). Además, se puede conseguir soporte de AMZI! Prolog, tanto para plataformas Microsoft Windows como para Linux.

La Figura 4.2 ilustra la relación entre las componentes de MaGeS, y los recursos de software empleados en su desarrollo.

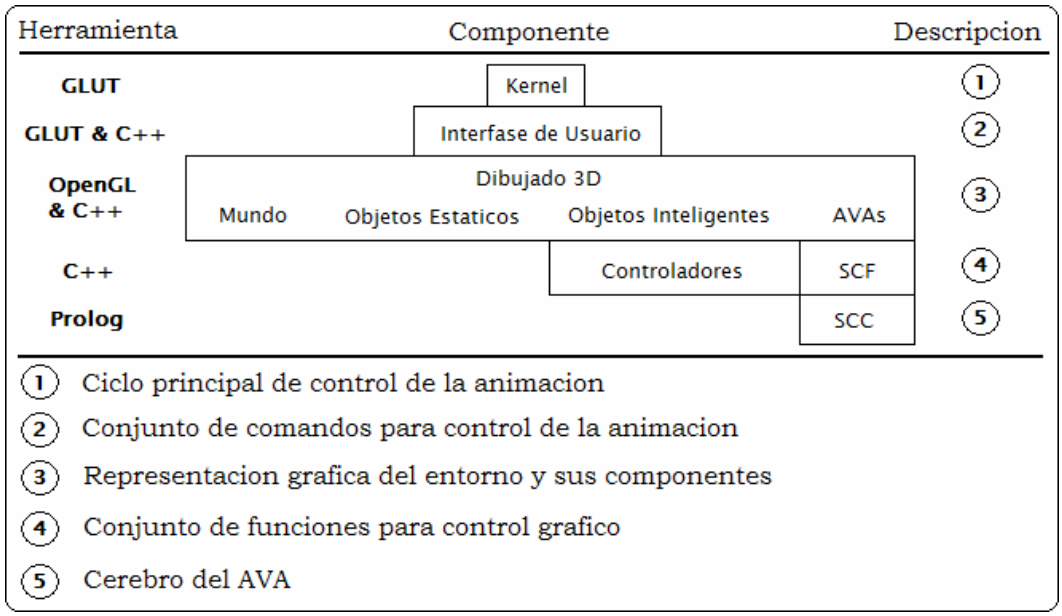


Figura 4.2. Herramienta de Software empleado en cada componente de MaGeS

Todas las APIs utilizadas (GLUT, GLU, GL) están relacionadas al dibujado gráfico, y son incluidas en el código C, desde Visual C++. Los primeros 3 niveles de la jerarquía (kernel, interfase, y dibujado 3D) son funciones del Motor de Simulación, que junto con el SCF del AVA completan los componentes que conforman el área denominada “Plataforma de Simulación” (ver Figura 4.1), ya que son todas desarrolladas íntegramente en Visual C++; mientras que el Sistema del Control del Comportamiento es totalmente desarrollado en Amzi! Prolog (plataforma independiente del entorno de desarrollo de la aplicación).

4.1.2 Implementación

En esta sección se discutirá en detalle el Motor de Simulación y su interconexión con el Sistema de Agente Virtual Autónomo, bajo distintas plataformas de hardware; específicamente, su implementación en ordenadores monoprocesador (PC tradicionales), y en sistemas distribuidos (red de ordenadores).

El Motor de Simulación es la aplicación principal que controla el ciclo de ejecución de la animación. Al iniciarlo, éste carga varios archivos de texto con la información del entorno y los elementos que participan en la animación, para luego dar inicio al ciclo de simulación.

Los archivos mencionados se dividen en tres tipos: los de entorno (.env), los parques (.prk), y los personajes (.per). A continuación se describen cada uno de ellos.

`Archivo de Personajes.` Cada humano virtual que aparece en la simulación tiene asociado un archivo .per donde se encuentra la información sobre sus características físicas, tales como, género (hombre, mujer, o niño), tipo de camisa (manga corta o larga), color del cabello, y el color de su ropa.

`Archivo de Parques.` El escenario seleccionado para las pruebas de simulación es un parque virtual. Este archivo almacena los datos sobre el parque (tal como sus dimensiones), los elementos que contiene (árboles, bancas, ruedas, subibajas, alpiste, y aves), y las cámaras (para observar la simulación desde diferentes puntos de vista). Adicionalmente, para cada elemento, incluye información sobre su ubicación y orientación dentro del parque. Estos archivos se reconocen por su extensión .prk.

`Archivo de Entorno.` Reúne todos los datos necesarios sobre el mundo virtual en el que se llevara a efecto la simulación, como lo son: el nombre del archivo del parque en el que se desarrollará la trama, y el nombre de los archivos de los personajes que participarán en ella. Además, para cada personaje se incluye información adicional, tal como rango de visión, cantidad de neuronas (capacidad de memoria), coeficiente de aprendizaje, nivel inicial de energía, velocidad de recuperación del cansancio, resistencia al cansancio, grado de sociabilidad, dinamismo, y relaciones de parentesco y afinidad; toda ella, para otorgarle cualidades particulares a cada personaje.

La explicación sobre la creación de estos archivos es ampliada en el Capítulo 7.

La Figura 4.3 muestra la organización de estos archivos. En el ejemplo, el procedimiento de inicialización del Motor de Simulación lee un archivo de entorno (.env), en el que se encuentra toda la información del mundo virtual (la referencia al archivo con los datos del parque virtual a utilizar, y la referencia a dos archivos con las características "físicas" de los personajes que participan en la simulación).

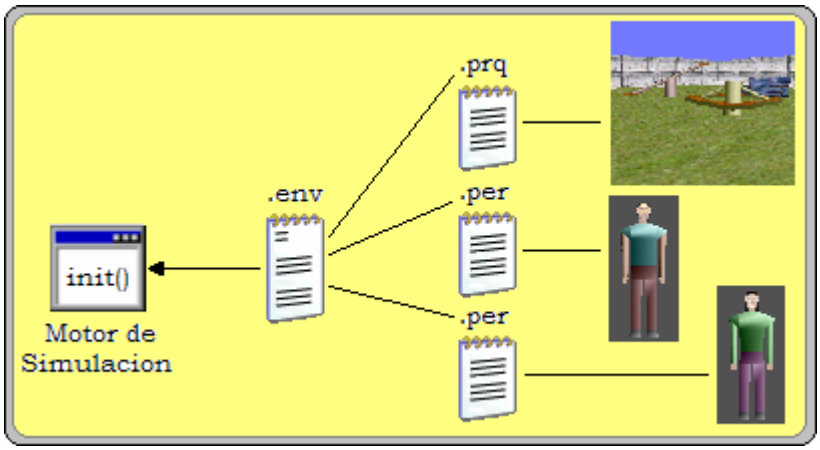


Figura 4.3. Organización de los archivos de datos del Mundo Virtual

La ventaja de esta distribución de archivos es que permite tener distintos diseños de personajes y parques en archivos individuales .per y .prk respectivamente, y de esta forma poder reunir en un solo archivo de entorno los elementos que se deseen que participen en una simulación. La posibilidad de preparar distintos archivos de entorno proporciona la suficiente versatilidad para evaluar los módulos de comportamiento de los AVAs bajo condiciones diversas pero controladas, con un mínimo de esfuerzo; ya que la información almacenada en los archivos .prk y .per puede ser reutilizada constantemente para construir distintas simulaciones. Recordemos que en los archivos .per solo aparecen los datos sobre las características físicas de los personajes, y es en el archivo de entorno en el que se reúnen las cualidades particulares asociadas a cada personaje; es por ello, por ejemplo, que en simulaciones que usen los mismos archivos (.prk y .per) se puede tener al mismo personaje (en apariencia) exhibiendo un comportamiento distinto, lo cual facilita la evaluación del comportamiento para distintos parámetros de personalidad.

Al tiempo que el Motor de Simulación va leyendo los datos de los archivos, también va construyendo las estructuras de control para el dibujado del escenario 3D y de todos sus elementos. Cada elemento es encapsulado en una *clase* que lo define; por lo que el Motor de Simulación

crea una lista para cada tipo de elemento, que contiene las instancias necesarias de cada uno de ellos. La estructura de dichas *clases* son descritas en la Sección 4.2.

