# Universidad de las Ciencias Informáticas facultad 5


## Título: Sistema de Inteligencia Artificial para el juego simulador de indicios aduaneros.

Trabajo de Diploma para optar por el título de

Ingeniero en ciencias Informáticas

**Autor: Dismel Echevarria Villafranca**

**Tutor:** Jaime Gonzalez Campintruz

**Co-tutor:** Lien Murgercia

"junio del 2009"

## DATOS DEL CONTACTO

Tutor: Ing. Jaime González Campistruz.
Institución: Universidad de las Ciencias Informáticas (UCI), La Habana, Cuba.
e-mail: jgonzalezc@uci.cu.

Co-Tutora: Ing. Lien Muguercia Torres.
Institución: Universidad de las Ciencias Informáticas, La Habana, Cuba.
e-mail: lmuguercia@uci.cu.

# AGRADECIMIENTOS

*A todas aquellas personas que influyeron en mi proceso educativo.*

*A mi tutor Jaime y a Lien.*

*A Liane por su ayuda.*

*A Divo por su apoyo.*

*A mis amigos Lizardo y Felipe que siempre estuvieron ahí.*

*A mis amigos de la Facultad 10 y 5; aprendí mucho de cada uno de ellos.*

*A mi familia, ya que por ellos estoy aquí.*

## DEDICATORIA

*...A mis padres por su ejemplo. Y a todas las personas que creyeron en mí.*

# RESUMEN

Uno de los pilares en el desarrollo de juegos electrónicos es la Inteligencia Artificial (IA). Con la ayuda de IA se aumenta el grado de realismo de los juegos, lo que contribuye en gran medida al éxito de estos.
Existen varios algoritmos y técnicas que se utilizan para el desarrollo de la IA, el hecho de escoger una u otra viene dado por la aplicación o el objetivo que vaya a tener el juego que se está desarrollando. El objetivo principal de esta investigación es desarrollar un sistema de IA para el juego simulador de indicios aduaneros; que intenta simular el flujo de pasajeros a través del Aeropuerto Nacional. Por lo que el sistema de IA pretende dotar a los *NPCs* (caracteres no personales por sus siglas en ingles) o caracteres del juego; con movimiento autónomo que necesitan estos para desplazarse por el entorno del juego. Para lograr este movimiento, a la hora del implementar el sistema se escogieron como algoritmos de movimiento los *Steering Behaviors* y como se pretende simular el movimiento de multitudes de personas también se hace uso de los comportamientos grupales.

Después de explicar brevemente el desarrollo de la IA en los juegos electrónicos a través de los años y algunas de las técnicas que utiliza la IA para proporcionar el realismo necesario para los juegos; se expone la propuesta para dar solución al objetivo planteado. Más adelante se muestra el diseño y la implementación de las clases que conforman el sistema de IA para el juego simulador de indicios aduaneros.

PALABRAS CLAVE: Inteligencia Artificial, movimiento autónomo, NPCs, Steering Behaviors, comportamientos grupales.

TABLA DE CONTENIDOS

**AGRADECIMIENTOS** I

**DEDICATORIA** II

**RESUMEN** III

**INTRODUCCIÓN** 1

**CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA** 3

**INTRODUCCIÓN.** 3

**1.1 ¿QUÉ ES LA INTELIGENCIA ARTIFICIAL?** 3

**1.2 EVOLUCIÓN DE LA INTELIGENCIA ARTIFICIAL EN LOS JUEGOS ELECTRÓNICOS.** 4

**1.3 TÉCNICAS DE INTELIGENCIA ARTIFICIAL EN JUEGOS.** 8
1.3.1 Máquina de Estados Finitos. 9
1.3.2 Búsqueda de Caminos. 11
1.3.3 Movimiento de Agentes Autónomos. 11
1.3.3.1 Steering Behaviors. 13
1.3.3.2 Comportamientos Grupales. 16
1.3.4 Redes Neuronales. 18

**1.4 SDKs DE INTELIGENCIA ARTIFICIAL EN JUEGOS.** 19

**CAPÍTULO 2: SOLUCIÓN PROPUESTA** 21

**INTRODUCCIÓN.** 21

**2.1 DESCRIPCIÓN DE LA PROPUESTA DE SOLUCIÓN.** 21

**2.2 METODOLOGÍAS Y HERRAMIENTAS PARA LLEVAR A CABO LA SOLUCIÓN.** 22

**2.2.1 Visual Studio 2005.** 22
2.2.2 UML. 22
2.2.3 RUP. 23
2.2.4 Visual Paradigm. 24
2.2.5 C++. 24

**CAPÍTULO 3: CARACTERÍSTICAS DEL SISTEMA** 26
3.2.1 Descripción de las clases conceptuales. 27

**3.3 REQUERIMIENTOS.** 28

**3.3.1 Captura de requisitos.** 28

**3.3.2 Requisitos funcionales.** 28

**3.3.3 Requisitos no funcionales.** 28

**3.4 MODELO DE CASOS DE USOS.** 29
3.4.1 Descripción de los actores del sistema. 29
3.4.2 Diagrama de casos de usos. 30
3.4.3 Descripción de los casos de usos en formato expandido. 30

**CAPÍTULO 4: DISEÑO DEL SISTEMA** 38

**INTRODUCCIÓN** 38

**4.1 DIAGRAMA DE CLASES DEL DISEÑO.** 39

**4.2 DIAGRAMA DE INTERACCIÓN POR CASOS DE USOS.** 41

**4.3 DESCRIPCIÓN DE LAS CLASES DEL DISEÑO.** 45

**CAPÍTULO 5: IMPLEMENTACIÓN** 54

**INTRODUCCIÓN** 54

**5.1 ESTÁNDARES DE CODIFICACIÓN.** 54

**5.2 DIAGRAMA DE DESPLIEGUE.** 58

**5.3 DIAGRAMA DE COMPONENTES.** 59

**CONCLUSIONES** 62

**RECOMENDACIONES** 63

**BIBLIOGRAFÍA** 64

**GLOSARIO** 65

**LISTA DE FIGURAS Y TABLAS** **66**

# INTRODUCCIÓN

El creciente desarrollo de las tecnologías ha permitido un incremento en el los sistemas de realidad virtual. Hoy en día surgen juegos y simuladores con mayores prestaciones para los usuarios, mayores niveles en la calidad visual y de realismo en la interacción con el mundo virtual.

Sin embargo, este incremento de calidad en las imágenes no ha venido acompañado de mejoras en factores de calidad no-visual, es decir, el tipo de estímulos que también aportan realismo al usuario y no depende directamente de la calidad de la imagen. Como ejemplo, se ve que la simulación de una bandada de pájaros en un entorno virtual adecuado constituye un elemento de este tipo, ya que el realismo que aporta a la escena no depende tanto de la calidad visual de los pájaros sino de su comportamiento. En entornos simulados donde la presencia de humanos virtuales se hace necesaria (simuladores de conducción o de tiro, en la educación, la formación, en situaciones de emergencia, en video-juegos,... etc.), el problema pasa por crear las arquitecturas inteligentes para los mismos, de modo que en el transcurso de las simulaciones, estos humanos virtuales sean capaces de aportar, a través de sus comportamientos autónomos, el realismo no visual previamente comentado.

Para crear un personaje virtual, y con un aspecto físico realista, es necesario también darle su propia personalidad capaz de expresar emociones e interactuar con el medio ambiente que los rodea.

Actualmente en la facultad 5 de la Universidad de Ciencias Informáticas se trabaja en el desarrollo de un juego para la simulación del ambiente del Aeropuerto Nacional. Este juego de simulación de indicios aduaneros carece de un sistema que permita el comportamiento autónomo[1] de actores virtuales[2] en el entorno. Por lo que surge el siguiente **problema científico**: ¿Cómo elaborar un sistema de inteligencia artificial que permita el comportamiento autónomo de actores virtuales?

[1] Se entiende por comportamiento autónomo, la capacidad que tienen aquellos agentes o actores virtuales de adaptarse, interactuar o reaccionar a los estímulos del entorno donde se desenvuelven.

Para el desarrollo de esta investigación se define como **objeto de estudio** el comportamiento autónomo de actores en entornos de realidad virtual. El **campo de acción** se basa en el uso de técnicas tales como los comportamientos grupales.

El **objetivo** principal de esta investigación es crear un sistema de inteligencia artificial para el comportamiento autónomo de actores en el entorno virtual perteneciente al juego simulador de indicios aduaneros.

Como apoyo al desarrollo del objetivo se han definido las siguientes **tareas de investigación:**

- ✓ Estudio de las tendencias actuales en la creación de juegos de simulación, para la identificación de las técnicas usadas en la elaboración de este tipo de juegos.
- ✓ Caracterización de las técnicas de inteligencia artificial que se usan en el desarrollo de video-juegos y simuladores para la identificación (obtención) de las que se usaran en el desarrollo del sistema.
- ✓ Análisis de la arquitectura del juego simulador de indicios aduaneros para acoplar el sistema de inteligencia artificial.
- ✓ Diseño de un modelo de clases que responda a las exigencias del sistema que se está desarrollando para lograr su acople al juego en desarrollo.
- ✓ Implementación de las clases que forman parte del modelo del sistema.
- ✓ Desarrollo de una documentación que recoja todo el proceso de investigación y desarrollo del sistema.

[2] Se entiende por actores virtuales a aquellos que se desarrollan e interactúan en un entorno virtual.

# CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

## INTRODUCCIÓN.

La inteligencia artificial (IA), en el de cursar de los años se ha tornado cada vez más importante en el desarrollo de los juegos electrónicos. Con ella los caracteres presentes en el juego se vuelven más fuertes, rápidos, mejoran la puntería e influyen en el éxito futuro del juego.

Este capítulo comienza haciendo referencia a conceptos de IA, seguido de una breve descripción sobre la evolución de la IA en los juegos electrónicos en sus diferentes géneros. También se hace una descripción de algunas de las técnicas que se utilizan en su implementación. Por últimos se mencionan algunos SDKs disponibles que facilitan la integración de la IA con el juego.

## 1.1 ¿QUÉ ES LA INTELIGENCIA ARTIFICIAL?

La IA se basa en hacer que las computadoras sean capaces de pensar y ejecutar tareas iguales a las que los humanos y animales son capaces de realizar.

Según Russell y Norvig en su libro *Artificial Intelligent: A Modern Approach* [Russell95] dicen que la IA es la creación de programas de computadoras que emulen la manera de actuar y pensar de los humanos, más bien de actuar y de pensar racionalmente.

Hoy en día se ha logrado programar computadoras con capacidades inalcanzables para los humanos, tales como: resolver problemas matemáticos complejos en muy poco tiempo, buscar grandes cantidades de información en cuestiones de segundo y otras muchas capacidades. Las computadoras pueden incluso jugar los tradicionales juegos de mesa mucho mejor que los humanos más talentosos. Muchos de estos problemas en un primer momento fueron considerados problemas concernientes a la IA, pero se han ido resolviendo de muchas maneras por lo que han dejado de formar parte del dominio de la IA.

Pero todavía hay muchas cosas que las computadoras no son capaces de hacer y las cuales son consideradas triviales por todos los seres humanos, tales como: reconocer rostros familiares, hablar en el idioma natal, decidir qué hacer en un momento dado, y ser creativos. Estos temas si están dentro del dominio de la IA: tratar de determinar qué tipos de algoritmos son necesarios para visualizar estas propiedades.

