VNIVERSIDAD NACIONAL AVTÓNOMA DE MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

EVOLUCIÓN DE LOS COMPORTAMIENTOS DE UN AGENTE UTILIZANDO ALGORITMOS GENÉTICOS EN AMBIENTES VIRTUALES

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN INGENIERÍA

P R E S E N T A:

ENRIQUE JAIMES ISLAS

DIRECTOR DE LA TESIS: DR. JESÚS SAVAGE CARMONA

MÉXICO, D.F. **2012.**

Dedicado a mis padres, Olivia Islas Gómez y Enrique Jaimes Juárez (Q.E.P.D.), pues sin ellos no habría rastro de mí en este mundo.

A Priscilla Hernández Rico por ser mi copiloto inseparable en este gran viaje llamado vida.

Agradezco a la UNAM, mi alma máter, por permitirme llevar sangre azul y oro en las venas.

Al Dr. Jesús Savage Carmona por su confianza y apoyo académico.

A CONACYT por el apoyo económico sin el cual no habría podido concluir mis estudios.

Al proyecto PAPIIT IN117612.

Índice de contenido

Resumen

Se presenta un método para el diseño y mantenimiento de comportamientos para agentes virtuales representados con máquinas de estados de forma automática. Este método, basado en el uso de algoritmos genéticos, busca generar comportamientos tomando en cuenta únicamente la estructura de un agente, compuesta por el conjunto finito de acciones que puede realizar y el de estímulos que puede percibir, una representación del entorno donde dicho agente se desenvuelve y un objetivo a cumplir.

Se genera un conjunto de comportamientos aleatorios que serán operados por un algoritmo genético con el fin de simular un proceso de evolución refinando el comportamiento para el objetivo dado, a través de un número específico de generaciones. El proceso de evolución se realiza sobre máquinas de estados finitos que ejecutará un agente móvil capaz de navegar por el entorno con un extintor, recargarlo y accionarlo para lograr el objetivo, apagar un conjunto de flamas presentes en dicho entorno con la mejor cantidad posible de recargas. Los comportamientos son evaluados con ayuda de un sistema de simulación y posteriormente se implementan en agentes inmersos en un entorno virtual. Al final del trabajo se reportan los resultados de los experimentos realizados para un agente y entorno específicos, tanto para la representación con máquinas de Moore como de Mealy.

1. Introducción

La graficación por computadora ha sido una disciplina con un crecimiento muy acelerado en los últimos años, principalmente en el área de los *videojuegos* debido al auge que ha tenido esta industria desde sus inicios en los años 70. En la medida que aparece hardware más poderoso y barato, es posible crear representaciones gráficas computacionales cada vez más complejas. Actualmente la interactividad y la retroalimentación son factores obligados más que deseables en estas aplicaciones, esto implica, entre otras cosas, producir y analizar imágenes y realizar cálculos matemáticos para física e inteligencia artificial a gran velocidad.

El desarrollo de los *ambientes virtuales* se ha impulsado por diversas fuentes, sin embargo, el principal catalizador de su crecimiento ha sido la industria de los juegos por computadora. Los grandes presupuestos utilizados para el desarrollo de videojuegos son muchas veces justificados por la garantía de ganancias económicas, lo cual ha hecho que disciplinas como el cómputo gráfico, la inteligencia artificial y la creación de ambientes virtuales crezcan enormemente en manos de sus desarrolladores. Es por esto que al hablar de ambientes virtuales es necesario tomar como referencia a los videojuegos, ya que las compañías dedicadas a la creación de ellos son la punta de lanza en investigación y desarrollo de gráficos por computadora, simulaciones de física, mundos virtuales, y creación de los mejores dispositivos de interfaz humano-máquina que van más allá de los *controles* de mando clásicos [3].

Los *videojuegos* son un tipo especial de *ambiente virtual* en los que se utiliza una terminología especifica por ejemplo: a la persona o personas que interactúan con dicho ambiente se les llama jugadores, al conjunto de medios por los cuales el ambiente introduce metas a los jugadores se le denomina dinámica de juego o *gameplay,* y a la actividad de interactuar con el ambiente se le conoce como jugar. Este vocabulario se utilizará indistintamente a lo largo de este trabajo.

1. Introducción

Parte de los esfuerzos por construir *ambientes virtuales* de mejor calidad consiste en crear mejores algoritmos y técnicas para modelar las acciones de los personajes controlados por la computadora, para esto se han utilizado satisfactoriamente varias técnicas de inteligencia artificial. Una de las más relevantes son las máquinas de estados finitos o FSM (por sus siglas en inglés) que también han sido utilizadas para resolver problemas en diversas áreas, como el diseño de circuitos lógicos, la construcción de compiladores, robótica y en el desarrollo de videojuegos y ambientes virtuales. Las máquinas de estados finitos proveen de una forma de modelar sistemas de comportamiento con base en un conjunto finito de estados, transiciones entre ellos y acciones de entrada y salida.

El presente trabajo se basa en el uso de máquinas de estados finitos, que es una técnica sencilla, de fácil implementación y que ha sido probada con éxito en numerosas aplicaciones en videojuegos comerciales. Los comportamientos de cada agente virtual estarán determinados en gran medida por el conjunto de acciones que éste puede realizar y los valores que pueden tomar sus sensores. Estos sensores determinan las transiciones entre estados, por ejemplo si un agente se encuentra en estado de "caminando" y sus sensores le indican que hay una pared cerca, un comportamiento esperado sería que el robot deje de caminar cambiando a un estado "detenido" para no chocar.

El modelado de comportamientos para *agentes virtuales* tiene mucho en común con la robótica en partes como el control de acciones y la planeación de rutas, ya que en una gran cantidad de ocasiones los mismos métodos con pocos cambios son aplicables a ambas disciplinas. Sin embargo la robótica se enfrenta a problemas muy complejos que no afectan a los *agentes virtuales* ya que estos se desempeñan dentro de un entorno en donde sus acciones, percepciones y su localización están controlados. A pesar de esto y gracias al avance de las técnicas de construcción de ambientes virtuales y el hardware especializado en gráficos, se pueden construir ambientes de simulación de bajo costo que permiten probar algoritmos de manera que se puedan detectar fallas antes de implementarlos en robots reales.

Comúnmente es tarea del desarrollador el diseñar los comportamientos para cada agente de la forma que a él le parezcan mejores, sin embargo es posible implementar técnicas o algoritmos que realicen este proceso automáticamente. En este trabajo se hace uso de los algoritmos genéticos debido a su capacidad de búsqueda en espacios de soluciones muy grandes y a la ventaja de poder trabajar con poblaciones en vez de un solo individuo. Así el sistema que se propone puede encontrar comportamientos deseables quitando el peso del diseño a los desarrolladores, además de que en un momento dado podría adaptar dichos comportamientos a cambios en el entorno o a la estrategia del jugador.

El trabajo está basado en los métodos evolutivos para máquinas de estados finitos generalizadas (*GFSM por sus siglas en inglés*) propuestos por Mattias Wahde en [25] y un trabajo reciente sobre comportamientos de evasión de obstáculos en robots móviles [19], así como la inteligencia artificial para videojuegos que desde hace tiempo ha utilizado máquinas de estados finitos con muy buenos resultados para el modelado de comportamientos en agentes controlados por la computadora. También el estudio de algoritmos meta-heurísticos aplicados a problemas de optimización combinatoria como herramientas para atacar aquellos cuyo espacio de soluciones es muy grande, en especial los algoritmos genéticos ya que trabajan con poblaciones de soluciones adaptándose satisfactoriamente al tipo de problemas en estudio.

1.1 Objetivos

Objetivo General: Desarrollar un método automático que encuentre comportamientos satisfactorios para agentes virtuales, utilizando máquinas de estados finitos y algoritmos genéticos. Para lograrlo se deben cumplir una serie de objetivos concretos:

- Diseñar un sistema de representación y ejecución de comportamientos para un agente virtual basado en máquinas de estado finitos de Moore y Mealy.
- Construir una representación simbólica de su entorno.
- Crear una codificación flexible de los comportamientos para poder ser operados por un algoritmo genético, así como sus operadores de selección, cruzamiento y mutación.

Crear un ambiente virtual de simulación donde los comportamientos encontrados sean asignados a un agente de prueba con el fin de evaluar el desempeño de cada uno de ellos.

- Construir un sistema que permita la comunicación entre el algoritmo genético y el ambiente de simulación.
- Hacer pruebas al sistema construido con algunos ejemplos específicos.

Se espera que los comportamientos estudiados mejoren progresivamente, partiendo de un conjunto de ellos generado aleatoriamente, hasta aquellos que cumplan de la mejor forma posible con los objetivos planteados.

1.2 Relevancia

Se propone un método para auxiliar al diseño de comportamientos para un agente de forma automática, evitando la complejidad de diseñarlos a criterio del programador, lo cual puede llevar a un esquema difícil de mantener para comportamientos muy complejos. Este método permite encontrar comportamientos muchas veces más robustos que los diseñados a mano [25], especialmente para agentes con un conjunto grande de acciones y transiciones cuyos comportamientos se pueden volver inmanejables debido a que el espacio de soluciones crece de forma exponencial en relación con el número de estados.

El sistema propuesto pretende dar un paso hacia el uso viable de los algoritmos genéticos en el desarrollo de comportamientos adaptables basados en máquinas de estados finitos para agentes inmersos en ambientes virtuales, como los utilizados en videojuegos.

1.3 Organización del trabajo

El presente trabajo está dividido en siete capítulos que tienen como objetivo explicar el proceso de desarrollo del sistema presentado, desde sus antecedentes hasta los resultados para casos particulares de estudio.

En el ***Capítulo 2: Antecedentes*** *s*e presenta una breve introducción resumiendo conceptos básicos necesarios para entender tanto la problemática como la solución propuesta a través de una perspectiva encaminada al desarrollo de ambientes virtuales y videojuegos. En este capítulo también se describe la necesidad de que los agentes en ambientes virtuales realicen acciones adecuadas en el momento adecuado para resolver problemas o cumplir tareas automáticamente, lo que es deseable para modelar personajes controlados por la computadora.

En el ***Capítulo 3: Análisis y diseño*** se hace un análisis del problema en estudio y una justificación de la resolución así como el diseño general de la solución y los componentes básicos del sistema. Estas partes del sistema pueden modelarse como procesos independientes con entradas y salidas específicas para posteriormente interconectarlos y resolver el problema deseado.

En el ***Capítulo 4: Agentes y el algoritmo genético*** se describe la estructura de los agentes utilizados, la representación simbólica del entorno en el que estarán inmersos y una forma de representar las máquinas de estados finitos que definen sus comportamientos que permita que sean operados por un algoritmo genético y utilizados por los agentes virtuales de forma eficiente. Se define también la estructura general del algoritmo genético utilizado, las clases que lo componen, algunas consideraciones para implementarlo en un ambiente virtual y también se muestran sus operaciones básicas de selección, mutación y cruzamiento.

El ***Capítulo 5: Construcción del ambiente virtual*** explica el procedimiento básico de creación del ambiente virtual de pruebas, de cada uno de sus módulos generales y las herramientas de software utilizadas. En él se simulan los agentes y su entorno con representaciones gráficas en tres dimensiones ejecutando los comportamientos encontrados por el algoritmo genético.

En el ***Capítulo 6: Pruebas y resultados*** se muestran algunos casos de prueba y sus resultados, en ellos se plantea una tarea a resolver así como la estructura del agente cuyo comportamiento se desea evolucionar y se ejecuta el algoritmo genético una cantidad suficiente de veces para calcular estadísticas básicas de su funcionamiento y también evaluar a los mejores individuos en instancias diferentes del problema y analizar sus resultados.

Por último, en el ***Capítulo 7: Conclusiones y trabajo futuro*** se hace un balance de la utilidad del sistema propuesto, se plantean también algunas mejoras y adiciones futuras al mismo con el fin de refinar el proceso de solución del problema en estudio y hacerlo más flexible.

2. Antecedentes

2.1 Ambientes virtuales

Un ambiente virtual es un sistema de visualización de imágenes interactivo enriquecido con elementos especiales como *dispositivos hápticos*, audio posicional, simulaciones físicas o de inteligencia artificial cuyo fin es crear en el usuario la ilusión de estar inmerso en un espacio sintético [8]. Estos ambientes virtuales describen potencialmente una nueva forma de comunicación humano-máquina y generalmente presentan una alternativa de bajo costo para entender, aprender y simular fenómenos complejos. Sus aplicaciones son muy diversas, por ejemplo en medicina se utilizan como una herramienta para realizar operaciones a distancia, en la milicia como una plataforma de entrenamiento para pilotos y para el entretenimiento. Adicionalmente la capacidad de crear personajes puramente virtuales da la posibilidad de implementar características de tipo visuales, físicas, auditivas e incluso comportamientos que no serían imposibles de realizar en un mundo físico.

- Para la construcción satisfactoria de ambientes virtuales, es necesario el uso de hardware clasificado en tres tipos:
- Periféricos o interfaces humano-máquina (Sensores) con el fin de detectar datos de entrada del usuario (teclado, mouse, controles de mando etc.) o bien señales de algún dispositivo externo (cámaras, micrófonos, sensores de movimiento etc.).
- Efectores, dispositivos de salida que tienen como objetivo general estimular los sentidos del usuario (pantallas estereoscópicas, audio posicional, retroalimentación a través de vibración etc.).
- Por último, para el caso de ambientes virtuales, es necesaria una pieza de hardware con un sistema de propósito específico que establezca una liga entre sensores y efectores para producir la experiencia deseada, es decir, la computadora donde se llevan a cabo los procesos de la simulación.

El software de un ambiente virtual por su parte debe realizar 3 tareas básicas:

- Proveer un mecanismo para representar gráficamente el entorno, los objetos y los personajes que componen el ambiente virtual. En el caso de los ambientes tridimensionales es necesario recurrir a técnicas de computación gráfica.
- Resolver y manejar las interacciones entre actores, objetos y entorno tales como su comportamiento, movimiento, leyes físicas y animación, entre otras.
- Manejar los datos de entrada/salida para crear la experiencia final del usuario utilizando sus acciones y la simulación del ambiente virtual.

Bajo este esquema, una computadora personal provee una plataforma básica para construir ambientes virtuales ya que el teclado y el mouse actúan como sensores, la pantalla y bocinas actúan como efectores y es posible desarrollar aplicaciones de simulación en ellas.

En conclusión, un *ambiente virtual* es un sistema computacional que reúne un conjunto de técnicas y características de hardware y software con el fin de crear una experiencia de *inmersión* en un entorno sintético real o imaginario. Este trabajo se centra únicamente en la parte gráfica y de inteligencia artificial, específicamente en la construcción de software para la visualización y el diseño de comportamientos de agentes virtuales que interactúan con el entorno y el usuario con el fin de enriquecer su experiencia final.

2.2 Videojuegos

Los videojuegos tienen muchas características en común con los ambientes virtuales ya que son sistemas de software que siguen el mismo modelo de sensor-simulador-efector expuesto anteriormente, pero con características adicionales: se presentan en diversas plataformas, desde computadoras personales hasta dispositivos móviles como teléfonos celulares y pasando por *consolas de juegos*. Los dispositivos de entrada de información varían en cada plataforma, para juegos en computadoras personales pueden ser mandos de control (*joysticks*) o incluso el mouse y el teclado, en *consolas* existen también mandos de control especializados inclusive con capacidad

de retroalimentación háptica. Las *consolas* de última generación como el Xbox 360, el Nintendo Wii y el PlayStation 3, por ejemplo, incorporan sensores de movimiento y técnicas de visión por computadora para enriquecer la experiencia del jugador.

La gran mayoría de los videojuegos presentan retroalimentación visual por medio de algún dispositivo de despliegue de video como una televisión o el monitor de una computadora independientemente de la plataforma. La interacción también se puede dar entre uno o varios jugadores humanos y la computadora e incorporan una serie de reglas y objetivos, los cuales el jugador debe cumplir mediante un conjunto de acciones disponibles.

Los videojuegos se han convertido en un medio muy rico para transmitir información y crear experiencias completas donde el jugador puede actuar y ser retroalimentado e incluso hacer su propia historia convirtiéndose así en un actor cuyas acciones y decisiones tienen repercusión en el ambiente virtual, dejando atrás el papel de observador que se limita a recibir información pasivamente. Este tipo de ambientes virtuales han demostrado cada vez más ser la narrativa del futuro pues pueden ser usados para fines más allá del entretenimiento como puede ser: la educación, la salud, fines sociales e incluso para crear conciencia política como es el caso de un juego llamado *Escape From Woomera* [9] que es una modificación (*mod*) de un conocido juego llamado Half-Life que trata temas reales sobre migración en Australia.

2.3 La Inteligencia Artificial en videojuegos

Los primeros videojuegos en los años 60 e inicios de los 70 incorporaban ya agentes controlados por la computadora, al principio los comportamientos estaban predefinidos y se repetían una y otra vez, posteriormente los juegos fueron más complejos ya que implementaban comportamientos con componentes aleatorias haciendo uso de los primeros microprocesadores con el objetivo de hacer dichos comportamientos un poco menos predecibles.

Herramientas como las máquinas de estado, árboles de decisión, métodos de búsqueda, robótica e incluso algoritmos genéticos, entre otras, han sido de gran utilidad para el desarrollo de Inteligencia Artificial (IA) para videojuegos, algunas con

mayor éxito que otras, irónicamente las técnicas más sencillas han sido las más satisfactorias al modelar agentes controlados por la computadora.

Actualmente la complejidad de los videojuegos y el tipo de mecánicas que incorporan hacen imposible el uso de algoritmos de fuerza bruta debido a las restricciones de hardware de las computadoras personales y consolas caseras, sin embargo esta restricción ha abierto la puerta a otras técnicas innovadoras de IA.

La IA que se utiliza en el desarrollo de videojuegos tiene características muy específicas comparadas con la IA académica, ya que sus objetivos son diferentes. Al desarrollar IA para videojuegos siempre se debe tener en cuenta el factor entretenimiento con todo lo que implica, a diferencia de la IA académica, donde regularmente se tiende a resolver los problemas de una manera óptima.

Los programadores de IA para videojuegos (Game AI), se enfrentan a condiciones peculiares de desarrollo: primero, trabajan con recursos de hardware limitados (generalmente con los que cuenta un jugador en su computadora personal, consola o dispositivo móvil) que varían dependiendo de la plataforma y la tecnología usadas. Segundo, ya que algunas de las habilidades de los agentes controlados por la computadora sobrepasan a las de los seres humanos, la IA implementada debe ser entretenida y equilibrada, de otra manera la mayoría de jugadores llegarán a sentirse frustrados o en desigualdad con agentes que nunca fallan en retos que requieren precisión, velocidad, problemas aritméticos y de ordenamiento o búsqueda (problemas que la IA puede resolver con facilidad) y, finalmente, un jugador debe de disfrutar la IA a la que se enfrenta, ésta debe poner a prueba sus capacidades y exigirle cada vez más pero nunca, como ya se dijo, debe ser invencible.

Si el desarrollador cumple con las condiciones anteriores entonces logrará que el usuario se sienta astuto, satisfecho y con ganas de seguir jugando, es decir, que su experiencia de juego sea cada vez mejor, en conclusión los jugadores humanos disfrutan jugar contra agentes que dan la ilusión de ser inteligentes.

Por otra parte existen situaciones que a los seres humanos nos parecen triviales, pero que para los desarrolladores y las computadoras son un verdadero reto, por ejemplo: el reconocimiento de caras, comunicarse por medio del lenguaje natural, crear agentes o entidades creativos, entre otras. En este tipo de tareas, los agentes controlados por computadora poseen una desventaja muy grande ante los jugadores humanos, debido a que las técnicas de IA desarrolladas para resolverlas no son lo suficientemente robustas o factibles de ser implementadas en un ambiente en tiempo real. Entonces, si se programan tareas donde la IA puede ser fácilmente superada por los humanos y se tendrá una IA predecible, tonta y por lo tanto aburrida.

Para combatir la situación anteriormente descrita, los programadores de inteligencia artificial para videojuegos se ven forzados a desarrollar aplicaciones que permitan enfrentar estos problemas. Un recurso utilizado por ejemplo, es el de incrementar los atributos de los personajes, como pueden ser puntos de vida de los enemigos, su velocidad o bien ignorando algunas reglas del juego para hacerlos más competitivos, esto es comúnmente conocido como *cheating* y es muy utilizado en una gran cantidad de videojuegos, no solo para poner en una situación competitiva a los agentes de la IA sino también, en muchos casos, para lograr mejor rendimiento del sistema. Un ejemplo de *cheating* es, hacer un análisis de todo el entorno en busca de obstáculos en un mapa arbitrario al inicio de la aplicación cuando se supone que los agentes no han explorado el mapa.

Hay quienes dicen que la inteligencia artificial en los videojuegos no debería hacer este tipo de "trampas", que debe de tener exactamente las mismas entradas de información y herramientas que un ser humano [21], sin embargo, hacerlo implicaría implementar técnicas muy demandantes como visión por computadora que se traducirían en mayores costos de desarrollo innecesarios.

Los agentes controlados por la computadora son capaces de escoger y combinar el conjunto de estrategias que los desarrolladores hayan implementado o les hayan enseñado, sin embargo, el jugador humano tarde o temprano las aprenderá y ya no encontrará reto alguno, por eso se hace necesario desarrollar técnicas de IA capaces de adaptarse a nuevos entornos o estrategias del jugador humano.

Es común pensar que mientras más compleja sea la inteligencia artificial en un videojuego, mejores serán los comportamientos de los personajes, sin embargo no siempre es cierto. Es importante encontrar los algoritmos correctos que generen comportamientos aceptables, y que sea factible su implementación en términos de tiempo, facilidad y rendimiento. Puede haber casos de implementaciones extremadamente complicadas que dan como resultado comportamientos aburridos o "tontos". Una técnica simple bien utilizada puede ser la más adecuada dependiendo de las necesidades del videojuego y el comportamiento específico.

2.4 Agentes en ambientes virtuales y videojuegos

En videojuegos *single-player* (de un solo jugador humano) gran parte de la experiencia de juego depende o está íntimamente relacionada con los personajes controlados por la computadora, estos personajes pueden modelarse como agentes, ya que perciben y actúan en un entorno dinámico, pueden ser tan sencillos como un simple jugador virtual que mueve una raqueta de ping-pong hacia arriba y hacia abajo o bien, tan complejos como verdaderos compañeros de juego que puedan ejecutar estrategias o evadir obstáculos (véase figura 2.1). Comúnmente estos agentes son conocidos como *NPCs* de sus siglas en inglés *Non-player Character* (personaje no jugador) y como su nombre lo dice, son personajes cuyos comportamientos están programados y son automáticos. Los roles que pueden tomar son muy diversos y están totalmente ligados al *gameplay*, por ejemplo un *NPC* puede ser un aliado ayudando al jugador a cumplir metas, puede ser un personaje hostil, los cuales suelen llamarse *enemigos* o bien podría comportarse como un guía de turistas o un narrador.

Al incorporar estos agentes virtuales a un ambiente es necesario dotarlos de algunos elementos básicos como: una representación gráfica, una voz o sonidos característicos, animaciones, funciones de movimiento para, por ejemplo, crear la ilusión de caminar o volar. Detalles como mirar fijamente un objeto que se mueve, moverse un poco más rápido y hasta el simple hecho de modificar su expresión facial pueden ayudar enormemente a que el usuario perciba a los agentes como inteligentes. Esto, junto con la ejecución de comportamientos que hacen lo correcto en el momento correcto son un gran paso para crear experiencias cada vez más ricas.

Figura 2.1: Personaje del videojuego *World of Warcraft. Blizzard Entertainment 2011.*

2.5 Autómatas

Un autómata finito es un dispositivo que acepta o reconoce un conjunto de elementos en Σ*, donde Σ es un alfabeto finito. A este subconjunto de Σ* se le llama lenguaje, al lenguaje ***L*** aceptado por un autómata ***M*** se le denota como ***M(L)***.Un *autómata finito determinístico* (AFD) se define como una quíntupla (Σ,Q,s_0,δ,F) donde:

Σ – Es un alfabeto finito de entrada.

Q – Es un conjunto finito de estados posibles.

s_0 – Denota el estado inicial, $s_0 \in Q$.

δ – Es la función de transición entre estados, $\delta: Q\times\Sigma \rightarrow Q$. Como entrada recibe un estado actual en Q y un elemento de Σ. Como salida produce un solo estado en Q al cual transitar.

F – Conjunto de estados finales o de aceptación, $F \subseteq Q$.

De esta forma, un autómata inicia en el estado s_0 y dada una cadena en Σ^*, el autómata leerá símbolo por símbolo empezando por el primero (si la cadena no es vacía) hasta llegar al último y se detendrá. Entonces, si el autómata se encuentra en un estado q y tiene como entrada el símbolo a, el siguiente estado será $\delta(q,a)$. Si el estado f en el que se encuentra el autómata después de leer la cadena completa cumple que $f \in F$, la cadena es aceptada, es decir, pertenece al lenguaje reconocido por dicho autómata. Si esta condición no se cumple, la cadena es rechazada.

En el caso de los autómatas *finitos no determinísticos* (AFND), también están definidos por la quíntupla (Σ,Q,s_0,δ,F) pero la función de transición δ está definida como:

$$\delta: Q \times \Sigma \rightarrow P(Q)$$

Donde $P(Q)$ denota al conjunto potencia de Q. Entonces, en un autómata finito no determinístico, dado un estado actual q y un símbolo leído a, existe un conjunto de estados posibles (subconjunto de Q) a los que se puede transitar. Por definición, todo AFD es un AFND pero no en sentido inverso.

2.6 Diagramas de estado

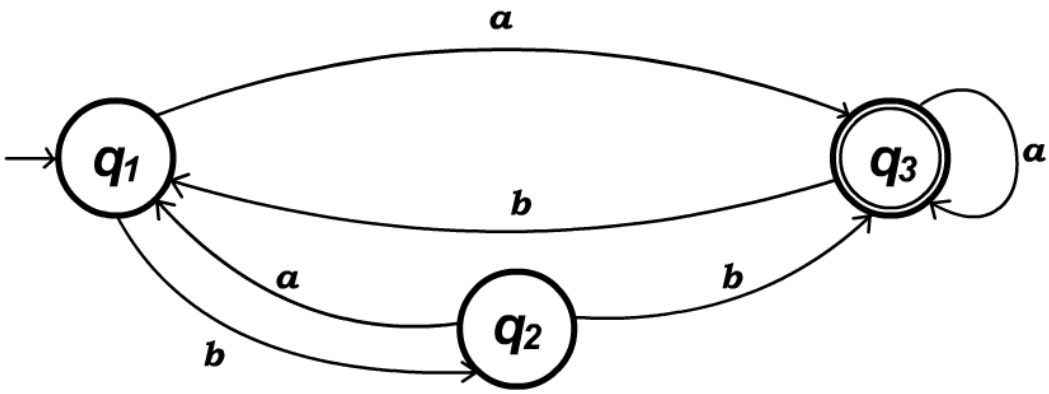
Se puede representar gráficamente a un autómata finito o una máquina de estados finitos por medio de un diagrama de estados, que es un grafo dirigido donde cada nodo representa un estado y si existe una transición de un estado q_i a otro q_j leyendo un símbolo a existirá un arco rotulado con a que parte del nodo que representa al estado q_i al que representa al estado q_j. Por lo general, los estados se dibujan como círculos simples a excepción del estado inicial al que se le antepone una flecha sin origen y los estados finales que se dibujan como círculos dobles.

Por ejemplo, al autómata definido por:

$\boldsymbol{\Sigma} = \{a,b\}$, $\boldsymbol{Q} = \{q1,q2,q3\}$, $\boldsymbol{s_0} = q1$, $\boldsymbol{F} = \{q3\}$

$\boldsymbol{\delta} = \{(q1,a,q3),(q1,b,q2),(q2,a,q1),(q2,b,q3),(q3,a,q3),(q3,b,q1)\}$

Le corresponde el siguiente diagrama de estados:



2.7 Máquinas de estados finitos

Una máquina de estados finitos (FSM) es un dispositivo matemático similar a un autómata finito determinístico pero con la diferencia de que produce salidas, cadenas de símbolos de un alfabeto de salida, en vez de solo aceptar o rechazar una cadena de entrada dada, es decir, su función de transición toma como entrada un estado actual y un símbolo del alfabeto de entrada y como salida entrega un estado final y un símbolo del alfabeto de salida o una acción. Las FSMs tienen un número finito de estados y solo pueden estar en uno de ellos a la vez. La idea general al utilizar FSM para representar comportamientos es el dividirlos en acciones básicas sencillas y ejecutar cada una de ellas en el momento indicado con base en sus entradas. Existen dos tipos de máquinas de estados cuyas diferencias son pequeñas pero importantes, las máquinas de Moore y las de Mealy.

2.7.1 Máquinas de Moore

Las máquinas de Moore (presentadas por Edward F. Moore en 1956) son máquinas de estados cuyas salidas dependen únicamente del estado actual, es decir, siempre que la máquina se encuentre en un estado dado, la salida será la misma independientemente de la entrada [15]. Formalmente una máquina de Moore se define como una séxtupla $(\Sigma, A, Q, s_0, \delta, \phi)$ donde:

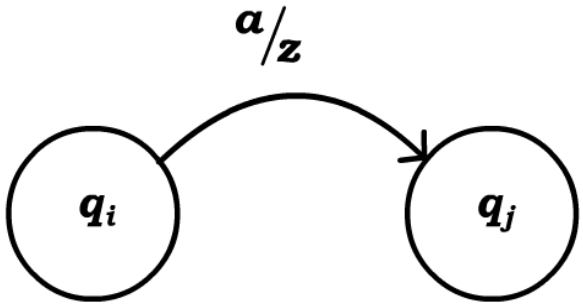
- $\boldsymbol{\Sigma}$ – Es el alfabeto de entrada
- $\boldsymbol{A}$ – es el alfabeto de salida.
- Al igual que los autómatas, tiene un conjunto finito de estados $\boldsymbol{Q}$ al que pertenece el estado inicial $\boldsymbol{s_0}$.
- $\boldsymbol{\delta}$ – Es la función de transición entre estados, $\delta: Q \times \Sigma \rightarrow Q$.
- $\boldsymbol{\phi}$ – Es llamada función de salida, $\phi: Q \rightarrow A$. Esta función relaciona cada estado con un símbolo de salida.