La Figura 4.4 muestra un ejemplo de la lista antes mencionada, para los árboles del parque virtual. Para cada especificación de un árbol en el archivo .prk, el Motor de simulación crea una instancia de la *Clase Árbol*, pasándole la información necesaria para su inicialización (tal como su posición), y añadiéndola a la lista correspondiente a los árboles. Lo mismo sucede para el resto de los elementos que aparecen en el archivo .prk, y para cada personaje incluido en el archivo de entorno.

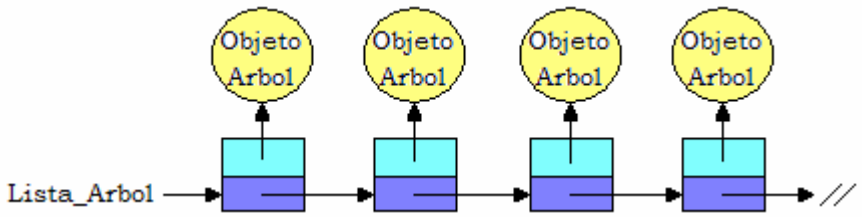


Figura 4.4. Lista de Instancias de la Clase Árbol

Una vez creadas todas las estructuras, el control pasa al *Kernel* del Motor de Simulación, el cual, mediante un manejador de eventos, espera alguna entrada del usuario para dar inicio a la simulación, detenerla, cambiar de cámara, o activar y desactivar la grabación de la secuencia de imágenes que se generan durante la simulación.

Comenzada la simulación, el ciclo de ejecución se repite

```
Ciclo de Simulación
        Para cada Humano Virtual
             Mirar();
             Oir();
             Animar();
        Fin Para Humano Virtual
        Para cada Ave Virtual
             Mirar();
             Animar();
        Fin Para Ave Virtual
Fin Ciclo de Simulación
```

Como puede verse, el ciclo de simulación solo activa la ejecución de los agentes virtuales autónomos, ya que son ellos mismo los encargados de interactuar directamente con los objetos inteligentes (los otros elementos del entorno que pueden exhibir animación). Los detalles de las funciones de percepción y animación son tratadas en la Sección 4.3.

4.1.2.1 Sistemas Monoprocesador y Distribuido

El marco de simulación presentado en las secciones previas puede ser implementado tanto sobre un sistema monoprocesador, como en una red de ordenadores para aprovechar el potencial de multiprocesamiento paralelo implícito en el diseño. Esta sección presenta los detalles de implementación de MaGeS para ambas arquitecturas, comenzando por las posibilidades para una PC estándar, y luego pasar a entornos distribuidos.

Para el primer caso, el kernel del Motor de Simulación puede ejecutar la animación de cada AVA secuencialmente, accediendo a cada uno de ellos mediante una lista dinámica (Figura 4.5a). Otra alternativa es ejecutar cada AVA en procesos separados de manera concurrente (Figura 4.5b). En cualquier caso, un solo ciclo de simulación pasa por ejecutar una sola vez el método de animación de cada AVA, antes de tener completada una imagen de la secuencia de animación. En el caso de la lista dinámica, como su recorrido es secuencial, el ciclo de simulación estará completado después de animar el último AVA. Por otro lado, ejecutando cada AVA en un proceso separado, el Controlador de Procesos es el encargado de sincronizarlos, para que ninguno de ellos comience un nuevo ciclo de simulación antes de que todos hayan terminado el actual.

Otro aspecto a tomar en cuenta para estas dos implementaciones en una arquitectura monoprocesador, tiene que ver con la detección de colisiones entre AVAs. Es usual que los actores virtuales se desplacen de un lado a otro del entorno. Si son animados secuencialmente, cada AVA conoce la disposición de los demás, antes de tomar una decisión. Por el contrario, cuando son ejecutados concurrentemente, dos AVAs pueden decidir avanzar a un mismo punto simultáneamente, por lo que el tratamiento para evitar colisiones debe ser distinto; aunque una implementación de esta técnica puede considerar ambos casos para hacerla independiente del diseño que se elija.

Como se aprecia en el diseño de MaGeS (Figura 4.1), cada agente virtual consta de dos componentes que pueden implementarse sobre plataformas de desarrollo distintas, en cuyo caso, deben considerarse los aspectos de comunicación entre ambos módulos. Las ventajas de dicha separación fueron discutidas a comienzos de este capítulo.

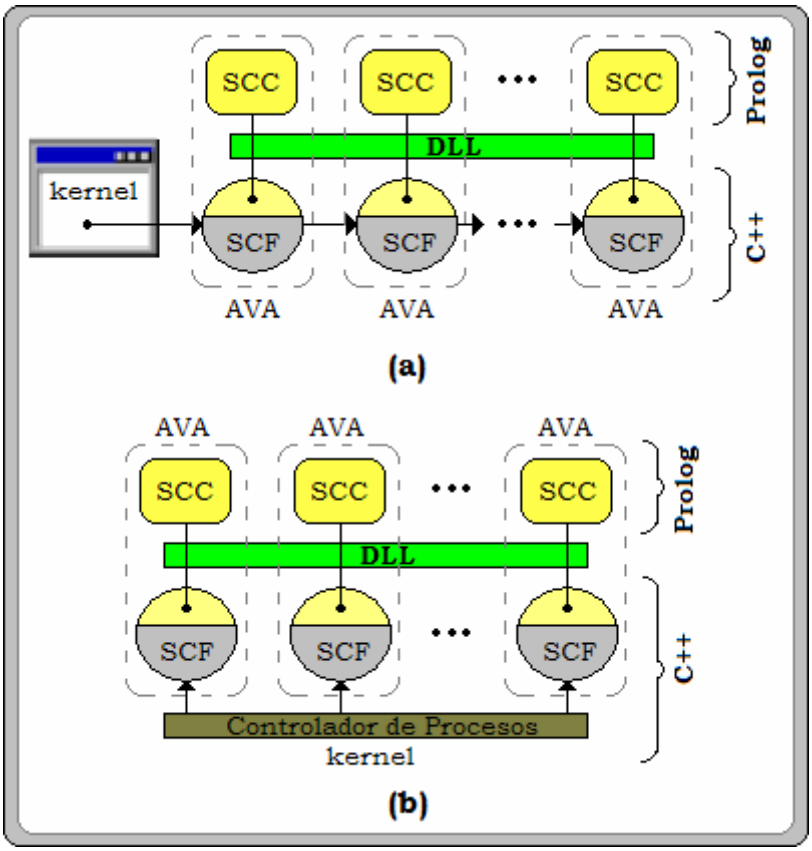


Figura 4.5. Implementación sobre arquitecturas monoprocesador

La Figura 4.5 muestra las componentes de los AVAs implementadas utilizando diferentes herramientas de programación. Es claro que el diseño no se restringe al uso exclusivo de estos lenguajes, los cuales fueron seleccionados por sus características específicas (ver Sección 4.1.1). Pues bien, para estas arquitecturas monoprocesador la comunicación entre los módulos se logra empleando librerías de enlace dinámico (DLL), ya que proporcionan un eficiente medio de comunicación entre aplicaciones.

El empleo de DLLs no es la única vía de comunicación entre aplicaciones. En el trabajo [THAL01] (ver Sección 3.1.3.3, Figura 3.11), los autores establecen la comunicación entre los IVAs y el ACE, mediante el protocolo TCP/IP, argumentando su portabilidad hacia arquitecturas distribuidas. Lo cierto es que en una PC estándar, utilizar TCP/IP para comunicación entre procesos esta lejos de ser el medio más eficiente, siendo el uso de DLLs lo idóneo para este caso.

Para efectos prácticos de este proyecto, la implementación de MaGeS desarrollada corresponde al de la Figura 4.5(a).

Con respecto al segundo caso, la Figura 4.6 ilustra un nivel de granularidad gruesa para la paralelización de MaGeS, en una red tipo estrella maestro-esclavo.