En los juegos electrónicos la IA se expresa en aquel código que hace que la computadora controle a los oponentes o elementos cooperativos y hace que parezca que toman decisiones inteligentes cuando el juego les pone múltiples opciones a escoger en una situación determinada, y además resulta un comportamiento que es relevante, efectivo y útil.

## 1.2 EVOLUCIÓN DE LA INTELIGENCIA ARTIFICIAL EN LOS JUEGOS ELECTRÓNICOS.

La evolución de la IA en los juegos electrónicos a tenido un enorme crecimiento en los últimos tiempos [Salvi., 2006].

En los primeros tiempos los juegos que contenían IA empleaban una lógica que podía ser fácilmente identificada por el jugador. Esto debido a que hacían uso de reglas simples, combinada con secuencia de acciones escogidas de forma aleatoria para representar sus características, así como patrones de movimiento que tornaban sus acciones bastante previsibles. No obstante, eran propuestas factibles para su tiempo si se ve desde el punto de vista de que cualquier juego que permita la confrontación del jugador con la máquina tiene algún nivel de inteligencia implementada, a menos que esta sea bien primitiva.

La historia de la IA en los juegos electrónicos tiene su inicio primeramente con los juegos de tipo *arcade*, que tuvieron gran éxito en las épocas del 70 al 80. En esta época entre los que más se destacan están: Pong, Pacman y Space Invaders.

Pacman [Midway Games West, Inc., 1979] fue uno de los primeros juegos que tenía en su programación una IA incipiente.

Pacman definía los caracteres enemigos de tal forma que parecían conspirar en contra del jugador, moviéndose alrededor del mundo y tornando la supervivencia más difícil en cada nivel de juego.

Este juego estaba implementado con una técnica de IA muy sencilla: una máquina de estados finitos (ver epígrafe 1.3.1). Cada uno de los monstruos que habían en el juego (más tarde se llamaron fantasmas) podían huir o perseguir al jugador. Es decir tenían dos estados implementados, huir y perseguir. La transición del estado perseguir al estado huir estaba condicionado por si el jugador se comía una de las pastillas de *poder* presentes en el juego y del estado huir a el estado perseguir estaba dado por la finalización del tiempo efecto de la pastilla de *poder*. Esta era una solución muy simple pero funcional para el momento en que fue creado este juego.

La IA en los juegos no cambio mucho hasta mediados de los años 1990. Hasta entonces la mayoría de los personajes controlados por la computadora eran tan inteligentes como los fantasmas del Pacman.

Se puede ver en un clásico como es el juego **Golden Axe** [SEGA Entertainment, Inc., 1987] a pesar de ser creado varios años después del Pacman, en este momento solo tenía pocos avances en comparación con el anterior.

A mediados de los años 1990 la IA en los juegos comienza a ser un parámetro en las ventas de estos. El juego **Beneath a Steel Sky** [Revolution Software Ltd., 1994] en su sistema de IA permitía a los caracteres caminar a través del entorno del juego y llevar una rutina de comportamiento, hecho este que constituyo un adelanto.

**Goldeneye 007** [Rare Ltd., 1997] probablemente fue el que mostró a los jugadores lo que la IA podía hacer para mejorar los juegos electrónicos. Los caracteres seguían dependiendo de un pequeño número de estados bien definidos. Además Goldeneye adicionó un sistema de simulación de sentidos: el carácter podía ver a sus compañeros y podía además notar si ellos habían sido asesinados. La simulación de sentido fue el tema del momento, **Thief: The Dark Project** [Looking Glass Studio, Inc., 1998] y **Metal Gear Solid** [Konami Corporation, 1998] basaron su diseño de juego en esta técnica.

También es en esta etapa cuando los *RTS games* (juegos de estrategia en tiempo real por sus siglas en ingles) comienzan a despegar. **Warcraft** [Blizzard Entertainment, 1994], fue uno de los primeros en disponer de una funcionalidad como la búsqueda de caminos o *pathfinding* (ver epígrafe 1.3.2). **Warhame: Dark Omen** [Mindscape, 1998] implementa un modelo emocional para caracteres tipo soldado en una simulación de un campo de batalla, también logran tener una formación moviéndose en la acción del juego.

Más adelante se siguieron implementando juegos donde la IA era el centro de estos.

**Creatures** [Cyberlige Technology Ltd., 1997] fue hecho en 1997, pero junto a juegos como **The Sims** [Maxis Software Inc., 2000] y **Black and White** [Lionhead Studios Ltd., 2001] llevan la delantera en el área. Creatures aún tiene uno de los sistemas de IA más complejos que se pueden ver en un juego, con una red neuronal para las criaturas del juego. Black&White es otro juego que se destaca bastante por su inteligencia. En este, el agente tiene un comportamiento según la enseñanza dada por el jugador, a través de la realización de tareas. La IA utilizada en este juego llegó a ser tan conocida que acabó creando un nuevo paradigma en el área: IA de tipo Black&White.

Los juegos de acción en primera persona son uno de los géneros más interesantes para extender su aplicación en disimiles simulaciones. Dentro de los juegos de acción en primera persona, uno de los más destacados es **Half-Life** [Sierra Entertainment., 1998], que proporciona enemigos con buenas cualidades tácticas para la ejecución de sus deberes. Este hace uso de scripts para el desarrollo de su IA.

Los juegos de deporte y los juegos de carreras en particular tienen sus propios desafíos en cuanto a IA; por otro lado se destacan los *RPG* (juegos de rol por sus siglas en ingles) con una compleja interacción entre caracteres.

Hoy en día hay una diversidad de juegos que agregan IA. Muchos de los géneros de estos juegos todavía continúan usando la IA sencilla de los primeros años cuando surgió el Pacman porque esto es lo que ellos necesitan.

Actualmente la IA tiene la meta de aplicar comportamientos y reacciones cada vez más realistas, posibilitando de este modo hacer que los agentes tomen decisiones diferentes y adecuadas para producir un efecto que se ajuste a la realidad. A raíz de esto, han surgido técnicas que se tornan bastante popular como: A-Life (vida artificial por sus siglas en ingles). Con esto se puede modelar el comportamiento de vida de un ser. Un juego que se hizo famoso por utilizar esta técnica es The Sims, que simula la vida de humanos a través de una interacción social simulada entre personajes virtuales, desarrollando así una personalidad en los personajes del juego.

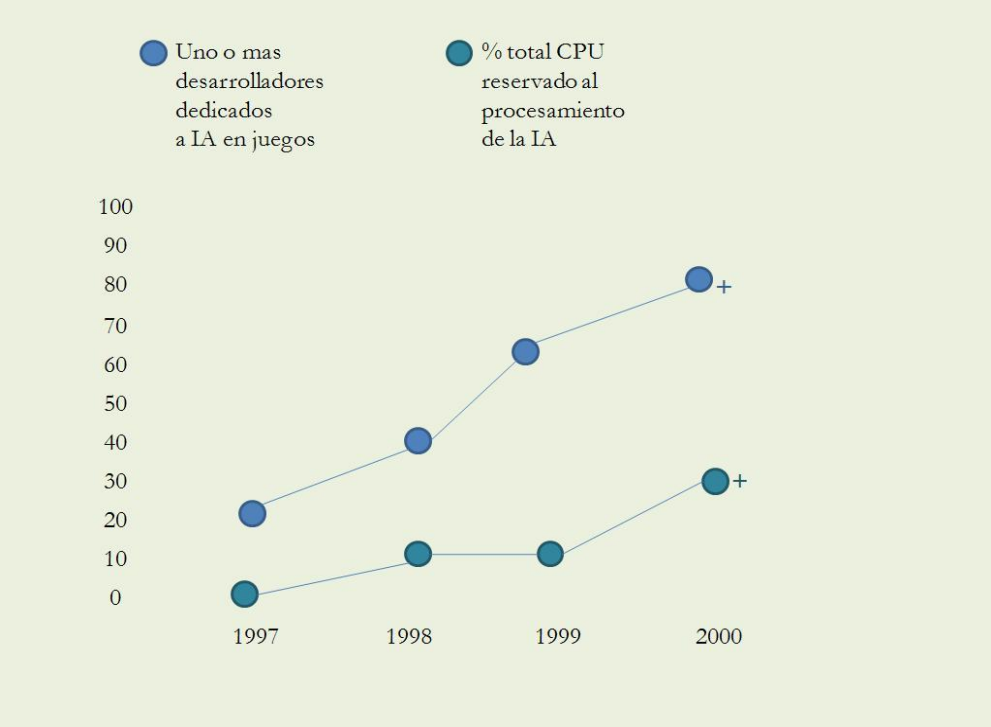


*Figura 1:* Importancia de la IA en los video juegos. [Salvi. 2006]

En la figura anterior se puede notar el crecimiento y el potencial del tema en cuestión. Esto es debido a la demanda cada vez mayor y objetivos más desafiantes impuesto por los jugadores. Por consecuencia también ha aumentado el tiempo de procesamiento de las rutinas dedicadas a la IA.

La IA en la mayoría de los juegos modernos necesitan tres factores básicos: la habilidad para mover caracteres, la habilidad para tomar decisiones hacia donde moverse, y la habilidad para pensar táctica y estratégicamente.

Aunque se vaya a usar un sistema de IA basado en estados (todavía son muy usados en muchos lugares) o incluso una gran gama de técnica; todos deben cumplir los mismos requerimientos.

## 1.3 TÉCNICAS DE INTELIGENCIA ARTIFICIAL EN JUEGOS.

En el desarrollo de los video-juegos, la IA es implementada como un módulo o un sistema a parte del sistema del juego, debido a que la creación de este organiza la estructura del juego además de proveer la independencia de las partes que la componen. De esta forma es típica la creación de módulos dedicados a la renderización, las redes y en este caso la IA.

El módulo de IA es responsable por la construcción del comportamiento de los agentes inteligentes que participan en el juego como adversarios o aliados del jugador; de manera independiente con respecto a los demás módulos que componen el sistema de juego [Salvi., 2006]. De acuerdo con las palabras del autor, la IA puede controlar tanto los adversarios como a los compañeros del jugador. Un ejemplo de esto se puede ver en el juego **Diablo II** - Expansión, donde el jugador puede contar con un mercenario, que tiene la función exclusiva de colaborar para que el jugador logre alcanzar los objetivos del juego y evolucionar junto al jugador. Además de esto, el desarrollo de un módulo, hace que la IA trabaje de forma independiente, sin la necesidad de estar directamente ligada a otras partes de la aplicación.

De acuerdo a las investigaciones de **Woodcock** [1999], una de las técnicas más populares y utilizadas por los desarrolladores de IA son las máquinas de estados finitos [FSM] y las máquinas de estados difusos [FuSM]. **Tozour** [2002] indica que el algoritmo de búsqueda A* es la principal técnica de búsqueda para efectuar la navegación, siendo utilizada en todos los géneros de juego. Para **Woodcock** [1999] y **Champandrad** [2003], los *scripts* en la generación de la IA son una técnica que proporciona mucha jugabilidad a los juegos y tiene gran aceptación entre los jugadores ya que pueden personalizar ciertas

características de los personajes del juego. En los próximos epígrafes se explicarán con un poco más de detalle algunas de estas técnicas que son utilizadas en el desarrollo de la IA en los juegos electrónicos.