Una máquina de Moore, al igual que un autómata recibe una cadena de Σ como entrada y lee uno a uno los símbolos que la componen. Al leer un símbolo de entrada $a \in \Sigma$ en un estado q_i y antes de determinar el siguiente estado $\delta(q_i, a)$ se procesa la salida en q_i que está dada por $\phi(q_i)$. De esta forma al terminar de procesar la cadena de entrada se tiene una cadena de salida de la longitud de la cadena de entrada más uno ya que antes de leer el primer símbolo de entrada se tiene el primer símbolo de salida dado por $\phi(s_0)$. Por lo tanto todas las cadenas de salida de una máquina de Moore inician con el símbolo $\phi(s_0)$. En un diagrama de estados la expresión $\phi(q_i) = z$ se representa como:

2.7.2 Máquinas de Mealy

Estas máquinas presentadas por George H. Mealy en 1955 son máquinas de estados en la que su salida depende no solo del estado actual sino también de la entrada [14]. En este modelo la salida no se procesa al entrar en un estado sino en el momento de hacer una transición, es decir, la salida no se asocia a un estado, se asocia a una pareja estado/transición o al arco que la representa en un diagrama de estados. Al igual que las máquinas de Moore se definen con una séxtupla $(\Sigma, A, Q, s_0, \delta, \gamma)$, con la única diferencia que la función de salida γ es de la forma γ: Q×Σ → A. En otras palabras, la función γ toma el estado actual y un símbolo de entrada para producir la salida. Entonces en un diagrama de estados la transición dada por:

$\delta(q_i, a) = q_j$, $\gamma(q_i, a) = z$ se representa como:



El funcionamiento de las máquinas de Mealy es muy parecido al de las máquinas de Moore, solo que las funciones de transición y salida se deben evaluar en orden inverso, en las máquinas de Mealy se evalúa primero la función de transición y luego la de salida, ambas con los mismos argumentos. Las máquinas de Mealy no generan salida en el estado inicial si no hasta que se lee el primer símbolo de entrada por lo que la cadena de salida tiene la misma longitud que la cadena de entrada. Se puede consultar [1] para una referencia más completa sobre autómatas y máquinas de estado.

2.7.3 Máquinas de estados generalizadas (GFSMs)

Las máquinas de estados son útiles para representar comportamientos, sin embargo el hecho que utilicen alfabetos finitos puede suponer un problema para aplicaciones reales, por ejemplo, si se tiene un robot que lee un valor numérico real de uno de sus sensores o bien en el caso de que un agente virtual que perciba movimiento en mandos analógicos como los de las consolas de videojuegos actuales. Éstas deben extenderse para dotarlas de la capacidad de manejar entradas y salidas cuyo dominio es continuo, para ello se utilizan máquinas de estados generalizadas.

Las GFSMs no necesariamente utilizan alfabetos finitos, esto permite, manejar entradas y salidas continuas. Las GFSMs tienen un conjunto finito de N estados, a cada estado *i* le corresponde una acción (correr, saltar, disparar etc.) que puede ser arbitrariamente compleja y un conjunto de M_i transiciones condicionales hacia otros estados, estas condiciones pueden involucrar lectura de sensores, variables internas o cualquier cosa que el agente o robot pueda percibir.

Es importante diseñar una estructura común para estas condiciones con el objetivo de hacerlas más manejables. Si una de estas condiciones se satisface, la GFSM salta al estado correspondiente ejecutando su acción asociada. Si no se satisface ninguna condición, la GFSM permanece en el estado actual [25]. También se les puede asociar una prioridad a estas condiciones, así en el caso de que más de una se satisfaga en un momento dado, la GFSM tomará en cuenta la que tenga mayor prioridad.

Las GFSMs son fáciles de programar y mantener cuando son de tamaño razonable, de igual forma su ejecución consume pocos ciclos de procesador, muy valiosos teniendo en cuenta que la IA es solo una parte de todo lo que se tiene que procesar en un ambiente virtual. Otra característica de las GFMS es que son flexibles, permitiendo realizar cambios en su estructura de forma sencilla, ya sea a criterio del desarrollador o automáticamente con ayuda de un sistema externo. Teniendo clara la existencia de las GFSM y por qué es necesario generalizar la definición básica de una FSM, de aquí en adelante se nombrará como FSM a cualquier máquina de estados presentada durante el desarrollo de este trabajo.

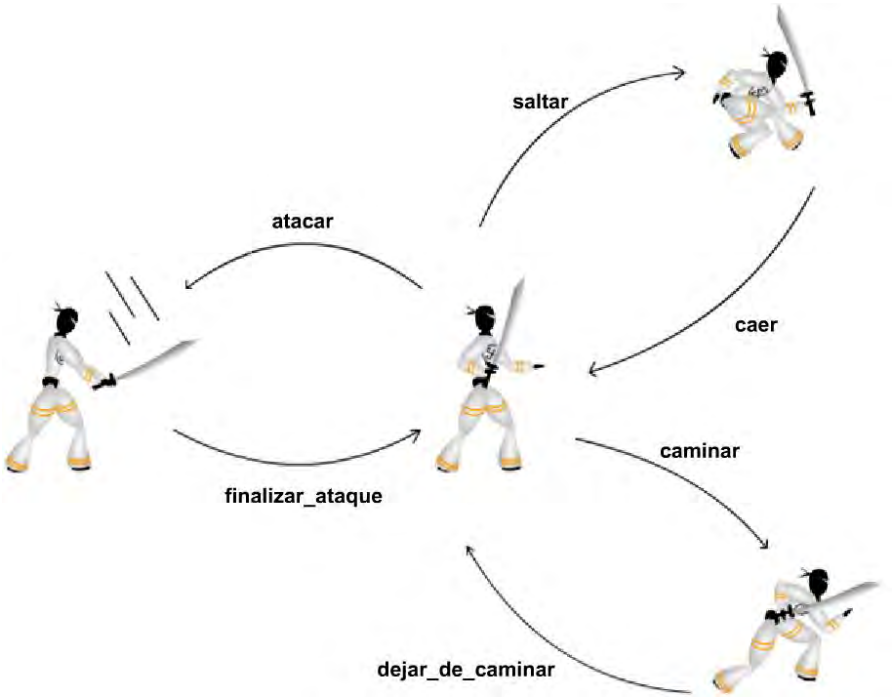


Figura 2.2: *Se puede utilizar una GFSM para controlar las animaciones de un personaje virtual.*

Como se muestra en la figura 2.2, crear comportamientos para agentes virtuales ha sido ampliamente utilizado por desarrolladores de videojuegos, por ejemplo, los fantasmas en el conocido juego *Pac-man* incorporan FSMs en sus comportamientos, cada uno de los cuatro fantasmas tiene implementado de diferente forma el estado de *perseguir*, sin embargo en el momento que el jugador "come" una *píldora de poder*, los fantasmas cambian su estado a *huir.* También los NPC en *juegos de estrategia en tiempo real* utilizan FSMs teniendo estados como *Mover*, *Atacar*, *Patrullar* o *Morir.*

2.7.4 Máquinas de estados jerárquicas

Las máquinas de estados jerárquicas (HFSM por sus siglas en inglés) son una extensión del concepto de máquina de estados. En las FSMs tradicionales, todos los estados se consideran en el mismo nivel, las HFSMs hacen uso de superestados, es decir, estados que a su vez están compuestos por otros estados de más bajo nivel. Estos superestados también tienen transiciones entre ellos. Cada estado debe pertenecer a un solo superestado con el fin de guardar una jerarquía estricta como en un árbol. Las HFSM son un mecanismo para reducir el tamaño del modelo de una FSM ya que permite ahorrar transiciones mediante el uso de jerarquías. Esto es útil para

hacer más manejable y entendible un modelo que sufre de explosión de estados, problema que surge cuando el sistema a modelar se hace cada vez más complejo y requiere un número de estados excesivamente grande.

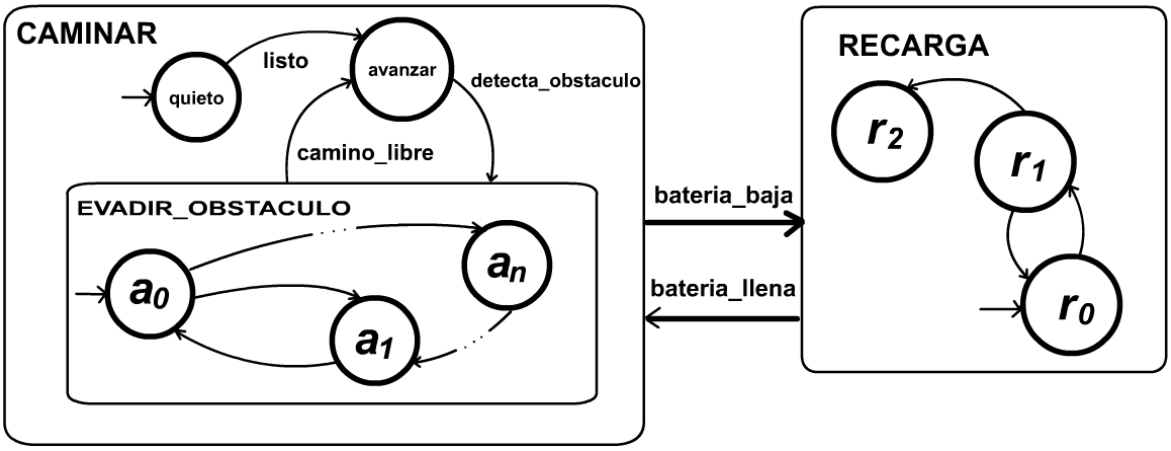


Figura 2.3: Maquina de estados jerárquica.

En la figura 2.3 se muestra un diagrama de estados para una HFSM, se pueden ver representados como rectángulos a los superestados y a los estados de jerarquía menor que los componen como círculos. En general, cuando una entrada se detecta, esta es capturada primero por el estado de mayor jerarquía, por ejemplo CAMINAR. Si no existe una transición activa para dicha entrada en ese nivel, pasa a ser procesada por los estados del nivel de jerarquía inmediato y así sucesivamente definiendo una especie de prioridad entre estados y transiciones.

Por ejemplo, si el robot está caminando (CAMINAR), y encuentra un obstáculo en su camino, entra en el estado EVADIR_OBSTACULO que a su vez tiene *n* estados internos, entonces la máquina se encuentra en 3 estados simultáneos (uno para cada nivel), en CAMINAR, EVADIR_OBSTACULO y a_0. Si en este momento entrara la señal de *batería_baja*, el estado de mayor jerarquía (CAMINAR) transitaría a RECARGA y la máquina se encontraría en RECARGA y r_0. De esta forma se evita representar transiciones entre cada uno de los estados a_i hacia r_0 cada que se detecta batería_baja y también se tiene un modelo más fácil de entender y manejar. Sin embargo aunque la representación disminuye en tamaño, la complejidad y el número de estados permanece.

2.8 Algoritmos genéticos

Los algoritmos genéticos (AG) son *métodos heurísticos de búsqueda* que simulan el proceso de evolución natural [11] siguiendo principios biológicos como la supervivencia de los organismos más aptos, herencia genética, reproducción sexual y la mutación. Sirven para atacar problemas cuyo espacio de soluciones es muy grande pero finito, complejo o poco entendido y donde los procedimientos matemáticos tradicionales fallan. Fueron estudiados por primera vez por John Holland de la universidad de Michigan en los años 70 con el objetivo de comprender mejor los procesos de adaptación natural y crear sistemas artificiales capaces de imitar estos procesos.

Algunas características particulares de los AG que los hacen diferentes a otros métodos heurísticos de búsqueda y optimización son que los AG trabajan con codificaciones de las soluciones candidatas y no con las soluciones mismas, trabajan con poblaciones y no con soluciones individuales y también utilizan únicamente el valor de la función a optimizar y no el de su derivada u otro conocimiento sobre ella por lo que se pueden utilizar con funciones no derivables. Además no suponen nada o casi nada del problema a resolver y se pueden aplicar a diversos problemas sin hacer cambios sustanciales en la estructura básica del algoritmo.

En la naturaleza, los organismos tienen características observables tanto físicas como conductuales, éstas sirven para identificar y definir las capacidades de dichos organismos, Al conjunto de características se les llama fenotipo y son la expresión de la información genética del individuo (genotipo) en interacción con un medio ambiente determinado. De esta forma la naturaleza tiene una representación codificada del dominio del problema (todos los individuos posibles), así en un AG es necesario codificar el espacio de soluciones del problema en estructuras equivalentes al genotipo en los seres vivos con el fin de ser operados. Se suele codificar a los individuos con cadenas de ceros y unos (binarias) pero se pueden utilizar diferentes representaciones como números enteros o reales.

El proceso general que siguen los AG se basa en un conjunto (población) de soluciones posibles codificadas (individuos). El proceso de evolución inicia sobre una población de individuos generados aleatoriamente y se crean poblaciones sucesivas en

las que proliferen individuos que representen mejores soluciones al problema, este proceso continúa hasta que se encuentre un individuo suficientemente apto que puede o no ser un óptimo global del problema. Durante cada generación, se obtiene una medida del desempeño relativo a la población de cada uno de los individuos con el objetivo de encontrar a los mejores y a los peores. Entonces, a cada individuo le corresponde una calificación, un número real obtenido al evaluar una función de adaptación (*fitness)* con dicho individuo, que mientras mayor sea, mejor es la solución a la cual representa.

Emulando al proceso de selección natural, en un AG debe existir un criterio de selección mediante el cual los individuos más calificados tengan mayor probabilidad de reproducirse o pasar a la siguiente generación asegurando así la posibilidad de tener individuos bien calificados en generaciones siguientes. El elitismo consiste en seleccionar a los *n* mejores individuos de la generación actual para pasar intactos a la siguiente, así se evita perder buenas soluciones encontradas en etapas previas.

Además de seleccionar a los individuos buenos, es necesario encontrar nuevos potencialmente mejores, esto se logra incorporando operadores de **cruzamiento** y **mutación**. El cruzamiento es una operación binaria entre individuos que al combinar sus códigos genéticos generan descendientes con características de ambos. La mutación se utiliza para dotar de diversidad a la población explorando regiones que probablemente no se han explorado donde se pueden encontrar nuevas soluciones mejores a las conocidas y evitar caer en óptimos locales, consiste en alterar el código genético de algunos individuos seleccionados aleatoriamente con cierta probabilidad. Tanto la mutación como el cruzamiento están ligados al tipo de codificación y el problema que se quiere atacar. Algunas veces también es necesaria una función de factibilidad que evalúe si un individuo resultante de un cruzamiento, mutación o generada aleatoriamente representa una solución válida.

En general, el procedimiento de un algoritmo genético básico es el siguiente:

- Primero se debe elegir una codificación del dominio del problema, representar cada posible solución como una cadena de símbolos binarios, enteros, reales o de algún otro tipo.
- Generar *N* códigos de posibles soluciones de forma aleatoria (población actual).
- Utilizando la función de adaptación, calificar cada uno de los individuos de la población actual.
- Seleccionar un par de individuos utilizando un criterio de selección / eliminación. En caso de incorporar elitismo se deben guardar o hacer pasar intactos los mejores *n* individuos a la nueva población.
- Cruzar los individuos seleccionados con una probabilidad *Pc.* En caso de cruzarse, se generan dos nuevos individuos hijos, en caso contrario los padres quedan intactos como los individuos nuevos.
- Mutar cada uno de los individuos nuevos con probabilidad *Pm.*
- Agregar los individuos nuevos (mutados o no) a la nueva población.
- Si la nueva población ya tiene *N* individuos, llamarla población actual y regresar al paso 3. En este punto ha transcurrido una generación y si se cumple una condición de terminación, el algoritmo termina.
- Si no, regresar al paso 4.

El **algoritmo genético simple** (AGS) llamado así por Goldberg [11] y propuesto por Holland ha servido como punto de partida para el desarrollo de nuevas formas de selección, cruzamiento y mutación, este algoritmo tiene las siguientes características básicas:

- Utiliza una codificación binaria.
- Tiene tamaño de la población fijo en todas las generaciones.
- Utiliza el método de ruleta como criterio de selección. Se basa en la selección proporcional al *fitness* de cada individuo, los mejores tienen mayor probabilidad de ser seleccionados.
- Utiliza *cruzamiento* de un punto con probabilidad constante. Este cruzamiento se basa en seleccionar un número al azar entre 1 y *l-1,* donde *l* es el tamaño del cromosoma, este punto define dos segmentos de cada padre, los cuales se intercambian para formar dos individuos nuevos.
- *Mutación uniforme*, todas las posiciones de la cadena genética tiene la misma

probabilidad de ser alteradas, probabilidad que permanece constante durante todo el proceso.

- No incorpora *elitismo.*

2.8.1 Algoritmos genéticos y GFSMs

El diseño de la arquitectura y los parámetros de las FSMs pueden optimizarse utilizando algoritmos evolutivos como los genéticos. Muchos de los comportamientos modelados con FSMs pueden ser realmente simples y fáciles de modelar a criterio del desarrollador. Sin embargo el uso de AGs tiene diversas ventajas como el poder encontrar soluciones muchas veces más robustas que las diseñadas a mano y principalmente para evitar que el desarrollador padezca la complejidad del diseño y mantenimiento cuando el modelo se hace muy grande [25].

2.9 Agentes

Ya que no existe una definición globalmente aceptada de lo que es un agente, se presentará la definición simple provista por Russel y Norvig en [16]:

Un agente es cualquier cosa que pueda percibir su ambiente mediante sensores y actuar sobre él mediante actuadores.

Esta definición es muy general y está ligada totalmente a lo que se pueda entender por ambiente, percibir y actuar. Pero existen agentes con características muy particulares que vale la pena hacer notar con el fin de clasificarlos, algunas de ellas son:

- **Reactividad**: Capacidad de responder inmediatamente (o casi) a los cambios en el ambiente.
- **Autonomía:** Capacidad de ejercer control sobre sus propias acciones.
- **Pro-actividad**: Que puede Actuar no necesariamente como respuesta a un estímulo.
- **Sociabilidad:** Que se comunica con otros agentes e incluso seres humanos.

- **Continuidad en el tiempo (persistencia)**: Existe como un proceso que se ejecuta continuamente.
- **Aprendizaje**: Capacidad de cambiar su comportamiento con base en experiencia previa.
- **Movilidad:** Capacidad de transportarse de un ambiente a otro.

La autonomía es una de las características más deseadas en un agente para los fines perseguidos en este trabajo al igual que la pro-actividad. Stan Franklin y Art Graesser proponen la siguiente definición formal en [10]:

Un agente autónomo es un sistema situado dentro y como parte de un ambiente, que percibe dicho ambiente y actúa sobre él a través del tiempo, buscando realizar sus metas o tareas y afectando lo que percibirá en un futuro.

Los agentes están situados en un ambiente específico gracias al cual existen, un sistema puede o no ser un agente dependiendo del ambiente en el que se encuentre, por ejemplo un agente con sensores de luz visible no lo será en un entorno sin luz. El agente debe ser capaz de planear sus acciones con base en sus percepciones del ambiente siempre persiguiendo sus objetivos sin la necesidad de intervención humana. De esta forma, el comportamiento de un agente está definido por una secuencia de interacciones entre él y su ambiente que afectan sus percepciones futuras y en consecuencia sus acciones.

En general, los agentes de software son agentes computacionales, es decir, son programas, sin embargo cumplen con ciertas características que los distinguen de cualquier otro programa arbitrario: Son persistentes, autónomos y reactivos por lo que el código que los compone se ejecuta continuamente y actúan durante un periodo de tiempo. Una vez invocado, un agente de software generalmente se ejecutará posiblemente hasta que él decida ya no hacerlo. Se podría decir que un programa arbitrario puede percibir su ambiente por medio de sus datos de entrada y actuar por medio de sus salidas, pero no es considerado agente mientras sus salidas no modifiquen el entorno afectando sus percepciones futuras. Todos los agentes de software son programas pero no todos los programas son agentes.

2.10 Comportamientos y su organización

Siguiendo el modelo de los sistemas basados en comportamientos [4], un agente está compuesto por un conjunto de éstos que definen su *repertorio conductual*. Estos comportamientos pueden ser tan simples o complejos como se requieran, por ejemplo evadir obstáculos, explorar un mapa, atacar un objetivo etc. Durante la existencia del agente, generalmente solo uno de ellos está activo en un momento dado. El problema de la organización de comportamientos se basa en cómo combinar cada uno de los comportamientos de un agente para lograr cumplir sus objetivos de la mejor manera. Debe existir un mecanismo que decida que comportamiento se ejecuta en cada momento definiendo prioridades dinámicamente entre ellos dependiendo de la situación.

Actualmente existen varios métodos de organización de comportamientos, uno de los más conocidos y utilizados es el de *subsunción*, ideado por Rodney Brooks. Este método introduce el concepto de niveles de competencia que define jerarquías entre comportamientos, por ejemplo, los niveles de competencia más bajos están compuestos por los comportamientos más básicos y en niveles altos se encuentra los más complejos. Cada comportamiento en los niveles altos (de mayor nivel de abstracción) subsume o incluye a los niveles más bajos. Cada uno de estos comportamientos puede ser implementado utilizando máquinas de estado aumentadas que tienen la característica de tener inhibidores y supresores mediante los cuales otro comportamiento puede inhibir sus entradas o suprimir sus salidas.

Otro método de organización de comportamientos es el propuesto por Pattie Maes en [13], en donde cada comportamiento tiene asociado un *nivel de activación* representado por un número real, un conjunto de condiciones que deben ser observadas para que el comportamiento pueda ejecutarse y un *umbral*. En el momento que se cumplan las condiciones para que un comportamiento se ejecute y su nivel de activación sobrepasa el umbral, se ejecuta. Adicionalmente, el agente tiene un conjunto de motivaciones, las cuales están asociadas a ciertos comportamientos, por ejemplo la motivación de *hambre* se asocia al comportamiento de *comer*, mientras más *hambre* se tenga, mayor será el nivel de activación de *comer*.

2. Antecedentes

En general, la organización de comportamientos debe estar presente en un sistema basado en comportamientos y opera sobre el conjunto de comportamientos de un agente independientemente de cómo fueron creados.

3. Análisis y diseño

3.1 Análisis del problema

Con el objetivo de modelar personajes virtuales basados en comportamientos con FSMs aptos para lograr metas complejas se pueden construir sistemas con agentes que puedan percibir y actuar cada vez más y mejor en su entorno. Mientras más complejo sea un comportamiento más lo será la FSM que lo representa, esto puede ocasionar una explosión de estados en el modelo haciéndolo inmanejable para el desarrollador. La naturaleza de las FSM hace que rápidamente incremente en complejidad y espacio su representación en función del número de estados.

Para hacer frente a esta situación, los desarrolladores de videojuegos han utilizado técnicas como las máquinas de estados jerárquicas que si bien logran reducir el número de transiciones a representar y proveen un diseño más comprensible, a la larga presentan el mismo problema conforme crece el número de estados.

Se explora la posibilidad de desarrollar un sistema automático que construya FSMs capaz de encontrar eficientemente comportamientos cercanos a los óptimos evitando que el desarrollador esté inmerso en la complejidad de su diseño directo. Utilizando un método de búsqueda heurística como los algoritmos genéticos como herramienta para explorar el espacio de soluciones que crece exponencialmente.

Además de crear los comportamientos para los agentes se desarrolla un entorno virtual de simulación. Para lograrlo se deben tomar en cuenta las capacidades y restricciones que se tienen al construir entornos virtuales así como implementar los subsistemas necesarios para la realización de tareas como el dibujo del entorno y los agentes, manejo de recursos, temporización, manejo de entrada y salida, una representación adecuada de los comportamientos, la simulación y las acciones de los agentes etc.

3.1.1 Comportamientos

El método presentado se basa únicamente en la creación de comportamientos, el tema de la organización de los mismos está más allá de los objetivos de este trabajo. Cada comportamiento creado por separado formará parte del repertorio de comportamientos de un agente dado. Cada uno de estos comportamientos será representado por una máquina de estados finitos (FSM) cuyos estados forman un subconjunto de todos los estados posibles en los que se puede encontrar el agente, por lo tanto, están basados en las capacidades del mismo.

Los sensores del agente están ligados con las condiciones asociadas a las transiciones de la FSM que representa su comportamiento, mientras más estados, sensores o capacidades de percibir su entorno y variables internas tenga el agente, más complejos podrán ser los comportamientos que ejecute.

Cada una de las FSMs utilizadas puede estar en un solo estado en un momento dado, por lo tanto el agente únicamente puede encontrarse realizando una acción a la vez. Para estados que involucran más de una acción, estas se ejecutarán en serie.

En el presente trabajo se toman en cuenta ambos enfoques: el de Moore, donde se tienen tantos estados como acciones y siempre el mismo estado ejecuta la misma acción y el de Mealy, donde cada estado puede ejecutar diferentes acciones en función de las entradas y el estado actual.

3.1.2 Estados

Cada estado debe realizar 3 procesos básicos:

Inicialización: Este proceso implica establecer las condiciones necesarias para la ejecución del estado, se ejecuta una sola vez al entrar al estado.

Actualización: Es el proceso que ejecuta las acciones propias referentes al estado. Generalmente se ejecuta una y otra vez mientras la máquina se encuentre en el estado en cuestión.

Finalización: Se ejecuta una sola vez cuando la máquina abandona el estado actual para transitar a uno nuevo, realiza las operaciones necesarias para terminar el estado actual.

Como máquina de Moore

Los estados de la FSM que representan un comportamiento están asociados a cada una de las acciones o combinaciones de éstas que puede realizar un agente en su entorno, por ejemplo: un agente puede tener estados como moverse, quedarse quieto, rotar, comer, saltar, atacar o tomar un objeto. Cada una de estas acciones está definida por el diseño del agente. Cuando el agente se encuentra en un estado, siempre ejecutará la misma acción o lista de ellas asociadas al estado, dichas acciones pueden ser ejecutadas una y otra vez hasta que la máquina cambie de estado o bien solo una vez y luego esperar hasta que se produzca este cambio.

Como máquina de Mealy

El agente de igual forma necesita un conjunto de acciones que puede realizar, sin embargo, el número de estados no necesariamente es el mismo que el de acciones. En este caso, se plantea un número máximo de estados que puede utilizar el comportamiento y cada acción del agente dependerá del estado actual y la transición que provocó el último cambio de estado, así, al transitar al mismo estado por diferentes transiciones, se pueden ejecutar diferentes acciones. Para lograr esto, es necesario añadir a cada transición del comportamiento, un identificador de acción que indica cuál de ellas se ejecutará al cumplirse dicha transición.

3.1.3 Condiciones

Una condición se puede definir como una sentencia o una función que devuelve un valor booleano y que puede tomar como entrada valores de las lecturas de los sensores, constantes numéricas e incluso otras condiciones, por ejemplo, una condición que evalúe si un enemigo se encuentra en rango de ataque puede tomar como entrada la distancia al enemigo *enemy_d* y el valor del rango de ataque *attck_r*.

Estos valores serán relacionados mediante un operador de comparación, en este caso será menor o igual, y el resultado será devuelto por la función que define la condición. Si el valor devuelto al evaluar una condición es verdadero (*true)*, indica que la condición se cumple haciendo posible ejecutar la transición a la que está asociada, si es falso (*false)* entonces no se tomará en cuenta dicha transición. El proceso interno que ejecuta una condición puede ser tan simple como leer un 1 (*true*) o un 0 (*false*) directamente de un sensor o tan complicado como ejecutar un algoritmo de *pathfinding*.

3.1.4 Transiciones

Para las máquinas de Moore, cada transición en un comportamiento tiene asociada una condición y dos estados, uno origen y otro destino. En el modelo de Mealy, cada transición tiene además una acción a ejecutar. De esta forma cuando se cumple su condición asociada estando en el estado origen, la máquina puede transitar al estado destino. Cada estado tiene tantas transiciones posibles como estados máximos en la máquina (una hacia cada estado, incluso él mismo) y más de una puede tener asociada la misma condición o en un momento dado se pueden cumplir las condiciones de varias transiciones. Para decidir que transición ejecutar, se les asigna una prioridad, un número entero no negativo que mientras mayor sea, mayor será esta prioridad. En la figura 3.1 se muestra una máquina de estados que representa un comportamiento básico de evasión de obstáculos.

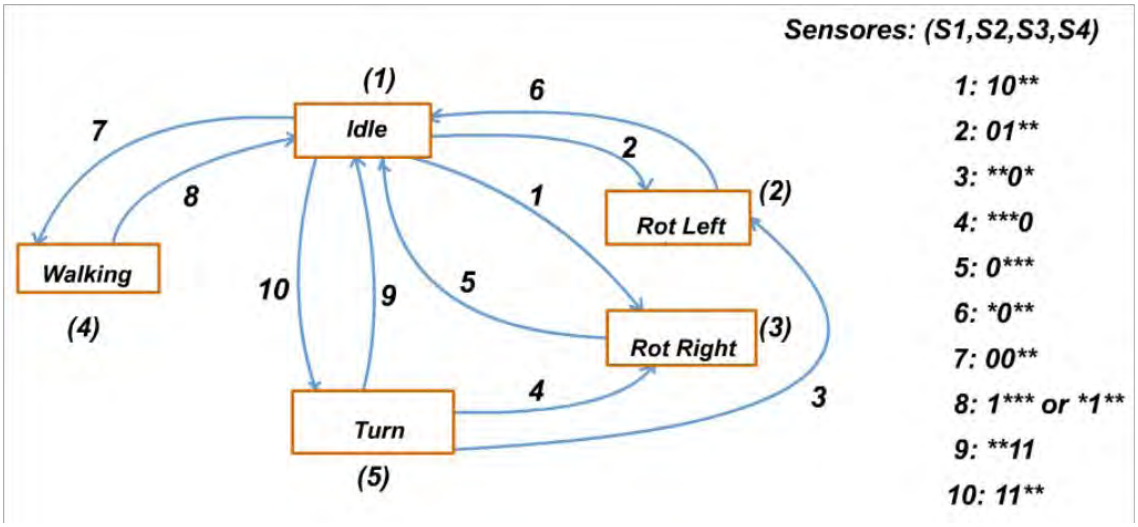


*Figura 3.1: Comportamiento compuesto por estados, condiciones y transiciones. Los rectángulos representan a los 5 estados numerados entre paréntesis y las líneas curvas que los unen sus transiciones etiquetadas con una condición. Hay 10 condiciones (derecha), cada una dada por una lectura de los cuatro sensores del agente. El * implica cualquier valor en el sensor (don't care).*

3.1.5 Espacio de búsqueda

Generalmente el diseño de comportamientos basados en FSM se realiza a criterio del desarrollador de inteligencia artificial ya que debe ser muy específico para cada agente y cada objetivo. Éste es un proceso de prueba y error en donde se proponen comportamientos y con base en el desempeño del agente que lo ejecuta se le hacen cambios o se le agregan cosas.

A medida que los sistemas crecen, se requieren comportamientos más complejos, esto implica modelar agentes con un mayor número de estados (acciones posibles) y el número de transiciones posibles entre ellos crece en un orden cuadrático con respecto al número de estados. Lo anterior es tomando en cuenta el diagrama de estados de una FSM arbitrario visto como una gráfica plana donde los nodos representan estados y las aristas dirigidas representan transiciones entre ellos.

Cada transición puede existir o no, el hecho de que no exista una entre un estado origen y un destino implica que no hay una condición que al presentarse provoque que la máquina pase directamente al estado destino desde el estado origen, por ejemplo, entre el estado *dormir* y el estado *comer* puede no haber conexión directa, habría que pasar por el estado *despertar* antes.