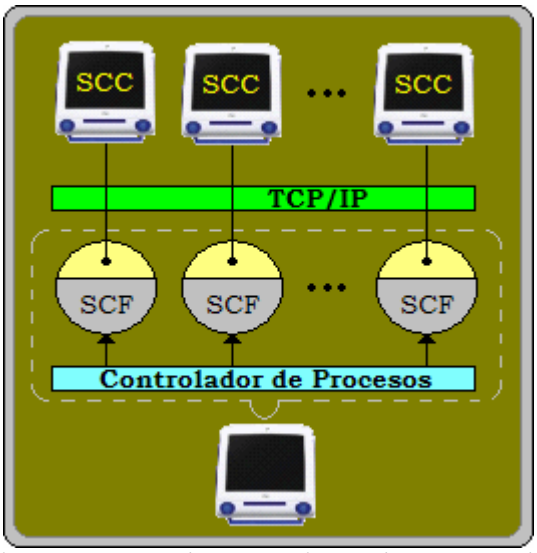


Figura 4.6. Implementación sobre una red de ordenadores

En una arquitectura maestro-esclavo, el maestro es el encargado de controlar el ciclo de simulación, y producir y dibujar la secuencia de imágenes de la misma. En este diseño, todos los AVAs son inicializados en el Maestro, el cual, mediante el Controlador de Procesos los maneja concurrentemente. Sin embargo, solo el procesamiento del Sistema de Control Físico (SCF) de los AVAs es llevado a cabo en el Maestro, ya que el Sistema de Control del Comportamiento (SCC) de cada uno es realizado en un ordenador independiente (esclavos); estableciéndose la comunicación entre el Maestro y los Esclavos mediante TCP/IP.

Dada la relación de procesamiento secuencial existente entre el SCF y el SCC de un AVA, (percepción(SCF)-toma de decisiones(SCC)-acción(SCF)), se podría estar tentado a implementar cada AVA completamente en un esclavo. Sin embargo, para una mayor eficiencia, el balanceo de las cargas de trabajo resulta esencial, a la vez que se busca minimizar el tiempo de comunicación entre ordenadores. El mayor porcentaje de trabajo computacional asociado a un AVA es debido al SCC, por la diversidad de técnicas algorítmicas que reúne y sus requerimientos de memoria; es por ello que resulta provechoso asignar a cada SCC un ordenador. Por otro lado, el tráfico de información requerido entre el SCF y el SCC es menor que el que ocurre entre el SCF y el Motor de Simulación para el dibujado de la escena. Por lo tanto, el tiempo de cómputo que emplea el Maestro para procesar los SCFs se compensa con la baja transferencia de datos entre maestro y esclavos.

Lo que muestra la Figura 4.6 es solo uno de muchos posibles diseños para paralelizar el sistema de simulación del comportamiento de agentes virtuales autónomos. El estudio de los algoritmos empleados a diferentes niveles de granularidad, y la cantidad y características de los equipos computacionales con los que se dispongan, determinara el mejor diseño para cada entorno distribuido.

Lo que resulta claro es la gran ventaja de tener separados el SCC y el SCF, del Motor de Simulación, ya que esto ofrece una mayor versatilidad para probar implementaciones con diferentes topologías, con el fin de obtener el mayor grado de paralelismo y optimizar el rendimiento.

4.2 Entorno Gráfico de Simulación

Como se ha comentado anteriormente, el escenario elegido para las pruebas de las simulaciones es un parque virtual. Las razones para tal elección son las siguientes:

- La amplitud del escenario les permite una mayor movilidad a los agentes, con lo que se obtienen simulaciones mas elaboradas.
- Ofrece versatilidad en su diseño, pudiéndose conseguir una gran diversidad de escenarios de prueba con solo variar los elementos que lo integran, o su disposición.
- Permite experimentar con una amplia variedad de interacciones: agente-objeto, agente-agente, agentes-objeto.
- Y finalmente, ofrece un entorno familiar, en el que es fácil identificar las diversas acciones que en él ocurren.

El parque virtual, como tal, es un área cuadrada, cuya dimensión y componentes son definidos en el archivo .prk (ver Sección 4.1.2), rodeado de paredes que establecen los limites del mismo. En su interior se distribuyen elementos que complementan el escenario final para las simulaciones. Los elementos que integran el parque virtual son: árboles, bancas, ruedas, subibajas, alpistes, y aves; los humanos virtuales son incorporados al configurar la simulación, tal como se describió en la Sección 4.1.2. La Figura 4.7 muestra lo antes descrito.

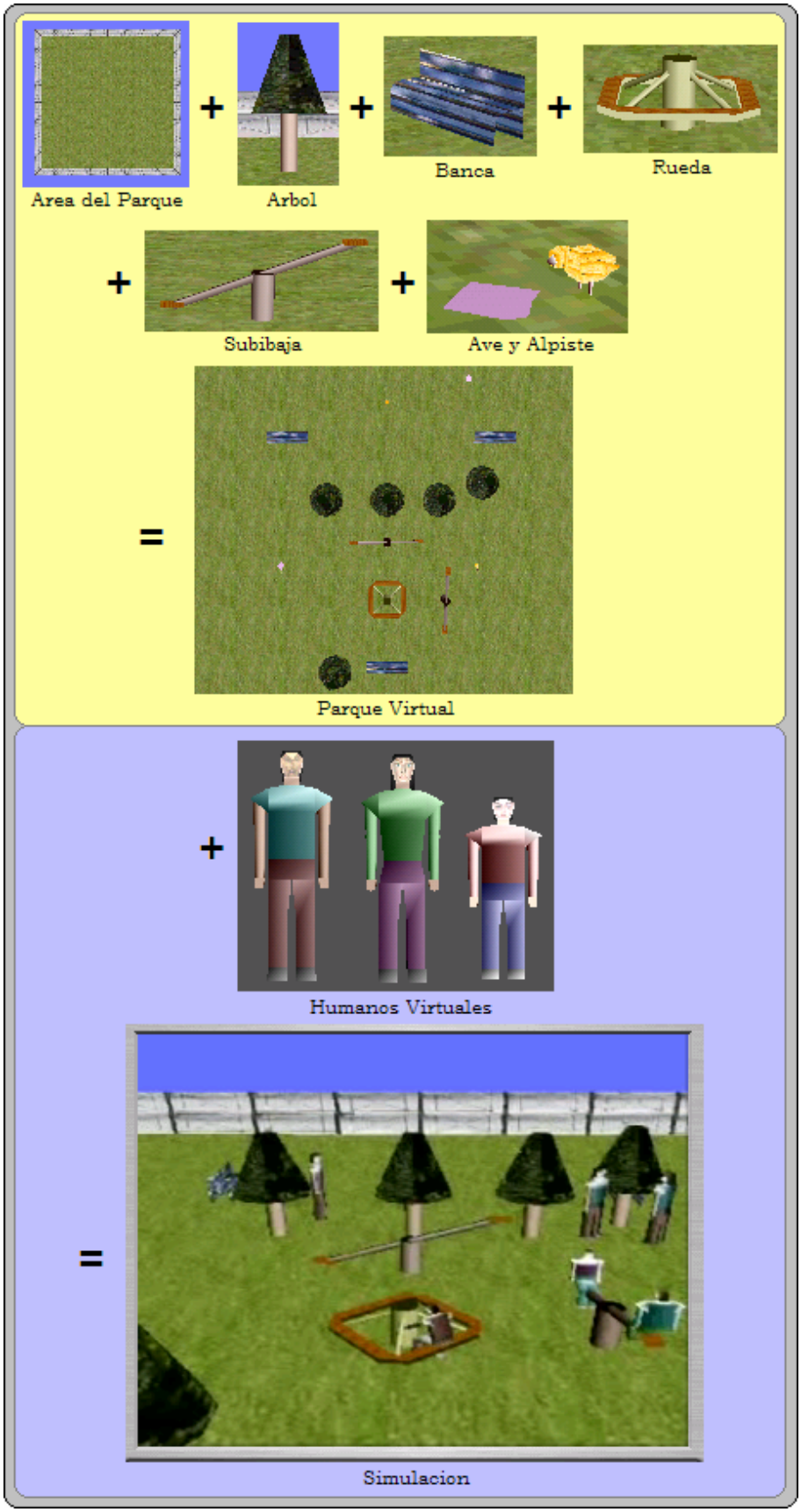


Figura 4.7. Construyendo un escenario para la Simulación

Como se puede apreciar en la Figura 4.7, las simulaciones incluyen dos tipos de AVAs: aves y humanos; esto es, dos tipos de actores virtuales que se desenvuelven de manera autónoma por el ambiente virtual según sus posibilidades motoras, y tomando decisiones conforme a sus respectivas estructuras de conocimientos y estados internos. La implementación de ambos tipos de agentes se rige según el diseño Sistema de Agente Virtual Autónomo de MaGeS, mostrado en la Figura 4.1, y serán descritos en detalle en la Sección 4.3, y en el Capitulo 5.