### 1.3.1 Máquina de Estados Finitos.

A veces, un carácter en el juego actuará de un limitado número de formas. Puede estar haciendo la misma acción hasta que ocurra un evento o sea influenciado a cambiar. En la mayoría de los casos una técnica bastante usada para gestionar este tipo de comportamiento son las máquinas de estados finitos.

Las Máquinas de Estados Finitos o *FSM* como comúnmente son conocidas, son probablemente una de las técnicas más antiguas, más fiables y más usadas por la mayoría de los juegos que incluyen IA; y es probable que permanezcan así por mucho más tiempo; ya que son rápidas y fáciles de implementar, además que el uso de CPU que necesitan estas es muy bajo.

A pesar de ser una técnica simple, puede proporcionar buenos resultados, además se puede integrar con otras técnicas de la IA, tales como redes neuronales y/o lógica difusa. La utilización de máquinas de estados finitos también permite crear elementos con comportamientos relativamente complejos, como los mostrados en el juego **Unreal Tournament** [EPIC GAMES, 1999], en el cual los enemigos tomaban una serie de decisiones basadas en las acciones del jugador.

Para desarrollar una *FSM*, cada situación debe ser representada como un estado. Un ejemplo de esto se ve en la siguiente figura, en la cual el agente representado puede ejecutar las siguientes acciones: estar A Salvo, Pelear o Huir.

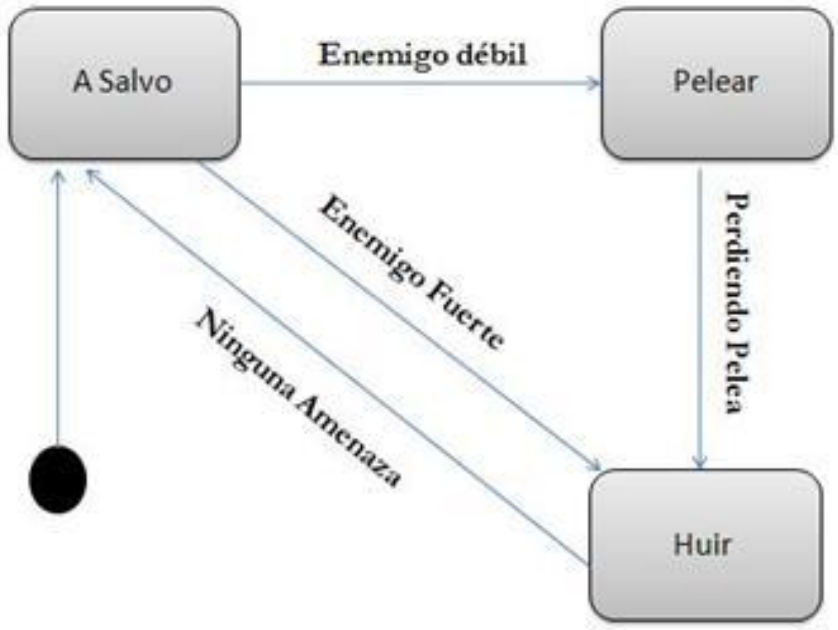


*Figura 2:* Ejemplo de una máquina de estado simple.

En la figura anterior se representa un agente inicialmente en estado A Salvo (estado1). En caso que en este estado se encuentre con un enemigo más débil entonces pasaría al estado Pelear (estado 2), en caso contrario que el enemigo sea más fuerte pasaría al estado Huir (estado 3). Se puede ver que para cada una de las situaciones hay una condición de transición (pueden ser más de una), una vez que se cumplen esta condición entonces se pasa al estado correspondiente.

Conceptualmente hablando, **Buckland** [Buckland., 2005] plantea que:

*Una maquina de estados finitos es un dispositivo, o modelo de un dispositivo, que tiene un número finito de estados, que puede estar en uno de ellos en un momento dado. La transición entre un estado y otro se puede dar mediante entradas internas o simplemente puede ser causado por una acción que se produzca externamente. Una máquina de estados finitos puede estar en un solo estado en un momento dado.*

La idea inicial de una *FSM* es descomponer el comportamiento de un determinado agente en varias partes o estados. Se da el caso en que la complejidad de las *FSM* aumenta y su utilización se torna

impracticable. Para resolver este problema se sugiere el uso de máquinas de estados jerárquicas (*HFSM*), en la cual cada estado puede ser una nueva *FSM*.

### 1.3.2 Búsqueda de Caminos.

Los mecanismos de búsqueda de caminos son una de las técnicas de IA más utilizadas en los juegos. El movimiento de una entidad de un lugar a otro a través de un camino razonable y desviando objetos es un requisito fundamental para demostrar alguna señal de inteligencia por los agentes en los juegos.

Para realizar la búsqueda de caminos pueden ser usados diversos algoritmos. Uno de los más conocidos y más utilizados es el A*, en el que la búsqueda se hace a través del cálculo de costos estimados para ir de un nodo origen a un nodo destino. Generalmente, el costo de cada nodo es asociado a una cuadricula del grid[3] regular utilizado en el juego, en el que cada celda representa el nodo de un grafo. Siendo así posible representar los caminos por donde el personaje puede caminar dentro del ambiente virtual.

Como el uso de la búsqueda puede consumir mucho tiempo de procesador es posible controlar ese problema a través de caminos pre-calculados. Este método es conocido como *waypoitns*. Estos son localizaciones en el mundo virtual que ayudan en el desplazamiento de un lugar (nodo actual) para otro (nodo destino) a través de caminos pre-calculados o métodos de búsqueda de bajo costo.

### 1.3.3 Movimiento de Agentes Autónomos.

Uno de los requerimientos fundamentales de la IA es el movimiento de caracteres dentro del entorno de juego. Incluso desde los primeros momentos en que la IA comenzó a mover caracteres(Los fantasmas en Pacman, por ejemplo.) ha sido el movimiento uno de los algoritmos que no se ha removido de los juegos hasta hoy en día.

Antes de ver como se produce el movimiento de los caracteres o agentes, se debe tener conocimiento de que es un agente autónomo.

Muchos autores dan sus conceptos referentes a lo que entienden por *agente autónomo* pero quizás esta sea la definición más general:

*Un agente autónomo es un sistema situado en un entorno; siente este entorno y actúa sobre él, todo el tiempo, siguiendo sus propios objetivos y afectando lo que siente sus futuras acciones.* [Buckland., 2005]

Es decir se entiende por agente autónomo a aquel agente que posee cierto grado de autonomía, en el caso que le corresponde a este epígrafe, cierto grado de movimiento autónomo. Si un agente autónomo se encuentra frente a determinada situación, tal como encontrar un obstáculo en su camino, él tendrá la habilidad de responder y ajustar su movimiento acorde a la situación. En la mayoría de los juegos estos agentes autónomos son llamados *NPC* (caracteres no personales por sus siglas en ingles).

El movimiento de un agente autónomo se puede dividir en tres partes o capas:

*Action Selection* (Acción Selección): Esta es la parte del comportamiento del agente responsable de seleccionar los objetivos y decidir qué plan seguir. Esta es la parte donde se dice “ve aquí” y “has A, B y después C”.

*Steering* (Dirección): Esta es la parte responsable de calcular la trayectoria deseada para satisfacer los objetivos y planes planteados por la capa de Acción Selección. Los *steering behaviors* (comportamientos de dirección (ver epígrafe 1.3.3.1)) son implementados en esta capa. Estos producen una fuerza de dirección que describe hacia donde un agente debe moverse y cuán rápido debe viajar para llevar ahí.

*Locomotion* (Locomoción): Esta es la capa base, representa los aspectos mecánicos en el movimiento de un agente. Es el cómo se debe viajar desde A hasta B. Por ejemplo, si se tiene implementado el mecanismo de movimiento de un camello, un tanque y un pez y se le da la orden de viajar al norte, ellos usaran diferentes procesos mecánicos para crear movimiento pero el intento de moverse al norte será el mismo. Separando esta capa de la capa de Dirección, es posible utilizar, con ligeras modificaciones, el mismo *steering behavior* para diferentes tipos de locomoción.

[3] Un grid es una teselación de polígonos. [Lógica de un video juego]

El modelo para la locomoción que se tiene en cuenta para este trabajo está basado en la idealización de un vehículo simple [Reynolds., 1999]. Este vehículo posee un centro de masas, este está definido por propiedades tales como la posición, la masa, la velocidad. Esta velocidad se modifica por las fuerzas aplicadas. Como es un vehículo estas fuerzas generalmente son auto aplicadas y por lo tanto limitadas, para este caso se toma en cuenta como parámetro la fuerza máxima. Para alcanzar una simulación más realista en el modelo del vehículo se incluye otro parámetro, la velocidad máxima. Por último el modelo incluye un parámetro referente a la orientación en el entorno.

La posición y la velocidad del vehículo representan valores vectoriales, la orientación puede estar representada por dos o tres vectores dependiendo del entorno (2D o 3D) o simplemente por un escalar que represente el ángulo de orientación.

El mecanismo de movimiento del vehículo está basado en cálculos físicos. En cada actualización de la simulación los comportamientos de dirección determinan la fuerza de dirección (esta está limitada por la fuerza máxima) que se debe aplicar al centro de masas del vehículo. Se produce una aceleración que es igual a la fuerza de dirección dividida entre la masa del vehículo. Con esta aceleración se obtiene la nueva velocidad que está limitada por la velocidad máxima y finalmente con esta velocidad se obtiene la nueva posición.

#### 1.3.3.1 Steering Behaviors.

Los comportamientos de dirección o *steering behaviors* por su terminología en ingles son el desarrollo lógico del modelo creado por **Craig Reynolds** en 1986 llamado "*boids*"[Reynolds, 1986]. El modelo "*boids*" está basado en la creación de movimiento simulado, tales como bandadas de pájaros, peces o multitudes de personas.

Reynolds fue capaz de crear una simulación bastante realista del comportamiento natural de bandada de pájaros y peces, mediante la combinación de varios comportamientos. El primer ejemplo del uso de este modelo se pudo ver en el cortometraje “Stanley and Stella in: Breaking the Ice”.

*Figura 3:* Escena tomada de *Stanley and Stella in: Breaking the Ice.*

Filme este creado por la *Symbolics Graphics Division* en cooperación con *Whitney / Demos Production* en el año 1987. Fue mostrado por primera vez en el evento *Siggraph* 1987.

Los *steering behaviors* tienen gran aceptación por los desarrolladores de juegos de PC y consola. En algunos géneros (como en los juegos de conducción) son dominantes; en otros géneros su uso está comenzando a ser tomados en serio. Estos generalmente son independientes de la forma de moverse que tenga cada carácter o agente en particular (la forma de moverse de un ave no es la misma que la de un automóvil).

Hay un gran número de diferentes *steering behaviors*, incluso con nombres muy similares que pueden llegar a confundir. Para su estudio se debe ver primero la estructura básica que comparten muchos de ellos antes de ver las variaciones.