Esta afirmación define un problema: *el de encontrar el subconjunto de transiciones que aparecerán en la FSM buscada*. Entonces si S es el conjunto de estados de tamaño N_S y T es el conjunto de transiciones posibles de tamaño $N_T = N_S^2$, existen $2^{N_T} - 1$ posibles conjuntos diferentes no vacios de transiciones que estarán presentes en la FSM.

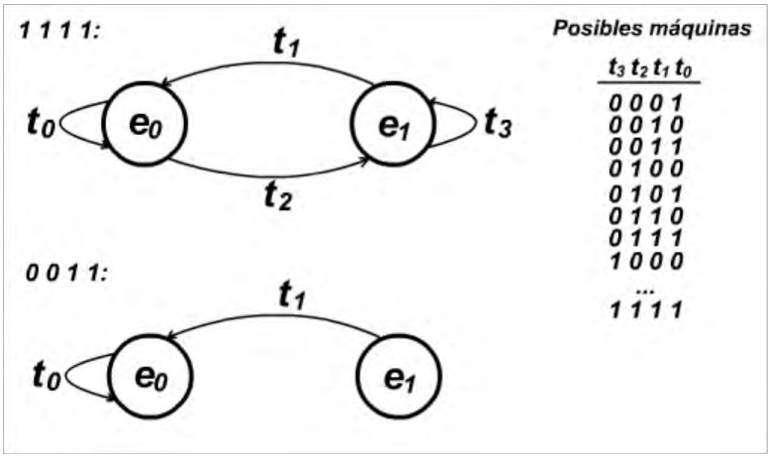


Figura 3.2: Algunas de las posibles máquinas *con 2 estados. t_i representa la transición i-ésima. Cada cadena de 4 bits define un conjunto de transiciones presentes diferente.*

Una vez definido el conjunto *R* de transiciones presentes de tamaño N_R menor o igual a N_T, cada una de estas transiciones debe estar etiquetada con una de N_M posibles condiciones, que pueden repetirse, entonces, existen $N_M{}^{N_R}$ máquinas posibles dado un conjunto *S* de estados y un conjunto *M* de condiciones. Estas condiciones dependen directamente del agente, representan el conjunto de sus perceptores como variables internas, lectura de un sensor o combinaciones entre ellos. En la *Figura 3.2* se muestra una FSM con 2 estados, para los cuales se tienen 4 transiciones posibles. Cada máquina que se puede formar está dada por una cadena de 4 bits, donde el bit *i-ésimo* representa la presencia (1) o ausencia (0) de la transición *i.* La cadena con puros ceros no interesa ya que define una máquina sin transiciones, lo cual no tiene sentido.

En general, para desarrollar un método automático que construya una máquina de estados se debe considerar lo siguiente:

- El problema de optimización de un comportamiento representado por FSMs con respecto a un objetivo se puede ver como un problema de búsqueda en el espacio de todas las máquinas posibles dado un conjunto de estados, transiciones y condiciones. Determinar el objetivo que debe cumplir el comportamiento es muy importante ya que algunas máquinas pueden ser malas para unos pero buenas para otros sin cambiar sus estados o condiciones.
- Para N_s estados, el número de transiciones posibles crece al cuadrado.
- El conjunto de las transiciones presentes en la máquina buscada será un elemento

no vacío de P(T), el conjunto potencia de todas las transiciones posibles.

- Todas las posibles condiciones de transición deben numerarse, formando el conjunto M. Estas condiciones pueden recibir y devolver parámetros de diversos tipos (booleano, entero, flotante). Por ejemplo la condición `bool batteryLessThan(50)` devolvería `true` si la carga de la batería en un robot es menor al 50% y `false` en caso contrario. Adicionalmente se le puede asociar una prioridad a cada condición.
- Cada una de las N_R transiciones presentes en la máquina se debe etiquetar con una de las N_M condiciones de transición, varias transiciones pueden tener la misma condición. Esto genera ${N_M}^{N_R}$ posibles configuraciones.

Dado esto, un comportamiento con N_S estados y N_M condiciones (incluyendo la transición nula o ausencia de transición) se puede representar como una cadena de ${N_S}^2$ números donde el *i-ésimo* número representa la condición de la transición entre el estado $\frac{i}{N_s}$ y el $i \,\%\, N_s$, donde % es la operación de módulo. Una de estas cadenas de números representa a un único comportamiento, esta representación, de longitud fija, se explica más a detalle en el capítulo 4.

Entonces, como existen ${N_S}^2$ posiciones en las que debe estar uno de N_M números que pueden repetirse, se tienen ${N_M}^{{N_S}^2}$ máquinas posibles con esta representación. Claramente el número de máquinas posibles crece de forma exponencial al incrementar el número de estados.

Este crecimiento hace que sea imposible numerar cada una de las máquinas para buscar la que representa al mejor comportamiento con un número suficientemente grande de estados. Por lo que se hace necesaria la aplicación de métodos heurísticos de búsqueda con el fin de explorar el extenso espacio buscando un óptimo global.

Número de Estados	Transiciones Posibles	Condiciones	Máquinas posibles aprox.
2	4	5	625
4	16	5	152,587,890,625
6	36	5	14×10^{24}
8	64	5	5.4×10^{44}
10	100	5	7.8×10^{69}
50	2500	5	2.6×10^{1747}
100	10,000	5	5×10^{6989}

Tabla 3.1: En la primera columna se muestra el número de estados de la máquina, las transiciones posibles es el número de estados al cuadrado. Si el agente puede evaluar 5 condiciones, se muestra en número de máquinas posibles en la última columna.

3.2 Diseño de la solución

3.2.1 Descripción general de la solución

El diseñar una máquina de estados a mano se puede ver como una búsqueda heurística, donde dicha heurística está relacionada con un proceso de prueba y error. Por esto, se propone construir un sistema automático que sea capaz de explorar el espacio de búsqueda de las máquinas de estado posibles encontrando soluciones cada vez mejores.

Para esto es necesario un sistema de simulación que implemente el comportamiento en los agentes virtuales y que sirva como plataforma de experimentación y pruebas, lo que implica el desarrollo de un sistema de cómputo gráfico que contenga una representación del mundo, de los agentes y sus comportamientos, un método heurístico de búsqueda y un mecanismo que permita que todos estos componentes trabajen en conjunto. El desarrollo del sistema en su totalidad se realiza utilizando el lenguaje de programación C++ y diversas bibliotecas de software que proveen funcionalidades desde lectura/escritura de archivos XML hasta el motor gráfico. Se propone entonces desarrollar un sistema integral con módulos funcionales trabajando en conjunto desde el proceso de creación y diseño del entorno hasta la ejecución de los comportamientos correctos en un ambiente virtual, para esto se deben realizar las siguientes tareas básicas:

- Construir una interfaz de diseño y representación simbólica del entorno en el que serán evaluados los agentes.

- Diseñar una representación flexible para agentes de diversos tipos y con diversas características.
- Diseñar una representación para FSMs así como su forma de ejecución.
- Implementar un sistema de simulación de comportamientos con el fin de evaluarlos y probar su funcionamiento.
- Implementar un algoritmo heurístico de búsqueda con el fin de explorar el espacio de soluciones que por su tamaño es imposible numerarlo. Se utilizará un algoritmo genético.
- Desarrollar un ambiente virtual que provea una interfaz gráfica tridimensional donde los agentes estudiados "vivan" ejecutando los mejores comportamientos.

Posteriormente, se plantean las siguientes actividades finales:

- Realizar pruebas al sistema con un comportamiento para un agente dado.
- Reportar y analizar resultados.
- Proponer mejoras y trabajo futuro.

El diseño general del sistema se muestra en la *Figura 3.3* donde se pueden ver los módulos principales y sus relaciones. Cada módulo puede verse como un programa independiente con entradas y salidas específicas que, interconectados entre sí resuelven el problema en cuestión. Algunos módulos están diseñados para ser totalmente automáticos durante el proceso de desarrollo de comportamientos, como el del algoritmo genético, el simulador rápido y el ambiente virtual. Otros como la representación del mundo y los parámetros de calificación requieren intervención directa del diseñador de los agentes y comportamientos, estos módulos son los que requerirán modificaciones al buscar comportamientos diferentes, cambiar las capacidades de los agentes o el entorno.

Se habla de una evolución de comportamientos ya que como algoritmo de búsqueda se eligió un algoritmo genético que emula el proceso de evolución natural. Aunado a esto, se llevará un control de los resultados intermedios para analizar el progreso de los comportamientos desde uno aleatorio inicial hasta el que realiza de mejor manera la meta propuesta.

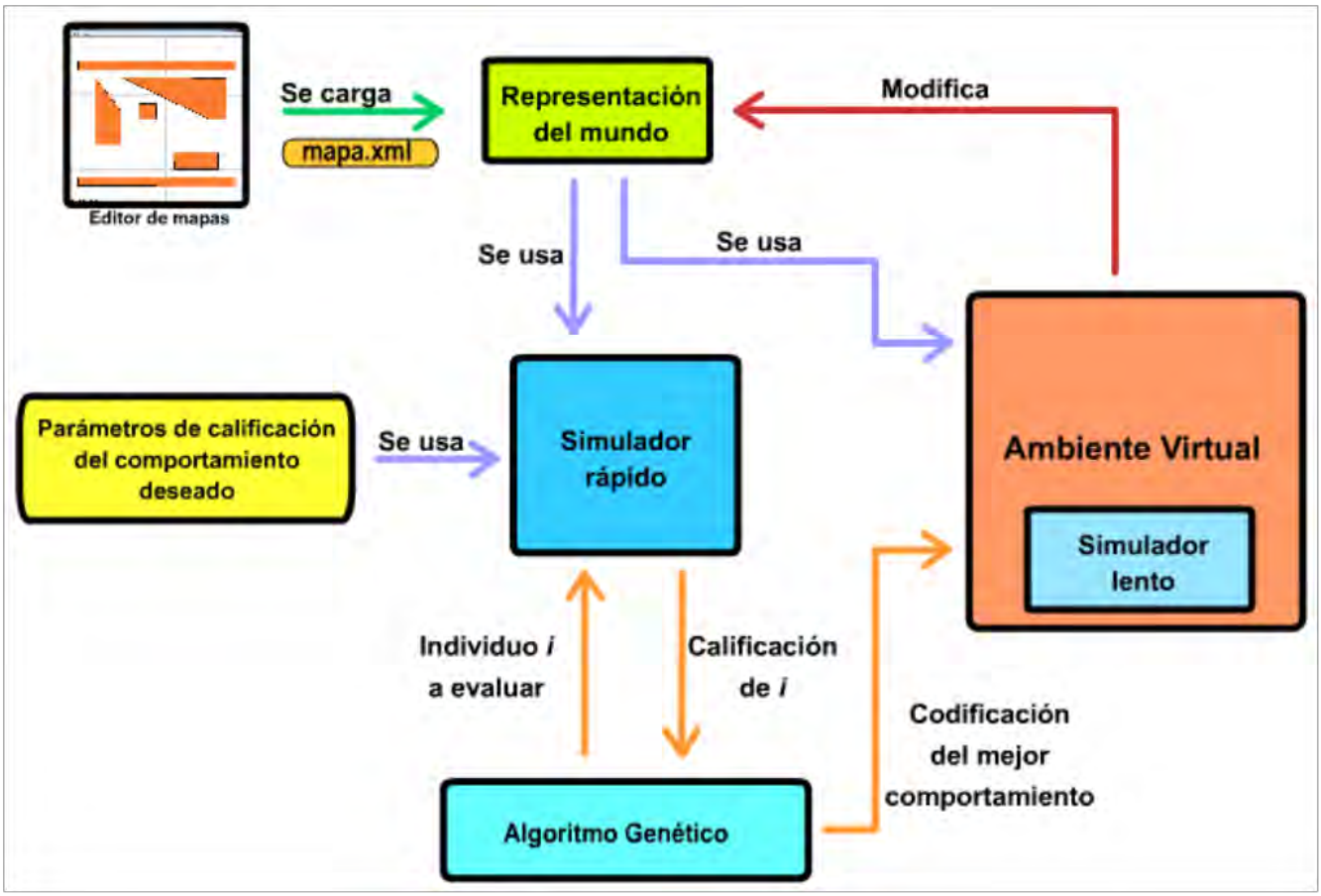


Figura 3.3: Diagrama general del sistema.

3.2.2 Agentes

El comportamiento general de los agentes puede tener tanto componentes simbólicas basadas en una representación del mundo, como componentes reactivas como el modelado de sensores para robots. Se creará un diseño de clases con el fin de modelar diversos tipos de agentes con características definidas. Como base del modelo se plantea la clase `Agent` que define un agente lógico en su forma más básica con un método de percepción y otro de acción. De `Agent` heredan clases más específicas, por ejemplo, `GameAgent` que define agentes con representación gráfica y propiedades de posición, tamaño, rotación, velocidad, entre otras. Dentro del ambiente virtual de simulación, cada agente tendrá una representación gráfica asociada, animaciones y mecanismos para realizar acciones sobre el entorno.

3.2.3 Mapas

Al hablar de agentes es necesario hablar del entorno en el que están inmersos, éste debe tener una representación flexible, sencilla, de fácil acceso y con todas sus propiedades de interés. Por eso es necesario diseñar un método de representación del entorno que sea utilizado por los simuladores. Se plantea el desarrollo de un programa externo, un editor como herramienta gráfica para la construcción de mapas poligonales que permita guardar sus especificaciones en un formato universal , el formato XML con el fin de poder ser leído por otro programa, algunas de las operaciones más importantes que realiza son:

- Crear un mapa nuevo con dimensiones deseadas.

- Mediante controles en la barra de herramientas permite agregar polígonos convexos que representan obstáculos o áreas con diferentes características, principalmente transitables y no transitables aunque también se pueden definir propiedades que afecten la percepción o acción de los agentes, por ejemplo un área con tierra donde el agente no pueda moverse muy rápido o un área de alta temperatura donde no se puede pasar a menos que sea absolutamente necesario.

- Guardar la especificación del mapa en un archivo de formato xml.

Este programa sirve como herramienta para el desarrollador y con su ayuda se pueden diseñar mapas de forma fácil y rápida para hacer diferentes pruebas. Es importante mencionar que el archivo xml resultante del editor representa únicamente el mapa (espacio transitable y obstáculos fijos) asociado al entorno, las demás propiedades como obstáculos móviles o su representación gráfica se deben definir aparte en la representación global del mundo que también provee información útil para los algoritmos de pathfinding como A* que pueden implementar internamente los agentes virtuales.

También se tiene la posibilidad de crear un mapa a partir de imágenes en *formato .tga* o texto plano para crear una representación del mundo basada en celdas, a partir de las cuales se genera el grafo de navegación para los agentes.

3.2.4 Simuladores

Los comportamientos encontrados se deben aplicar a agentes inmersos en un mundo virtual y con el fin de cuantificar que tan bueno o malo es un comportamiento dado, éste debe simularse en un sistema dedicado. Esta simulación requiere implementar un modelo de agentes con diversas características así como cada una de sus acciones y la forma en que percibe el entorno.

La simulación se realiza en 2 modos diferentes, uno independiente de la representación gráfica que se ejecuta a alta velocidad y en segundo plano y otro modo cuyos resultados se muestran gráficamente, esta última debe ejecutarse en tiempo real, retroalimentar al usuario y permitir la comunicación con el mismo mediante dispositivos periféricos. El simulador rápido ejecuta cada ciclo a la velocidad máxima que la computadora lo permita, el hecho de no tener que realizar ningún proceso gráfico incrementa en mucho esta velocidad. Ambos simuladores basan sus acciones en un contador de tiempo, para el simulador rápido cada ciclo representa una fracción de segundo, por ejemplo *time_delta* = 0.1s., entonces cada 10 ciclos el contador interno de tiempo aumentará un segundo independientemente del tiempo "real" que haya pasado.

El simulador "lento" o dependiente de los gráficos se basa en FPS (frames por segundo) ya que la realización de los procesos gráficos requiere un tiempo considerable de procesador. Más aún, todas las acciones del agente deben estar basadas en un contador de tiempo "real" donde 1 segundo en el simulador equivale a 1 segundo en el mundo real contado por el reloj de la computadora. Esto hace que independientemente de los FPS a los que se ejecute la simulación, si un agente debe avanzar 10 unidades por segundo, después de 1 segundo "real", éste se encontrará a 10 unidades de donde partió sin importar si dio 5 pasos de 2 unidades o 2 pasos de 5, véase *Figura 3.4*. Es importante notar que no todos los ciclos (frames) tardan lo mismo, esto dependerá de la complejidad de los procesos de cada uno y de la carga de trabajo que tenga la computadora en el instante de procesarlo, no siempre se deberá dibujar la misma cantidad de polígonos. La medida de los FPS es un promedio del número de ciclos que ejecuta la computadora en un segundo.

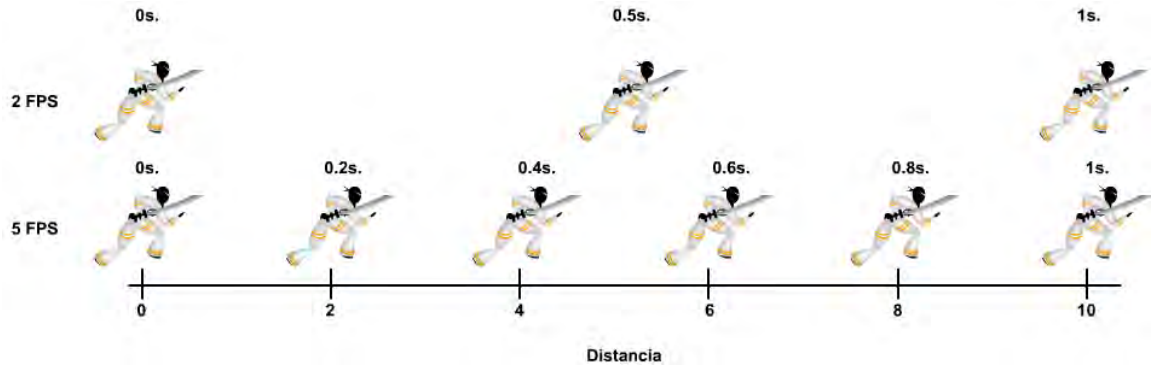


Figura 3.4: Se muestra un agente moviéndose 10 unidades en 1 segundo a 2 y 5 FPS, más FPS resultan en mayor suavidad en el movimiento.

Este esquema es necesario ya que cada uno de los comportamientos estudiados (posibles soluciones) debe ser evaluado, esto implica simular un agente virtual que ejecute el comportamiento en cuestión en un entorno dado durante *x* número de ciclos de la FSM. No es funcional simular cada uno de ellos en un simulador lento ya que tardaría demasiado, por ejemplo, para simular 50 grupos de comportamientos 100 veces; para esto se utiliza el simulador rápido. En el simulador lento se ejecutan los mejores comportamientos encontrados para realizar las pruebas de su funcionamiento. Bajo este esquema de simulación es necesario que ambos simuladores obtengan datos equivalentes.

3.2.5 Algoritmo de Búsqueda

La búsqueda en el espacio de máquinas posibles se realiza utilizando un algoritmo genético (AG) que trabaja con conjuntos de soluciones y hace más natural el proceso de "evolucionar" poblaciones de comportamientos. Para implementar satisfactoriamente el AG es necesario:

- Diseñar una representación de las máquinas, en este caso, una representación de longitud fija basada en una matriz de transiciones.
- Los operadores de selección, cruza y mutación para cada una de las representaciones.
- Una implementación de un AG genérico que realice el proceso evolutivo independiente de la representación y el mecanismo de calificación de los agentes.

Los parámetros a tomar en cuenta son propios del algoritmo genético como probabilidad de mutación, cruzamiento, tipo de selección, número de individuos por generación y número máximo de generaciones. También los parámetros propios del problema como el número de estados, el número de transiciones por estado y las condiciones de transición. Estos parámetros se deben afinar manualmente con base en el desempeño del sistema y el criterio del desarrollador, aunque también se puede hacer uso de métodos automáticos para lograrlo.

El simulador independiente del tiempo (rápido) utiliza la representación del mapa para simular un agente inmerso en él, dicho agente posee un comportamiento dado por el algoritmo genético. También utiliza una serie de condiciones o parámetros que son dependientes del objetivo o tarea específica que se quiere calificar, estas son definidas por el desarrollador y están relacionadas directamente con el cálculo de la calificación del individuo, por ejemplo, se puede calificar como "mejor" un agente que esquiva obstáculos a otro que choca con ellos, en cambio si el comportamiento deseado fuera el de un misil que debe impactar un objetivo, estas condiciones deben cambiar.

El AG ejecuta los procesos referentes a la evolución, maneja poblaciones de comportamientos y se encarga de la selección, mutación y cruzamiento de los individuos. Para poder realizar esto, cada individuo necesita ser calificado externamente. El simulador rápido evalúa el desempeño del agente en función de estos parámetros y como resultado del proceso se tiene la calificación del comportamiento, un número entero que se pasa al algoritmo genético. Posteriormente el AG toma como medida de desempeño este valor para cada individuo. En general la serie de pasos que sigue el AG durante su ejecución:

1. Crea población inicial de comportamientos.
2. Para cada comportamiento (individuo) en la población actual
    a. Se pasa el código del comportamiento al simulador rápido.
    b. Este simula x pasos y obtiene una calificación del agente, ésta depende de un conjunto variables asociadas al resultado de la simulación.

   c. El AG toma la calificación y la asigna como medida de desempeño a cada individuo.
3. Una vez calificados todos los individuos de la población, el AG prosigue con el proceso de selección, cruzamiento y mutación para generar la siguiente generación.
4. Si se encontraron soluciones que satisfacen el criterio de búsqueda (se encontró un comportamiento aceptable que apaga todas las flamas), pasa al paso 6.
5. Si no se ha alcanzado la generación máxima, pasa al paso 2.
6. Las mejores *n* soluciones encontradas se pasan al ambiente virtual para su simulación gráfica en el simulador lento.

3.2.6 Ambiente virtual

El *ambiente virtual* realiza los procesos necesarios para dibujar el mundo y los agentes con base en los resultados numéricos del simulador lento haciendo uso de un motor de graficación (OGRE3D en este caso). También se encarga del manejo de entrada/salida y se comunica con la representación global del mundo para actualizar cambios que pudieron haber ocurrido producto de alguna acción del usuario o de un agente (como destruir una pared). Dentro del *ambiente virtual* se realiza una serie de pruebas con el fin de comprobar la efectividad del método propuesto, estas pruebas implican realizar todo lo necesario para dotar a los agentes de "vida", una representación gráfica y de un entorno sintético.

El *ambiente virtual* incorpora módulos funcionales básicos como: una pantalla de introducción, un menú que permita acceso a las opciones de simulación posibles, un mecanismo de interacción con el jugador o usuario y la parte de simulación donde se creará una población de agentes, cada uno con comportamientos que pueden o no ser iguales. Este ambiente también provee funcionalidad para que los agentes realicen sus acciones y para retroalimentar al usuario por medio de animación, sonido o texto en pantalla.

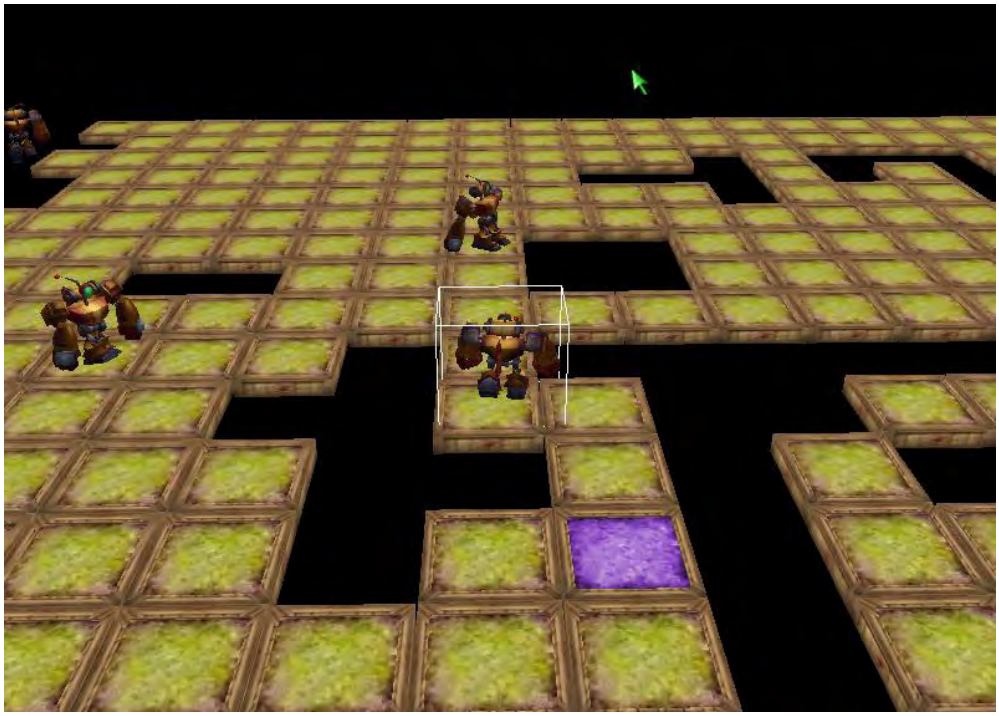


Figura 3.5: Ambiente virtual que muestra la representación gráfica del mapa y algunos agentes en él.

4. Agentes y el algoritmo genético

4.1 Estructura de los Agentes

Con el fin de implementar un modelo flexible de agente que pueda ser usado en el simulador, que permita reutilizar código y extenderlo para crear diversos tipos de modelos, se definen una serie de características básicas comunes a todos los agentes, éstas características se implementan en la clase `Agent`, raíz de la jerarquía. En `Agent` se declara un agente básico con percepciones, acciones y variables relativas a su comportamiento.

El primer método que implementa es `init()`, que tiene como fin realizar los procesos básicos de inicialización del agente, como establecer el número de estados utilizados en el comportamiento (menor o igual al número de estados en los que puede estar el agente), dar valores iniciales a algunas variables específicas de sus acciones o regresar al agente al estado inicial. Este método se llama en el constructor del `Agent` y cada que se modifica su comportamiento. También implementa un método de ejecución del mismo, `Behave()`, este método será independiente de las acciones disponibles y las percepciones del agente y se encarga básicamente de ejecutar su FSM asociada.

Cada objeto de la clase `Agent` define también un método de asignación de comportamiento `setBehavior()`, que permite asignar o cambiar el comportamiento del agente en cualquier momento, de esta forma, para evaluar a toda una población de *n* comportamientos no es necesario crear *n* agentes. Se crea un solo agente y se le asigna un comportamiento a la vez para ser evaluados.

En general cada objeto de la clase `Agent` define una serie de métodos, de los cuales algunos son *virtuales*, es decir, se deben de implementar en las clases hijas ya que definen procesos específicos de cada tipo de agente. De tal forma que se tiene una

jerarquía con las siguientes clases (véase *Figura 4.1*):

- Clase `Agent`: Clase que define un agente general, sin representación gráfica y con métodos de acción, percepción y ejecución de comportamiento, declara los métodos virtuales:
  - `void executeAction(int actionIndex, float time = 0.0)` – Ejecuta una acción del agente dado su índice y el tiempo transcurrido desde la llamada anterior. De esta forma, este método llamará a la acción correspondiente dado el estado actual de la FSM.
  - `float getSensorReading(int sIndex)` – Evalúa una condición dado su índice, el resultado de esta evaluación es un número real.
- Clase `GameAgent`: Hereda de `Agent` y define un agente situado con atributos de posición, orientación, tamaño etc.
- Clase `OgreActor`: Define una entidad estática con representación gráfica en el ambiente virtual. Utiliza funciones del motor gráfico para cargar la geometría asociada y dibujarse en escena.
- Clase `OgreMovileActor`: Hereda de `OgreActor` y define un Actor, una entidad móvil y con representación gráfica. Agrega atributos como velocidad, funciones de movimiento, animación, orientación entre otras.
- Clase `SimpleRobot`: Es una clase de ejemplo que define un agente específico para ser simulado en el ambiente virtual.

El simulador rápido utiliza una instancia de alguna clase que herede de `GameAgent` para simular el comportamiento de los agentes que no requieren representación gráfica. En cambio, el simulador lento del ambiente virtual utiliza objetos de alguna clase específica que herede de `GameAgent` y `OgreMovileActor`. Es importante notar que se permite el uso de herencia múltiple en el lenguaje C++ a diferencia de otros, sin embargo existen mecanismos para implementar este modelo en lenguajes orientados a objetos como java haciendo modificaciones ligeras.

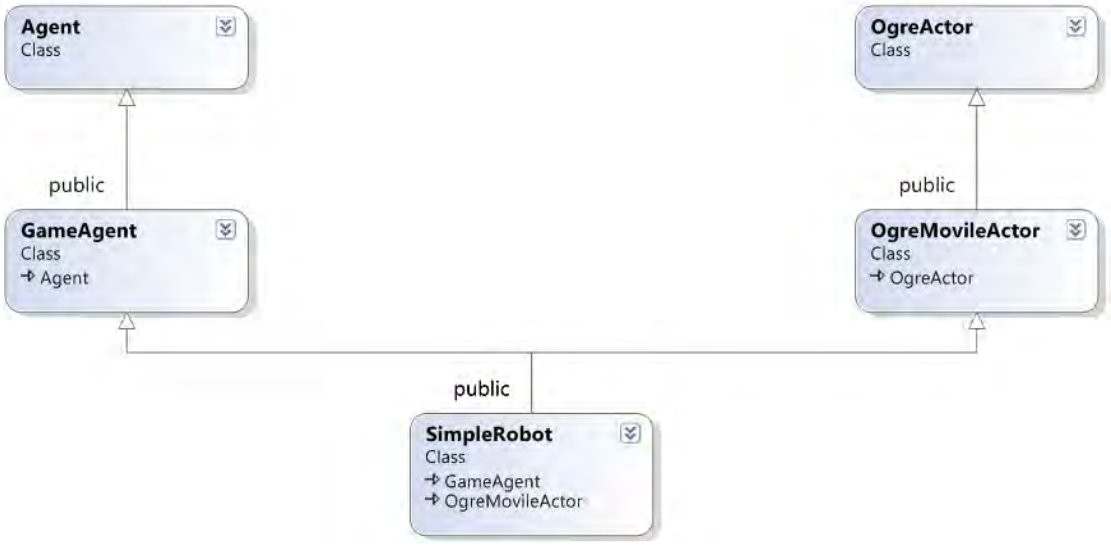


Figura 4.1: Jerarquía de clases de los agentes en el ambiente virtual.

4.1.1 Acciones y estados

Cada agente tiene un conjunto de acciones, que son procesos o tareas básicas independientes que puede realizar, combinaciones de estas tareas básicas pueden resultar en acciones nuevas más complejas, por ejemplo, si un agente puede caminar y evadir obstáculos, eventualmente podrá explorar el entorno en su totalidad. Su comportamiento define el orden en el que se ejecutan estas acciones dado un conjunto de percepciones de entrada.