Pasaremos ahora a describir cada uno de los elementos que forman parte del entorno del parque virtual.

4.2.1 Objetos Inteligentes

En el Capítulo 2 hemos visto lo que son los objetos inteligentes, y como sus diseños permiten la fácil interacción con los agentes.

Todos los elementos que forman parte del parque virtual (salvo el ave, claro está) son objetos inteligentes, ya que su diseño incorpora algún nivel de cambio de estado que un actor virtual es capaz de percibir para su interacción con él.

Dado que hay dos tipos de AVAs, cada objeto inteligente se diseñó para interactuar con solo uno de ellos, por lo que para el otro tipo de agente, dicho objeto no es más que un elemento del decorado. Tal es el caso, por ejemplo, de La Rueda, donde los humanos pueden sentarse y jugar, mientras que un ave solo la percibe como un obstáculo en su camino. Lo mismo ocurre para el resto de los objetos inteligentes.

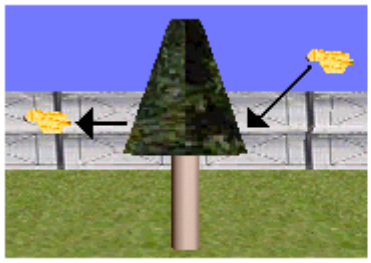
La Programación Orientada a Objetos nos posibilita tener a cada elemento del parque virtual definido dentro de una *clase*, la cual establece las características y funcionalidades del objeto. Todas las clases tienen un método en común: `dibujar(void)`, el cual es el encargado de transmitirle al Motor de Simulación la información gráfica sobre el objeto en su estado actual, necesaria para su dibujado. El resto de los métodos de cada clase entra en una de las siguientes categorías: 1) nativos (refiriéndose a los métodos *constructor* y *destructor* de la clase), 2) informativos (refiriéndose a aquellos métodos que nos proporcionan datos sobre el estado del objeto), y 3) acción (para aquellos métodos que cambian el estado del objeto).

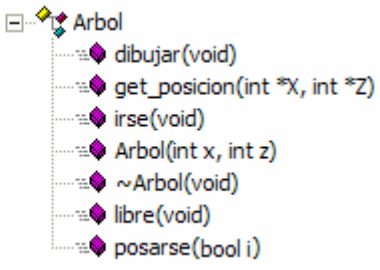
A continuación pasamos a describir los métodos de cada clase.

4.2.1.1 Árbol

Es el más simple de los objetos inteligentes, y su funcionalidad (como objeto inteligente) se limita a servir de refugio a un ave virtual.

Los métodos que forman la Clase Árbol son los siguientes:





Nativos	**Descripción**
Árbol y ~Árbol	El constructor recibe la ubicación del árbol dentro del parque
Informativos	
get_posicion	Devuelve la posición del árbol
libre	Indica si hay algún ave dentro del árbol
Acción	
posarse	Actualiza el estatus del árbol, si un ave se posa dentro de él, o lo abandona

4.2.1.2 Alpiste

El alpiste es el otro objeto inteligente relacionado al ave virtual, y representa su comida. Cada objeto Alpiste posee un valor asociado a la cantidad de comida que posee, y que a su vez establece su nivel de opacidad para hacerlo más visible. Cada vez que un ave siente hambre, se acercara a este objeto, e irá decrementando el valor de su cantidad de comida, simulando el acto de comer, y haciendo al objeto cada vez mas traslucido. Cuando su nivel de comida llega a 0 indica que ya no contiene más, y se torna transparente, por lo que no podrá verse gráficamente.

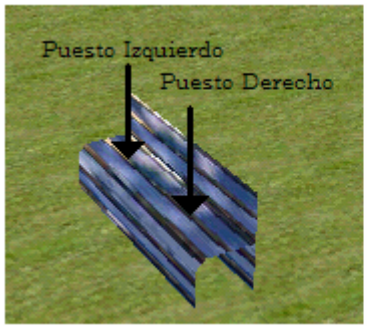
Los métodos que forman la Clase Alpiste son los siguientes:

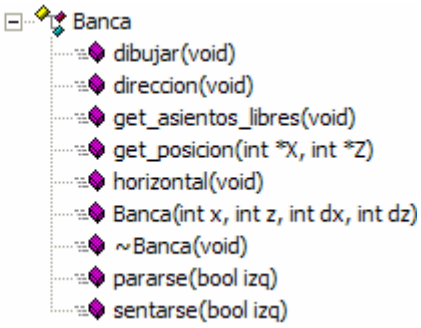
Nativos	Descripción
Alpiste, ~Alpiste	El constructor recibe la ubicación del alpiste dentro del parque
Informativos	
get_cantidad	Devuelve la cantidad de comida que aun le queda
get_posicion	Devuelve su posición
Acción	
picar	Lo activa un ave cada vez que come, decrementando la cantidad de comida

4.2.1.3 Banca

Es un lugar para el reposo de los humanos virtuales. En ellos se pueden llegar a sentar hasta dos de estos agentes. El objeto Banca lleva control sobre el estatus de su disponibilidad; esto es, sobre la disponibilidad de sus puestos.

Los métodos que forman la Clase Alpiste son los siguientes:





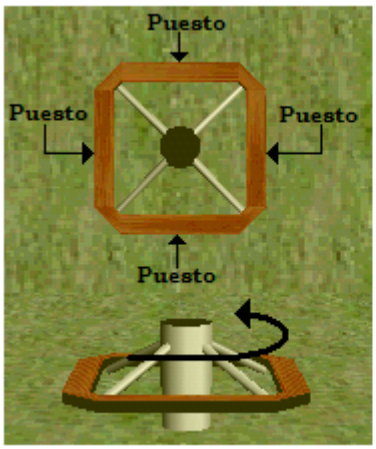
Nativos	Descripción
Banca, ~Banca	El constructor recibe la posición de la banca y la orientación hacia donde da el frente

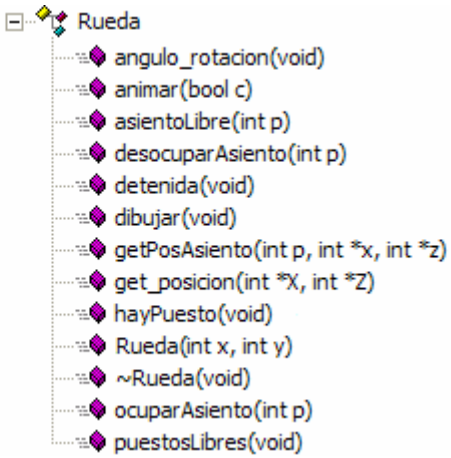
Informativos	
dirección	Indica en cual dirección (norte, sur, este, u oeste) da el frente
get_asientos_libres	Indica cuales asientos están libres (si es que los hay)
horizontal	Indica si la banca es paralela al eje de las X o no
get_posicion	Devuelve la posición de la banca dentro del parque
Acción	
pararse	Le indica a la banca que un determinado asiento se ha desocupado
sentarse	Le indica a la banca sobre el asiento que se está ocupando

4.2.1.4 Rueda

Este es uno de los dos objetos inteligentes diseñados para ofrecer un grado más de interacción con el humano virtual, incorporando una componente de animación que proporciona más dinamismo a la simulación. La Rueda ofrece hasta 4 asientos en el que los humanos se sientan para girar alrededor del eje de la rueda, lo que les sirve de diversión. Una característica interesante de este tipo de objetos es que en su interacción con el agente, también altera el estatus de este último, cambiando su posición, por ejemplo.