Por lo general, la mayoría de los *steering behaviors* tienen una estructura similar. Ellos toman como entrada las propiedades cinemática del carácter que se está moviendo tales como posición, velocidad y otros datos necesarios para la locomoción e información sobre los objetivos que debe alcanzar. La

información sobre los objetivos depende de la aplicación. Un comportamiento para la evasión de obstáculos toma como entrada la representación de las colisiones en el entorno. También es posible especificar un camino como objetivo para un comportamiento de seguimiento de camino.

A la hora de discutir un *steering behavior* en particular se asume que la locomoción esta implementada, en este caso mediante el modelo de un vehículo simple (ver epígrafe 1.3.3).

Comportamientos Seguir y Evadir.

Los comportamientos de seguir (*seek*) y evadir (*flee*) son implementados complementariamente. La lógica tras de ellos es la misma para los dos. La única diferencia es el resultado de las fuerzas de dirección. El comportamiento *seek* es usado para dirigir el agente hacia una dirección o un objetivo especifico. Un ejemplo puede ser un pez nadando hacia su comida. El objetivo puede ser un punto en el espacio u otro agente en movimiento. El comportamiento *flee* es usado para simular un simple comportamiento de evasión. Como en el caso del *seek*, el objetivo de la evasión puede ser un punto en el espacio u otro agente en movimiento.

Comportamiento Arribar.

El comportamiento arribar (*arrive*) es una extensión del seek. Igual que el *seek*, este comportamiento es usado para dirigir a un agente a un objetivo especificado. La diferencia es en la forma en la que se llega al objetivo. El seek llega al objetivo con toda su velocidad, desviándose del objetivo y virando hacia atrás una y otra vez, como oscilando sobre el objetivo una vez que llega. Esto normalmente no es el resultado que se espera. Por lo que el comportamiento arribar lo que hace es que el agente en movimiento vaya disminuyendo la velocidad mientras se va acercando a su objetivo y deteniéndose una vez llega a él.

Comportamiento Errático.

Muchas veces este comportamiento resulta de mucha ayuda al crear el comportamiento de los agentes. Está diseñado para producir una fuerza que da la impresión de un caminar aleatorio a través del entorno.

El comportamiento errático (*wander*) les permite a los agentes moverse con un comportamiento autónomo y sin la especificación de ningún objetivo en el entorno. Es comparable con el comportamiento que tienen las hormigas cuando están buscando comida.

Comportamiento Seguir Camino

Este comportamiento seguir camino (*path following*) le permite al carácter dirigirse a través de un camino predeterminado, tal como una carretera, un pasillo o un túnel.

#### 1.3.3.2 Comportamientos Grupales.

Muchas veces en los video juegos, los agentes necesitan moverse en grupos de manera cohesiva para logar sus objetivos. Los comportamientos grupales les dan a los agentes un comportamiento con la capacidad de formar grupos con otros agentes. Un ejemplo en la naturaleza puede ser una multitud de animales o personas.

Los comportamientos grupales son *steering behaviors* que toman en consideración algunos o todos los agentes en el entorno de juego. El movimiento colectivo o movimiento grupal conocido como *flocking* en su terminología en ingles, es la combinación de tres comportamientos grupales - **cohesión**, **separación**, **alineación** - todos trabajando juntos.

Para determinar la fuerza de dirección para un comportamiento grupal, el agente debe de considerar a todos los otros agentes dentro de un área circular predefinida con anterioridad conocida como radio de vecinos de la cual el centro será el agente con este comportamiento.

#### Separación.

Este comportamiento crea una fuerza que pone al agente fuera de la región de sus vecinos. Cuando esto se aplica a un número de vehículos, estos se abren, tratando de maximizar la distancia entre ellos.

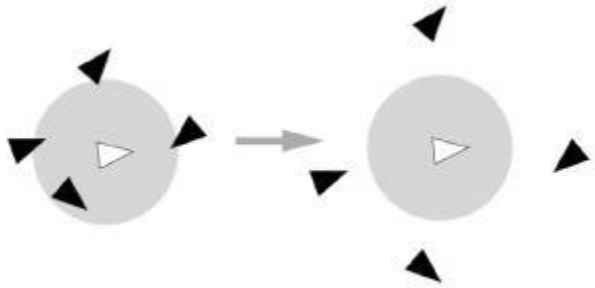

*Figura 4:* Ejemplo del comportamiento separación.

Alineación.

Este comportamiento le da al agente de mantener su alineación en correspondencia con la alineación de los agentes vecinos.

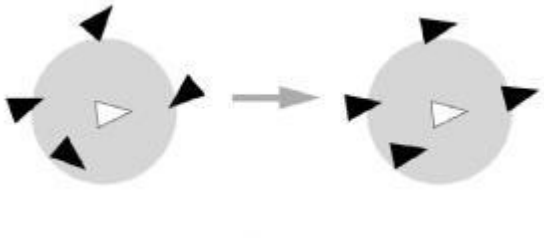

*Figura 5:* Ejemplo del comportamiento alineación.

Cohesión.

La cohesión produce una fuerza de dirección que mueve al vehículo hacia el centro de masa de sus vecinos. Se usa esta fuerza para mantener a un grupo de agentes unidos.

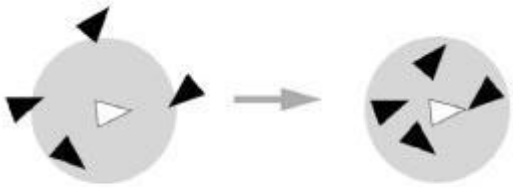

*Figura 6:* Ejemplo del comportamiento cohesión.

### 1.3.4 Redes Neuronales.

Las redes neuronales artificiales *ANN* (Artificial Neural Network por sus siglas en ingles) [MILLINGTON., 2006] son un tipo de técnicas de aprendizaje, que se basa en la forma en la que las neuronas biológicas trabajan en el cerebro de los seres humanos. Las redes neuronales pueden ser usadas en varias aplicaciones como en la clasificación de patrones; en las cuales una entrada puede ser descrita como un problema que está presente en la red, los cuales son clasificados en las salidas. Antes de esta clasificación, la red neuronal debe ser entrenada mediante la presentación de un conjunto de patrones para los cuales se conoce la clasificación correcta, y poniendo en orden la configuración interna de la red para permitirle el reconocimiento de estos patrones.

Hay muchos modelos de redes neuronales cada uno de estos con diferentes funcionalidades y aplicaciones. En el mundo de los video juegos uno de los modelos más utilizados es el perceptrón multicapa [MILLINGTON., 2006], junto a uno de los muy utilizados algoritmos como es el *backpropagation*.

Más conceptualmente una red neuronal consiste en un número relativo de nodos, los cuales ejecutan el mismo algoritmo. Estos nodos individualmente constituyen una neurona artificial; esta neurona se comunica con otras neuronas artificiales en la red. Están conectadas basadas en un patrón que identifica al tipo de red neuronal.

## 1.4 SDKs DE INTELIGENCIA ARTIFICIAL EN JUEGOS.

Los sistemas de desarrollo integrado o s*ystem development kits* (SDKs por sus siglas en ingles), de IA son básicamente una capa de software(o un conjunto de componentes) que proveen servicios al motor de juego para la realización de funciones de IA. No obstante son considerados como un middlewar[4] entre el motor de juego y la IA [Eric Dybsand., 2003]. Ellos se encargan del proceso de producir el comportamiento deseado o la toma de decisión por parte de los agentes presentes en el juego.

De entre los SDKs comerciales más conocidos se puede citar a **DirectIA** [Mathematiques Appliquees, 1999]. DirectIA es, en esencia, un SDK que permite a los desarrolladores crear agentes autónomos (o grupos de agentes) que tienen la capacidad de aprender, anticiparse, y seleccionar sus acciones. Los agentes controlados por este SDK pueden adaptarse al entorno y aprender en tiempo real. Tiene una serie de módulos de apoyo tales como comportamientos reactivos, percepción y motivaciones [Woodcock., 2001].

Entre otros se encuentra **Character Studio** [www.discreet.com] este controla a los agentes en multitudes; **IA-Implant** [www.ia-implant] que permite el control de los caracteres del juego mediante un sistema basado en reglas; y **Renderware A.I** (www.renderware.com) que permite la búsqueda de caminos y comportamientos muy útiles para el desarrollo de la IA. Entre los que se pueden considerar como libre u *open source* se encuentra **Game::AI++** (www.iagamedev.com).

A pesar de facilitar la implementación de la IA en juegos, Salvi [Salvi., 2006] afirma que de cualquier forma ningún SDK es plenamente aceptado por la comunidad desarrolladora de juegos, ya que en primer lugar muchas veces restringe el acceso al código y a la mayoría de las funcionalidades del SDK y en segundo lugar traen consigo una inmensidad de características y funcionalidades que no siempre serán utilizadas.

[4] Se entiende como middleware aquel software que sirve de conectividad y ofrece un conjunto de servicios que hace posible el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas.

La clave del éxito de un SDK de IA va a ser el aceptable balance entre la inclusión de muchas funcionalidades así como el hacer que el sistema no sea trivial, y al mismo tiempo no complicar el sistema haciendo su uso restringido.

# CAPÍTULO 2: SOLUCIÓN PROPUESTA

## INTRODUCCIÓN.

En el primer capítulo se da una introducción al concepto de la inteligencia artificial en el desarrollo de videos juegos. También se pretendió dar una pequeña descripción de algunas de las técnicas que son utilizadas para el control del comportamiento de agentes autónomos en los videos juegos. Este capítulo se centra en la descripción del sistema que se va a desarrollar y el fin que se persigue con el desarrollo del mismo. Además se mencionan las metodologías y herramientas que ayudaran en el desarrollo del trabajo.

## 2.1 DESCRIPCIÓN DE LA PROPUESTA DE SOLUCIÓN.

Los conceptos de IA presentados en el capítulo anterior son de extrema importancia para escoger las técnicas y métodos que integraran el sistema propuesto. Sabiendo esto, a continuación se presentan las técnicas y métodos escogidos para la implementación de este sistema, se expone también el porqué de su utilización.

Primeramente se debe tener en cuenta que el juego simulador de indicios aduaneros pretende simular el ambiente del Aeropuerto Nacional; para esto es necesario recrear el flujo de pasajeros por este entorno, además de la interacción de los agentes aduaneros con los pasajeros.

Para darle solución al problema y enfrentar las necesitad de toma de decisión de los agentes que integraran el juego se decide utilizar como técnica para la toma de decisión una máquina de estados para la gestión de los estados de los agentes.

Para la gestión del movimiento autónomo de los agentes por el entorno de juego se decide utilizar como técnica, los *steering behaviors*. Se utilizaran algunos básicos descritos en el capítulo uno junto con la combinación de los comportamientos grupales para lograr que los agentes del juego tenga la capacidad de moverse en colectivo.

Los comportamientos simples que se deben implementar son el *seek* y el *arrive.* Estos comportamientos son los que le van a permitir a un determinado agente dirigirse hacia un lugar determinado y detenerse correctamente una vez llegue al lugar indicado. Otro de los comportamientos simples que se deben implementar es el *pathfollowing* este es el comportamiento que le va a permitir a los agentes del juego dirigirse a través de un camino determinado. Como comportamiento grupal se debe implementar el *separation.* Se deben combinar los comportamientos *pathfollowing* y *separation* para lograr un comportamiento que se conoce como *Crowd Path Following* o seguir camino en multitud. En este caso cada uno de los agentes se dirige a través de un camino predeterminado manteniendo la separación indicada con los agentes vecinos. Este comportamiento se debe observar en el primer nivel del juego donde un grupo de caracteres bebe caminar a través de un pasillo, manteniendo un movimiento coordinado entre los agentes.