Los tipos de acciones que puede realizar un agente son diversas, desde moverse, que se considera una acción sobre el *entorno* ya que afecta sus percepciones futuras, hasta encontrar un camino entre dos puntos utilizando un algoritmo de búsqueda como A*, que si bien no modifica el entorno, si modifica las variables internas del agente y afecta sus acciones futuras. Éstas acciones dependen del diseño previo del agente, a grandes rasgos, se crea uno con un conjunto de acciones tan general o particular como el diseñador decida, y al hacerlo, se están creando los estados posibles de la FSM que define su comportamiento, entonces la tarea del algoritmo genético es *encontrar las transiciones entre estos estados*.

El diseñador debe establecer el conjunto de estados y percepciones posibles de cada agente, cada uno de manera individual sin preocuparse por las transiciones entre ellos. Este conjunto debe estar relacionado con el tipo de comportamiento o metas que

se espera que el agente logre. Un agente que no sabe volar no podrá realizar tareas que requieran esta habilidad o si se requiere un agente que persiga un objeto, se le debe de proveer de un mecanismo de percepción de dicho objeto ya que si no se da cuenta que este existe nunca podrá perseguirlo.

En un principio cada acción se codifica en un método por separado en la clase que define a cada agente, cada método está asociado a un estado de la FSM. Si cada acción se implementa por separado, se puede incurrir en un problema de duplicidad de código ya que muchas acciones pueden utilizar los mismos procesos básicos.

Para esto se pueden crear un conjunto de primitivas de acción, que son procesos básicos de los que se componen las acciones, por ejemplo, cada agente tiene un vector de dirección asociado y moverse en dicha dirección es una primitiva, las acciones de correr, evadir o huir harán uso de ella evitando que se duplique código de movimiento en cada una. Si las acciones no tienen procesos internos en común puede no ser necesario crear dichas primitivas.

Cada acción tiene asociadas tres funciones generales: `enter`, `update` y `exit`. La primera se ejecuta justo al entrar al estado dado y después de la función de finalización del estado anterior (`exit`), en ella se pueden inicializar valores o ejecutar procesos que sirvan para crear las condiciones necesarias para la ejecución del estado, posteriormente se ejecuta `update` una y otra vez hasta que se den las condiciones necesarias para un cambio de estado.

4.1.2 Evaluación de condiciones

Al igual que el conjunto de acciones, también se debe diseñar e implementar el conjunto de percepciones del agente (sensores), por convención, cada percepción o valor de un sensor en un tiempo dado será representado por un número real, aún cuando la lectura sea booleana, asignando 0.0 al valor *falso* y *verdadero* en cualquier otro caso. De esta forma, en la clase del agente se definen una serie de funciones que representan condiciones. Pueden recibir como parámetros de entrada la lectura de un sensor o una constante e internamente operan estos datos utilizando operadores aritméticos, lógicos o de comparación para evaluar algún aspecto de interés para el desarrollador. También se les puede asignar prioridades para casos en los que se

cumplan más de una condición en un momento dado.

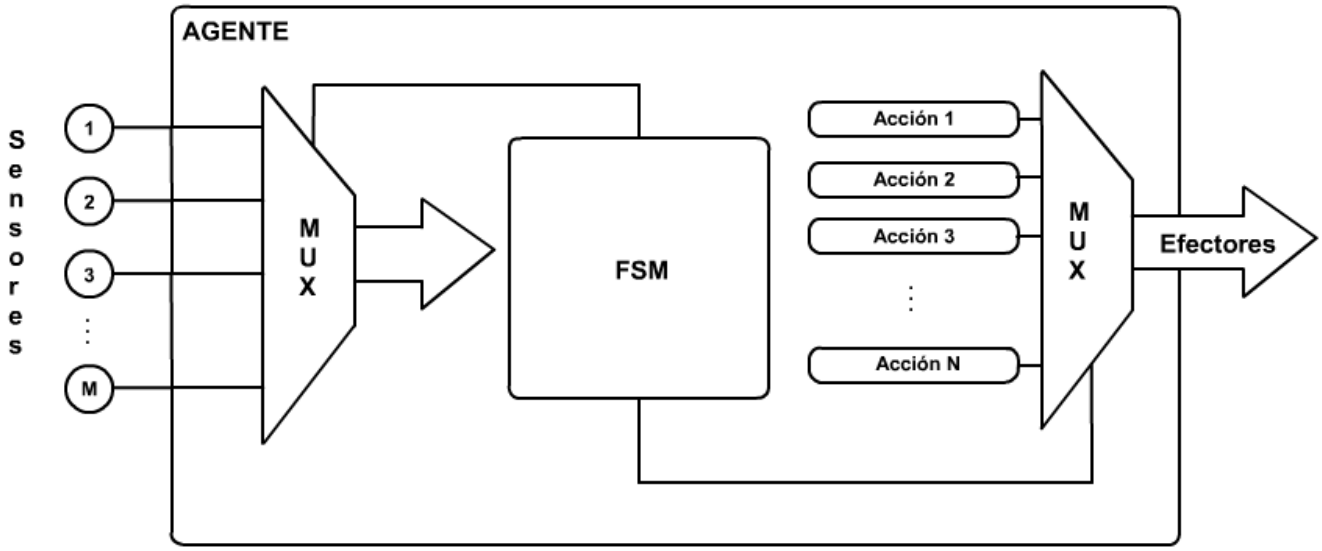


Figura 4.2: Estructura general de los agentes.

Cada estado tiene un conjunto de transiciones asociadas a él, cada una de estas transiciones tiene en principio una condición aunque pueden ser varias conectadas con operadores lógicos. Durante el proceso de ejecución de comportamientos, en cada ciclo de la FSM se deben evaluar todas las condiciones asociadas al estado actual en algún orden específico. Dependiendo del diseño del agente, este puede cambiar de estado al cumplirse la primera condición, la última o tomando en cuenta la de mayor prioridad entre todas las que se cumplieron.

4.1.3 Ejecución del comportamiento

Cada agente conoce su comportamiento, es decir, tiene acceso en todo momento a todos los datos que lo definen. Estos datos deben estar organizados de tal forma que el agente pueda implementar un método universal para ejecutarlos independientemente de las características específicas de la FSM, esto mientras el número de estados en la FSM sea menor o igual al número de acciones del agente así como el número de condiciones.

Éste ciclo de ejecución sigue los siguientes pasos básicos y como se mencionó

anteriormente, se implementa en el método `Behave` de la clase `Agent`:

1. Mientras el agente exista.
    a. Para cada condición *i* asociada al estado actual:
    b. Si *i* se cumple
        - Agrega a la a lista de prioridad el estado destino de la transición asociada a la condición *i.*
    c. Si lista de prioridad NO está vacía.
        - Cambiar al estado con mayor prioridad:
            1. Llamar *exit()* de estado actual.
            2. Hacer estado destino el estado actual.
            3. Llamar *enter()* de estado actual.
    d. Si la lista SI está vacía
        - Llamar *update()* de estado actual.

Para los agentes con representación gráfica, se deberá ejecutar al final de cada ciclo su función de dibujo correspondiente para actualizar la geometría en el ambiente virtual.

4.1.4 Pong: Un ejemplo básico

Se muestra un ejemplo sencillo de un agente denominado `SimpleRobot` que tiene como objetivo jugar el juego *Pong (http://www.ponggame.org/*) con el objetivo de mostrar a grandes rasgos el diseño de los componentes básicos de un agente. Como se mencionó anteriormente, primero se debe crear una clase que herede de `GameAgent`, y de `OgreMovileActor` en caso de requerir representación gráfica.

Un `SimpleRobot` representa a un jugador de *Pong* controlado por la computadora, se le concibe como un agente que puede actuar moviendo la "raqueta". El proceso de diseño requiere de varios pasos:

- Definir e implementar el conjunto de primitivas de acción, en caso de ser requeridas. Las acciones básicas que el agente puede realizar se basan en el movimiento vertical de la raqueta. Entonces, se puede crear una primitiva de movimiento, por ejemplo `MoveToDirection(dx,dy)`, que permite mover al

agente hacia alguna dirección dada por el vector (dx,dy).

- Diseñar y construir el conjunto de acciones del agente. Se busca que el agente pueda jugar *Pong*, lo que requiere mover la raqueta hacia arriba o hacia abajo, entonces se definen 2 acciones:
  - `MoveUp()` - Mueve hacia arriba *delta* unidades, hace uso de `MoveToDirection`: `MoveToDirection(0,delta);`
  - `MoveDown()` - Mueve hacia abajo *delta* unidades, hace uso de `MoveToDirection`: `MoveToDirection(0,-delta);`
  - Por último, el agente simplemente puede no moverse, lo que plantea la necesidad de una tercera acción, estar quieto. `Idle()` - El agente entra en modo de espera, no se mueve. Una vez definido el conjunto de acciones que el agente puede realizar, se numeran.

Índice	Acción
1	`Idle()`
2	`MoveUp()`
3	`MoveDown()`

- Definir e implementar el conjunto de percepciones del agente con base en su estructura propia y en el entorno.

  El agente propuesto necesita percibir 2 cosas básicamente:
  - La posición de la pelota: basta con la posición relativa a él, en particular saber si la pelota se encuentra más arriba de él (coordenada *y* mayor) o más abajo (coordenada *y* menor).
  - Los límites de la "mesa": Será necesario que el agente pueda percibir los límites del entorno ya que no debe poder salir del área de juego. Como el movimiento solo es vertical, solo el límite inferior y superior son de interés.

Con base en estas consideraciones se plantean las siguiente condiciones que el agente puede evaluar o percibir.

Índice	Condición
1	*reachTop*
2	*reachBottom*
3	*ballAbove*
4	*ballBelow*

- Contar con una representación simbólica del entorno en el que el agente estará inmerso. En este caso, se trata de un espacio bidimensional de forma rectangular en donde se encuentra una "pelota" representada por un punto y otra "raqueta" representada por un rectángulo.

- Definir la representación gráfica del agente. El método de dibujo dibujará la raqueta como un rectángulo centrado en la posición del agente.

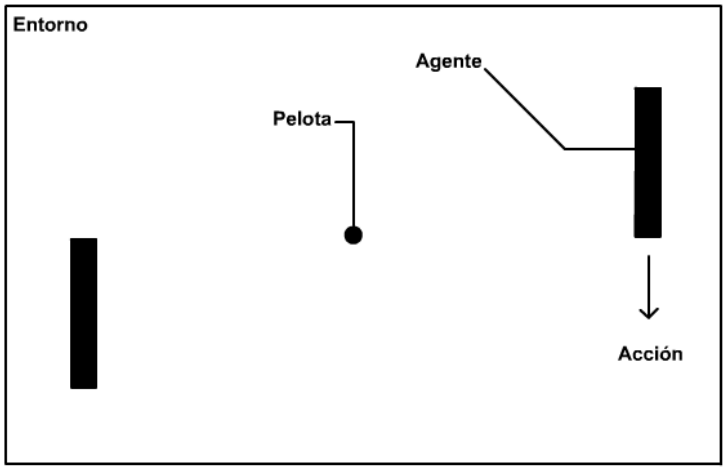


Figura 4.3: Representación gráfica del agente que juega Pong y su entorno.

De esta forma se tiene un agente que puede realizar 3 acciones básicas y puede evaluar 4 condiciones de su entorno, esto da como resultado 5 posibles valores para cada transición tomando en cuenta el 0 o ausencia de la misma, por lo tanto, existen $5^9 = 1{,}953{,}125$ máquinas posibles. En principio, con el comportamiento adecuado que relacione las acciones correctas con las percepciones correctas, el agente puede cumplir satisfactoriamente la meta de jugar *Pong*. Es un ejemplo muy básico y pequeño, diseñar un comportamiento manualmente para este problema no es una tarea difícil, sin embargo se complica para agentes más complejos que tengan que cumplir metas más complejas.

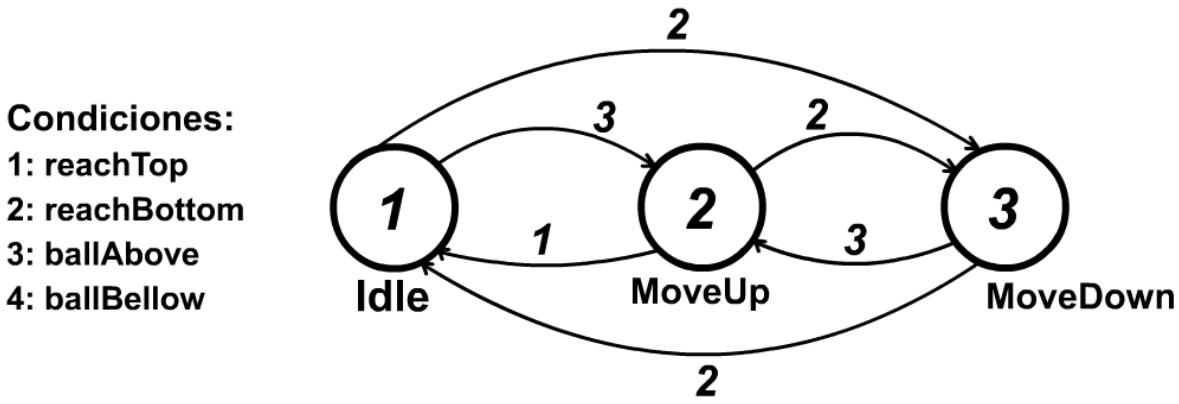


Figura 4.4: Ejemplo de un comportamiento que juega Pong.

4.2 Representación del Entorno

El entorno se implementa utilizando el *patrón de diseño singleton* para poder ser visible por todos los agentes, además de asegurar que haya una sola instancia del mismo durante toda la aplicación, así los cambios que haga un agente en él, se reflejan de inmediato a los demás. Este modelo es parecido a tener una memoria compartida, sin embargo los problemas de concurrencia no se presentan ya que cada agente es evaluado secuencialmente, nunca dos agentes podrán modificar el entorno al mismo tiempo. En caso que se desee realizar una implementación concurrente se debe de tomar en cuenta el control de acceso a las variables del entorno como memoria compartida para asegurar la integridad de los datos.

El entorno es bidimensional, aunque su representación gráfica sea mediante objetos en tres dimensiones, esto facilita muchos cálculos relacionados con la detección de colisiones y la búsqueda de caminos. Esta representación se hace mediante un conjunto de polígonos en el plano que representan obstáculos fijos. El entorno puede tener diversas características específicas dependiendo del tipo de simulación, por ejemplo, gravedad, velocidad del viento, temperatura o cualquier otra variable perceptible por los agentes en estudio. Sin embargo las características básicas mínimas del mapa serán sus dimensiones y la lista de áreas poligonales que definen su estructura.

4.2.1 Áreas poligonales

Todo objeto estático en el mapa se representa con uno o varios polígonos convexos, a nivel simbólico cada uno de estos objetos del mapa definen áreas: una pared, un río, un conjunto de árboles o una casa. Esta área definida por polígonos convexos representa la proyección de dichos objetos en el plano horizontal, donde se encuentra el piso. Cada polígono está definido por una lista de puntos ordenados en sentido horario y tiene propiedades de penalización al movimiento, para una penalización de 0% se trata de un área caminable al 100% de velocidad para los agentes, para el 100% de penalización se trata de un área no pasable como una pared o un barranco. En la figura 4.5 se muestra un ejemplo de este tipo de representación.

Valores intermedios implican dificultad para caminar, peligro o terreno difícil como un pasillo con fuego, una sección con arena o pendientes muy prolongadas donde el agente gasta más energía al pasar. Esta información se toma en cuenta para encontrar caminos mínimos con el algoritmo A*, ya que modifica los costos de movimiento entre nodos. Se utilizan polígonos convexos para simplificar procesos como cálculo del área, detección de colisiones o evaluar si un punto se encuentra dentro de él, útil para saber sobre qué área está parado un agente.

Para asegurar que cada polígono es convexo, se miden cada uno de sus ángulos internos, estos deben ser menores a 180°, en otras palabras, al moverse de un vértice a otro adyacente en un sentido dado (horario o anti-horario), se debe rotar siempre en la misma dirección. El procedimiento es simple:

- Para cada vértice $v_i = (x_i, y_i, 0)$ en el polígono
- Se toman los vectores $a = v_{i-1} - v_i$, $b = v_{i+1} - v_i$
- Se calcula axb
- La componente z del vector resultante debe tener el mismo signo en todos los casos.

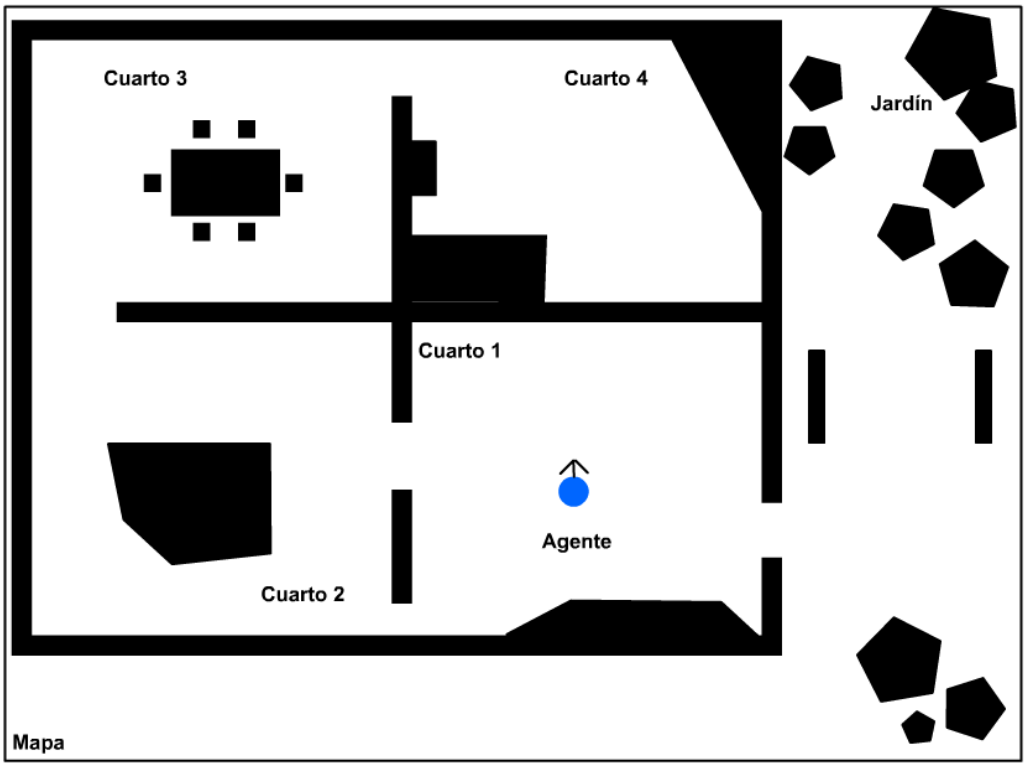


Figura 4.5: Ejemplo de mapa poligonal de una casa.

4.2.2 Obstáculos

La representación con áreas poligonales define el conjunto de obstáculos fijos en el ambiente, es decir, que su posición, tamaño u orientación no varían con el tiempo. Cada una de las aristas de los polígonos define límites entre áreas, de tal forma que es necesario tener un método que permita saber si un agente o un punto arbitrario sobre el entorno se encuentran dentro o fuera de un polígono. En principio, un agente no puede estar parado dentro de un polígono que define un área no pasable ni atravesar sus aristas de afuera hacia adentro.

Para lograr esto, se puede proveer a los agentes de sensores simulados representados por segmentos de recta que "salen" de él en distintas direcciones, si alguno de estos segmentos se cruza con una arista de algún polígono puede calcularse el punto de intersección entre ellos y así calcular la distancia a la que se encuentra el agente de colisionar. El comportamiento del agente debe determinar qué acciones realizar al detectar o no un obstáculo cerca.

Otra forma de hacer frente a la evasión de obstáculos es implementando un algoritmo de búsqueda de rutas, por ejemplo, A* que encuentra caminos óptimos entre dos nodos de un grafo siempre que existan. Esto provee al agente de acciones de alto nivel que pueden ser utilizadas en su comportamiento, sin embargo, antes de hacer uso de A* es necesario definir a las áreas pasables del entorno por medio de un grafo, para eso se utiliza una representación basada en celdas.

4.2.3 Representación basada en celdas

Dada la representación poligonal del mapa, se puede definir una malla de celdas cuadradas superpuesta al mapa, esta malla tendrá las mismas dimensiones definidas en esta representación poligonal. La cantidad de filas y columnas estará dada por la resolución de la malla. Si las dimensiones del mapa son *w* (ancho), *h* (alto) y el tamaño del lado de cada celda es de *r* unidades, la malla tendrá un total de $\left(\frac{w}{r}\right) \times \left(\frac{h}{r}\right)$ celdas. Es recomendable que *w* y *h* sean múltiplos de *r* para no lidiar con celdas incompletas.

Cada celda, al igual que cada área poligonal, tiene asociado un nivel de penalización que tomará del área a la cual pertenezca. También, el centro de cada celda define un nodo en el grafo que representa al mapa, toda celda tiene conexión con sus celdas adyacente que sean pasables, entonces cada una tiene un máximo de ocho vecinos, cuatro en diagonal, dos en dirección horizontal y dos en vertical. El costo asociado a cada arista entre dos nodos será modificado por el nivel de penalización del nodo destino, definiendo así el costo real de moverse de un nodo a otro. El costo de moverse a nodos diagonales es 1.4142 veces mayor al de moverse horizontal o verticalmente ya que la distancia es mayor en esta proporción. Este mapa de celdas se crea a partir del mapa poligonal realizando el siguiente procedimiento:

- Para cada *t* celda en la malla
  - Asignar a la celda propiedad de pasable
  - Para cada polígono *p* en la representación poligonal
    - Si el centro de *t* se encuentra dentro de *p*
      - Se asigna a *t* el valor de penalización de *p*

Este proceso se ejecuta una sola vez al inicio de la aplicación ya que al contener información de obstáculos fijos, su estructura no se espera que cambie.

En la *Figura 4.6* se muestra un ejemplo de equivalencia entre un mapa poligonal y una malla de celdas. Ambas representaciones son útiles para diversos procesos, por ejemplo, para saber en qué área se encuentra un agente simplemente se lee este valor de la celda sobre la cual está parado evitando calcularlo para todos los polígonos. Por otro lado el mapa poligonal es más útil para agentes que basan su movimiento en sensores y en la detección de fronteras entre áreas.

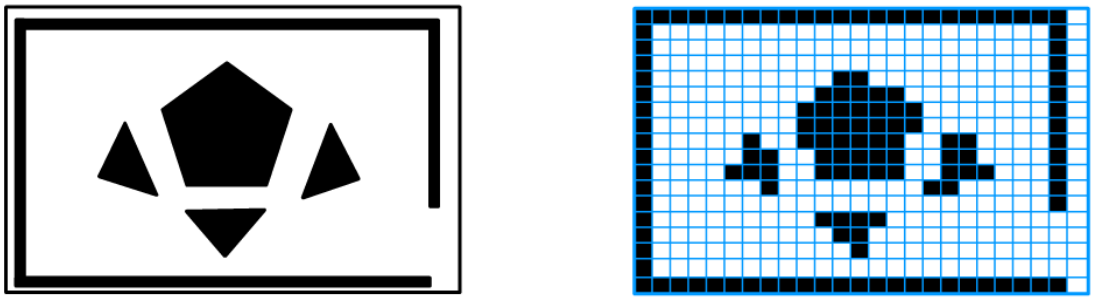


Figura 4.6: Un mapa representado con polígonos (izquierda) y con una malla de celdas (derecha). Las áreas negras indican obstáculos y las blancas áreas transitables.

La representación por celdas se puede representar como una matriz de enteros y sobre ella trabaja el algoritmo A* para encontrar caminos mínimos. Esta forma de representar el grafo es bastante compacta ya que no se utiliza espacio adicional para representar las aristas entre nodos. Es importante notar que mientras mayor resolución tenga la malla, mayor será la cantidad de nodos del grafo, lo que se refleja en una representación más fiel del mapa poligonal pero también en mayor carga de trabajo ya que además del consumo de memoria al guardarlos, A* debe buscar en una mayor cantidad de nodos.

4.2.4 Archivos .mp

Los *archivos .mp* son básicamente archivos con formato XML que contienen la información estructurada de los polígonos que componen un mapa. Estos archivos son generados por el editor de mapas, un programa que provee una interfaz gráfica para dibujar estos polígonos y asignarles características de penalización. El simulador carga este archivo para construir la representación poligonal y de celdas del entorno donde los agentes ejecutarán su comportamiento.

Dentro del archivo XML, la definición del mapa se realiza con la etiqueta `<World>` que tiene como atributos:

- min_X: define el valor mínimo en x.
- max_X: define el valor máximo de x.
- min_Y: define el valor mínimo en y.
- max_Y: define el valor máximo de y.

Entonces la línea:

```
<World min_X="0" max_X="100" min_Y="0" max_Y="200"></World>
```

Define un área cuadrangular de 100 unidades de ancho y 200 de alto cuyo origen se encuentra en 0,0.

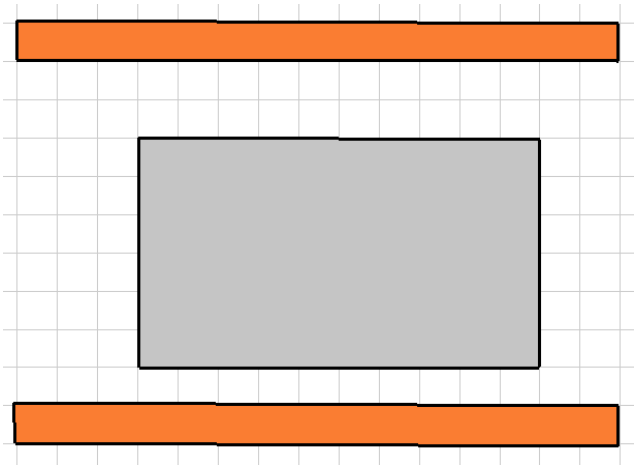


Figura 4.7: Mapa poligonal creado por el editor.

Cada polígono en el mundo se define utilizando la etiqueta <Polygon> que tiene como atributos:

- place: Nombre del grupo al que pertenece el área poligonal.
- name: Nombre del área que define el polígono.
- penalty: Porcentaje de penalización.

A su vez cada polígono está compuesto de vértices, cada vértice se define utilizando la etiqueta <vertex> con sus coordenadas x,y como atributos. Cada vértice aparece en el orden dado al recorrer el polígono en sentido horario.

Listado 4.1: Muestra el archivo .mp correspondiente al mapa de la Figura 4.6

```
<World min_X="0" max_X="100" min_Y="0" max_Y="200">
	<polygon place="MAP_NAME" name="MAP_POLYGON_0" penalty="0">

		<vertex x="15" y="85.2304"/>
		<vertex x="15" y="90.3794"/>
		<vertex x="89.7436" y="89.9729"/>
		<vertex x="89.7436" y="85.0948"/>

	</polygon>
	<polygon place="MAP_NAME" name="MAP_POLYGON_1" penalty="0">

		<vertex x="14.8718" y="35.0949"/>
		<vertex x="14.6154" y="40.3794"/>
		<vertex x="89.7436" y="39.9729"/>
		<vertex x="89.7436" y="34.6883"/>

	</polygon>
	<polygon place="MAP_NAME" name="MAP_POLYGON_2" penalty="58">

		<vertex x="30.1282" y="75.0677"/>
		<vertex x="80" y="74.7967"/>
		<vertex x="79.8718" y="44.9865"/>
		<vertex x="30.2564" y="44.9865"/>

	</polygon>
</World>
```

4.2.5 Carga de mapas desde archivos .tga

El sistema también permite cargar mapas a partir de una imagen en *formato .tga*, esta imagen debe tener algunas características específicas como: utilizar únicamente escala de grises, que sus dimensiones sean de preferencia una potencia de dos y tiene que ser guardada sin ningún tipo de compresión.

El proceso consiste en dibujar una imagen en algún programa de edición (por ejemplo Gimp, http://www.gimp.org/) y guardarla en *formato .tga* sin compresión RLE. Posteriormente se carga desde el sistema con un objeto de la clase `CASMap`, la representación del mapa se guarda en una estructura de datos llamada `asGridMap` que contiene toda la información necesaria para la simulación de los agentes como el mostrado en la figura 4.8.

Así, existen tres formas para generar la representación por celdas de un mapa: tomando como entrada el archivo .mp del editor de mapas, utilizando una imagen .tga y por último, utilizando un archivo de texto plano con la matriz de nodos escrita explícitamente, cada una de ellas tiene sus ventajas y desventajas y dan como resultado el mismo `asGridMap` necesario para la simulación.

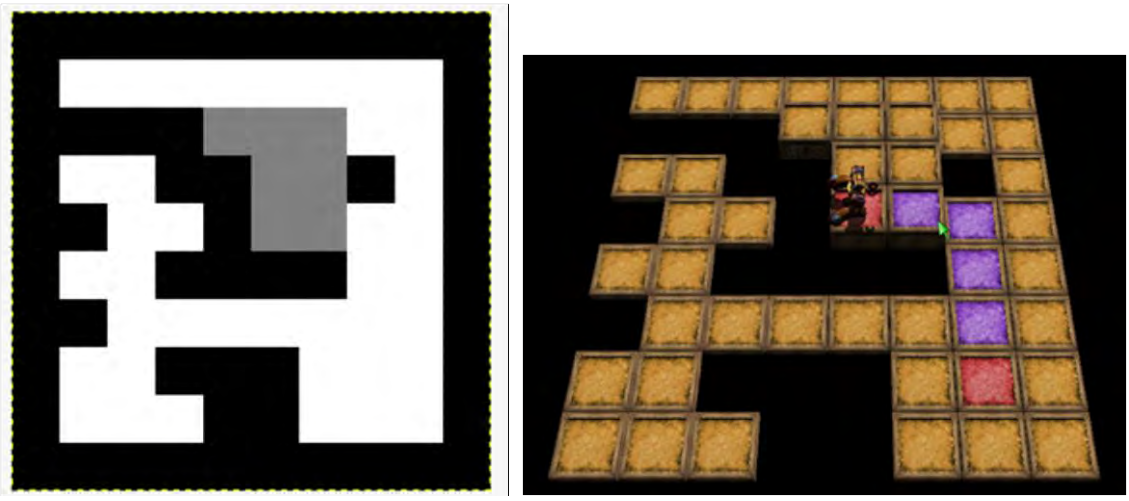


Figura 4.8: Un mapa en una imagen .tga (izquierda) y su representación gráfica en el ambiente virtual (derecha).

4.3 Representación de los comportamientos

Para poder trabajar con FSMs es necesario diseñar una forma fácil y flexible para su representación dentro del sistema, que contenga todos los aspectos de su estructura y a partir de la cual se pueda ejecutar fácilmente el comportamiento al que representa así como operarla para lograr el proceso de evolución. Ésta representación puede hacerse utilizando clases o diversas estructuras de datos dependiendo de la aplicación y las operaciones que se pretenda realizar sobre los datos.