Los métodos que forman la Clase Rueda son los siguientes:





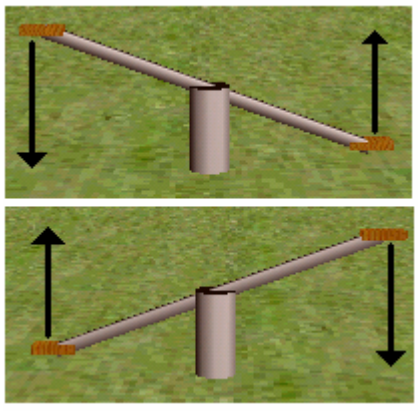
Nativos	**Descripción**
Rueda, ~Rueda	El constructor recibe la posición de la rueda dentro del parque
Informativos	
angulo_rotacion	Devuelve el ángulo de rotación de la rueda alrededor de su eje, para que el agente pueda actualizar su posición sobre ella, o conocer la posición de los asientos
asientoLibre	Permite saber si uno de los 4 puestos esta libre o no

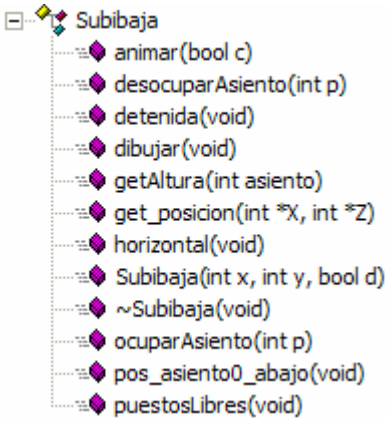
detenida	Devuelve el estado de la rueda, si todavía esta girando o ya se ha detenido
getPosAsiento	Para un asiento en particular, devuelve las coordenadas en la que se encuentra, para que el agente se pueda dirigir a él y sentarse
get_posicion	Retorna la posición de la rueda dentro del parque
hayPuesto	Informa si la rueda dispone de puestos libres o no
puestosLibres	Indica cuales puestos están desocupados
Acción	
animar	Activa o desactiva el movimiento de rotación de la rueda
desocuparAsiento	Marca un asiento como libre
ocuparAsiento	Marca un asiento como ocupado

4.2.1.5 Subibaja

Finalmente, el subibaja es el otro elemento que proporciona mayor dinamismo a la simulación, siendo el más complejo de los objetos inteligentes por su mecánica de interacción. Al igual que la Rueda, es auto animado; pero su animación consiste en subir y bajar alternativamente sus asientos. El subibaja constituye otro elemento de juego para los humanos virtuales, pero a diferencia de la rueda (la cual puede ser activada para jugar por un solo agente), el uso del subibaja requiere de la colaboración de dos agentes para poder activarla para jugar. Un solo agente puede cambiar la posición de los asientos del subibaja; sin embargo, requiere de la colaboración de otro humano virtual para poder jugar sobre él. El reto que plantea la interacción con este objeto inteligente es la sincronización que debe haber entre los agentes para su uso.

Los métodos que forman la Clase Subibaja son los siguientes:





Nativos	Descripción
Subibaja, ~Subibaja	El constructor recibe la posición del subibaja dentro del parque, y si esta orientado paralelo al eje de las X o no
Informativos	
detenida	Indica si la animación de está o no activada
getAltura	Devuelve la altura a la que esta el asiento indicado, para que el agente pueda ajustar la suya (si se encuentra sentado sobre dicho asiento)
get_posicion	Devuelve la posición del subibaja dentro del parque
horizontal	Indica si el subibaja esta paralelo al eje de las X o no
pos_asiento0_abajo	Indica si el asiento de la derecha esta abajo o no
puestosLibres	Indica cuales asientos están desocupados
Acción	
animar	Activa o desactiva la auto animación del subibaja
desocuparAsiento	Marca como libre un asiento específico
ocuparAsiento	Marca como ocupado un asiento específico

4.2.2 Agentes Virtuales

En este punto solo se hará una breve introducción al diseño de los agentes, ya que en las siguientes secciones se discutirán los detalles de su implementación.

Como se describiera en la Sección 2.1.1 sobre el modelado de los AVAs, estos deben ser diseñados de manera de facilitar su identificación, y permitir su animación a través de controles sobre sus partes móviles.

El modelado físico de un AVA se ha dividido en dos niveles. Un nivel base, donde se establece su apariencia (vértices, polígonos, texturas) y movilidad (conjunto de funciones que permiten un control primario sobre sus partes móviles). Y un segundo nivel, formado por una colección de métodos que, mediante combinaciones de las funciones del nivel básico y usando keyframing (para ahondar sobre ésta y otras técnicas de animación, ver Sección 2.1.2), proporcionan movimientos mas elaborados para la animación, como el volar del ave, o el caminar del humano virtual.

A continuación se presentan los detalles del modelado de los AVAs en su nivel básico.

La Figura 4.8 muestra el diseño del Ave Virtual, y sus capacidades de movilidad. Tal como se aprecia en la figura, el ave tiene un limitado espectro de movimiento, pero el mismo es suficiente para poder simular el comportamiento básico que lo caracteriza. El ave esta habilitada para mover

sus alas, balancearse hacia los lados, inclinarse al frente, y esconder sus patas.

El modelado del humano virtual es mucho más elaborado en cuanto a sus capacidades de movimiento se refiere. La Figura 4.9 muestra todos sus puntos de movilidad, donde cada uno de ellos aporta por lo menos un grado de libertad para la animación del agente. Como puede verse de la figura, el cuello es el que aporta más flexibilidad de movimientos. Con todos estos grados de libertad se pueden conseguir un buen número de animaciones que contribuyen a la vistosidad de la simulación y a incorporar comportamientos más complejos.

En la siguiente sección se presenta el Sistema de Control Físico, donde se dan los detalles de la implementación de los AVAs.