## 2.2 METODOLOGÍAS Y HERRAMIENTAS PARA LLEVAR A CABO LA SOLUCIÓN.

### 2.2.1 Visual Studio 2005.

Para el desarrollo del sistema se utilizara como herramienta de desarrollo el Visual Studio 2005, esta es una herramienta orientada a desarrollar aplicaciones profesionales. Es una herramienta estable que se puede integrar con varios lenguajes de programación como C#, Asp.Net y en este caso C++. Además contiene muchas librerías con códigos pre-escritos que ayudan al desarrollo de aplicaciones y posee buena detección y corrección de errores.

### 2.2.2 UML.

Como lenguaje de modelado para el diseño del sistema se utilizara UML (Lenguaje Unificado de Modelado) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad. UML permite visualizar, especificar, construir y documentar un sistema. Ofrece un estándar para describir un modelo de un sistema, incluyendo aspectos conceptuales tales como procesos de negocio y funciones

del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes reutilizables.

Es importante resaltar que UML es utilizado para especificar o para describir métodos o procesos. Se utiliza para definir un sistema, para detallar los artefactos en el sistema y para documentar y construir. En otras palabras, es el lenguaje en el que está descrito el modelo.
Se puede aplicar en el desarrollo de software entregando gran variedad de formas para dar soporte a una metodología de desarrollo de software (tal como RUP), pero no especifica en sí mismo qué metodología o proceso usar.

### 2.2.3 RUP.

Actualmente no existe una metodología de desarrollo de software que sea global, es decir que encierre características que puedan aplicarse a cualquier tipo de proyecto. Las características de cada proyecto conjuntamente con su equipo de desarrollo, recursos, y requisitos exigen que se escoja una que se adapte en la mayor medida posible a estas características. Se escoge entonces la metodología RUP, su ciclo de vida se caracteriza por:

**Está dirigido por Casos de Usos:** Los casos de uso reflejan lo que los usuarios futuros necesitan y desean, lo cual se capta cuando se modela el negocio y se representa a través de los requerimientos. A partir de aquí los casos de uso guían el proceso de desarrollo ya que los modelos que se obtienen, como resultado de los diferentes flujos de trabajo, representan la realización de los casos de uso (cómo se llevan a cabo).

**Centrado en la arquitectura:** La arquitectura muestra la visión común del sistema completo en la que el equipo de proyecto y los usuarios deben estar de acuerdo, por lo que describe los elementos del modelo que son más importantes para su construcción, los cimientos del sistema que son necesarios como base para comprenderlo, desarrollarlo y producirlo económicamente.

**Iterativo e Incremental:** RUP propone que cada fase se desarrolle en iteraciones. Una iteración involucra actividades de todos los flujos de trabajo, aunque desarrolla fundamentalmente algunos más que otros. Por ejemplo, una iteración de elaboración centra su atención en el análisis y diseño, aunque refina los requerimientos y obtiene un producto con un determinado nivel, pero que irá creciendo incrementalmente en cada iteración.

Es práctico dividir el trabajo en partes más pequeñas o miniproyectos. Cada miniproyecto es una iteración que resulta en un incremento. Las iteraciones hacen referencia a pasos en los flujos de trabajo, y los incrementos, al crecimiento del producto. Cada iteración se realiza de forma planificada es por eso que se dice que son miniproyectos.

La metodología escogida, RUP agrupa las actividades en grupos lógicos, para ello delimita 9 flujos de trabajo. A los seis primeros se les denomina flujos de ingeniería y los tres últimos son conocidos como de apoyo. Cada uno de estos flujos pasa, en mayor o menor medida, por cuatro fases: Inicio, Elaboración y Construcción.

### 2.2.4 Visual Paradigm.

Visual Paradigm es una herramienta CASE de diseño que utiliza “UML”: como lenguaje de modelado. Es una herramienta muy potente y de fácil uso. Te permite dibujar todo tipo de diagramas UML, revertir código fuente a modelos UML, generar código fuente desde los diagramas UML. Además generar documentación automáticamente en varios formatos como html, pdf o doc y permite el control de versiones. Por otro lado la herramienta es colaborativa, es decir, soporta múltiples usuarios trabajando sobre el mismo proyecto.

### 2.2.5 C++.

En principio cualquier lenguaje de programación puede ser utilizado para desarrollar sistemas de inteligencia artificial. Pero en este caso se toma en cuenta que en el desarrollo de este tipo de sistemas y

la mayoría de los sistemas de gráficos 3d en el mundo suelen implementarse en C++. Es uno de los lenguajes de programación que con más alta velocidad ejecuta el código. Es un lenguaje bastante flexible, de propósito general; además es un lenguaje de programación orientado a objetos tan importante para el desarrollo de este tipo de sistemas. También como una de las ventajas esta su portabilidad, es decir, un programa con el código escrito en C++, se podrá compilar en cualquier sistema operativo o sistema informático sin necesidad de cambiar casi el código fuente.

# CAPÍTULO 3: CARACTERÍSTICAS DEL SISTEMA

## INTRODUCCIÓN.

En este capítulo se presentan las características del sistema. El modelo de domino representado por las clases conceptuales del sistema, las principales funcionalidades del sistema que se expresan en los casos de usos y la descripción de los casos de usos.

## 3.1 REGLAS DEL NEGOCIO.

El programador que utilice el sistema de IA debe tener conocimientos básicos sobre programación de inteligencia artificial para video juegos.

Las entidades del juego deben ser de tipo entidades de IA.

## 3.2 MODELO DE DOMINIO.

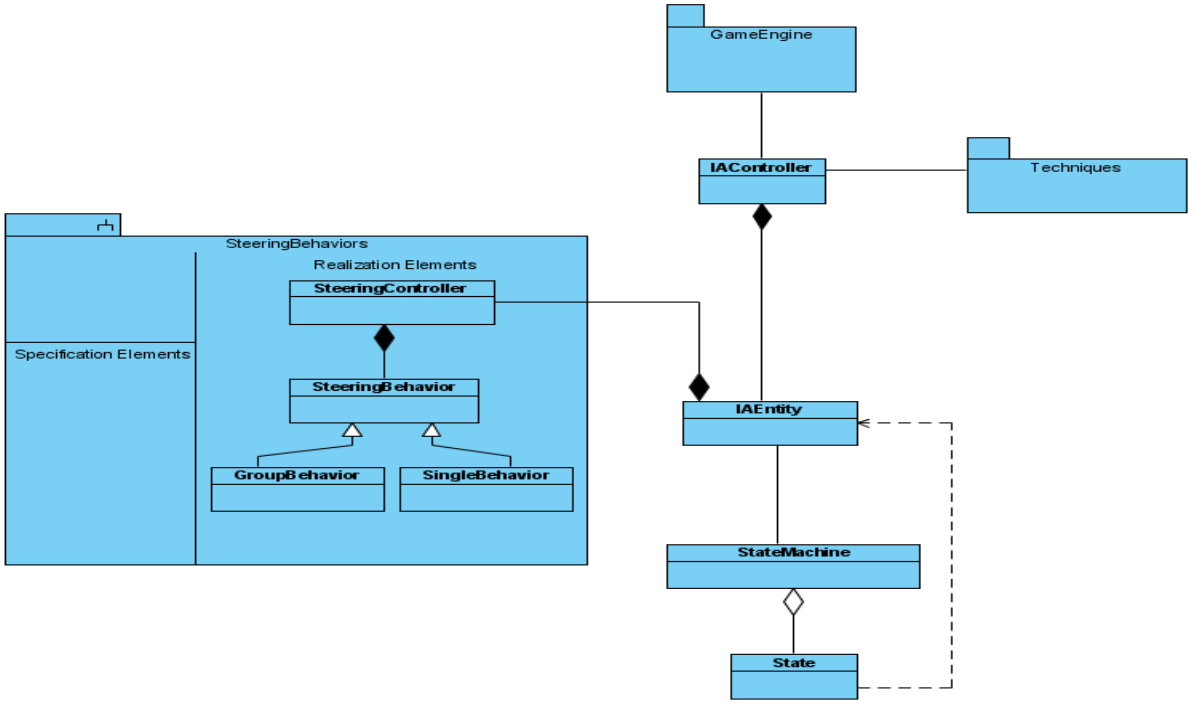


*Figura 7:* Modelo de Dominio.

### 3.2.1 Descripción de las clases conceptuales.

**GameEngine**: Utiliza el sistema de IA.

**IAController**: Se encarga del control y actualización del sistema de IA. Posee lista de entidades.

**Techniques**: Aquí se deben empaquetar algoritmos de búsquedas, caminos, redes neuronales y otras técnicas.

**IAEntity**: Es el tipo de entidad que será controlada por el sistema de IA.

**State**: Representa el estado en el cual debe encontrarse una entidad en un momento dado. Define el comportamiento de una entidad en un determinado momento.

**StateMachine**: Se encarga de gestionar la transición entre estados.

**SteeringBehavior**: Sirve de interfaz a los comportamientos de dirección.

**SingleBehavior**: Representa comportamiento de dirección simple. Este tipo de comportamiento utiliza un único objetivo para el cálculo de la fuerza de dirección.

**GroupBehavior**: Representa comportamiento de dirección grupal. Este tipo de comportamiento tiene en cuenta una o varias entidades para el cálculo de la fuerza de dirección.

**SteeringController**: Se encarga del control y gestión de los diferentes comportamientos de dirección. Le proporciona a la IAEntity la fuerza de dirección resultante de la combinación de los diferentes comportamientos de dirección.

## 3.3 REQUERIMIENTOS.

### 3.3.1 Captura de requisitos.

### 3.3.2 Requisitos funcionales.

R1 Inicializar sistema de IA.
R2 Actualizar el sistema de inteligencia artificial.
R3 Finalizar IA.
R4 Crear Máquina de estados.
R5 Inicializar comportamiento de dirección.
R6 Actualizar entidades de inteligencia artificial.
R7 Calcular fuerza de dirección.
R8 Actualizar máquina de estados.
R9 Cambiar estados.

### 3.3.3 Requisitos no funcionales.

Los requerimientos no funcionales son propiedades o cualidades que el producto debe tener. Debe pensarse en estas propiedades como las características que hacen al producto atractivo, usable, rápido o confiable.

A continuación se presentan los requerimientos no funcionales definidos para el sistema a desarrollar.

- Usabilidad**:** Los futuros usuarios del sistema serán programadores con conocimientos básicos con respecto al tema del desarrollo de inteligencia artificial para video juegos.
- Soporte: Debe ser compatible con la plataforma Windows.
- Software: Sistema operativo. Windows.

- Hardware**:** Cualquier tipo de hardware y al menos 128 megabyte de RAM.

## 3.4 MODELO DE CASOS DE USOS.

### 3.4.1 Descripción de los actores del sistema.

| Actor | Justificación |
|---|---|
| Juego | Es el que utiliza las funciones del sistema de IA para el desarrollo lógico del juego. |
| EntidaddeJuego | Es quien crea la máquina de estado para la toma de decisión e inicializa el comportamiento de dirección para la gestión de la locomoción. |
| SistemaIA | Gestiona los procesos de la IA. Es quien controla las entidades del juego y se encarga de mantenerlas actualizadas con el ciclo del juego. |

*Tabla 1:* Descripción de los actores del sistema.