Para hacer posible el proceso de evolución de comportamientos, la representación de las FSMs debe poder ser operada por un algoritmo genético, esto implica que se puedan aplicar operadores de selección, cruzamiento y mutación a conjuntos de estos elementos, de preferencia, sin hacer procesos adicionales como conversiones o decodificaciones para salvar ciclos de reloj. Esto se puede lograr con una representación por medio de una cadena finita de números, de preferencia enteros para simplificar operaciones entre ellos.

En el caso de un robot real, se puede tener un conjunto bien definido de entradas, por ejemplo las lecturas binarias de cada sensor instalado, lo cual da como resultado 2^k combinaciones posibles para k sensores diferentes. Esto permite considerar en la FSM que define su comportamiento todas las posibles entradas que quedan a su vez definidas por la estructura misma del robot.

En agentes virtuales, este conjunto de entradas posibles debe ser cuidadosamente diseñado por el desarrollador ya que en general no se trata de sensores simples, si no de funciones complejas las que definen cada percepción. Esto hace necesario representar explícitamente cada una de las capacidades sensoriales deseadas en el agente así como las combinaciones u operaciones lógicas entre ellas.

Tomando como referencia la construcción de FSMs utilizando memorias por direccionamiento entrada-estado descrita en [18] y su utilización en robots móviles, se presenta un método de codificación para máquinas de estados generalizadas que utilizarán los agentes virtuales. Este método toma en cuenta el tipo de percepciones que puede tener un agente virtual que en general pueden ser de cualquier tipo, siempre y cuando pueda simularse en el entorno.

4.3.1 Representación de longitud fija

A cada comportamiento de los agentes le corresponde una FSM que se puede representar como un conjunto de relaciones del tipo: Si estando en el estado Q se cumple la condición C, ejecuta la acción X y pasa al estado R. A lo más existen N_S^2 transiciones presentes en la máquina de Moore, donde N_S es el número de estados posibles, y cada una de ellas se puede representar por un número entero asociado al índice de la condición que debe de cumplirse. La acción a ejecutar para cada estado está definida por el diseño mismo del agente, siempre un estado ejecutará la misma serie de acciones. Los estados origen y destino están definidos por la posición de las condiciones en una matriz de adyacencia.

La matriz de adyacencia es de tamaño N_SxN_S, las posiciones verticales representan los estados origen y las horizontales los estados destino. Entonces, el elemento $M_{QR} = C$ representa la transición entre el estado Q y R al cumplirse la condición C. El número de valores posibles para cada elemento de la matriz es N_m+1, el número de condiciones posibles más la transición vacía o ausencia de transición que se representa con el 0. Así, los elementos de la matriz forman una cadena de N_S^2 enteros, donde cada uno representa una de las transiciones posibles en una máquina dada.

La representación es de longitud fija, siendo ésta N_S^2 para toda máquina independientemente de cuantas transiciones estén presentes en ella y cuantos estados se tomen en cuenta, cada máquina tiene tantas transiciones como elementos diferentes a cero en su representación.

Para extender el diseño y poder representar máquinas de Mealy, se le incorpora a cada transición el identificador de la acción a ejecutar cuando se cumpla su condición. Para esto, la longitud de la cadena de enteros que la representa crece al doble ya que a cada transición se le asigna una pareja de números (c,a) que determina la condición y la acción a ejecutar. Este cambio da como resultado máquinas de Mealy, aunque no se puede representar cualquiera de ellas, solo las que tienen un máximo de N_S transiciones por estado. Sin embargo, esta deficiencia se puede subsanar aumentando el número máximo de estados.

Las ventajas principales de utilizar esta representación son las siguientes:

- Es una representación natural, sencilla y fácil de entender.
- Por su estructura puede ser fácilmente operada por un algoritmo genético de forma que se convierte directamente en un individuo para el cual se pueden definir operaciones de cruzamiento y mutación que den como resultado otro individuo válido.
- También, dada esta cadena de enteros, un agente puede ejecutar el comportamiento de forma eficiente sin "decodificarlo" o realizar procesos adicionales, esto permite mantener una comunicación entre el algoritmo genético y el simulador de forma directa, fácil y rápida.
- Se puede implementar fácilmente con estructuras de datos básicas como arreglos o listas simples, lo que se encuentra en casi cualquier lenguaje de programación.

Algunas desventajas:

- Al ser una representación de longitud fija, para máquina con pocas transiciones se utiliza espacio innecesario ya que su representación debe guardar muchos ceros.
- Poco flexible a la hora de representar condiciones compuestas, utilizando conectores lógicos, pero se puede subsanar en el diseño del agente aunque le quita flexibilidad al algoritmo. Sólo se puede tener una condición entre 2 estados.
- No permite que el algoritmo genético explore condiciones diferentes a las diseñadas por el desarrollador.

4.3.2 Otras representaciones

Existen otras representaciones para FSMs, cada una de ellas con diferentes ventajas y desventajas, por ejemplo se puede pensar en una representación de longitud variable donde cada individuo pueda tener un número de estados diferente o más de una condición asociada a una transición entre dos estados conectadas por operadores lógicos.

Esto daría la posibilidad de que el genético no solo evolucione el conjunto de transiciones presente y sus condiciones si no el número de estados y hasta sus acciones asociadas. Sin embargo, manejar individuos con longitudes diferentes hace más compleja su operación, en específico el diseño de los operadores genéticos. También, necesariamente se debe definir un espacio finito de condiciones posibles y modificar la estructura de los agentes que ejecutan los comportamientos.

En este trabajo se experimenta únicamente con la representación de longitud fija expuesta anteriormente para verificar su factibilidad en aplicaciones reales. El estudio de otras representaciones de comportamientos basados en FSMs está más allá de sus objetivos.

4.4 Descripción General del algoritmo genético

El proceso de evolución de FSMs se realiza por medio de un algoritmo genético, ya que se requiere realizar una búsqueda en el espacio de las máquinas posibles, este espacio puede ser demasiado grande haciendo difícil o imposible explorarlo por medio de métodos exhaustivos. Éste algoritmo genético debe tener una serie de características particulares que le permitan operar y comunicarse con un ambiente virtual de simulación.

En primer lugar debe ser flexible, en el sentido de poder utilizar diversos tipos de individuos y poder ser extendido para aplicarlo a problemas particulares donde, si bien se utilizan operadores genéticos, individuos y parámetros específicos, la estructura general del algoritmo no cambia. Dado esto, y utilizando las capacidades de programación genérica del lenguaje C++ , se creó una *clase template* para un algoritmo genético genérico que pueda operar individuos cuyos elementos pueden ser

números enteros, reales, valores booleanos o cualquier otro tipo de dato soportado por esta característica del lenguaje de programación

Esta clase genérica provee de las funcionalidades generales así como de una plantilla para la construcción de cualquier algoritmo genético en donde únicamente se deben implementar las funciones específicas como son los operadores genéticos o definir el tipo de individuos que se utilizarán y cambiar los parámetros como el número de individuos por generación, el número total de generaciones y las probabilidades de cruzamiento y mutación.

Además, el algoritmo genético diseñado tiene características muy particulares para poder trabajar de forma fácil y eficiente con el ambiente de simulación y el ambiente virtual. Por ejemplo, para salvar tiempo de proceso, es posible distribuir su operación completa entre frames, es decir, en vez de calcular todas las generaciones del algoritmo en un solo *frame* de dibujo, se puede calcular hasta una generación por *frame*, así se evita sobrecargarlos evitando que el sistema se trabe.

También, es importante notar que el cálculo del *fitness* de cada individuo puede no ser realizado por el algoritmo genético, por eso se le ha dotado de una función que sirve como interfaz para mandar a los individuos a calificar y recibir el valor de su calificación desde un proceso o sistema externo. Esto se utiliza para recibir la calificación de cada comportamiento calculada en el simulador rápido.

Entonces, el proceso de calificación es independiente del AG, por lo que el comportamiento a calificar dependerá totalmente del simulador, pues él decide cual individuo es bueno y cual no lo es. Así, el mismo algoritmo genético sirve perfectamente para encontrar diferentes comportamientos incluso si las acciones y percepciones del agente cambian, sin embargo, por parte del simulador, se necesita crear una función de *fitness* diferente para cada uno de ellos e incluso modificar las simulación en su totalidad si es que el agente cambia.

4.4.1 Individuos

Los algoritmos genéticos tienen la característica de trabajar con conjuntos de soluciones llamados poblaciones, a cada una de estas posibles soluciones se le llama individuo y que, en este caso es una cadena de enteros que representa una FSM. Sin embargo, en diferentes problemas, se puede requerir una representación binaria o incluso con números reales o hasta de diferentes longitudes, es por eso que al igual que con el AG se creó una *clase template* que permita crear individuos cuyos datos sean de diferentes tipos evitando duplicar código.

También se hace uso de una estructura de datos lineal que ocupa memoria dinámica con el fin de poder variar la longitud de cada individuo, durante el proceso se necesita crear al menos dos poblaciones de individuos, la actual y la siguiente, además de aquellos que aparecen como resultado de la operación de cruza. Se debe ser cuidadoso para no tener problemas de fugas de memoria ya que el lenguaje C++ no tiene un recolector de basura o mecanismos que lo eviten.

En el *Listado 4.1* se muestra la declaración de la *clase template* `Individual`. Lo primero a notar es el parámetro de tipo que recibe `T`, este puede tomar como valor cualquier tipo de dato simple o compuesto. Como miembros de esta clase se tiene un apuntador `mData` de tipo `T`, que apunta al bloque de memoria donde se guardarán los datos. `mSize` guarda el tamaño de éste arreglo para poder recorrerlo.

Se sobrecarga el operador de asignación, esto para evitar fugas de memoria copiando manualmente cada elemento siempre que se igualen dos individuos. También se implementa el constructor copia que crea un individuo nuevo a partir de otro del mismo tamaño y con los mismos elementos pero ocupando memoria diferente.

Listado 4.2: Declaración de la clase Individual.

```
template <class T>

class Individual{

	int mSize;

public:
	T *mData;


	Individual(int size  = 0);
	Individual(int size,T initValue);
	Individual(const Individual<T> &copy);
	~Individual();

	int  getSize(){return mSize;}
	void print();

	Individual<T> & operator = (const Individual<T> &op);
};
```

Así, la clase Individual básicamente es una envoltura que permite crear arreglos dinámicos de cualquier longitud y tipo en sus datos. Por ejemplo, un individuo con datos booleanos de longitud 256 con todos sus valores inicializados en falso, se crea de la siguiente forma:

```
Individual<bool> myIndividual(256, false);
```

La principal ventaja de este esquema es que se tiene una sola definición de clase que permite crear una variedad muy grande de individuos para problemas diversos, y que además de reutilizar código, se aprovecha una de las características más importantes del lenguaje de programación utilizado, la *programación genérica*.

4.4.2 Operadores genéticos

Cada algoritmo genético particular debe implementar sus operadores genéticos a la medida con base en sus necesidades y estructura. Para el problema de evolución de FSMs se eligieron algunos de los más conocidos para utilizarlos en los casos de prueba. El funcionamiento general de los mismos se describe a continuación.

4.4.2.1 Selección

El proceso de selección en un algoritmo genético emula el proceso de selección natural mediante el cual los individuos más aptos se reproducen y pasan a la siguiente generación, incrementando la probabilidad de tener mejores individuos en el futuro. En este caso, los mejores individuos serán los comportamientos que al ser se aproximen más a cumplir la tarea asignada. El método de selección utilizado en este trabajo es el método determinístico desarrollado por Ángel Kuri [11] llamado Vasconcelos. Los métodos de selección determinísticos no toman en cuenta la calificación o *fitness* de los individuos a diferencia de aquellos de selección proporcional donde las probabilidades de selección de cada individuo dependen de dicho valor.

4.4.2.2 Método Vasconcelos

Este método de selección determinística, al igual que otros del mismo tipo, buscan proveer de mayor diversidad a la población cruzando parejas de individuos en posiciones predefinidas. En este modelo en particular, se favorece la cruza entre los mejores y los peores individuos con el objetivo de explorar un espectro más amplio del espacio de búsqueda. El método consiste en cruzar, en cada generación, al individuo *i* con el individuo *n-i+1* en una lista de los *n* individuos ordenados con respecto a su aptitud (*fitness*) de mayor a menor. Podrá parecer que al hacer esto se destruyen características deseables de los individuos, pero no es así, ya que si se combina este método con un elitismo total, se logra explorar una variedad más amplia de características.

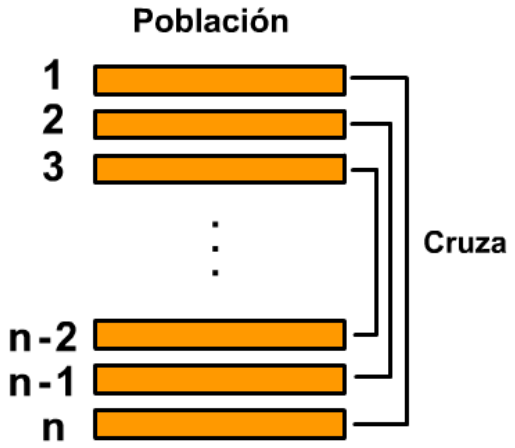


Figura 4.9: Método de selección determinístico Vasconcelos.

4.4.2.3 Elitismo

El elitismo es un mecanismo que consiste en mantener intactos los *e* mejores individuos en cada generación de tamaño *n*, para que pasen a la siguiente, esto puede mejorar el funcionamiento del algoritmo ya que evita que en generaciones posteriores se pierdan las mejores soluciones encontradas durante todo el proceso. El elitismo puede generalizarse y pasar a la generación siguiente a los *n* mejores individuos de las *k* generaciones anteriores.

Para $e < n$ se habla de un elitismo parcial, sin embargo si $e = n$ se dice que el elitismo es total ya que en cada generación se tiene registro de los mejores *n* individuos de los *kn* que han aparecido durante todo el proceso del algoritmo genético. En el algoritmo presentado se utiliza el elitismo total que como se mencionó anteriormente complementa al método de selección determinística Vasconcelos.

4.4.2.4 Cruzamiento

Las operaciones de cruzamiento emulan el proceso de reproducción sexual en la naturaleza durante el cual dos individuos con cargas genéticas diferentes se mezclan para producir otros cuyo código genético es una combinación del de sus padres, es híbrido. Este proceso hace posible que se conserven características de un individuo a otro o mezcladas de ambos padres para producir nuevas. La descendencia de individuos bien calificados tiene alta probabilidad de serlo también.

En la selección determinística se seleccionan dos individuos para cruzarse y posteriormente se ordenan todos los hijos y se mezclan con los padres, ordenados por aptitud para obtener a los *n* mejores que pasarán a la siguiente generación. Una vez seleccionados los individuos, se cruzan, utilizando operadores binarios de cruzamiento que operan con individuos (padres) y producen también dos individuos (hijos) con características heredadas de sus padres. En este trabajo se utiliza el cruzamiento uniforme.

4.4.2.5 Cruzamiento uniforme

En el cruzamiento uniforme, cada uno de los genes que forman las cadenas hijas tiene la misma probabilidad de provenir de uno u otro padre. El procedimiento es sencillo, se genera una cadena de ceros y unos de la misma longitud que los individuos, cada posición tiene 0.5 de probabilidad de ser cero o uno. Esta cadena sirve como una máscara, de forma que para formar el primer hijo, se recorre la máscara, si el gen con índice *i* en la máscara es un 1, el gen *i* del hijo se copiará del primer padre, y si es un 0, se copiará del segundo. Para el segundo hijo se sigue el mismo proceso, con la misma máscara, pero con los padres intercambiados.

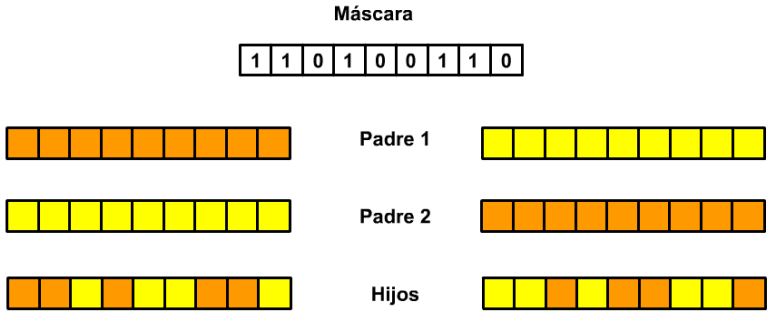


Figura 4.10: Cruzamiento uniforme.

4.4.2.6 Mutación

La mutación es un proceso mediante el cual se dota de variedad a la población permitiendo al algoritmo explorar regiones en el espacio de soluciones que probablemente no se han explorado aún, son modificaciones del código genético de cada individuo que se hacen a propósito siguiendo diversas reglas, dependiendo de la aplicación y representación de las soluciones. En este trabajo se toma en cuenta una mutación que define un criterio de eliminación y creación de transiciones, además de cambiar sus condiciones aleatoriamente.

El método consiste en recorrer cada uno de los genes del individuo y con cierta probabilidad modificarlos. El valor de cada gen en la representación de longitud fija representa una condición y su posición una transición, si su valor es cero, implica que la transición no existe. Entonces en primer lugar, con probabilidad p_e se quita o pone la transición en cuestión según sea el caso (si es diferente de cero se hace cero). Si la transición no se eliminó, con probabilidad p_m se cambia su condición por otra aleatoria.

Esta operación permite crear nuevas transiciones entre estados que tal vez no tenían así como asignar condiciones diferentes entre ellos. Es importante notar que se utilizan dos probabilidades para realizar la mutación, p_e que en general es baja y determina que tanto se agregan o quitan transiciones a la máquina y p_m que, para transiciones presentes determina el cambio de condición asociada a ellas.

El proceso general de mutación es el siguiente:

- Para cada gen *i* en el individuo *a*
  - Genera número aleatorio *r* entre 0 y 1
  - Si *r* <= p_e
    - Si *a[i] == 0*
      - *a[i]* = número aleatorio entre 1 y N_m − 1 (cantidad de condiciones posibles).
    - Si no
      - *a[i] = 0*
  - Si no
    - Si *a[i] != 0*
      - Genera número aleatorio *q* entre 0 y 1
      - Si *q* <= p_m
        - *a[i]* = número aleatorio entre 1 y N_m − *1 (cantidad de condiciones posibles).*

4.4.3 Implementación

El algoritmo genético se implementa en la clase `GGA`, que es una *clase template* al igual que `Individual` y describe un algoritmo genético genérico, es decir, implementa la estructura básica general de este tipo de algoritmos. Esta clase tiene los métodos necesarios para ejecutarlo y declara métodos virtuales para aquellas tareas que deben escribirse específicamente para cada aplicación particular como los operadores genéticos o la creación de individuos válidos. Así, cada algoritmo particular se define en una clase que hereda de `GGA`.

Para esta aplicación, es necesario que el *fitness* de cada individuo se calcule fuera del algoritmo genético ya que es el simulador rápido el que realiza esta tarea. Para este fin se implementa la función `setFitness(int index, float fitness)`

que sirve de interfaz entre el simulador y el algoritmo genético, así mismo, la función `getIndividual(int index)` permite obtener una referencia al individuo con índice *i* de la generación actual. El proceso de calificación empieza con una llamada a `getIndividual` del genético, el individuo *i* se pasa al simulador para calcular su fitness y por último se llama a `setFitness` con el índice del individuo y su calificación dada por el simulador.

También la clase `GGA` implementa la función `createInitialPop()` que tiene como objetivo generar la primera población con individuos creados aleatoriamente, sin embargo, para cada tipo de individuo esta tarea será diferente por lo que se declara un método virtual que implemente el procedimiento específico de generar un individuo aleatoriamente, este método se llama `newRandomIndividual()` y al ser virtual debe implementarse en las clases hijas dependiendo del problema.

Otra consideración importante es que la ejecución del algoritmo genético debe poder distribuirse en varios frames ya que si se ejecutan todas las generaciones en uno solo puede consumir mucho tiempo dicho *frame* y causar un efecto desagradable en la animación y respuesta del sistema. Para lograr esto, la clase `GGA` implementa la función `nextGeneration` que procesa únicamente una generación, permitiendo así que se procese una generación cada *x* frames o cada que se necesite. Por ejemplo, si el algoritmo necesita ejecutarse para 600 generaciones y la animación está corriendo a 60 frames por segundo, se pueden ejecutar 10 generaciones por frame, lo que permite distribuir en 1 segundo todo el trabajo del genético y no saturar 1 solo frame.

El algoritmo se ejecuta para todas sus generaciones y no existe algún otro criterio de parada debido a que, en principio, no se conoce el comportamiento óptimo para tomarlo como referencia. Sin embargo dependiendo de la aplicación, los criterios de parada pueden ser diversos, por ejemplo, se puede parar cuando las soluciones ya no mejoren o cuando el fitness de la mejor solución encontrada alcance un umbral predefinido. En general, un comportamiento aceptable para los agentes será aquel que cumpla la tarea asignada, sin embargo unos pueden ser mejores que otros con relación a la velocidad en que lo hace o el consumo de energía. Debido a esto se debe diseñar cuidadosamente la función de fitness para que tome en cuenta todos los aspectos de interés relativos al comportamiento de los agentes.

El simulador rápido se ejecuta en la función `computeRobotFitness` que recibe el robot y el comportamiento a simular. Una vez que se han calificado todos los comportamientos (individuos) se llama al método `nextGeneration` que realiza los procesos necesarios para obtener la siguiente generación de individuos, una vez cumplidas las condiciones de paro evaluadas en el método `mGA.done()` termina su ejecución.

5. Construcción del ambiente virtual

5.1 Descripción General

Para desarrollar e implementar satisfactoriamente la evolución de los comportamientos de un agente en ambientes virtuales es necesario construir el entorno virtual sobre el cual ésta se ejecute, lo que implica un desarrollo importante en diversas áreas además de la IA. Se requiere desarrollar un sistema gráfico para la representación de objetos tridimensionales, manejo de eventos de entrada/salida, sistema de interfaz *gráfica de usuario (GUI)* entre otras cosas. En este capítulo se describe el proceso de construcción del entorno virtual utilizado como plataforma de experimentación en el presente trabajo. También se muestran algunas consideraciones importantes a la hora de implementar IA en ambientes virtuales como las capacidades del hardware disponible ya que muchos algoritmos demandan mucho tiempo de procesamiento como es el caso de los AG.

El sistema se realizó en su mayoría utilizando el lenguaje de programación C++ debido a que una gran cantidad de bibliotecas gráficas (OpenGL, DirectX, el mismo OGRE3D etc.) se encuentran escritas en este lenguaje así como la mayoría de videojuegos comerciales, motores de física, bibliotecas de audio; prácticamente toda la base de código existente en el campo del desarrollo de *videojuegos AAA* utiliza C++. Además es el lenguaje en el que el desarrollador tiene más experiencia.

Es necesario que el sistema planteado tenga las siguientes características mínimas para realizar los experimentos propuestos y evaluar resultados de forma ágil y eficiente.

- La posibilidad de crear un entorno de visualización tridimensional en tiempo real sobre el cual ejecutar algoritmos de IA. Debe ejecutarse a una velocidad media de 30 *frames por segundo* (FPS) para ser capaz de retroalimentar al usuario a una velocidad suficientemente alta para percibir cambios en el entorno instantáneamente.

- Manejar estados del mundo y de la aplicación en general (estado de carga, menú, estado de ejecución principal y de finalizar). Cada estado debe tener características independientes y se debe poder ir de uno a otro fácilmente.
- Dar la posibilidad de ejecutar el entorno a diferentes velocidades e incluso pausar el sistema para observar detalladamente el estado actual de los agentes y el entorno.
- Debe contar con la capacidad de presentar información gráfica y de texto sobre los elementos del entorno, por ejemplo, indicadores de vida de los agentes o mostrar información sobre su desempeño.
- Permitir al usuario interactuar con el entorno y sus elementos mediante comandos o acciones a través de dispositivos periféricos.
- Incorporar un sistema de manejo de cámara que permita tener una vista global del entorno o hacer acercamientos a elementos individuales para observarlos a detalle.
- Capacidad de manejar información lógica ligada a los elementos geométricos del entorno (representación del grafo de navegación, posiciones iniciales de los agentes, variables propias de cada agente y sus comportamientos). Comúnmente conocidos como *metadatos*.
- Para ejecutar el AG de manera eficiente será necesario poder ejecutarlo por partes o distribuido en varios *frames.*

Tomando en cuenta las necesidades mencionadas anteriormente se eligieron diversas herramientas de software que proveen de funcionalidades específicas muy útiles para el desarrollo del sistema.

5.2 Herramientas de desarrollo

5.2.1 OGRE 3D (Object Oriented Rendering Engine)

OGRE [17] es un *motor de software* enfocado a la programación de sistemas gráficos en tiempo real, gratuito y de código abierto escrito en C++ que incorpora un conjunto de características que facilitan la construcción de aplicaciones gráficas tridimensionales. Aunque es ampliamente utilizado para desarrollar videojuegos (junto con otras bibliotecas de software), no está hecho específicamente para este fin, en

cambio, los desarrolladores han enfocado este proyecto para aplicaciones gráficas 3D en general como simulaciones, visualización científica, de negocios o videojuegos. Al estar construido con C++, puede ser compilado y utilizado en la gran mayoría de sistemas MS Windows, Linux y Mac OSX. La versión de OGRE utilizada en este trabajo es la 1.6.5 que se liberó el 27 de diciembre de 2009 ya que el desarrollo del sistema presentado se inicio a principios de 2010.

OGRE únicamente provee soluciones para la parte gráfica como carga de modelos, manejo de la escena, cámaras, entre otras, por lo que si se tiene la necesidad de audio, conectividad en red o inteligencia artificial se deben implementar o utilizar otras bibliotecas para estos fines. Su diseño basado en componentes provee de una interfaz amigable para la integración de dichas bibliotecas de forma fácil y robusta. La comunidad de desarrolladores de Ogre es muy activa, el motor cuenta con soporte de un equipo dedicado y en el sitio *http://www.ogre3d.org* se puede encontrar toda la documentación necesaria para trabajar con Ogre.

Algunas de sus características principales:

- Provee de una interfaz amigable y fácil de utilizar independiente del *API* de programación gráfica utilizada (OpenGL o Direct3D).
- Resuelve requerimientos básicos como manejo del grafo de escena, carga de modelos y animación.
- Soporta Direct3D y OpenGL.
- Incorpora efectos especiales como sistemas de partículas, efectos de post producción, billboards y manejo de transparencias de forma fácil y rápida.
- Provee un sistema de manejo de recursos con carga y descarga de archivos.
- Soporta archivos zip y pk3.

Para empezar a trabajar con OGRE solo basta descargar el SDK (*Software Development Kit*) que es un paquete con todo lo necesario pre-compilado para varios entornos de desarrollo como *Microsoft Visual Studio, Code Blocks*, *Ubuntu*, entre otros. También está disponible el código fuente en caso de que el desarrollador busque utilizar OGRE en otros entornos de desarrollo, esta opción requiere compilar y construir las bibliotecas de OGRE con la ventaja de poder hacer modificaciones directas al código para corregir errores o incorporar nuevas funcionalidades propias al motor. Junto con el

SDK vienen varios ejemplos que muestran funcionalidades particulares del motor de forma clara.

5.2.1.1 Arquitectura de Ogre

Todas las clases y tipos de OGRE se encuentran organizados bajo un espacio de nombres (*namespace*) llamado *Ogre,* por lo que para acceder a cualquiera de sus elementos se debe utilizar el prefijo `Ogre::`, por ejemplo la clase `Ogre::Camera` permite el acceso a la clase que incorpora funciones de manejo de cámaras. El uso de espacios de nombre es una forma fácil y eficiente de evitar colisiones o nombres duplicados de tipos ya que muchas bibliotecas pueden tener clases que se llamen igual, por ejemplo, casi cualquier motor de gráficos o de física tiene una clase para definir vectores en dos o tres dimensiones generalmente llamada `Vector2` o `Vector3`, `Ogre::Vector3` se refiere a la clase para manejo de vectores en 3 dimensiones de OGRE.

En el diagrama de la *Figura 5.1* se puede ver que en la parte más alta se encuentra el objeto `Root`, este objeto es el punto de entrada al sistema de OGRE y el encargado de crear los componentes principales del sistema como el manejador de escena (`SceneManager`) o la ventana de dibujo (`RenderWindow`). En OGRE es posible incorporar *plugins* con el fin de extender sus funcionalidades ya que muchas de las clases que componen el sistema permiten "conectar" nuevas funcionalidades al motor, gracias a esto se puede realizar una implementación propia para carga de recursos o modificar los algoritmos de manejo de escena, en general casi cualquier funcionalidad se puede ampliar. Las clases de OGRE se clasifican en 3 roles principales [23]:

- Las clases responsables del *Manejo de Escena* se encargan de todo lo relacionado al contenido y estructura de la misma. Llevan el control del grafo de escena así como de los objetos que la componen y sus características como geometría, texturizado, iluminación o posición.
- La parte de Manejo de Recursos se encarga de la carga, descarga y administración de archivos diversos que pueden ser modelos, texturas, sonido o fuentes útiles para la generación de contenido.

- Las clases en Rendering son las encargadas de todo el proceso de conversión de la representación de una escena tridimensional en una imagen 2D para presentarla en la ventana de dibujo. Estos procesos tienen que ver directamente con el *Rendering pipeline*.