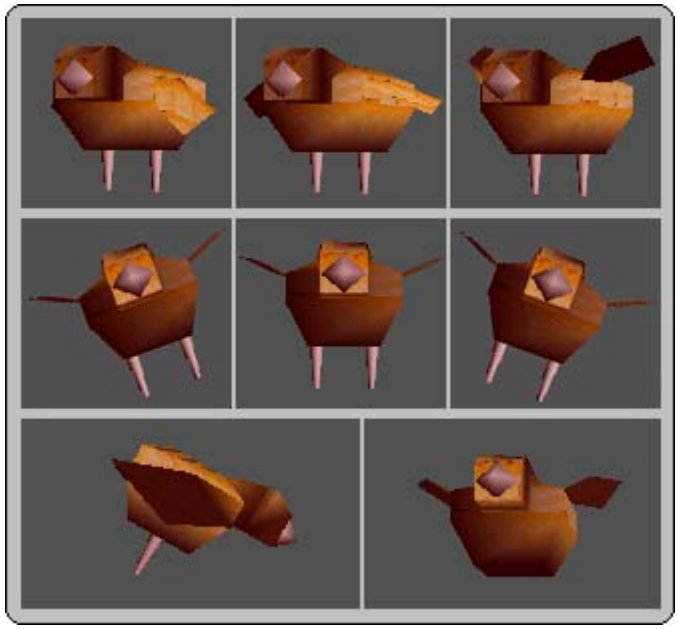


Figura 4.8. Ave Virtual

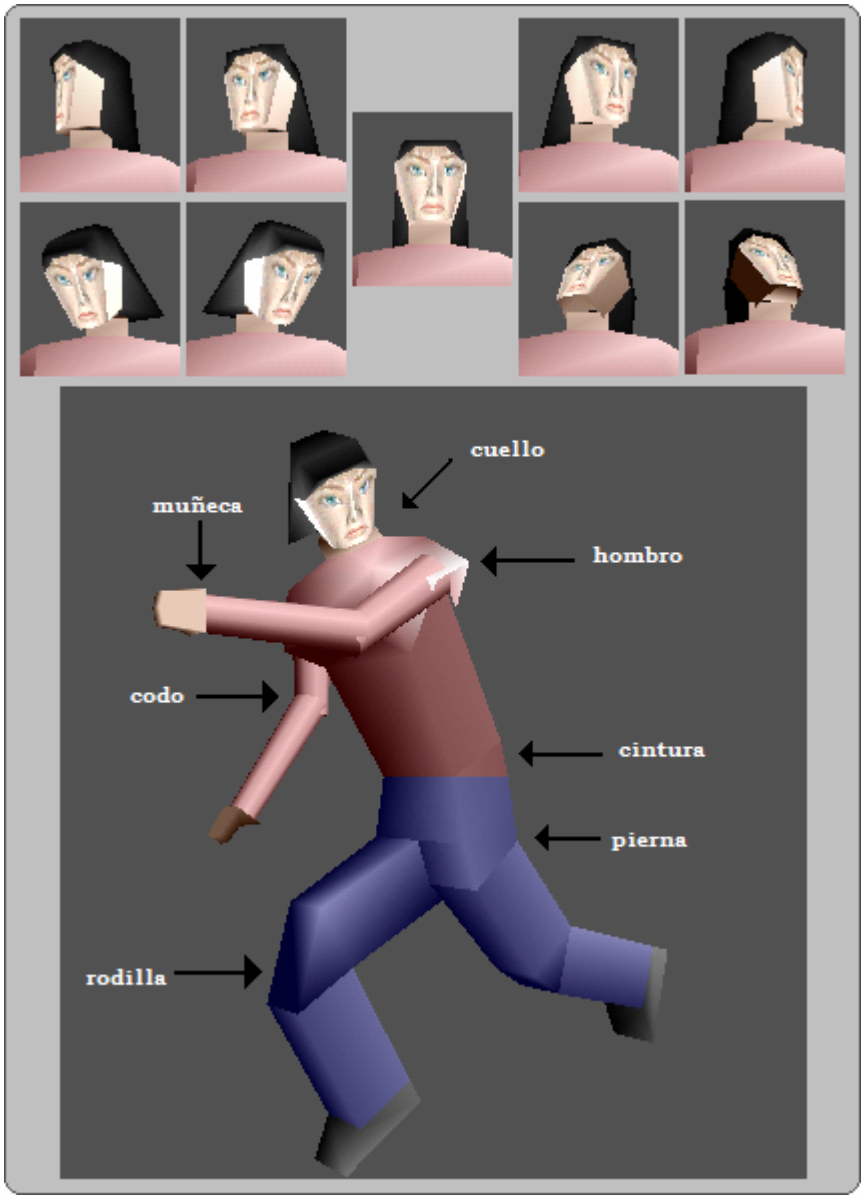


Figura 4.9. Humano Virtual

4.3 Sistema de Control Físico

Acorde al diseño MaGeS (Figura 4.1), el Sistema Agente Virtual Autónomo es representado por una Clase AVA que implementa fundamentalmente al Sistema de Control Físico (SCF), reuniendo un conjunto de atributos y métodos que definen tanto al Subsistema de Movimiento como al de Percepción; además contiene métodos especiales que permiten la comunicación con un modulo externo correspondiente al Sistema de Control del Comportamiento (SCC).

La Clase AVA, para los humanos virtuales, se ha implementado con 89 métodos para diferentes propósitos (y con solo 18 para las aves), que se agrupan en 4 categorías: 1) de diseño, 2) de movimientos básicos y compuestos (como los descritos en la sección anterior), 3) informativos, y 4) de comunicación con el SCC.

En la categoría de Diseño (solo aplicada a los humanos virtuales) se encuentran los métodos destinados a cambiar la apariencia del modelado del humano para conseguir varios personajes; permitiendo cambiar su tipo (si se quiere que sea hombre, mujer, o niño), o el color de su ropa.

Los movimientos básicos fueron comentados en la sección anterior (referente a Agentes Virtuales). Entre los movimientos compuestos para el ave tenemos: aletear, caminar, comer, dormir, y volar. Los movimientos compuestos para el humano son: caminar, animación para dos tipos de ejercicios físicos, leer, sentarse, posición de fatiga, bajar un asiento del subibaja, posición de sentado en la rueda, y recuperar postura original.

Los métodos informativos, al igual que para los objetos inteligentes, proporcionan información sobre el estado del agente, tal como su ubicación dentro del entorno, la dirección hacia donde miran, y la acción que se encuentran realizando.

Por último, los métodos de comunicación deben garantizar la transferencia de datos bidireccional entre el SCF y el SCC. Dado que en la implementación desarrollada para este trabajo, el SCC fue creado como una DLL (librería de enlace dinámico, Figura 4.5), el SCF tiene acceso, no solo al espacio de memoria del SCC, sino también a ejecutar sus funciones mediante llamadas directas.

La Figura 4.10 muestra un modelo general de la implementación de la Clase AVA. El método constructor crea un objeto de la Clase AVA con características específicas que lo personalizan, por ejemplo, rango de visión, resistencia al cansancio, capacidad de memoria, apariencia externa, etc. El método dibujar comunica al Sistema Motor toda la información necesaria para el dibujado del agente en su estado actual. Y el método animar, ejecuta un ciclo interno de animación del AVA, tal como se vio al final de la Sección 4.1.2.

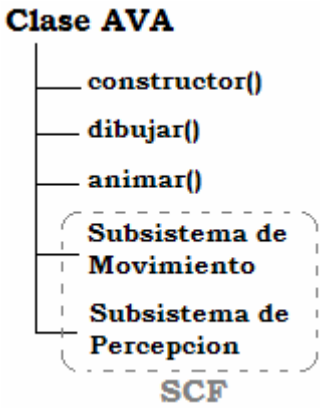


Figura 4.10. Clase AVA

Las siguientes secciones describen el funcionamiento de los subsistemas del SCF, a través de los principales métodos que los componen.

4.3.1 Subsistema de Percepción

Para que un agente virtual pueda exhibir un comportamiento autónomo debe ser capaz de percibir su entorno; esto es, identificar la información, recolectarla, y transformarla en conocimiento para actuar acorde a ella.

Poder identificar los elementos que componen el entorno implica que los AVAs tienen (o deben tener) conocimiento previo del tipo de elementos con que se pueden conseguir; y que cada elemento posee un identificador único que lo diferencia de otros similares. Por otro lado, el tipo de información que se toma del entorno dependerá de los sentidos simulados que posea el agente; y para su transformación en conocimiento, debe transferirlo al SCC.

Para cada elemento percibido se almacena en el espacio de memoria del SCC la siguiente información: a) identificador del sentido que lo percibió, b) identificador del elemento percibido, y c) información sobre el elemento, tal como su posición y estatus.

La siguiente sección presenta un nuevo esquema de codificación general para la identificación de los elementos que participan en una simulación. Posteriormente, se describen los mecanismos para simular la visión y el oído, como medios de percepción para un humano virtual.

4.3.1.1 Esquema de Codificación de Elementos

El esquema de codificado presentado a continuación permite identificar de manera unívoca a todos los elementos del mundo virtual susceptibles a ser percibidos por un AVA. Una ventaja adicional de dicho esquema, es que posibilita establecer relaciones de similitud entre los elementos.

Inspirados en la literatura sobre algoritmos genéticos, llamaremos *cromosoma* a la representación codificada de un elemento del mundo virtual. Estrictamente hablando, un *cromosoma* es una colección de *tramos*, los cuales a su vez están formados por una secuencia de campos llamados *genes*, como se muestra en la Figura 4.11.

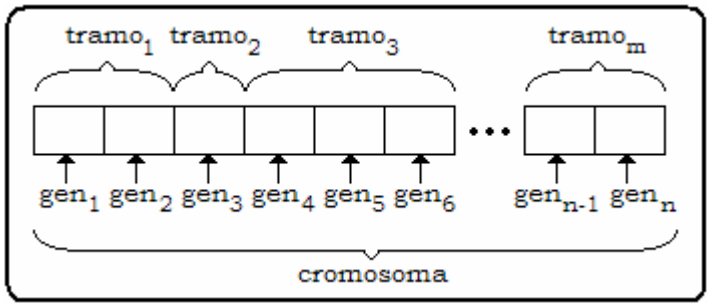


Figura 4.11. Estructura de un Cromosoma

Cada tramo de un cromosoma corresponde a una cierta característica del elemento a identificar, siguiendo una estructura jerárquica como la mostrada en la Figura 4.12.