3.4.2 Diagrama de casos de usos.

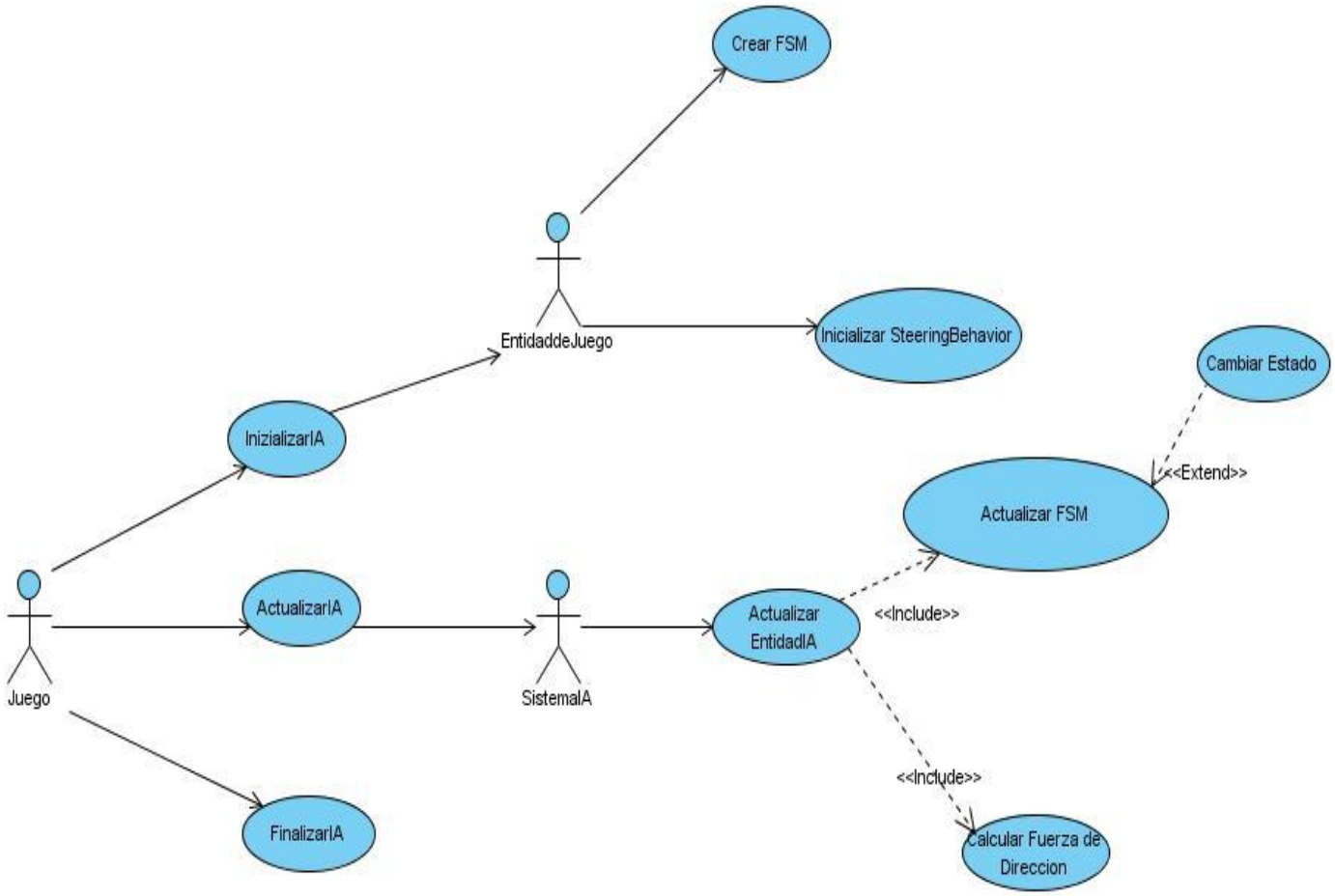


*Figura 8:* Diagrama de Casos de Usos.

3.4.3 Descripción de los casos de usos en formato expandido.

| **Caso de Uso** | InicializarIA |
|---|---|
| **Actor** | Juego |
| **Propósito** | Las entidades creadas en el juego deben adicionarse al sistema de IA |
| **Resumen** | El caso de uso se inicia cuando el juego decide adicionar las entidades al |

| | |
|---|---|
| | sistema de IA. |
| **Referencia** | R1 |
| **Precondiciones** | Las entidades deben estar creadas con anterioridad. Todos los atributos necesitados por el sistema de IA deben tener un valor inicial. |
| **Pos-condiciones** | El sistema de IA tiene control de las entidades. |
| **Curso Normal de los Eventos** | |
| **Acciones del Actor** | **Respuesta del Sistema** |
| 1- El juego decide después de creadas las entidades cederle el control de estas al sistema de IA. | 1.1- El sistema de IA adiciona cada entidad proveniente del juego a su lista de entidades. |
| **Curso Alterno** | |
| | |
| **Prioridad** | Crítico |

*Tabla 2:* Descripción del CU InicializarIA en formato expandido.

| | |
|---|---|
| **Caso de Uso** | ActualizarIA |
| **Actor** | Juego |
| **Propósito** | Actualizar el sistema de IA con el ciclo del juego. |
| **Resumen** | El caso de uso se inicia cuando el sistema decide actualizar el sistema de IA pasándole el tiempo transcurrido de la última actualización a la actualización actual. |
| **Referencia** | R2 |
| **Precondiciones** | Juego en ejecución. |
| **Pos-condiciones** | Se actualiza el sistema de IA. |
| **Curso Normal de los Eventos** | |
| **Acciones del Actor** | **Respuesta del Sistema** |
| 1- El juego pasa la secuencia de tiempo transcurrida desde la última actualización, para la actualización del sistema de IA. | 1.1- El sistema de IA se actualiza con esta secuencia de tiempo pasada por el juego. |

| **Curso Alterno** | |
|---|---|
| | |
| **Prioridad** | Crítico |

*Tabla 3:* Descripción del CU ActualizarIA en formato expandido.

| **Caso de Uso** | FinalizarIA |
|---|---|
| **Actor** | Juego |
| **Propósito** | Finalizar las acciones del sistema de IA. Liberar memoria asignada al sistema de IA. |
| **Resumen** | El caso de uso se inicia cuando el juego decide finalizar las acciones del sistema de inteligencia artificial. |
| **Referencia** | R3 |
| **Precondiciones** | El juego detiene la ejecución. |
| **Pos-condiciones** | Se libera la memoria asignada al sistema de IA. |
| **Curso Normal de los Eventos** | |
| **Acciones del Actor** | **Respuesta del Sistema** |
| 1- El juego llama a la funcionalidad finalizar del sistema de IA. | 1.1- El sistema elimina las entidades de la lista de entidades.<br>1.2- El sistema de IA libera la memoria asignada a las entidades que estaban siendo controladas. |
| **Curso Alterno** | |
| | |
| **Prioridad** | Medio |

*Tabla 4:* Descripción del CU FinalizarIA en formato expandido.

| **Caso de Uso** | Crear Máquina de Estados |
|---|---|
| **Actor** | Entidad de Juego |
| **Propósito** | Crear la máquina de estado que se encargara de la gestión de la transición entre los estados que tendrán las entidades. |
| **Resumen** | El caso de uso se inicia cuando la entidad de juego decide crear la máquina |

| | |
|---|---|
| | de estados que gestionará la transición entre los estados de la entidad. Después de la creación se le debe asignar el estado inicial a la máquina de estados y activar el estado inicial. |
| **Referencia** | R4 |
| **Precondiciones** | Los estados de la entidad deben estar previamente creados.<br>Los estados deben tener implementada la lógica para las transiciones.<br>En la lógica de los estados es donde se activan y se desactivan los comportamientos de dirección.(ver caso de uso calcular fuerza de dirección) |
| **Pos-condiciones** | La máquina de estados tiene control sobre el estado inicial. |
| **Curso Normal de los Eventos** | |
| **Acciones del Actor** | **Respuesta del Sistema** |
| 1- La entidad de juego decide crear máquina de estados. | 1.1- Se recibe la entidad de juego y con esta se crea la máquina de estado.<br>1.2- Se le asigna a la máquina de estados el estado inicial de la entidad de juego.<br>1.3- La máquina de estado activa el estado inicial. |
| **Curso Alterno** | |
| | |
| **Prioridad** | Medio |

*Tabla 5:* Descripción del CU Crear Máquina de Estados en formato expandido.

| | |
|---|---|
| **Caso de Uso** | Inicializar Comportamiento de Dirección |
| **Actor** | Entidad de Juego |
| **Propósito** | Asignarle al comportamiento de dirección la entidad que se moverá. |
| **Resumen** | El caso de uso se inicia cuando la entidad de juego decide inicializar el comportamiento de dirección. Para esto se le debe asignar al comportamiento de dirección la entidad a mover. |
| **Referencia** | R5 |

| **Precondiciones** | Debe de estar creado el comportamiento de dirección. | |
|---|---|---|
| **Pos-condiciones** | El comportamiento de dirección tiene control sobre la entidad. | |
| **Curso Normal de los Eventos** | | |
| **Acciones del Actor** | | **Respuesta del Sistema** |
| 1- La entidad de juego decide inicializar el comportamiento de dirección. | | 1.1- El comportamiento de dirección recibe la entidad que deberá moverse. |
| **Curso Alterno** | | |
| | | |
| **Prioridad** | Critico | |

*Tabla 6:* Descripción del CU Inicializar Comportamiento de Dirección en formato expandido.

| **Caso de Uso** | Actualizar EntidadIA | |
|---|---|---|
| **Actor** | SistemaIA | |
| **Propósito** | Mantener las entidades actualizadas con el ciclo del juego. | |
| **Resumen** | El caso de uso se inicia cuando el sistema de IA recibe la secuencia de tiempo en la cual se debe actualizar el sistema. El sistema de IA actualiza cada una de las entidades tiene en su contenedor de entidades. | |
| **Referencia** | R6 | |
| **Precondiciones** | El juego se encuentre en ejecución. | |
| **Pos-condiciones** | | |
| **Curso Normal de los Eventos** | | |
| **Acciones del Actor** | | **Respuesta del Sistema** |
| 1- El sistema decide actualizar cada entidad en correspondencia con el tiempo recibido de la secuencia de juego. | | 1.1- La entidad actualiza su ciclo interno en correspondencia al tiempo de actualización recibido del sistema de IA. |
| **Curso Alterno** | | |
| | | |
| **Prioridad** | Crítico | |

*Tabla 7:* Descripción del CU Actualizar EntidadIA en formato expandido.