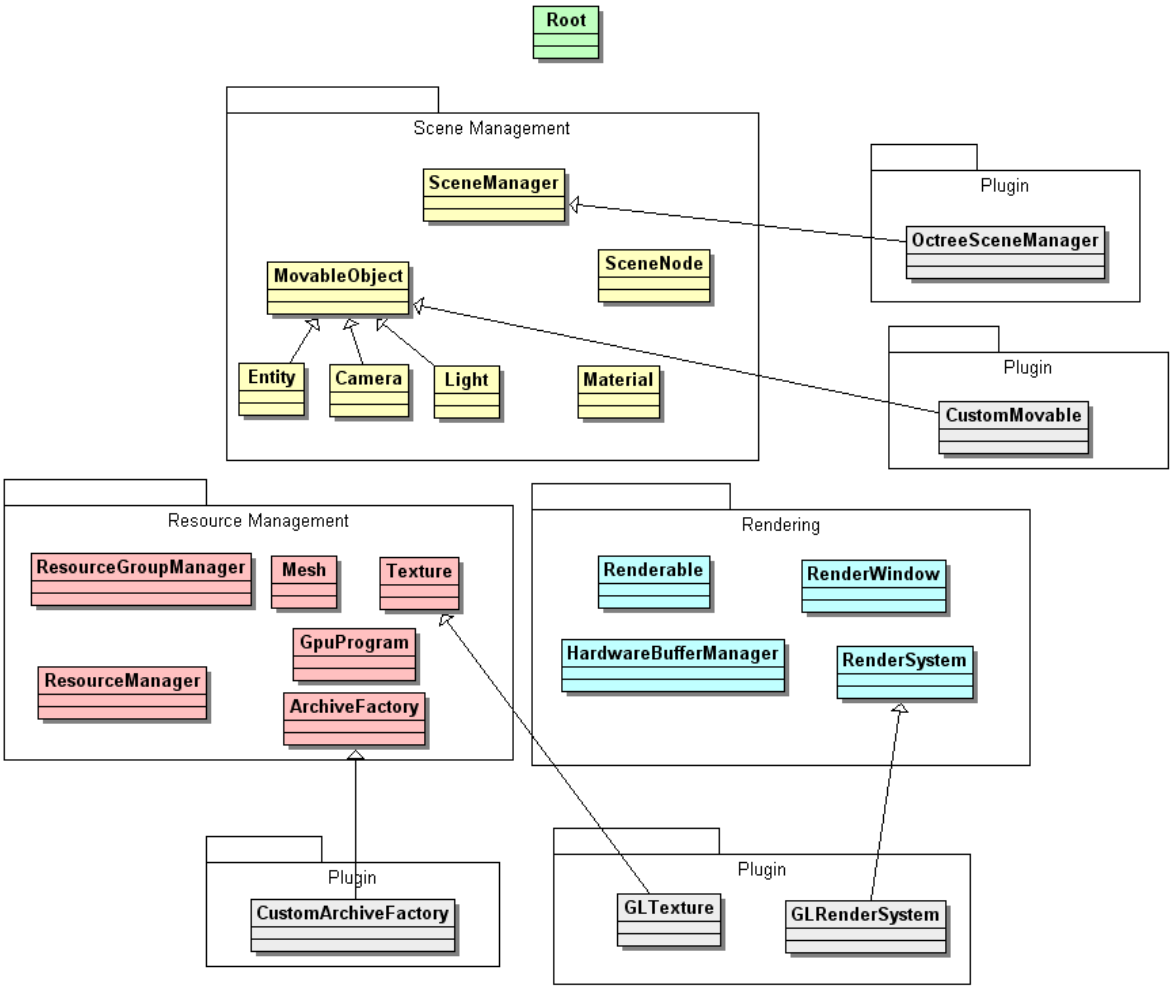


Figura 5.1. Diagrama de la arquitectura general de Ogre. Se muestran solo algunas de las clases principales como ejemplo. Tomado de http://www.ogre3d.org/

5.2.2 Otras Herramientas

5.2.2.1 Microsoft Visual C++ 2008 Express Edition

Es un entorno de desarrollo integrado (IDE por sus siglas en inglés) para programar en C++ creado por Microsoft y distribuido gratuitamente como versión disminuida de la versión de paga Microsoft Visual Studio. Es una versión ligera y puede ser utilizada para compilar aplicaciones .NET y aplicaciones que utilicen la *Win32 API* [27]. A pesar de ser una versión disminuida cuenta con todas las herramientas necesarias para programar en C++, de hecho se puede encontrar en la página oficial de OGRE el SDK pre-compilado para varias versiones de Visual Studio que funcionan perfectamente con la *Express*.

Algunas limitaciones con las que cuenta la versión *Express* son las siguientes:

- No cuenta con un editor de recursos.
- No cuenta con soporte integrado para las Microsoft Foundation Classes.
- No soporta aplicaciones de 64 bits de forma nativa.
- No soporta macros.

Esta herramienta se utilizó como entorno integrado de desarrollo en la construcción del sistema gráfico y los simuladores.

5.2.2.2 CEGUI (Crazy Eddie´s Graphic User Interface)

CEGUI [6] es una biblioteca de software libre orientada a objetos y escrita en C++ que provee funcionalidad para crear una interfaz gráfica de usuario (GUI) con ventanas y controles en *APIs* gráficas que no cuentan con esta funcionalidad nativa. CEGUI está dirigida a los desarrolladores de videojuegos que necesitan un sistema flexible, fácil de utilizar, gratuito y que ahorre tiempo de desarrollo. En versiones de OGRE anteriores a la 1.7 CEGUI venía incluida como dependencia, siendo el sistema de GUI utilizado por defecto. Para la versión 1.7 y posteriores los desarrolladores de OGRE utilizan su propia implementación para interfaz de usuario básica.

CEGUI cuenta con diversos controles como ventanas, botones, sliders, áreas de texto, barras de desplazamiento, entre otros. Estos controles son totalmente configurables vía archivos XML. Se puede modificar desde la apariencia de cada uno de los controles hasta su funcionamiento, incluso se pueden crear controles personalizados de forma muy fácil extendiendo las clases que provee esta biblioteca. Existen también editores que permiten crear una interfaz gráfica de forma fácil como el `CELayoutEditor` [5] que se distribuye como el editor oficial de CEGUI. Este editor permite exportar a XML para posteriormente cargarla dentro de la aplicación vía código.

5.2.2.3 TinyXML

Es una herramienta de código libre que permite cargar, analizar y guardar documentos en *formato XML* de forma muy simple y eficiente. TinyXML [24] está escrita en C++, es independiente del sistema operativo y puede incluirse fácilmente cualquier proyecto que utilice este lenguaje. Esta herramienta se utilizó para leer configuraciones y propiedades de la escena así como las propiedades del piso, mapa, número de agentes, entre otros datos desde archivos XML.

5.3 Estructura del Sistema

El entorno virtual está construido utilizando el *patrón de diseño singleton* bajo una estructura de managers. Este patrón de diseño se utiliza generalmente para clases que realizan tareas de administración centralizada para las cuales es necesario restringir la creación de objetos o instancias. En el presente sistema se hace uso extensivo de *singletons* ya que Ogre los utiliza también y además provee una clase base para implementarlos de forma muy sencilla.

El sistema implementa una serie de managers utilizando éste patrón de diseño, es una estructura tomada de Ogre y que también es ampliamente utilizada por su comunidad de desarrolladores. Cada manager es un objeto único de acceso global que se encarga de la administración de un aspecto específico del sistema y son implementados como clase *singleton*.

Sobre el motor gráfico se construyeron una serie de módulos especializados para hacer posible la simulación, por ejemplo, el sistema de representación y dibujo de agentes descrito en el capítulo 3. La mayoría de éstos utiliza funciones o características provistas por Ogre no sólo para graficar elementos en la pantalla sino también para realizar el manejo de escena, de eventos, carga de recursos o dibujo del mapa.

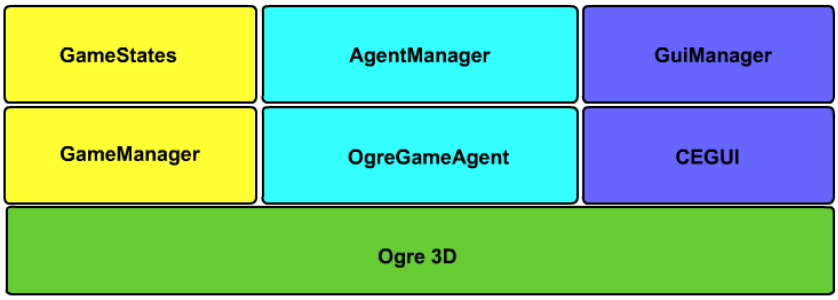


Figura 5.2: Ejemplos de managers construidos sobre Ogre.

5.3.1 Game Manager

El `GameManager` es el módulo encargado de administrar los estados de la aplicación, controla su inicio, su ejecución y su final así como las transiciones entre ellos. También es el encargado de "escuchar" por eventos del teclado y mouse a través de OIS, una biblioteca orientada a objetos con funciones específicas para esta tarea. También lleva el control general de la aplicación durante todo su tiempo de vida y es el encargado de realizar las tareas globales de inicialización y finalización de todos los módulos utilizados en el sistema.

Este módulo realiza tareas básicas como la inicialización y finalización de Ogre, la carga y descarga de recursos globales utilizados por la aplicación como modelos, texturas, sonidos, entre otros. También enmascara los métodos de manejo de eventos de teclado y mouse para pasarlos al estado que debe procesarlos.

En todo momento, durante la ejecución de la aplicación, existe sólo un estado activo en proceso, que se ejecuta hasta que exista una condición de salida o de cambio de estado que haga que el `GameManager` salga de la aplicación o active otro estado finalizando el anterior.

El funcionamiento del `GameManager` no tiene sentido sin los módulos llamados **game states**, que representan a los estados de la aplicación, ya que se encarga de administrarlos, por lo que al menos debe haber uno para poder inicializar el ambiente gráfico. La cantidad de estados finales y su funcionalidad depende del diseño general de la aplicación, en este caso se manejan tres.

5.3.2 Game States

La clase `GameState` define una porción del sistema, un estado diferenciable en el que éste se puede encontrar. A nivel de diseño, cada estado puede verse como un programa aparte con sus propios procesos implementados como clases hijas de GameState y que se comunican por medio del `GameManager`.

Dentro de la aplicación pueden existir muchos estados, sin embargo solo uno se ejecuta a la vez, esta ejecución implica que el `GameManager`, en su bucle principal ejecutará el bucle de dicho estado. De igual forma, todos los eventos de teclado o mouse que reciba el `GameManager` serán pasados al estado activo para que los procese.

Cada estado tiene un bucle principal que se ejecutará mientras éste se encuentre activo, en él se ejecutan todos los procesos que componen al cuerpo del estado. Así mismo tiene dos funciones, una de inicialización y otra finalización que se ejecutan al entrar y salir del estado respectivamente, ya sea al inicio o final de la aplicación o bien, al cambiar entre estados.

El proceso de cambio de estados lo realiza el `GameManager`, sin embargo, éste no "sabe" de antemano la secuencia de estados o a que estado hay que cambiar, si no cada estado, al manejar sus propios eventos decide cual es el siguiente y solicita al `GameManager` el cambio pasando como argumento dicho estado destino. Esto es posible ya que cada `GameState` tiene acceso a una instancia única y global de `GameManager`. Así, los estados pueden solicitarle realizar diversas acciones además del cambio de estado como pausar la simulación, reanudarla o terminar el programa.

La estructura del sistema propuesto contempla el uso de tres estados, los cuales se describen a continuación:

- `IntroState`: Es el estado de inicialización encargado de levantar el entorno gráfico y cargar los recursos de los grupos shared e intro. Su ejecución muestra en pantalla una imagen de fondo y la desvanece después de algunos segundos, terminando esto, manda el mensaje al `GameManager` de cambio de estado a `MenuState`.

- `MenuState`: En este estado se muestra el menú de opciones a las que puede acceder el usuario, básicamente un botón que permite iniciar la simulación cambiando al estado `PlayingState` y otro que permite salir de la aplicación.

- `PlayingState`: Es donde se realiza la simulación principal de los agentes con base en la configuración cargada desde un archivo xml.

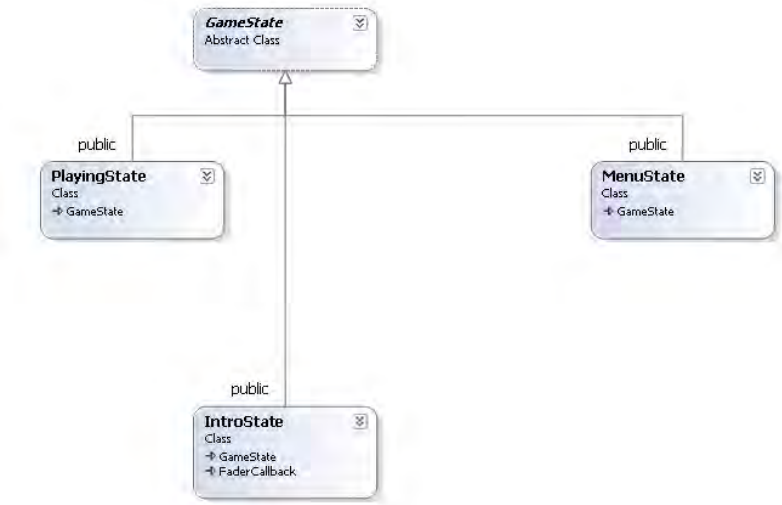


Figura 5.3: Estados de la aplicación.

5.3.3 Manejo de Recursos

Ogre 3D provee un sistema de manejo de recursos que permite cargar y descargar archivos externos (modelos, texturas, fuentes, sonidos etc.) bajo demanda e incluso organizarlos por grupos. Cada `GameState`, en su fase de inicialización carga los recursos que utilizará durante su ejecución para posteriormente descargarlos al finalizar. Esto permite que, por ejemplo, que se libere la memoria utilizada por texturas, modelos o sonidos utilizados en un estado de la aplicación y de esta forma optimizar el uso de memoria, es decir, tener cargado únicamente lo que se está utilizando.

Existen también recursos globales, que están presentes durante toda la ejecución de la aplicación como la imagen del cursor del mouse o la fuente para escribir el texto en pantalla. Estos se cargan al iniciar el `IntroState` pero no se descargan hasta salir de la aplicación. Esto permite evitar el proceso de carga y descarga de los mismos elementos en cada estado.

En el sistema propuesto se manejan básicamente tres grupos de recursos:

- **Intro**: Este grupo de recursos lo componen aquellos elementos presentes en los estados de introducción y el menú. Son texturas de fondo en su mayoría.

- **Main**: Este grupo contiene los elementos necesarios para el funcionamiento del `PlayingState`, es decir, se cargan a su inicio y se descargan cuando éste termina. Se compone principalmente de los modelos y texturas de los agentes y el entorno.

- **Shared**: Este grupo está formado por aquellos recursos que están presentes en todos los estados, está compuesto por elementos como texturas de la interfaz gráfica y archivos de fuentes de texto.

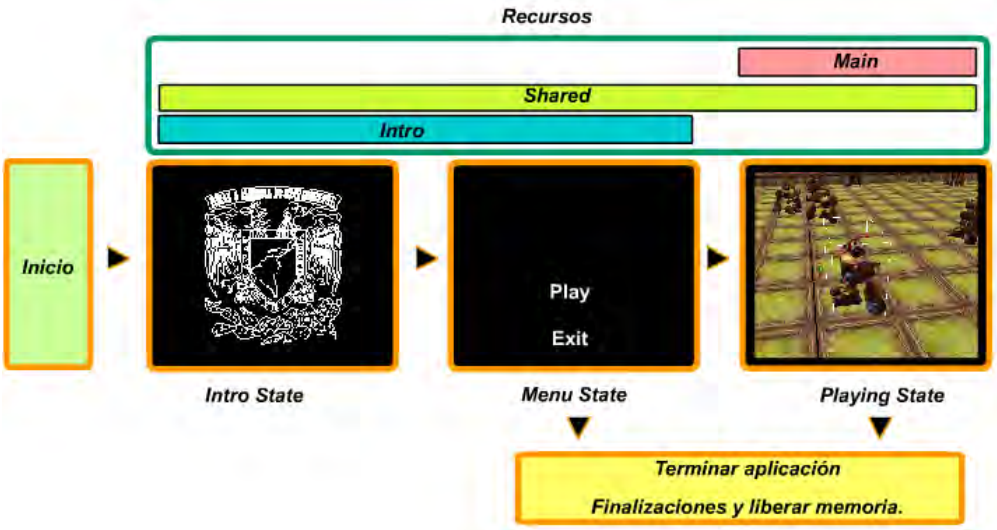


Figura 5.4: Diagrama general de estados y recursos en la aplicación.

5.3.4 Dibujo de los agentes

Los agentes en el ambiente virtual están definidos por objetos de clases hijas de `OgreGameAgent` e incorporan funcionalidad para cada una de las tareas que necesitan, incluyendo su representación gráfica en el entorno. Ésta representación se logra asociando a cada uno de ellos un modelo tridimensional en *formato .mesh*, que es el formato nativo de los archivos de geometría en Ogre.

La clase `Ogre::Entity` permite utilizar dicho modelo tridimensional y asociarlo a una entidad en la escena para posteriormente unirlo a un nodo de escena y dotarlo de posición, tamaño y orientación, entre otras propiedades. Entonces, cada agente con representación gráfica debe tener una liga a una entidad que lo representa y otra a un nodo de escena que define sus propiedades en el espacio, los cambios de posición, orientación o tamaño del modelo se realizan a través de dicho nodo de escena.

Figura 5.5: Un modelo utilizado para representar a los agentes, tomado del videojuego World of Warcraft (http://us.battle.net/wow) con ayuda del programa WoW Model Viewer.

Cada modelo tiene una serie de animaciones predefinidas que sirven para dar realismo a la escena y que se pueden activar mediante un estado de animación que provee la clase `Ogre::Entity`. Así mismo es posible aumentar o disminuir la velocidad de las mismas o bien, detenerlas dependiendo de las acciones del agente. El modelo utilizado para los experimentos en este trabajo se muestra en la figura 5.5.

5.3.5 Movimiento

El movimiento es la principal acción que realizarán los agentes estudiados, esto implica trasladarse de un punto a otro en el entorno. Se ha elegido una trayectoria recta con velocidad constante para trasladarse de un punto a otro debido a su facilidad de implementación. Esto se logra calculando el vector de movimiento y variando la posición del agente en relación con él, antes de dar cada paso se evalúa si el paso siguiente se encuentra más allá del punto final, con el fin de evaluar la llegada a su destino. Una interpolación lineal con intervalos constantes causará problemas en la simulación ya que el intervalo de tiempo entre *frames* es variable.

Es importante tener en cuenta que la velocidad de movimiento debe ser independiente de la velocidad de simulación, es decir, si se busca que el agente avance 100 unidades en un segundo, debe hacerlo tanto a 50 cuadros por segundo (dos unidades por cuadro) como a 10 *cuadros* por segundo (diez unidades por cuadro).

5.3.5.1 Planeación de rutas

Los agentes presentados tienen la capacidad de calcular rutas óptimas como acción básica, es decir, encontrar el camino más corto entre dos puntos y posteriormente caminarlo hasta llegar a su destino. Así el agente puede esquivar obstáculos fijos y trasladarse de un punto a otro del mapa. El algoritmo utilizado para encontrar caminos mínimos es *A* (A estrella)* y se implementa una clase llamada `AStar`. Utiliza la representación del mundo basada en celdas, sobre el grafo cuyos nodos son los centros de cada una de las celdas pasables. Así, este algoritmo encuentra el camino mínimo entre dos puntos siempre que exista y lo guarda en una pila para posteriormente retirar cada uno de los nodos que el agente debe recorrer.

5.3.6 Dibujo del entorno

El dibujo del mapa se realiza también a partir de la representación basada en celdas, simplemente se recorre la matriz de enteros y para cada elemento de la matriz se dibuja un cubo cuya altura es proporcional a la penalización de movimiento de la celda. Para hacer esto, se debe determinar un factor de escala que define el ancho y largo del mapa así como un intervalo de altura que defina el límite inferior para una celda pasable (1%) y el superior para una celda no pasable (100%) que se verá como una pared, los valores de penalización intermedios se obtienen haciendo una interpolación lineal.

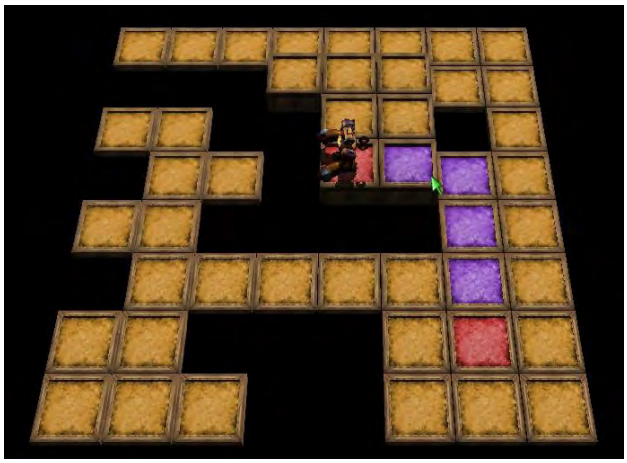


Figura 5.6: Representación gráfica de un mapa, las celdas no pasables no se dibujan.

Además del factor de escala para cada celda, se le suma un vector de desplazamiento, de esta forma se puede dibujar el mapa en cualquier parte del entorno virtual. Este vector define la posición $(x0, y0, z0)$ de la celda $0,0$ y la posición de las demás deldas serán relativas a ésta, así la celda i, j estará dibujada en la posición $(x0 + i * s, y0, z0 + j * s)$, donde s es el factor de escala del mapa, véase figura 5.7.

Figura 5.7: Mapa con celdas con diferentes niveles de penalización.

La matriz de enteros en la que se encuentra guardada la representación del mundo puede cambiar sus valores, en el transcurso de la simulación, el usuario puede decidir modificar el mapa, por esto, no bastará con dibujarlo una sola vez al inicio de la simulación, se deberá volver a dibujar periódicamente o cuando haya sufrido cambios.

5.3.7 Acciones externas

Durante el desarrollo de la simulación de los agentes en el ambiente virtual, el usuario o jugador podrá interactuar con él para mejorar el proceso de experimentación y acelerar las pruebas del sistema. Algunas de ellas sobre los agentes directamente, otras sobre el entorno o sobre la simulación. Estas acciones se realizan a través de los dispositivos periféricos de entrada básicos: el mouse y el teclado.

A continuación se numeran y describen las acciones disponibles durante la simulación, es decir, durante la ejecución del `PlayingState` de la aplicación. Estas acciones son accesibles a través de un menú gráfico en las partes izquierda e inferior de la pantalla.

- Acciones sobre los agentes

  - *Seleccionar un agente*: El usuario puede dar clic izquierdo sobre un agente para seleccionarlo. La selección implica que la siguiente acción se realizará sobre dicho agente.
  - *Mover un Agente*: Permite mover, desde la posición actual, hasta otra posición válida del mapa al agente seleccionado.
  - *Inspeccionar las propiedades de un agente*: Muestra información detallada de las propiedades y variables internas del agente seleccionado y su comportamiento. Estos valores son de interés para refinar los comportamientos y reportar resultados.
  - *Reiniciar un agente*: Reinicia el comportamiento de un agente en una posición dada. Mueve el agente si es necesario y hace que la máquina de estados de su comportamiento pase al estado 0 (idle) y reinicia todas las variables necesarias.

- Acciones sobre el entorno

  - *Mover un Objeto*: Permite mover, desde la posición actual, hasta otra posición válida del mapa un objeto seleccionado.

- Acciones sobre la cámara

  - *Elegir entre dos modos diferentes de cámara:* La simulación permite elegir entre dos modos de cámara diferentes que se pueden cambiar en cualquier momento.
  - *Cámara libre:* Permite al usuario mover la cámara libremente por el entorno utilizando las flechas del teclado y el mouse para tener una visión global o desde la perspectiva que éste desee.
  - *Que la cámara siga a un objeto:* La cámara siempre mira al objeto o agente seleccionado permitiendo acercar y alejar la vista (zoom).

- Acciones sobre la simulación

  - Reiniciar la simulación: Reiniciar con los parámetros iniciales a todos los agentes en posiciones aleatorias. Si hubo cambios en el entorno, permanecen.
  - Modificar la velocidad de simulación: Para poder ver el desarrollo de los comportamientos con detalle o acelerarlos para que los agentes lleguen a un punto deseado. Se puede modificar el tiempo entre cuadros desde 1 cuadro por segundo hasta el máximo permitido por el hardware.
  - Salir de la simulación: Permite finalizar la simulación y toda la aplicación desde el `PlayingState`.

5.3.8 Proceso general de simulación

La simulación en el ambiente virtual empieza cargando sus propiedades desde un archivo de configuración *.xml*, este archivo contiene los siguientes datos generales:

1. Nombre del archivo que contiene el mapa (el mismo que utiliza el simulador lento).
2. Cantidad de agentes a simular.
3. El comportamiento que debe asignar a cada agente (provenientes del algoritmo genético).
4. Velocidad de movimiento de cada agente (mayor velocidad implica mayor consumo de energía).
5. Nivel de batería/energía inicial de cada agente (ésta se consume cada que el agente realiza una acción).
6. Datos sobre objetos auxiliares del entorno (por ejemplo nivel de carga, posiciones y número de extintores en el caso de prueba).

Después de cargar los datos, se crean los agentes con los valores deseados y se sitúan en posiciones aleatorias pero válidas del mapa para empezar su simulación. Ésta empieza cuando el usuario presiona la tecla 'enter' y continúa hasta que decida interrumpirla o bien, las condiciones del mapa hagan imposible que los agentes cumplan sus metas.

6. Pruebas y resultados

6.1 Descripción del experimento

En el presente capítulo se muestran las pruebas realizadas al sistema así como los resultados obtenidos. Para esto es necesario describir la estructura del agente cuyo comportamiento se evoluciona, el entorno en el que se desarrolla y los objetivos que debe cumplir el comportamiento. El proceso de prueba consiste en diseñar e implementar un agente con un conjunto finito de acciones y percepciones que definen su estructura y las tareas que puede realizar. Se deben definir también las propiedades y estructura del entorno, por ejemplo, sus áreas pasables y no pasables u objetos con los que el agente puede interactuar.

Una vez hecho lo anterior, se plantea un problema, el cual debe ser resuelto por el agente ejecutando un comportamiento válido (basado en sus acciones y percepciones), esto implica diseñar e implementar la función de aptitud (*fitness*) del algoritmo genético, pues éste es el encargado de realizar la búsqueda del mejor comportamiento en el espacio de soluciones definido por la cantidad de acciones y percepciones del agente. Este espacio de búsqueda crece exponencialmente (véase capítulo 3), lo que va complicando la búsqueda al añadir metas que requieran de nuevas acciones agregadas al agente.

La función de aptitud es calculada por el simulador, de tal forma que para calificar cada comportamiento (individuo), éste debe ser asignado al agente de prueba, posteriormente se simula su desarrollo en el entorno por un número de ciclos dado y al final de la simulación se calcula su calificación con base en un conjunto de variables que definen el estado final de la ejecución del comportamiento.

En este problema de búsqueda, no se sabe de antemano el comportamiento óptimo, de hecho puede no ser único ya que pueden existir diferentes comportamientos que realicen la tarea de la misma forma, en el mismo tiempo o cualquiera de las variables medidas por la función de aptitud. Éstas variables pueden ser el número de ciclos que tarda la FSM en alcanzar el objetivo, el número de transiciones que tiene, el número de cambios de estado que se realiza, la energía gastada por el agente, entre otras.

6.1.1 Objetivos del comportamiento

El agente tiene como objetivo apagar todas las flamas que están distribuidas por todo el entorno, cada una de ellas tiene una posición que debe representar un área transitable ya que si no es así, el agente nunca podría llegar a ella. Para poder apagar cada flama, el agente cuenta con un extintor, que es un objeto que inicialmente el agente trae consigo. Este extintor tiene un número definido de cargas máximas, cada que se apaga una flama, éstas disminuyen, así, cuando la carga del extintor es igual a cero se debe de recargar para continuar apagando flamas.

Entonces, se espera que el algoritmo genético encuentre el comportamiento apropiado para que el agente que lo ejecute e iniciando desde una posición aleatoria en el mapa pueda apagar todas las flamas del entorno tomando en cuenta la prioridad de cada una de ellas, se deben apagar primero las flamas con mayor prioridad. La fuente de carga del extintor es ilimitada durante la simulación, aunque se espera que el comportamiento recargue el extintor un número mínimo de veces.

El extintor tiene un rango de acción de 2 unidades, entonces el agente debe encontrarse al menos a esta distancia de cada flama para poder apagarla, esto hace necesario que el agente se mueva a través del mapa. Después de apagar todas las flamas, el agente debe detenerse esperando que aparezcan más para seguir apagándolas.

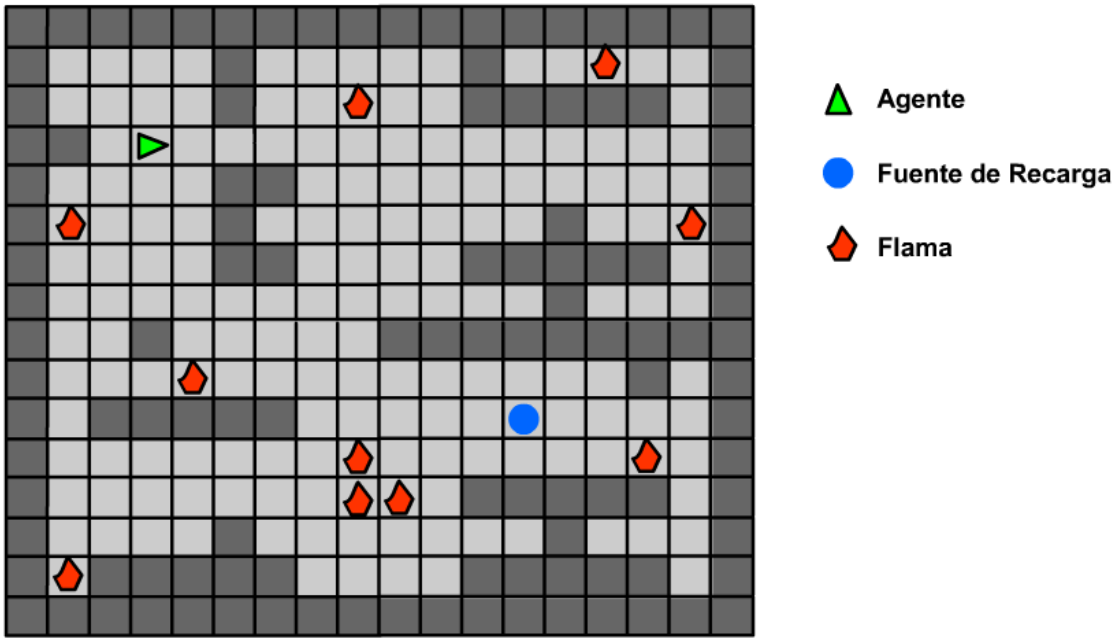


Figura 6.1: Mapa que muestra un conjunto de flamas que el agente debe apagar.

6.1.2 Características del agente

El agente a simular tiene acciones, percepciones y un conjunto de variables internas que definen algunos aspectos de su comportamiento, estos elementos se deben especificar en función de las metas del comportamiento. Para cumplir el objetivo anterior, se plantea un conjunto mínimo de las acciones básicas que el agente puede realizar. Cada acción corresponde a un estado en su comportamiento e incorporan funciones de inicialización, actualización y finalización.

Como funciones primitivas comunes a varios estados se tiene:

- **Moverse**: El agente se mueve, si existe un camino, hacia una posición dada del mapa.
  - *enter*: Ejecuta A* buscando un camino entre la posición actual del agente y la posición destino dada. Si el camino existe, hace `mbPathFound = true`.
  - *update*: Si `mbPathFound = true`
    - Si hay más nodos en el camino.
    - Se mueve hacia el siguiente nodo.
  - *exit*: Hace `mbPathFound = false`.