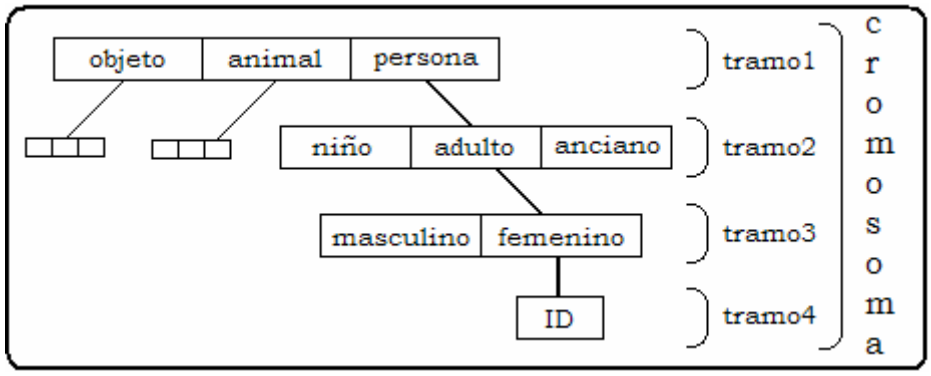


Figura 4.12. Ejemplo de la jerarquía de los tramos de un cromosoma

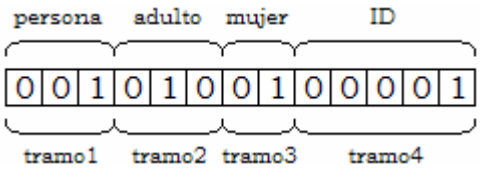
En el ejemplo mostrado en la Figura 4.12, se utiliza la estructura de cromosoma para identificar una amplia variedad de elementos del mundo virtual. Para este ejemplo, cada cromosoma esta formado por 4 tramos. El primer tramo identifica al elemento de manera general. Los subsiguientes tramos van complementando la información del anterior aportando nuevas

características para su identificación, hasta llegar al último tramo, el cual añade un valor que lo hace único.

Siguiendo con el ejemplo de la Figura 4.12, supongamos que cada tramo esta formado por 3, 3, 2, y 5 genes para cada uno de ellos, respectivamente; y cada $gen \in \{0,1\}$. Con lo que se podría tener el siguiente esquema de codificado:

Tramo1 (género del elemento)
[1 0 0] = objeto [0 1 0] = animal [0 0 1] = persona
Tramo2 (tipo, asociado al género persona)
[1 0 0] = niño [0 1 0] = adulto [0 0 1] = anciano
Tramo3 (sexo)
[1 0] = masculino [0 1] = femenino
Tramo4: cualquier combinación de ceros y unos para sus 5 genes.

Con este esquema, cualquier mujer quedaría codificada, utilizando solo los tres primeros tramos, con [0 0 1 0 1 0 0 1]; pero para identificar una mujer virtual de manera única se debe incluir un código que ninguna otra mujer posea (tramo4). Finalmente, la mujer virtual numero 1 quedaría codificada como sigue:



La estructura tipo árbol del esquema propuesto permite codificar a todos los elementos del entorno, y formar los cromosomas sencillamente estableciendo un recorrido que va desde la raíz (tramo1) hasta cada hoja (último tramo).

Una ventaja adicional que se desprende de este esquema es que posibilita el establecer criterios de similitud, o discrepancia, entre los elementos que identifican.

A continuación se define una función *Error* para medir el grado de discrepancias (diferencia) entre dos cromosomas, donde $A_i^j \in [0,1]$ y $B_i^j \in [0,1]$, denotan al gen i del tramo j de los cromosomas A y B, respectivamente, entonces,

$$Error(k, j, A, B) = \frac{1}{k} \sum_{i=1}^{k} \left| A_i^j - B_i^j \right| ,$$

y donde k representa la cantidad de genes del tramo *j*.

Con esta medida de discrepancia, podemos decir que dos cromosomas son similares hasta el tramo m si $Error(k_i, i, A, B) = 0$, para $i = 1 \ldots m$.

Para ilustrar mejor el uso de la función *Error*, veamos el siguiente ejemplo:

Sea una estructura de cromosoma de 3 tramos, con 3, 2 y 2 genes para cada tramo, respectivamente; y dados los siguientes cromosomas:

$$A = [1001011],\ B = [0101111],\ C = [1000000],\ \text{y}\ D = [0010111]$$

Al comparar los cromosomas *B*, *C*, y *D*, con el *A*, tenemos,

Error(3,1,A,B)=2/3	*Error(2,2,A,B)*=1/2	*Error(2,3,A,B)*=0
Error(3,1,A,C)=0	*Error(2,2,A,C)*=1/2	*Error(2,3,A,C)*=1
Error(3,1,A,D)=2/3	*Error(2,2,A,D)*=1	*Error(2,3,A,D)*=0

Por el sistema de jerarquías de los tramos, los cromosomas mas parecidos son el *A* y el *C*, ya que se comienzan a diferenciar en el segundo tramo. Si tomásemos la interpretación dada en la Figura 4.12 para el tramo1, ambos cromosomas corresponderían a un objeto, mientras que el B y el D estarían asociados a un animal y a una persona, respectivamente.

Para la implementación del parque virtual se empleó una estructura de cromosoma de 4 tramos, con 3, 3, 1, y 10 genes por tramo, respectivamente. El significado de la codificación de cada tramo es la siguiente,

Tramo1	Tramo2	Tramo3
100 = objeto	100 = banca 010 = rueda 001 = subibaja	0 = funciona 1 = no funciona
110 = vegetal	100 = árbol	1 = grande
010 = animal	100 = ave 010 = perro	0 = pequeño 1 = grande
011 = persona	100 = niño 110 = adulto 010 = anciano	0 = masculino 1 = femenino

Tabla 4.1. Esquema de codificado para los elementos del parque virtual

El Tramo4 es utilizado como identificador.

Esta codificación es suficiente para representar todos los elementos del parque virtual, y posee las siguientes características:

- No es necesario forzar la interpretación de todas las combinaciones de genes, pudiendo utilizar solo las requeridas; por lo que es factible incorporar nuevos elementos al parque sin cambiar el tamaño de los tramos ni su interpretación, cuando sea necesario.
- Los tramos codifican con más realismo la distancia a la que se encuentran unos elementos de otros. Por ejemplo, la persona esta más cerca de parecerse a un animal que a un objeto; y un adulto esta a igual distancia del niño que del anciano, siendo mayor dicha distancia que la del niño al anciano.
- Aunque la longitud del tramo4 permite codificar hasta 2^{10} elementos que compartan los mismos genes para los tres primeros tramos, puede resultar más conveniente utilizar un gen para cada elemento; con lo que solo se pueden tener, por ejemplo, hasta 10 árboles, ruedas, o niños. Además, el tamaño de este tramo puede variar sin cambiar la interpretación.

La utilidad práctica de este esquema de codificación se discutirá con mayor detalle en el siguiente capítulo.

4.3.1.2 Vista

Para simular el sentido de la vista, la Clase AVA dispone de un método `mirar`, que tiene acceso a un apuntador de cada lista dinámica que contienen los elementos del parque. Con acceso a dichas listas, el agente puede consultar la posición y el área de la envolvente convexa de cada elemento para determinar si se encuentra dentro de su punto de visión.

Cada AVA posee un *rango* y un *ángulo* de visión que definen un área de visibilidad para el agente, limitando su capacidad para ver objetos fuera de dicha área. El objetivo de limitar su visión consiste en lograr comportamientos mas realista, al tener, por ejemplo, que buscar objetos que no ven a simple vista, o a cambiar de objetivos por nueva información visual de que dispongan; tal como ocurre en la realidad. Adicionalmente, para el caso del humano virtual, el área de visión es dividida en dos sectores, un área de visión clara (donde los elementos son identificables), y otra de visión difusa (donde solo se conoce su posición, pero no su identificación). La Figura 4.13 muestra lo antes expuesto.