| **Caso de Uso** | Actualizar Máquina de Estados | |
|---|---|---|
| **Actor** | SistemaIA | |
| **Propósito** | Actualizar la máquina de estado que posee la entidad de juego. | |
| **Resumen** | El caso de uso comienza cuando la entidad de juego recibe la secuencia de tiempo de actualización del sistema de IA. | |
| **Referencia** | R7 | |
| **Precondiciones** | Debe de haberse iniciado la actualización el sistema de IA así como la actualización de las entidades de IA. | |
| **Pos-condiciones** | Se ejecuta el estado correspondiente. | |
| **Curso Normal de los Eventos** | | |
| **Acciones del Actor** | | **Respuesta del Sistema** |
| 1- El sistema de IA le permite a la entidad de juego actualizar la máquina de estado. | | 1.1- La máquina de estado ejecuta el estado en el que se encuentra la entidad. |
| **Curso Alterno** | | |
| | | 1.1- Si ocurre algún evento o algún cambio de variable significativo, se cambia de estado (ver caso de uso cambiar estado). |
| **Prioridad** | Crítico | |

*Tabla 8:* Descripción del CU Actualizar Máquina de Estados en formato expandido.

| **Caso de Uso** | Calcular Fuerza de Dirección |
|---|---|
| **Actor** | SistemaIA |
| **Propósito** | Calcular la fuerza de dirección que permita el movimiento de la entidad. |
| **Resumen** | El caso de uso comienza cuando la entidad de juego recibe la secuencia de tiempo de actualización del sistema de IA. Dadas las características cinemáticas de la entidad en cuestión(posición, velocidad, masa, fuerza) o teniendo en cuenta otras entidades u obstáculos del entorno, se calcula la fuerza de dirección que se necesita para lograr un objetivo determinado, tal |

| | |
|---|---|
| | como llegar a una posición determinada o mantener la separación con otra entidad. Con esta fuerza la entidad calcula la nueva velocidad y la nueva posición. |
| **Referencia** | R8 |
| **Precondiciones** | Debe de haberse iniciado la actualización el sistema de IA y la actualización de las entidades de IA.<br>Debe de haber algún comportamiento de dirección activado. |
| **Pos-condiciones** | La entidad se mueve en correspondencia a la fuerza de dirección. |
| **Curso Normal de los Eventos** | |
| **Acciones del Actor** | **Respuesta del Sistema** |
| 1- El sistema de IA le permite a la entidad de juego calcular la fuerza de dirección.<br>2- La entidad recibe la fuerza de dirección calculada, con esta calcula la nueva velocidad y la nueva posición. | 1.1- Dadas las características cinemáticas de la entidad el comportamiento de dirección calcula la fuerza de dirección en dependencia del objetivo que se quiera lograr.<br>1.2- El algoritmo de dirección le retorna la fuerza calculada a la entidad de juego. |
| **Curso Alterno** | |
| | |
| **Prioridad** | Crítico |

*Tabla 9:* Descripción del CU Calcular Fuerza de Dirección en formato expandido.

| | |
|---|---|
| **Caso de Uso** | Cambiar Estados |
| **Actor** | SistemaIA |
| **Propósito** | Realizar la transición entre estados de la entidad en cuestión. |
| **Resumen** | El caso de uso comienza cuando ocurre algún evento (disparo de una alarma, finalización de un tiempo determinado,..etc.) o algún cambio de variable que coincide con la lógica para la transición de estado. |
| **Referencia** | |
| **Precondiciones** | Debe de haberse iniciado la actualización el sistema de IA. |

| | |
|---|---|
| | Debe de estarse ejecutando algún estado. |
| **Pos-condiciones** | Se ejecuta el nuevo estado. |
| **Curso Normal de los Eventos** | |
| **Acciones del Actor** | **Respuesta del Sistema** |
| 1- Ocurre un evento. | 1.1- La entidad tiene conocimiento del evento que ha ocurrido.<br>1.2- La entidad le dice a la máquina de estado que cambie del estado actual al nuevo estado.<br>1.3- La máquina de estado desactiva el estado actual y activa el nuevo estado. |
| **Curso Alterno** | |
| | |
| **Prioridad** | Media |

*Tabla 10:* Descripción del CU Cambiar Estados en formato expandido.

# CAPÍTULO 4: DISEÑO DEL SISTEMA

## INTRODUCCIÓN

Este capítulo está dedicado al diseño del sistema. Se encuentran los diagramas de diseño y los de secuencia que intervienen en el ciclo de desarrollo del sistema.

## 4.1 DIAGRAMA DE CLASES DEL DISEÑO.

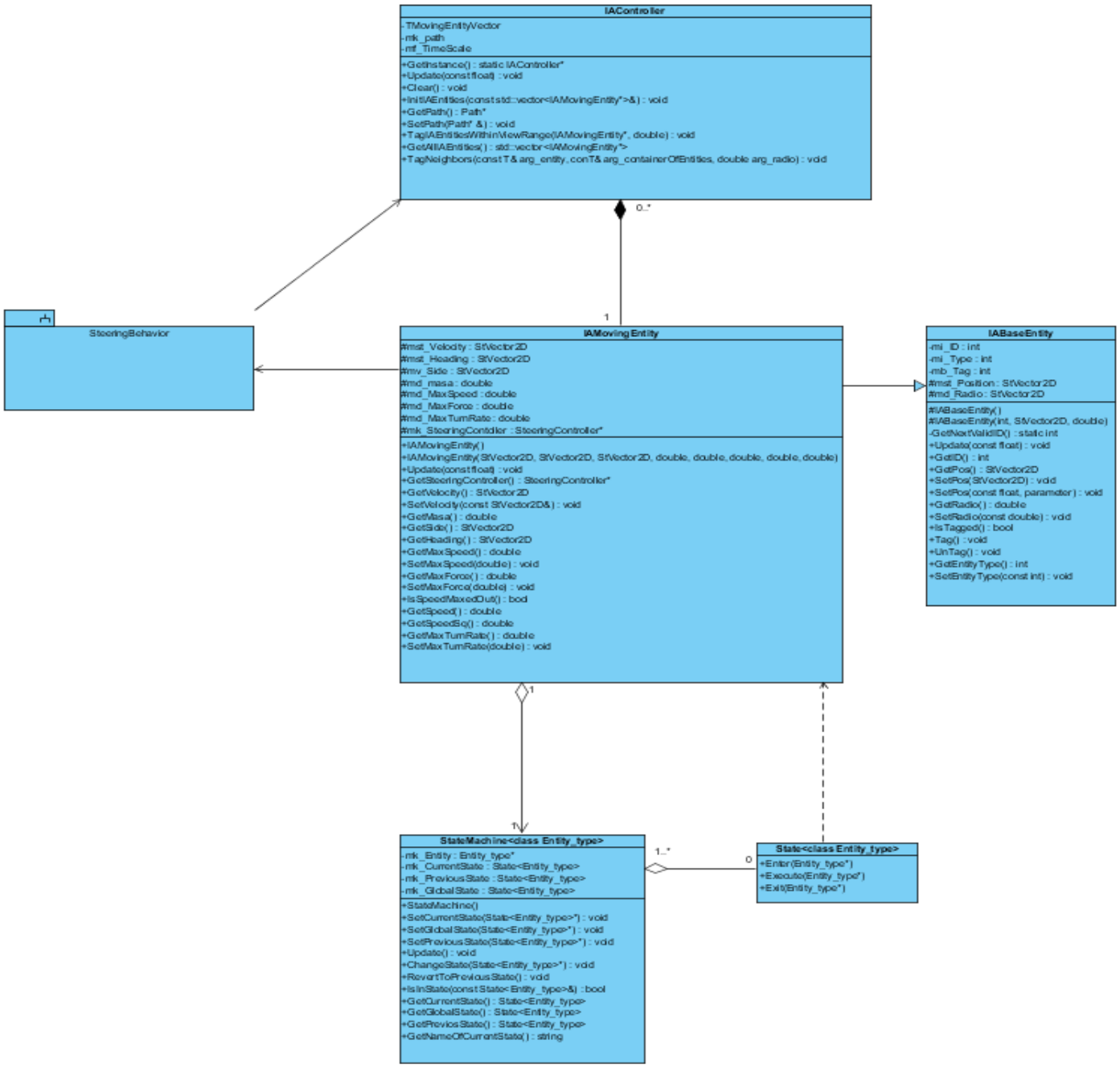


*Figura 9:* Diagrama de clases del diseño.

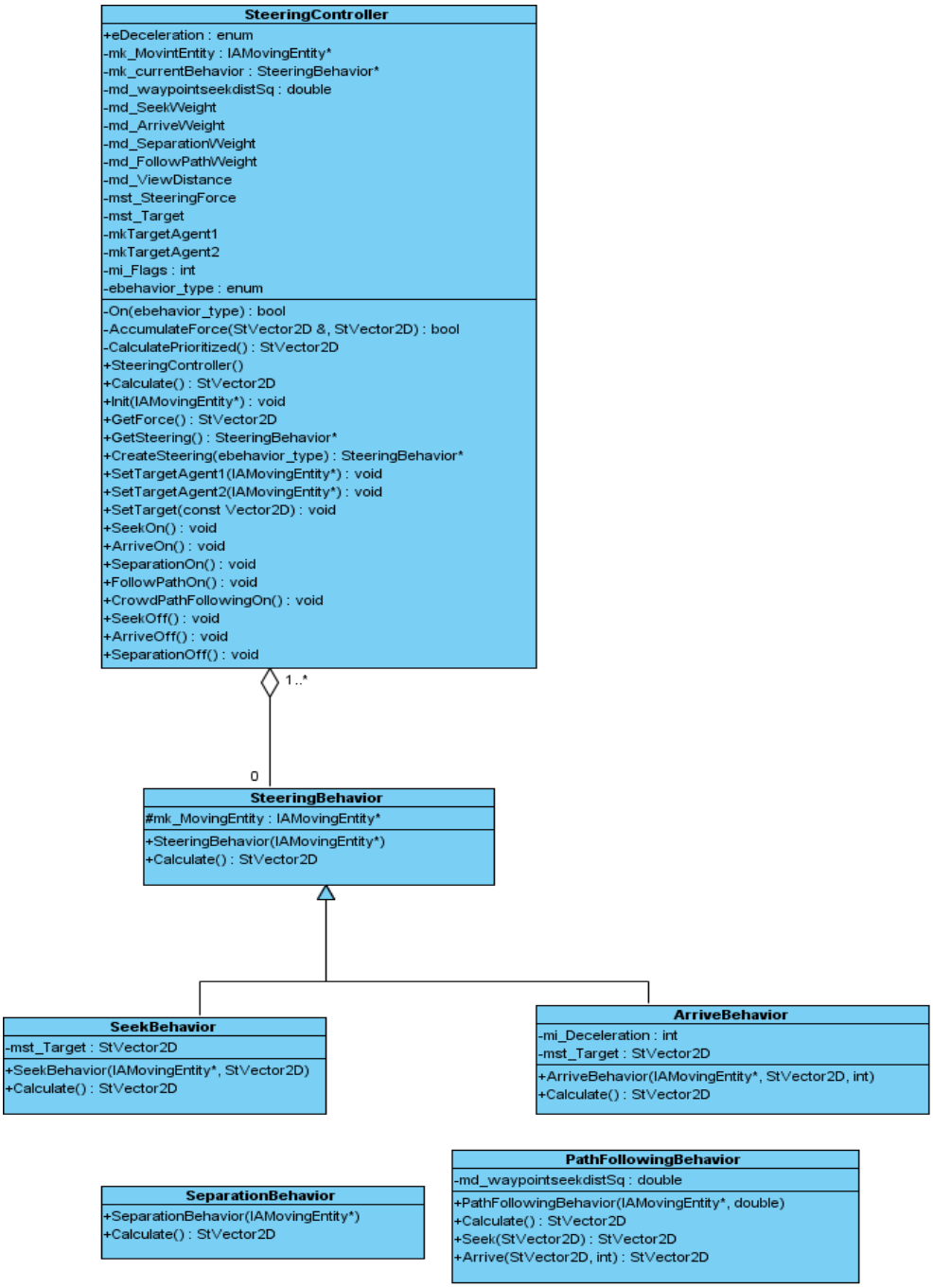


*Figura 10:* Subsistema Steering Behavior.