Sus acciones:

- **1: Quieto**: El agente está en modo de espera, inmóvil.
  - *enter*: No hace nada.
  - *update*: No hace nada.
  - *exit*: No hace nada.
- **2: IrASiguienteFlama**: Utiliza **Moverse** con la posición de la siguiente flama a apagar como destino.
  - *enter*: Si existen más flamas en el mapa, lee la posición de la siguiente (la de mayor prioridad). Esta información es proporcionada por la representación del entorno. Por último, ejecuta *enter* de **Moverse** con este destino.
  - *update*: *update* de **Moverse**.
  - *exit*: *exit* de **Moverse**.
- **3: Disparar**: Si el extintor tiene carga, lo dispara hacia una flama objetivo, si la flama se encuentra dentro del rango del extintor, la apaga.
  - *enter*: Calcula la distancia entre el agente y la flama objetivo, si es menor al rango del extintor, avisa al mapa que dicha flama está apagada. Si no se encuentra en rango, no hace nada.
  - *update*: No hace nada.
  - *exit*: No hace nada.
- **4: RecargarExtintor**: Recarga el extintor a su número máximo de cargas.
  - *enter*: Calcula la distancia entre el agente
  - *update*: No hace nada.
  - *exit*: No hace nada.
- **5: IrARecarga**: El agente se mueve hacia la fuente de recarga del extintor.
  - *enter*: Pregunta al entorno la posición de recarga del mapa y ejecuta el *enter* de **Moverse** para éste destino.
  - *update*: *update* de **Moverse**.
  - *exit*: *exit* de **Moverse**.

Además de las acciones que el agente puede realizar, éste debe contar con un conjunto de percepciones que le permiten leer variables propias y del entorno para, con base en ellas, dirigir su comportamiento. Como se dijo anteriormente, las percepciones definen las condiciones con las que se etiqueta cada una de las transiciones en la FSM que representa su comportamiento. El agente en estudio tiene 4

percepciones que al evaluarlas dan como resultado un valor booleano que indica si se cumple o no, y son las siguientes:

- **1: ExtintorVacio**: Percepción interna que lee el número de cargas del extintor, si está vacío devuelve *true* y *false* en caso contrario.
- **2: MasFlamas**: Percepción del entorno que indica si existen más flamas que apagar, si se han apagado todas las flamas devuelve *false*. Al contrario, si aún quedan flamas por apagar devuelve *true* y se obtiene la posición de la siguiente a apagar tomándola como objetivo del agente.
- **3: CercaDeFlama**: Evalúa si el agente se encuentra a la distancia necesaria (rango del extintor) para apagar la flama objetivo. Para que la acción **Disparar** tenga éxito, esta condición se debe cumplir.
- **4: CercaDeRecarga**: Evalúa si el agente se encuentra a la distancia necesaria de la fuente de recarga, si es así podrá cargar el extintor, si no, la acción **RecargarExtintor** no tendrá éxito.

Las variables internas del agente son:

- **Rango del extintor (`ROBOT_WATER_RANGE`)**: Distancia máxima a la que el agente debe encontrarse de una flama para poder apagarla en el momento de disparar el extintor.
- **Cargas iniciales del extintor (`ROBOT_WATER_INIT_CHARGES`)**: Al iniciar la simulación, el extintor tiene este número de cargas, puede ser cualquiera entre cero y el número máximo.
- **Cargas Máximas (`ROBOT_WATER_MAX_CHARGES`)**: Capacidad total del extintor, al recargarlo siempre se llena completo independientemente de su carga actual.
- **Rango de la fuente de carga (`ROBOT_WATER_CHARGE_RANGE`)**: Distancia máxima a la que el agente debe encontrarse de la fuente de recarga de agua para poder cargar el extintor.

Se tiene un agente con cinco acciones o estados en su comportamiento y cuatro percepciones que combinados en una FSM formarán el comportamiento que resuelva la tarea propuesta de una manera cercana a la óptima.

6.1.3 Entorno

El entorno está representado por celdas pasables y no pasables, que a su vez forman un grafo sobre el cual se controlará el movimiento del agente. También, el entorno tiene objetos que representan al conjunto de flamas y la fuente de carga de agua, que al iniciar la simulación se encuentran en posiciones válidas aleatorias. El entorno utilizado para las pruebas del sistema se muestra en la *Figura 6.2,* cada una de las celdas representa nodos transitables.

Figura 6.2: Mapa utilizado para las pruebas.

La representación del entorno lleva el control de la cantidad de flamas apagadas y encendidas, de sus posiciones y prioridades. Ésta es capaz de contestar al agente preguntas como: ¿Hay más flamas activas?, ¿Dónde está la siguiente flama a apagar?, ¿Dónde está la fuente de carga? o ¿Existe un camino entre el agente y una posición dada?

6.1.4 Función de aptitud

La medida de aptitud (función objetivo) para cada individuo se calcula utilizando una suma ponderada de un conjunto de variables asociadas al resultado de la simulación del agente ejecutando el comportamiento al que representa. Cada variable está multiplicada por una constante *ki* y son seis en total. La primera constante, *k1* multiplica a la cantidad de flamas apagadas durante la simulación.

Mientras más flamas apague el agente, mejor es el comportamiento evaluado ya que éste es su objetivo principal. Sin embargo, no basta con que el agente apague flamas, se deben de tomar en cuenta una serie de restricciones del comportamiento mismo, por ejemplo, el número de transiciones que tiene la máquina, ya que se busca aquella que logre realizar el objetivo con el menor número de ellas. Por esto se hace uso de la constante *k2* que multiplica el número de transiciones, tiene un valor negativo ya que penaliza el exceso de las mismas.

Además, en el caso de ser necesario, se espera que el agente recargue el extintor un número mínimo de veces para apagar todas las flamas, así que la función de aptitud penaliza la cantidad de veces que el agente recarga el extintor sin estar éste vacío. La constante *k3* multiplica al número de veces que el agente recargó el extintor sin estar vacío. La constante *k4* multiplica a la cantidad de cambios de estados que realizó la máquina. Así, se buscan máquinas que cumplan el objetivo en la menor cantidad de tiempo y cambios de estado.

Constante	Multiplica
k1	Número de flamas apagadas (**NF**)
k2	Número de transiciones de la máquina (**NT**)
k3	Número de recargas innecesarias (**UL**)
k4	Número de cambios de estado (**SC**)

La función de aptitud tiene la siguiente forma:

$$F = k1 * \boldsymbol{NF} + k2 * \boldsymbol{NT} + k3 * \boldsymbol{UL} + k4 * \boldsymbol{SC}$$

6.1.5 Selección

Como método de selección se utilizó el Vasconcelos, ya que este trabajo está basado en el artículo [19], en el cual, se reporta el algoritmo de un robot móvil que evade obstáculos obtenido por un algoritmo genético, después de varias pruebas se comprobó que el método Vasconcelos fue siempre el mejor convergiendo de una manera más rápida hacia buenas soluciones. Además, en [12] se hace un estudio general sobre varios algoritmos genéticos con características diferentes y aquel con selección Vasconcelos fue de los mejor calificados.

6.1.6 Cruzamiento

Como se mencionó anteriormente, para todos los experimentos se utiliza cruzamiento uniforme, en donde cada transición del primer padre tiene 0.5 de probabilidad (excepto en experimento 4) de pertenecer al primer hijo o al segundo. Así, se espera que cada hijo tenga un número igual de transiciones heredadas del primer padre que del segundo.

6.1.7 Mutación

La mutación de cada máquina sirve para eliminar, crear o modificar transiciones, cambiar condiciones y en caso de las máquinas de Moore, acciones a ejecutar. Se basa en dos probabilidades: p_e y p_m. La primera es la probabilidad de crear o eliminar una transición de la máquina, es decir, si la transición es cero (no existe) con probabilidad p_e se crea y si es diferente de cero, se elimina. El segundo valor (p_m) es la probabilidad de modificar una transición existente que no fue eliminada en el paso anterior. A las transiciones que se crean, se les asigna una condición aleatoria.

6.1.8 Representación como máquinas de Moore

Para esta representación se toman en cuenta las N_s^2 transiciones posibles que puede haber entre los N_s estados, cada una de ellas está representada por un número entero, por lo tanto, la longitud de la representación de cada individuo es de longitud N_s^2. Por ejemplo, la transición entre el estado 2 y el 3 estará representada por el elemento $N_s + 3$ y en general la transición del estado i al j estará representada por el

elemento $(i-1)*N_s+j$. La posición en la representación define la transición, si ésta es igual a cero indica que dicha transición no está presente en la máquina, si no lo es, su valor indica la condición que se debe de cumplir para que dicha transición se active.

6.1.9 Representación como máquinas de Mealy

En el caso de las máquinas de Moore, la salida (acción a ejecutar) depende de cada transición y no de los estados. Así, su representación, utilizada en los experimentos 3 y 4, se realiza de forma similar a la de una máquina de Moore pero con la diferencia de que cada transición es representada por una pareja de enteros, el primero, representa la condición de activación y de igual forma, si es igual a cero, la transición no existe. El segundo número representa la acción a ejecutar cuando se active la transición, debido a esto, la representación con máquinas de Mealy tiene una longitud de $2*N_s^2$.

6.2 Experimento 1

Se busca que el agente apague 10 flamas en el entorno con un número suficiente de cargas del extintor, en este caso 10 también, es decir, no será necesario recargarlo ya que siempre se apagarán todas las flamas antes que se quede vacío. La representación de los comportamientos es de longitud fija como maquinas de Moore.

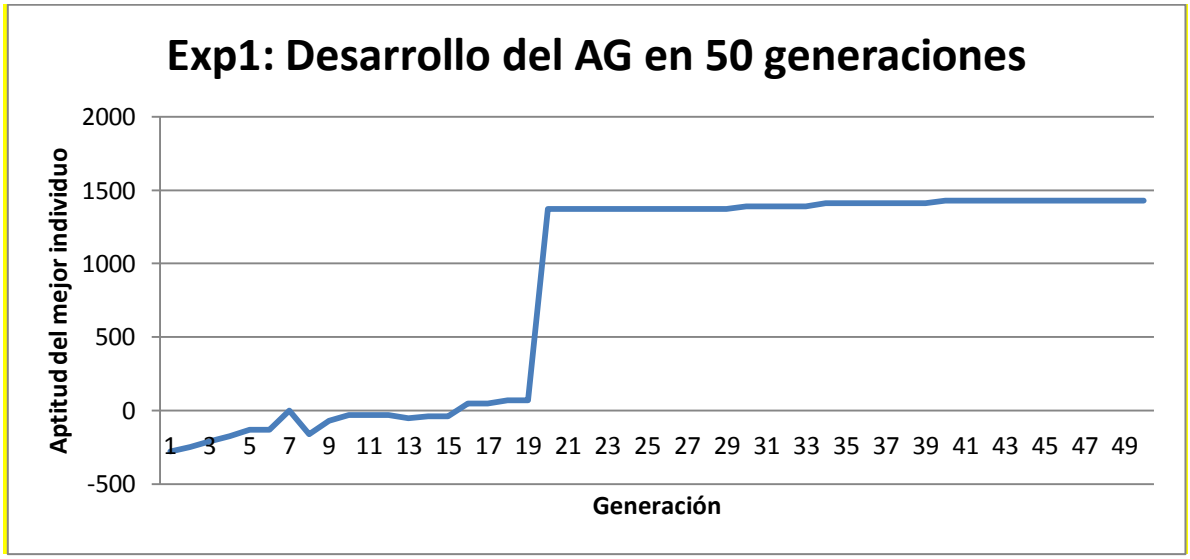
```
FLAMES_NUMBER = 10
ROBOT_WATER_INIT_CHARGES = 10
ROBOT_WATER_MAX_CHARGES = 10
```

Los valores de las constantes de la función de aptitud, los cuales se fijan a partir de un proceso de prueba y error, los que mejores resultados reportaron son los siguientes:

```
k1 = 150.0f          k3 = 0.0f
k2 = -20.0f          k4 = -0.5f
```

6.2.1 Resultados

El algoritmo se ejecutó 100 veces con 50 generaciones de 50 individuos cada una, selección determinística Vasconcelos, con probabilidades de mutación p_e = 0.05 y p_m = 0.2, y utilizando cruzamiento uniforme, el algoritmo encontró el **92%** de las veces un comportamiento que cumpliera con el objetivo. Con una aptitud promedio de 1284.21 y el mejor calificado con 1430.



Gráfica 6.1: Desarrollo del AG para el experimento 1 por 50 generaciones.

En la figura 6.3 se muestran los diagramas de estado de las dos mejores máquinas encontradas por el algoritmo, se puede ver que el primer individuo (arriba) tiene una medida de aptitud de 1430 y el segundo (abajo) de 1425 a pesar que ambos cumplen con la meta. Esto se debe a que la segunda FSM hace un cambio de estado innecesario después de apagar cada flama, regresando al estado **Quieto** después de **Disparar**. El valor negativo de la constante *k4* en la función de aptitud indica esta penalización (-0.5 por cada una de las 10 flamas).

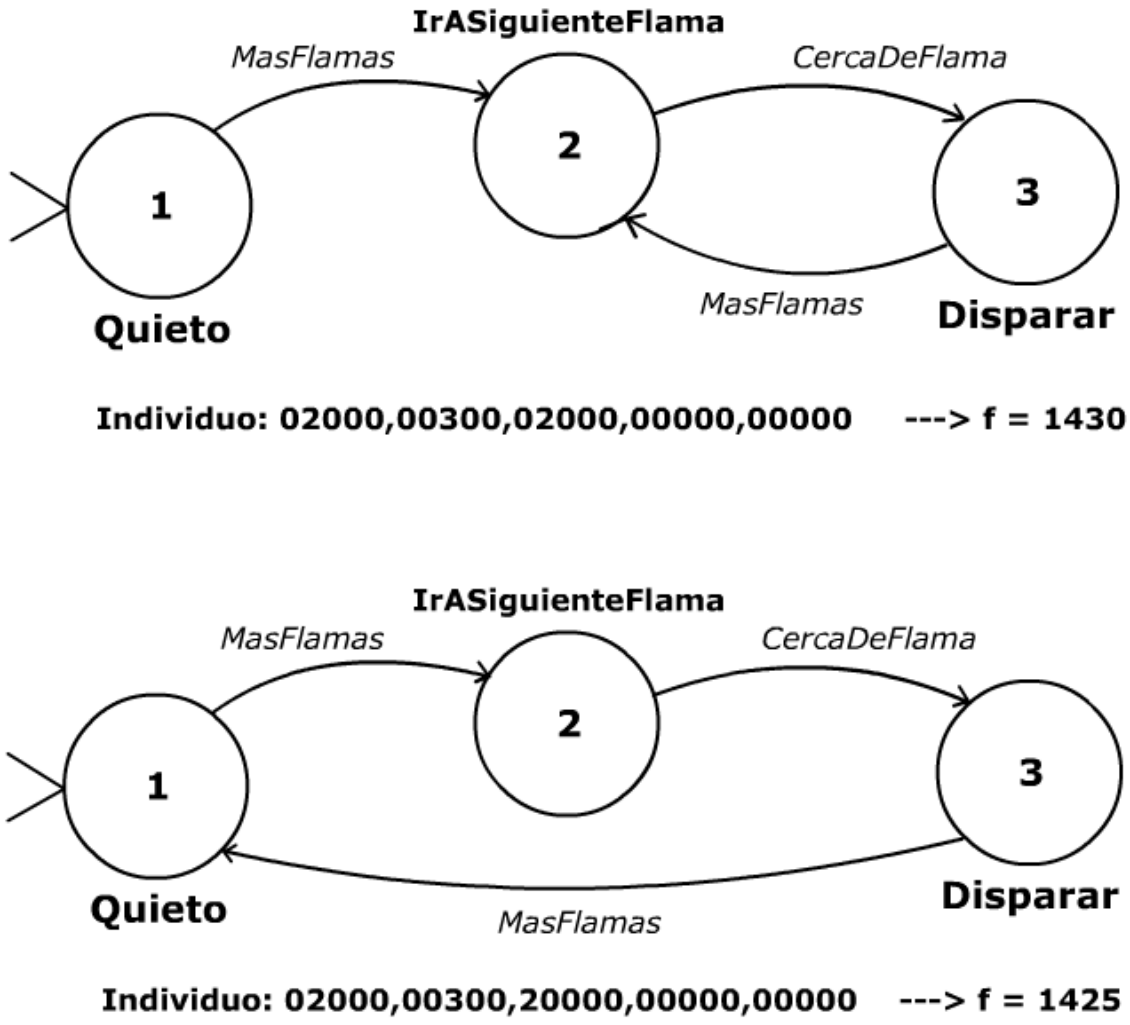


Figura 6.3: Los dos mejores individuos encontrados por el algoritmo para el experimento 1.

Se puede ver que los comportamientos cumplen la meta utilizando solo tres de los cinco estados posibles del agente ya que por las características del extintor (que nunca queda vacio), dos no son necesarios. Los estados 4 y 5 (últimos 10 dígitos del individuo) aparecen con ceros ya que no requieren transiciones porque nunca se llega a ellos. En realidad podrían tener cualquier condición en sus transiciones y seguirían cumpliendo la meta.

Los 10 mejores individuos encontrados en el experimento se muestran en la figura 6.4, se puede ver que a partir del tercer individuo, empiezan a tener transiciones inútiles, esto no evita que cumplan el objetivo de la misma forma que los otros, sin embargo su calificación es menor gracias a la constante *k2* en la función de aptitud que penaliza a las máquinas que tienen transiciones de más (figura 6.5).

```
[1430]-Individual: 0,2,0,0,0 || 0,0,3,0,0 || 0,2,0,0,0 || 0,0,0,0,0 || 0,0,0,0,0 ||
Individual: 0,2,0,0,0 || 0,0,3,0,0 || 0,2,0,0,0 || 0,0,0,0,0 || 0,0,0,0,0 ||
Individual: 0,2,0,0,0 || 0,0,3,0,0 || 2,0,0,0,0 || 0,0,0,0,0 || 0,0,0,0,0 ||
Individual: 0,2,0,0,0 || 0,0,3,0,0 || 2,0,0,0,0 || 0,0,0,0,0 || 0,0,0,0,0 ||
Individual: 0,2,0,0,3 || 0,0,3,0,0 || 0,2,0,0,0 || 0,0,0,0,0 || 0,0,0,0,0 ||
Individual: 0,2,3,0,0 || 0,0,3,0,0 || 0,2,0,0,0 || 0,0,0,0,0 || 0,0,0,0,0 ||
Individual: 0,2,3,0,0 || 0,0,3,0,0 || 0,2,0,0,0 || 0,0,0,0,0 || 0,0,0,0,0 ||
Individual: 0,2,0,0,0 || 0,0,3,0,0 || 0,2,0,0,0 || 0,0,0,0,0 || 0,0,0,3,0 ||
Individual: 0,2,0,0,0 || 0,0,3,0,0 || 0,2,0,0,0 || 0,0,0,0,0 || 0,4,0,0,0 ||
Individual: 0,2,4,0,0 || 0,0,3,0,0 || 0,2,0,0,0 || 0,0,0,0,0 || 0,0,0,0,0 ||
Individual: 0,2,3,0,0 || 0,0,3,0,0 || 0,2,0,0,0 || 0,0,0,0,0 || 0,0,0,0,0 ||
```

Figura 6.4: Los 10 mejores individuos encontrados por el algoritmo para el experimento 1.

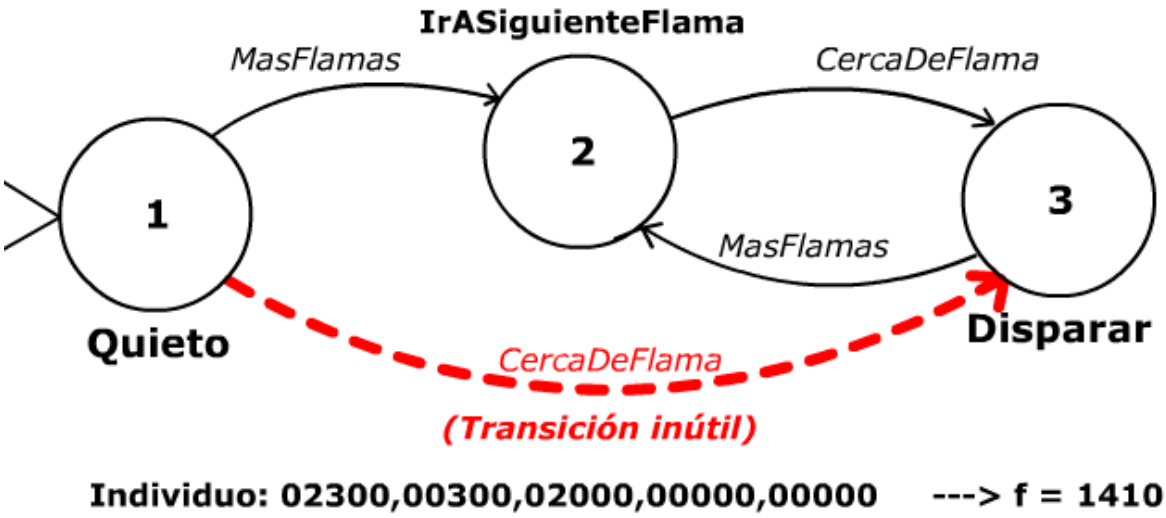


Figura 6.5: Individuo con una transición inútil, es penalizado.

6.3 Experimento 2

Esta vez el extintor tendrá un número menor de cargas iniciales para obligar al agente a que cargue el extintor al menos una vez si quiere apagar todas las flamas. Este cambio es propio del agente, y ahora, el comportamiento del experimento 1 ya no logra la meta (ahora hay menos cargas, puede verse también como que el número de flamas en el entorno aumentó) por lo que es necesario que el AG encuentre una nueva solución al problema.

```
FLAMES_NUMBER = 10
ROBOT_WATER_INIT_CHARGES = 4
ROBOT_WATER_MAX_CHARGES = 10
```

Los valores de las constantes de la función de aptitud, los cuales se fijan a partir de un proceso de prueba y error, los que mejores resultados reportaron son los siguientes:

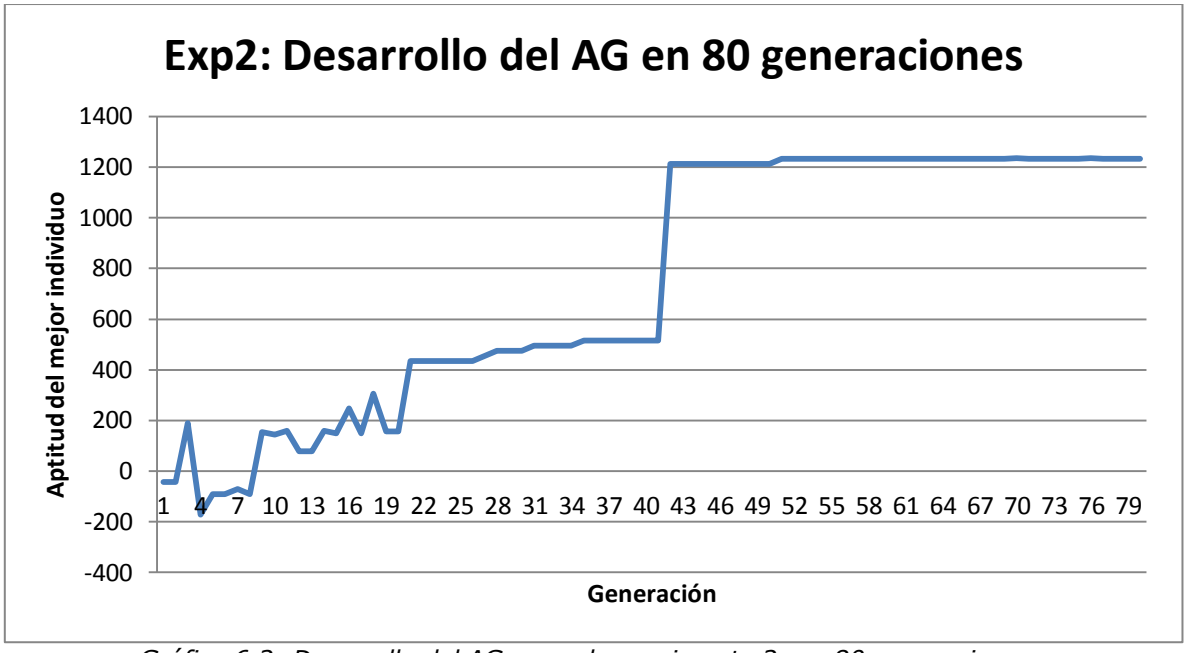
```
k1 = 150.0f          k3 = -45.0f
k2 = -20.0f          k4 = -0.5f
```

6.3.1 Resultados

El algoritmo se ejecutó 100 veces con 80 generaciones de 80 individuos cada una, selección determinística Vasconcelos, con probabilidades de mutación p_e = 0.05 y p_m = 0.2, y utilizando cruzamiento uniforme, el algoritmo encontró el **85%** de las veces un comportamiento que cumpliera con el objetivo. Con una aptitud promedio de 1168.1 y el mejor calificado con 1234. Esta vez las condiciones del problema hacen necesario el uso de las dos acciones adicionales que puede realizar el agente ya que como el extintor tiene cargas limitadas, en algún momento quedará vacío obligando al agente a cargarlo.

En la gráfica 6.2 se muestra el desarrollo del algoritmo genético durante sus 80 generaciones, se puede ver que en la generación 42, la gráfica tiene un salto súbito. Esto se debe a que a partir de este punto, aparecen individuos que han "aprendido" a recargar el extintor, justo en este paso se puede ver la capacidad de adaptación que provee el algoritmo. El proceso empieza con comportamientos aleatorios, agentes que

no saben hacer nada. Posteriormente aparecen aquellos que saben apagar flamas y por último aquellos que saben apagar flamas y recargar el extintor.



Gráfica 6.2: Desarrollo del AG para el experimento 2 por 80 generaciones.

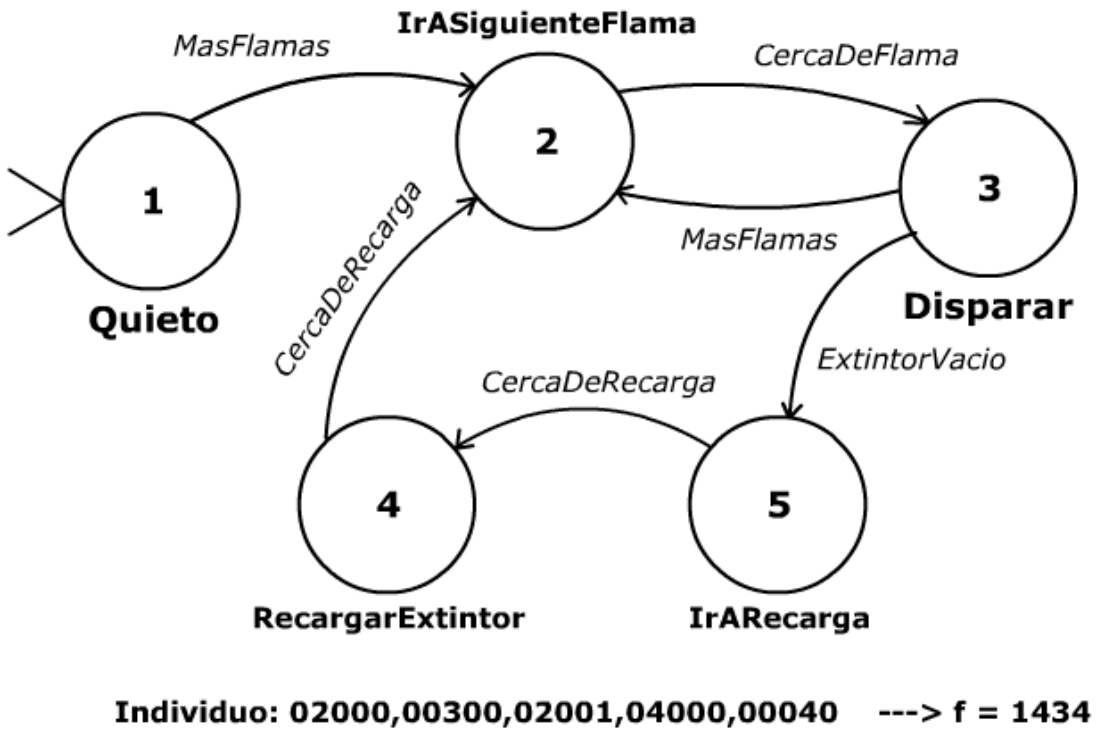


Figura 6.6: El mejor individuo encontrado por el algoritmo para el experimento 2.

El mejor individuo encontrado en este experimento (figura 6.6) tiene las mismas transiciones que el encontrado en el experimento 1, además tiene tres adicionales que permiten realizar las acciones de recarga. De hecho, se comporta exactamente igual mientras el extintor no se quede sin cargas.

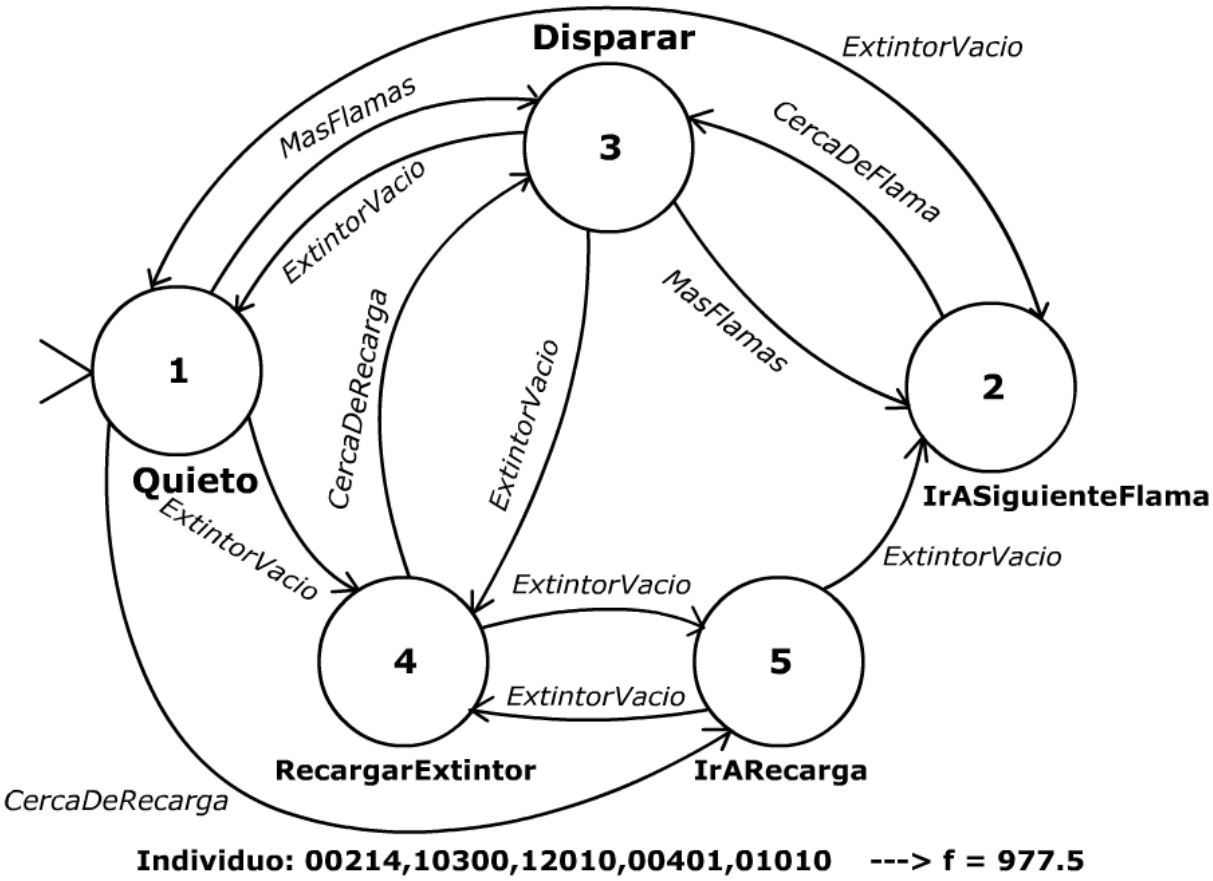


Figura 6.7: Un individuo con baja calificación pero que cumple con el objetivo para el experimento 2.