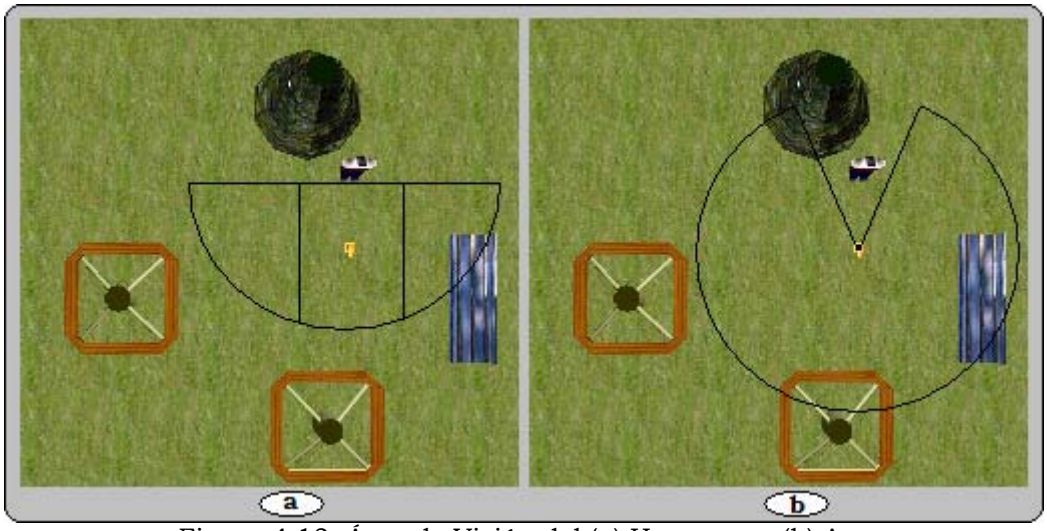


Figura 4.13. Área de Visión del (a) Humano, y (b) Ave

El medio circulo de la Figura 4.13(a) muestra el área de visión del humano, donde el radio de la circunferencia corresponde al *rango* de visión del agente. Los 180 grados de visibilidad están divididos en dos áreas, la central correspondiente a la visión clara, y el de los extremos, correspondiente al sector de visión difusa. Por su limitado rango de visión, el humano no es capaz de percibir las dos ruedas que tiene relativamente cerca; sin embargo, el ave que tiene en frente cae dentro de su área de visión clara, por lo que es capaz de identificarla plenamente, al contrario de la banca, que al estar (aunque sea parte de ella) dentro del área de visión difusa, el agente sabe que hay "algo" allí, pero no es capaz de identificarlo, por lo que solo lo considera un obstáculo al momento de definir una ruta a seguir.

Para el caso del ave, su ángulo de visión es mucho más amplio, 315 grados; y toda el área se considera de visibilidad clara. Como puede verse en la Figura 4.13(b), el ave virtual no alcanza a ver a una de las ruedas, por quedar fuera de su rango de visión, y al humano, por tenerlo justo detrás. Sin embargo, cuando un humano esta lo suficientemente cerca del ave, aunque ésta no lo vea, se ha dispuesto que lo perciba para dar mas dinamismo a las simulaciones a través de las persecuciones de aves.

Otro aspecto a considerar para la vista es la obstrucción de la visión por obstáculos. Con esto se pretende lograr que si dos elementos caen dentro del área de visibilidad de un agente, uno de ellos pueda ocultar al otro, según el punto de vista del agente. Para tal fin, se lanza una recta desde el punto de visión del observador a cada vértice de la envolvente convexa del objeto observado; si algún punto de cada recta toca la envolvente convexa de otro elemento antes que la del observado, este último no será visible para el

observador. La envolvente convexa de un elemento es el paralelepípedo de menor área que lo recubre completamente.

La Figura 4.14 ejemplifica lo antes explicado, donde la visión de un hombre es obstruida por un niño, lo que le impide ver una de las dos aves que tiene en su área de visión clara.

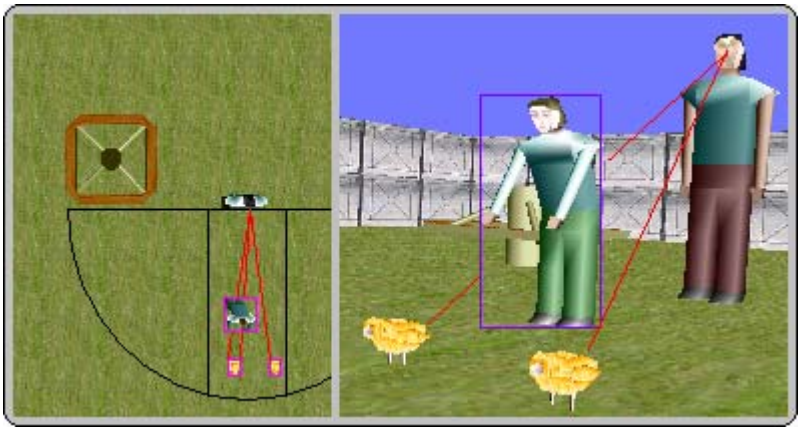


Figura 4.14. Obstrucción de la Visión

El procedimiento descrito para la detección de elementos ocultos es simple, pero suficiente para lograr los efectos deseados, dadas las características del entorno de simulación.

4.3.1.3 Oído

Igual que sucede para los elementos del entorno, los diferentes sonidos deben identificarse (codificarse) para poder ser reconocidos por los AVAs. Por otro lado, cuando un agente o un objeto emite un sonido, éste pasa ha encontrarse en el entorno, de donde otros agentes pueden "escucharlo".

Para implementar el sentido de la audición, se define un área de memoria global (`buffer_oido`) al que todos los AVAs tienen acceso. Cada entrada del buffer es una estructura con los siguientes campos: código del sonido (identifica si es un ruido o una frase hablada), código del mensaje (en caso de ser una frase hablada, especificaría cual), intensidad (hasta que distancia se podría escuchar desde su punto de origen), identificador del emisor (quien lo produce), identificador del destinatario (en caso de ser una frase hablada, a quien va dirigida), y las coordenadas del emisor (para saber desde que punto se emite).

Para "escuchar", un AVA debe revisar cada entrada en el `buffer_oido`. Si encuentra un código de sonido correspondiente a una frase, verifica que

él es el destinatario y que se encuentra dentro de la distancia a la que la puede escuchar. Siendo así, toma la información y la transfiere al SCC, tal como lo hace el sentido de la vista. Si el código del sonido corresponde a un ruido, el agente solo verifica estar a dentro del área en que puede escucharlo, para enviarlo a su SCC.

A cada elemento capaz de emitir un sonido le corresponde una entrada en el `buffer_oido`, la cual actualiza durante su turno en cada ciclo de simulación.

4.3.2 Subsistema de Movimiento

El Subsistema de Movimiento es el encargado de ejecutar las acciones motoras del agente, necesarias para alcanzar sus objetivos; y para ello cuenta con una colección de rutinas (brevemente comentadas al comienzo de la Sección 4.3) que permiten el control sobre partes específicas de su cuerpo. A continuación se listan las rutinas asociadas al Subsistema de Movimiento:

Los métodos listados abarcan una amplia variedad de movimientos y posturas, así como la posibilidad de simular la capacidad del habla, bien sea a través de exhibir una frase escrita en una ventana de dialogo o reproduciendo un archivo de audio.

Las órdenes para activar un método proceden del Sistema de Control del Comportamiento (ver Figura 4.1), el cual determina las acciones físicas requeridas por el agente, y que el Subsistema de Movimiento se encarga de llevar a efecto. El mecanismo de comunicación entre ambos sistemas será discutido en detalle en el siguiente capitulo.

El diseño incremental de la arquitectura MaGeS permite, por ejemplo, que se puedan añadir más métodos al Subsistema de Movimiento para ofrecer una mayor gama de expresiones corporales, sin que esto afecte el desarrollo o el desempeño del resto de las componentes de la arquitectura.