La máquina de la figura 6.7 muestra un comportamiento que al igual que el de la figura 6.6 logra que el agente apague todas las flamas del entorno, sin embargo su representación es mucho más compleja, tiene el doble de transiciones, muchas de ellas inútiles y además realiza cambios de estado innecesarios, por lo que consume más ciclos para lograr su objetivo. Debido a esto fue una de las peor evaluadas a pesar de apagar todas las flamas.

6.4 Experimento 3

Las condiciones para este experimento son las mismas que para el experimento 1, el extintor nunca quedará vacío, con la diferencia que esta vez los comportamientos estarán representados como máquinas de Mealy. Ahora no es necesario que haya tantos estados como acciones, así que el experimento se realiza para 3 estados máximos (**MEALY_MAX_STATES)**.

```
FLAMES_NUMBER = 10
ROBOT_WATER_INIT_CHARGES = 10
ROBOT_WATER_MAX_CHARGES = 10
MEALY_MAX_STATES = 3
```

Los valores de las constantes de la función de aptitud, los cuales se fijan a partir de un proceso de prueba y error, los que mejores resultados reportaron son los siguientes:

```
k1 = 150.0f          k3 = 0.0f
k2 = -20.0f          k4 = -0.5f
```

6.4.1 Resultados

El algoritmo se ejecutó 100 veces con 80 generaciones de 80 individuos cada una, selección determinística Vasconcelos, con probabilidades de mutación p_e = 0.05 y p_m = 0.2, y utilizando cruzamiento uniforme, el algoritmo encontró el **82%** de las veces un comportamiento que cumpliera con el objetivo. Con una aptitud promedio de 1167 y el mejor calificado con 1458.

El algoritmo encontró comportamientos que cumplen el objetivo utilizando únicamente 2 estados de los 3 máximos que se plantearon, también se puede ver que la acción **Quieto** ya no es necesaria, por lo que solo se utilizan 2 de ellas en contraste con el experimento 1. Además, esto tiene como consecuencia que la representación tenga menos transiciones y que su ejecución se realice en menos ciclos con menos cambios de estado, lo que da como resultado una medida de aptitud mayor. En la figura 6.8 se muestra el mejor individuo encontrado

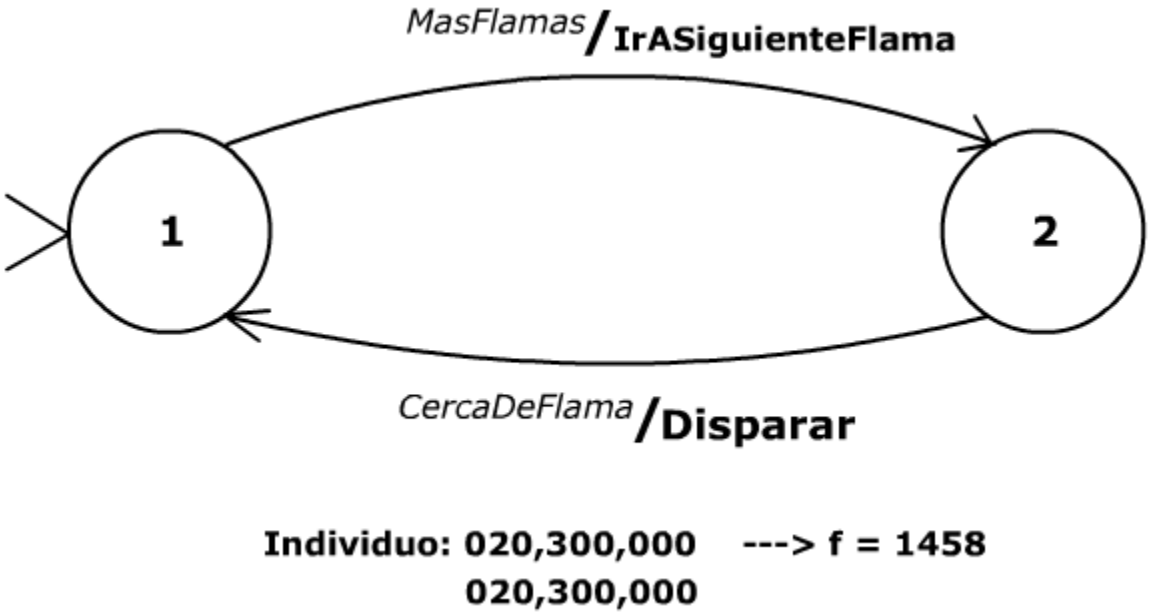


Figura 6.8: El mejor individuo encontrado por el algoritmo para el experimento 3.

6.5 Experimento 4

Las condiciones para este experimento son las mismas que para el experimento 2, en el entorno hay más flamas que cargas en el extintor, con la diferencia que esta vez los comportamientos estarán representados como máquinas de Mealy con 4 estados como máximo.

```
FLAMES_NUMBER = 10
ROBOT_WATER_INIT_CHARGES = 4
ROBOT_WATER_MAX_CHARGES = 10
MEALY_MAX_STATES = 4
```

Los valores de las constantes de la función de aptitud, los cuales se fijan a partir de un proceso de prueba y error, que mejores resultados reportaron son los siguientes:

```
k1 = 150.0f          k3 = 0.0f
k2 = -20.0f          k6 = -0.5f
```

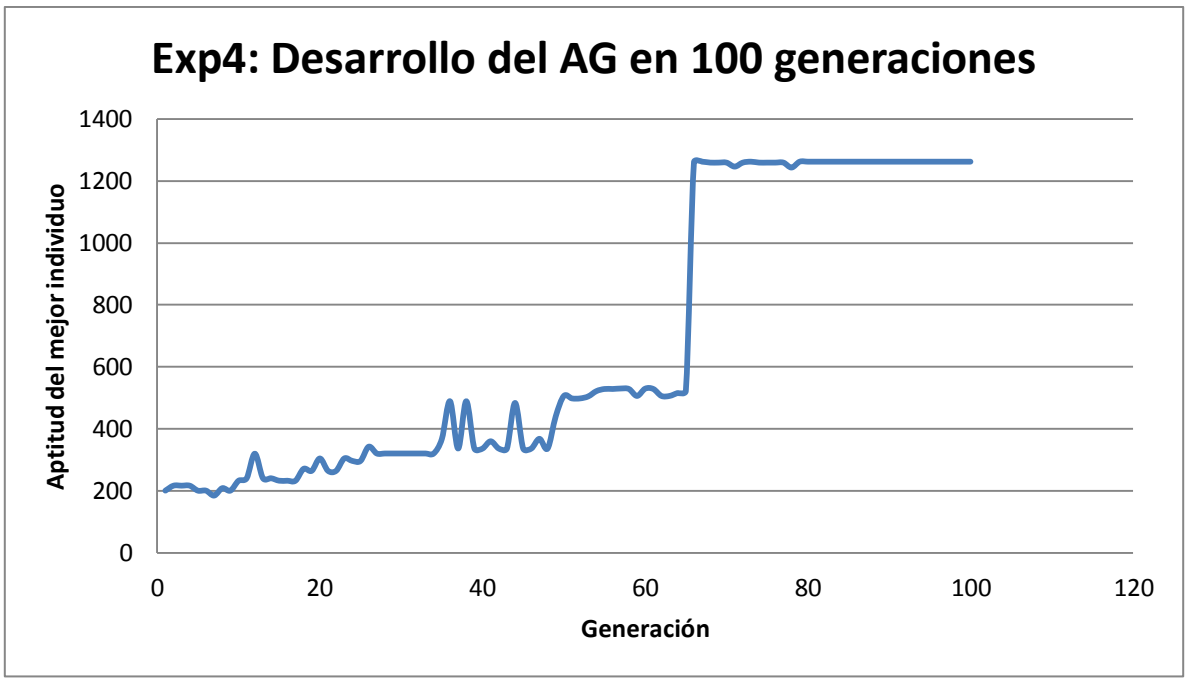
En este experimento, se utiliza de igual forma el cruzamiento uniforme pero para cada transición del primer padre, la probabilidad de pertenecer al segundo hijo es muy baja (0.05), por lo tanto, la probabilidad de que el primer hijo sea igual al primer padre aumenta, lo que reduce considerablemente el impacto del operador de cruzamiento en el algoritmo. Esto se hizo así ya que se notaban efectos destructivos generados por el cruzamiento en el algoritmo para este experimento.

6.5.1 Resultados

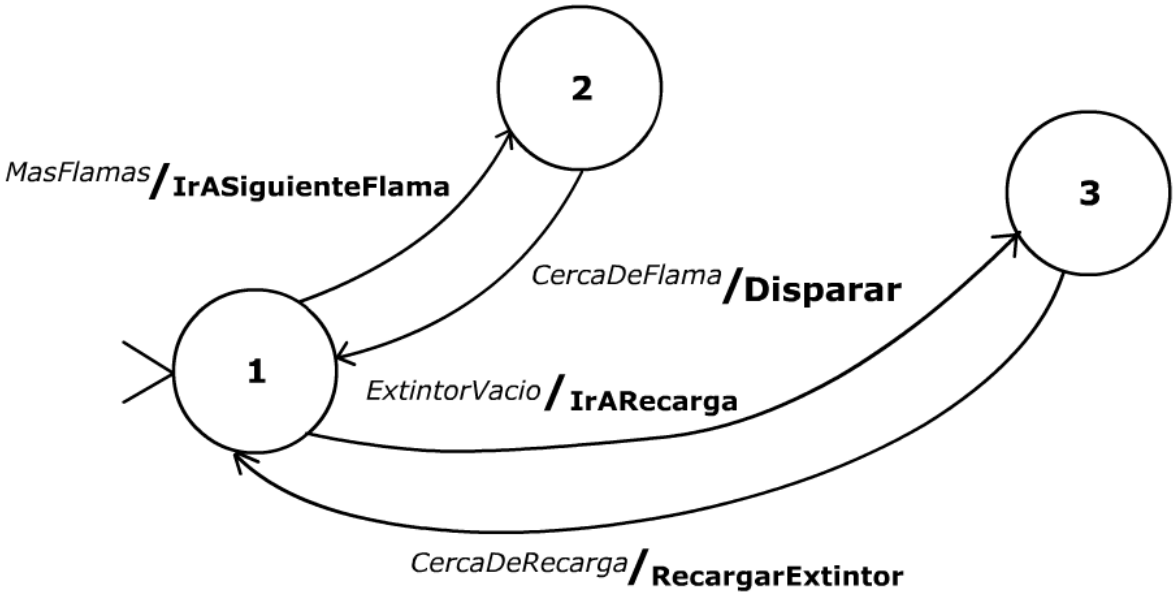
El algoritmo se ejecutó 100 veces con 100 generaciones de 100 individuos cada una, selección determinística Vasconcelos, con probabilidades de mutación p_e = 0.05 y p_m = 0.2, y utilizando cruzamiento uniforme, el algoritmo encontró el **77%** de las veces un comportamiento que cumpliera con el objetivo. Con una aptitud promedio en los comportamientos satisfactorios de 1213 y el mejor calificado con 1269.

Esta vez el algoritmo logró encontrar un comportamiento que logra el objetivo con 3 estados en vez de 5 como la presentada en el experimento 2. También se encontraron comportamientos que utilizan 4 estados como el mostrado en la parte inferior de la figura 6.9 que al requerir más transiciones y cambios de estado son penalizados y tienen menor medida de aptitud.

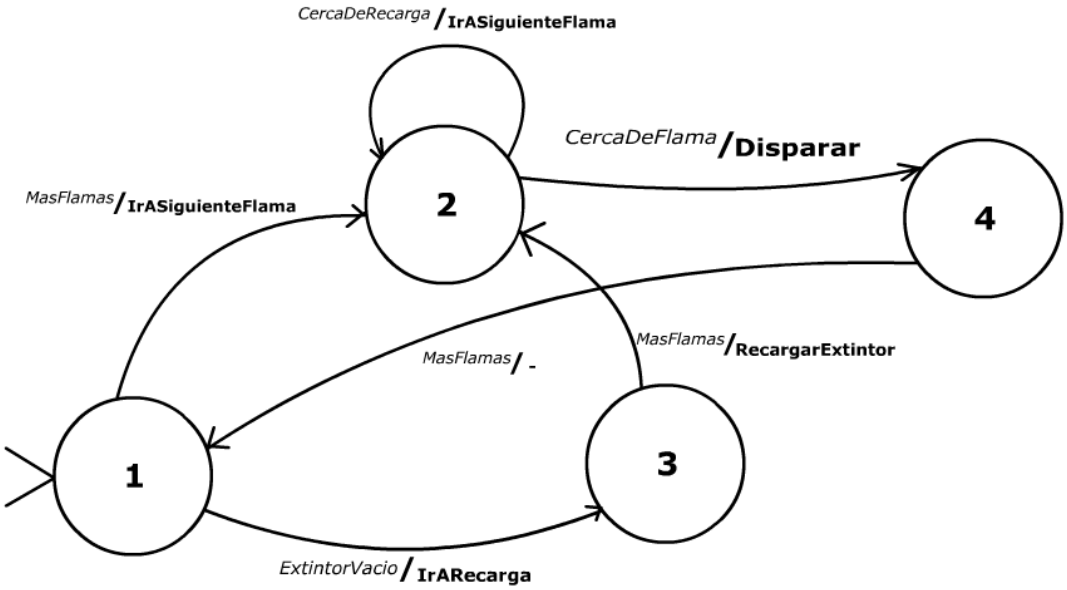
El número de individuos por generación se incrementa en cada experimento con el fin de realizar una exploración del espacio, que es mayor cuando se requiere recargar el extintor, y encontrar soluciones lo más rápido posible sin que el algoritmo tardara demasiado en su ejecución, después de algunas pruebas se fijaron estos valores.



Gráfica 6.3: Desarrollo del AG para el experimento 4 por 100 generaciones.



Individuo: 0201,3000,0000,4000 ---> f = 1269.5
0205,3000,0000,4000



Individuo: 2210,0403,0200,2000 ---> f = 1245.5
0250,0203,0410,0000

Figura 6.9: Arriba: El mejor individuo encontrado por el algoritmo para el experimento 3. Utiliza 3 estados para cumplir con el objetivo. Abajo: Un individuo que utiliza 4 estados para cumplir el objetivo, es penalizado.

7. Conclusiones y trabajo futuro

7.1 Conclusiones

Se comprobó la utilidad de los algoritmos genéticos en la evolución de máquinas de estados que representan comportamientos para agentes virtuales. La representación de longitud fija funciona correctamente para agentes con conjuntos de acciones y percepciones definidas y para representar máquinas de Moore y algunas de Mealy. La ejecución de comportamientos a partir de esta representación es rápida, fácil de implementar y permite al agente cambiar instantáneamente su comportamiento. Además, al ser una cadena de números enteros de tamaño fijo, puede ser operada directamente por un algoritmo genético.

El método cumplió con el objetivo general del trabajo, sin embargo, el tamaño del espacio de soluciones y su rápido crecimiento con respecto al número de estados hace que para una cantidad relativamente pequeña de estados (seis), el algoritmo consuma suficiente tiempo para no hacer viable su ejecución continua en tiempo real. Sin embargo, estos procesos pueden realizarse sin problema en la etapa de diseño, al iniciar la aplicación o durante intervalos destinados a estos procesos, por ejemplo, al pasar de una escena a otra durante el proceso de carga de recursos.

El motor de gráficos, Ogre, comprueba ser una buena alternativa para construir ambientes virtuales debido a su flexibilidad, robustez y a que brinda la capacidad de construir un sistema a la medida modificando si así se desea, cualquier aspecto de su arquitectura. Además, al estar construido sobre C++, que es un lenguaje compilado, provee de gran rendimiento, lo que se traduce en más polígonos, efectos especiales o texturas en pantalla. El principal problema de utilizar dos simuladores es que ambos deben calcular datos exactamente iguales. Para esto, es necesario diseñar un sistema de ejecución de comportamientos determinístico, de esta forma los resultados de ambos simuladores serán iguales para las mismas entradas y más aún, se podrán predecir.

En caso de aplicar el método a robots reales, el problema está en realizar una simulación virtual lo más fiel posible a la realidad, lo que implica tomar en cuenta la mayor cantidad posible de variables relevantes. Si la simulación virtual no es lo suficientemente apegada a la realidad del robot, los comportamientos no serán consistentes.

El método presentado es una buena alternativa para lograr adaptabilidad en comportamientos basados en máquinas de estado ya que dota al sistema de la capacidad de modificar la estructura de los comportamientos de un agente basado en su estado mismo y el del entorno. Esta capacidad recae directamente en la modificación de las variables operadas por la función de aptitud y la representación simbólica del entorno.

7.2.1 Algunas consideraciones

Durante el desarrollo del algoritmo genético, se puede ver en las gráficas correspondientes (6.1, 6.2 y 6.3) que el fitness del mejor individuo en cada generación no siempre es mayor que el de la generación anterior. Esto es aparentemente contradictorio ya que al utilizar el método de selección Vasconcelos combinado con el elitismo total, se asegura que el mejor individuo de cada generación, es el mejor de todas las generaciones pasadas. Esto es cierto, siempre y cuando la función de aptitud califique con el mismo valor al mismo individuo evaluado varias veces; en el problema presentado esto no sucede ya que la configuración del entorno (la posición de las flamas) cambia en cada evaluación.

Cada comportamiento debe ser evaluado en entornos aleatorios para así encontrar aquel que pueda cumplir los objetivos en la mayor cantidad de configuraciones posible del entorno. Entonces, el valor de aptitud de cada individuo es un promedio de tres evaluaciones en tres configuraciones aleatorias. Un mismo individuo puede tardar unos pocos más de ciclos para configuraciones donde las flamas estén muy alejadas con respecto a otras donde las flamas estén más cercanas a él. Es por esto que la penalización por cantidad de ciclos debe ser pequeña con relación a la ganancia por apagar una flama. Este tipo de cálculos causa que un mismo individuo pueda ser calificado diferente al evaluarlo varias veces.

Además de esto, el algoritmo encontró comportamientos que no utilizaban algunas condiciones aparentemente básicas para realizar una acción, por ejemplo se puede pensar que la condición si *ExtintorVacio* entonces **IrARecarga** debe estar en un comportamiento exitoso, sin embargo no aparece en algunos individuos. En su lugar aparecen condiciones como: si *MasFlamas* entonces **IrARecarga** después de una como: si *ExtintorVacio* entonces **No hagas nada**, de hecho no vale la pena recargar el extintor si no existen más flamas.

Esto se puede presentar cuando la última flama se apaga con la última carga de extintor, así se evita cargar el extintor antes de que el agente se "de cuenta" que ya no hay flamas, consumiendo ciclos. En este tipo de lógica inesperada y ajena al desarrollador es justo donde está la magia de utilizar un método automático.

Las FSM se han utilizado para modelar comportamientos, manejar los estados de la aplicación, organizar animaciones, reproducción de audio y un sinfín de aplicaciones en videojuegos. Es una técnica que ha probado funcionar por mucho tiempo y con seguridad se seguirá aplicando ya que es fácil de implementar. Es cierto que existen métodos mucho más complejos y modernos, sin embargo no siempre son necesarios en videojuegos y ambientes virtuales ya que al hacer inteligencia artificial para estos sistemas se debe tener un equilibrio entre entretenimiento y rendimiento.

7.3 Trabajo futuro

Como trabajo futuro se planten las siguientes extensiones:

- Hacer pruebas con robots reales implementando un simulador basado en sensores, el control de motores y su localización.

- Implementar un método automático de optimización para afinar los valores de las constantes *k* de la función de aptitud.

- Explorar la programación evolutiva como nuevo enfoque para la resolución del problema de evolución de comportamientos para agentes en videojuegos.

- Implementar este método en una aplicación real para dispositivos móviles con el fin de evaluar su rendimiento bajo restricciones de memoria y procesador.

- Diseñar un sistema de generación de condiciones. Esto evitaría tener un conjunto de condiciones fijo permitiendo que se generen nuevas utilizando las percepciones básicas y combinándolas con operadores de comparación.

- Explorar la posibilidad de hacer una implementación en paralelo para acelerar el tiempo de evolución.

8. Glosario

- **API**.- Todos los sistemas operativos proporcionan una forma para que las aplicaciones utilizan los recursos del sistema usando una interfaz de programación de aplicaciones o API. Esto es generalmente definido por una extensa lista de funciones y clases y variables. Todas las bibliotecas de programación (y hay miles disponibles, tanto comerciales o libres) poseen un API. (http://cplus.about.com/od/introductiontoprogramming/g/apidefn.htm)

- **Cheating**.- En el contexto de los videojuegos, Cheating se refiere a la ruptura de las reglas (por parte de un jugador o de la inteligencia artificial) para obtener ventaja en una situación competitiva.

- **Clase template**.- Las plantillas de clases (template class, en inglés) son una herramienta característica del lenguaje de programación C ++, que permite a las funciones y a las clases trabajar con tipos de datos genéricos. Esto permite que una función o una clase pueda trabajar con varios tipos de datos diferentes sin que la definición de la clase tenga que ser reescrita para cada uno, mas información en : http://www.bgsu.edu/departments/compsci/docs/templates.html

- **Code Blocks**.- Es un IDE de código libre para el lenguaje C++ , diseñado para satisfacer las necesidades más exigentes de sus usuarios, puede utilizando en todas las plataformas, integrando todas las necesidades de programadores exigentes, para mayor información: http://www.codeblocks.org/

- **Consolas de videojuegos**.-(sinónimo: videoconsola) es un aparato y sistema electrónico especializado de entretenimiento para el hogar que ejecuta juegos electrónicos (videojuegos), dichos juegos que están contenidos en cartuchos, discos ópticos, discos magnéticos o tarjetas de memoria, los primeros sistemas de videoconsolas fueron diseñados únicamente para jugar videojuegos pero a partir de la sexta generación de videoconsolas han sido incorporadas características

importantes de multimedia, internet, tiendas virtuales, algunos ejemplos de consolas modernas son: Xbox-360,PlayStation 3,Nintendo Wii.

- **Dispositivos hápticos**.- Los dispositivos hápticos permiten la retroalimentación (con algún tipo de fuerza) al individuo que interactúa con entornos virtuales o remotos. Tales dispositivos trasladan una sensación de presencia al operador.

- **Fitness**.- En los algoritmos genéticos, se le llama fitness a una función de aptitud, que se utiliza para resumir, con una sola cifra de mérito, la proximidad de una solución de diseño dado el logro de los objetivos establecidos.

- **Formato XML**.- XML,por sus siglas en inglés de eXtensible Markup Language ('lenguaje de marcas extensible'), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C), actualmente se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas como en bases de datos, editores de texto, sistemas de realidad virtual, etc.

- **Gameplay**.-Es la forma específica en el que los jugadores interactúan con un un juego de vídeo. El gameplay está definido por reglas de juego (conexión entre jugador y el juego), los problemas que resolver, la superación de ellos, y finalmente el argumento de juego.

- **Inmersión**.-En realidad virtual la inmersión es la capacidad del ambiente de poder concentrar toda la atención del individuo en una situación o actividad específica de forma que éste se sienta dentro de un ambiente sintético.

- **Joysticks**.- Es un dispositivo de control de dos o tres ejes que se usa desde una computadora o videoconsola, poseen botones de acción que pueden incorporar controles deslizantes siendo estos últimos más precisos.

- **Videojuegos de estrategia en tiempo real**.- Los videojuegos de estrategia en tiempo real o RTS (siglas en inglés de real-time strategy) son videojuegos de estrategia en los que no hay turnos sino que el tiempo transcurre de forma continua para el o los jugadores,son un subgénero de los juegos de estrategia en la

que las acciones de os jugadores tienen efecto inmediato en el ambiente.

- **Microsoft Visual Studio**.- Microsoft Visual Studio es un entorno de desarrollo integrado para sistemas operativos Windows. Soporta varios lenguajes de programación tales como Visual C++, Visual C#, Visual J#, ASP.NET y Visual Basic .NET. http://www.microsoft.com/spain/visualstudio

- **Motor de Software** – Un motor de software se refiere a la parte central de un sistema, aquella que provee la funcionalidad principal. Un motor de gráficos provee funcionalidad básica para manipular y dibujar gráficos en pantalla.

- **Namespace.-** es un conjunto de nombres en el cual todos los nombres son únicos.Un espacio de nombres es un contexto en el que un grupo de uno o más identificadores pueden existir. Un identificador definido en un espacio de nombres está asociado con ese espacio de nombres, C++ utilizar esta propiedad.

- **NPC**.- Un NPC **(non-player character)**, es un personaje controlado por la computadora en un videojuego, y no por un humano.

- **Pac-man**.- es un videojuego arcade creado por el diseñador de videojuegos Toru Iwatani de la empresa Namco, y distribuido por Midway Games al mercado estadounidense a principios de los años 1980. El protagonista del videojuego Pac-Man es un círculo amarillo al que le falta un sector por lo que parece tener boca. Aparece en laberintos donde debe comer puntos pequeños (llamados Pac-dots en inglés), puntos mayores y otros premios con forma de fruta.

- **Pathfinding**.- Se refiere al cálculo, por una aplicación informática, de la ruta más corta entre dos puntos en un grafo ponderado. Usualmente se basa en algoritmos voraces, exhaustivos o heurísticos como el de Dijkstra, A* o BFS.

- **videojuegos AAA**.- El término AAA (triple-A) se refiere a los juegos de la más alta calidad desarrollados con grandes presupuestos, equipos de trabajo y que generalmente generan una gran cantidad de ventas.

9. Referencias

- **[1]Anderson** James A., *Automata theory with modern applications,* Cambridge University Press, USA 2006.

- **[2]Arkin**, R.C. Behavior-Based Robotics, MIT Press, Cambridge, MA, 1998.

- **[3]Bartle** Richard A., *Designing Virtual Worlds,* New Riders Publisher, 2003.

- **[4]Brooks**, R. A. "Intelligence without Representation", Artificial Intelligence 47 (1991), 139–159, 1991.

- **[5]CEGUI** Official Layout Editor CELayoutEditor.
  url: http://www.cegui.org.uk/wiki/index.php/CELayoutEditor_Manual,
  Último acceso: viernes 25 de Noviembre 2011.

- **[6]*Crazy*** *Eddie's GUI System* http://www.cegui.org.uk,
  Último acceso: viernes 25 de Noviembre 2011.

- **[7]Dawson**, Michael, *Beginning C++ game programming*, Thomson Course Technology PTR, Premier Press, USA, 2004.

- **[8]Ellis** Stephen R., *What are Virtual Environments?,* Journal IEEE Computer Graphics and Applications archive: Volume 14 Issue 1, 1994.

- **[9]*Escape*** *from Woomera*, Octubre 2004,
  url: http://www.moddb.com/mods/escape-from-woomera,
  Último acceso: viernes 25 de Noviembre 2011.

- **[10]Franklin**, Stan, Art Graesser*, Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents,* Institute for Intelligent Systems, University of Memphis,USA 1996.
  url: http://www.msci.memphis.edu/~franklin/AgentProg.html,

Último acceso: viernes 25 de Noviembre 2011.

- **[11]Kuri** A., Casas José G. Algoritmos genéticos, Instituto Politécnico Nacional, Universidad Nacional Autónoma de México, Fondo de Cultura Económica, México, 2002.

- **[12] Kuri**-**Morales**, A., "A Methodology for the Statistical Characterization of Genetic Algorithms", Lecture Notes in Artificial Intelligence: No. 2313, Springer-Verlag, pp. 79-88, Editor(s)): Coello, C., Albornoz, A., Sucar, E., Cairó, O., ISBN: 3-540-43475-5, ISSN: 0302-9743, 01/04/2002.

- **[13]Maes**, Pattie, *A bottom-up mechanism for behavior selection in an artificial creature*. In J. A. Meyer and S. Wilson editors, Proceedings of the First International Conference on Simulation of Adaptive Behavior. The MIT Press, Cambridge, 1991.

- **[14]Mealy**, George H., *A Method for Synthesizing Sequential Circuits*. Bell Systems Technical Journal 34, pag: 1045–1079, 1995.

- **[15]Moore**, Edward*, Gedanken-experiments on Sequential Machines*. Automata Studies, Annals of Mathematical Studies, Princeton University Press, 1956, pag: 129–153.

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

- **[17]*Ogre*** *engine*, url: www.ogre.org, Último acceso: viernes 25 de Noviembre 2011.

- **[18]Savage** Jesús, Vázquez Gabriel, Diseño de Microprocesadores, Facultad de Ingeniería, Ciudad Universitaria, México, 2004.

- **[19] Savage** Jesús, Obstacle Avoidance Behaviors for Small Mobile Robots Implemented in FPGAs, presented in 2012 ACM/SIGDA 20th International Symposium on FPGAs.

- **[20]Schildt**, Herbert, **C++ The Complete Reference**, Osborne McGraw-Hill,USA 1998, third edition.

- **[21]Scott**, Bob. "The Illusion of Intelligence". En Rabin, Steve. *AI Game Programming Wisdom*. Charles River Media. pp. 16–20, 2002.

- **[22]Stroustrup**, Bjarne, *Why C++ is not just an Object-Oriented Programming Language*, AT&T Bell, Laboratories Murray Hill, USA 1995.

- **[23]The Core Objects**, Ogre Manual

  url: http://www.ogre3d.org/docs/manual/manual_4.html#SEC4,

  Último acceso: viernes 25 de Noviembre 2011.

- **[24] TinyXML** Project Page. http://sourceforge.net/projects/tinyxml/,
  Último acceso: viernes 25 de Noviembre 2011.

- **[25] Wahde**, Mattias, An Introduction to Adaptive Algorithms and Intelligent Machines. Chalmers University of Technology, Goteborg, Sweden, 2002.

- **[26] Wilson**, Kyle, *Why C++?,Game Architect*, Game architect, USA 2006,
  url: http://www.gamearchitect.net/Articles/WhyC++.html ,
  Último acceso: viernes 25 de Noviembre 2011.

- **[27]*Windows** API*,
  url: http://msdn.microsoft.com/en-us/library/cc433218.aspx ,
  Último acceso: viernes 25 de Noviembre 2011.