## 4.2 DIAGRAMA DE INTERACCIÓN POR CASOS DE USOS.

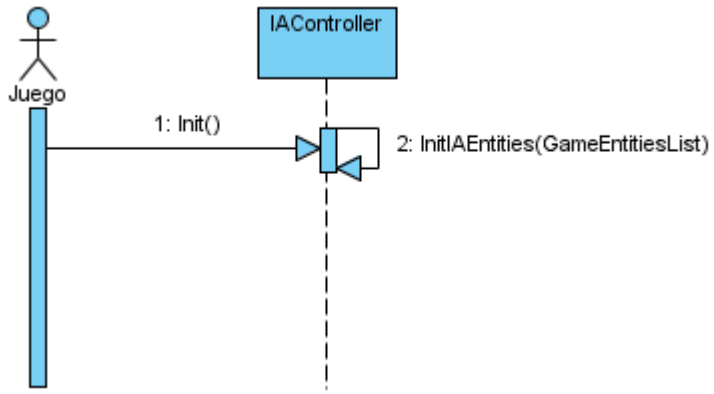


*Figura 11:* CU Inicializar IA.

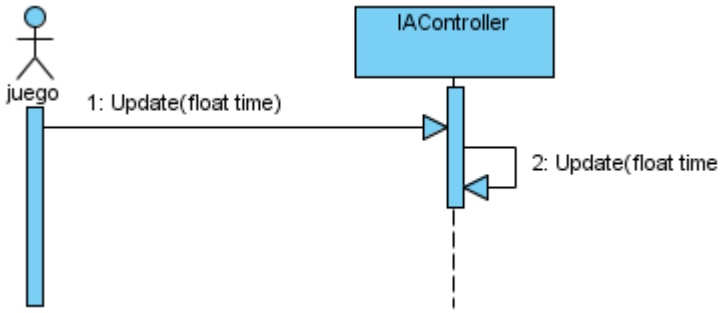


*Figura 12:* CU Actualizar IA.

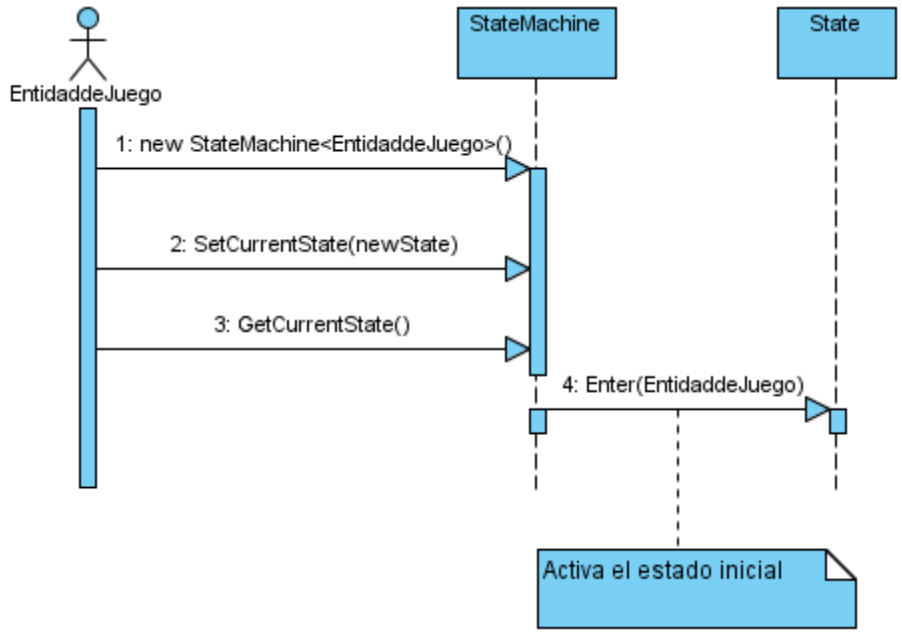


*Figura 13:* CU Crear Máquina de estado.

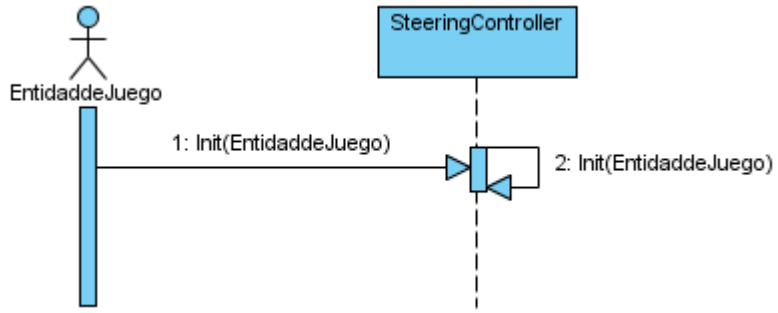


*Figura 14:* CU Inicializar comportamiento de dirección.

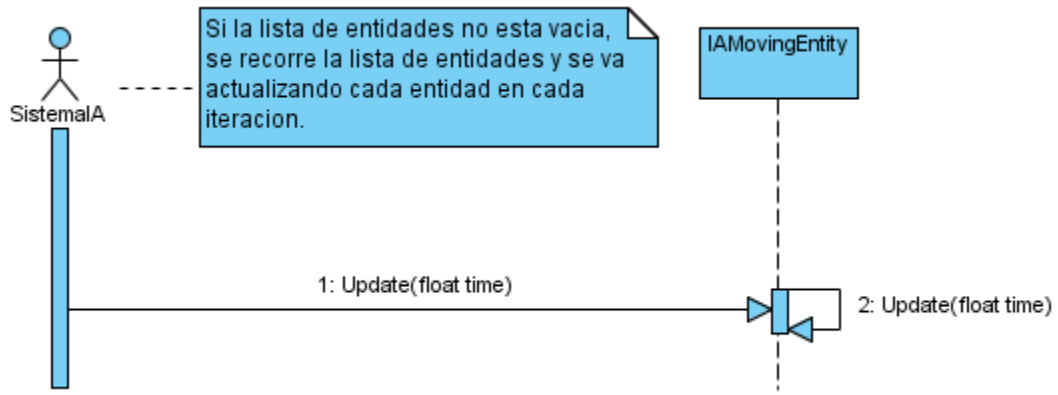


*Figura 15:* CU Actualizar Entidad de juego.

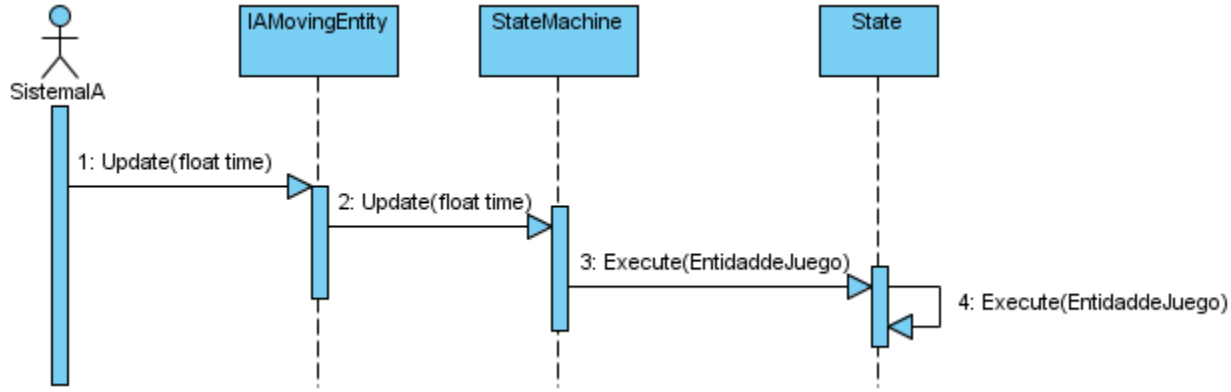


*Figura 16:* Actualizar Máquina de estado.

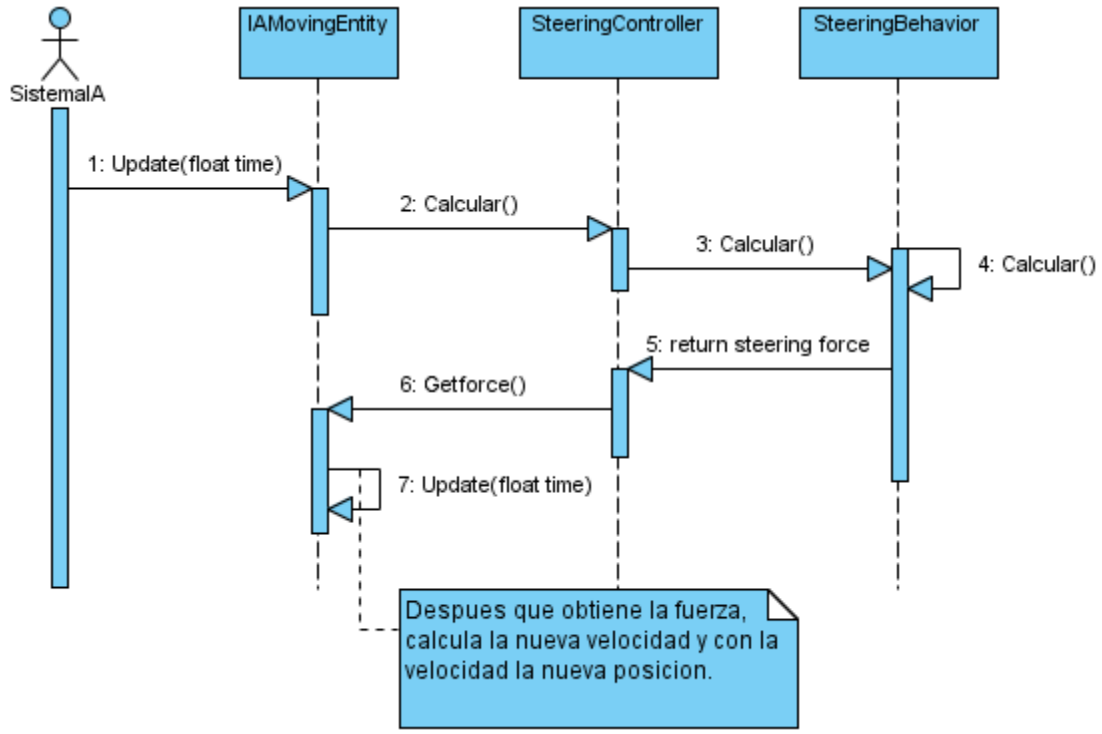


*Figura 17:* Calcular Fuerza de dirección.

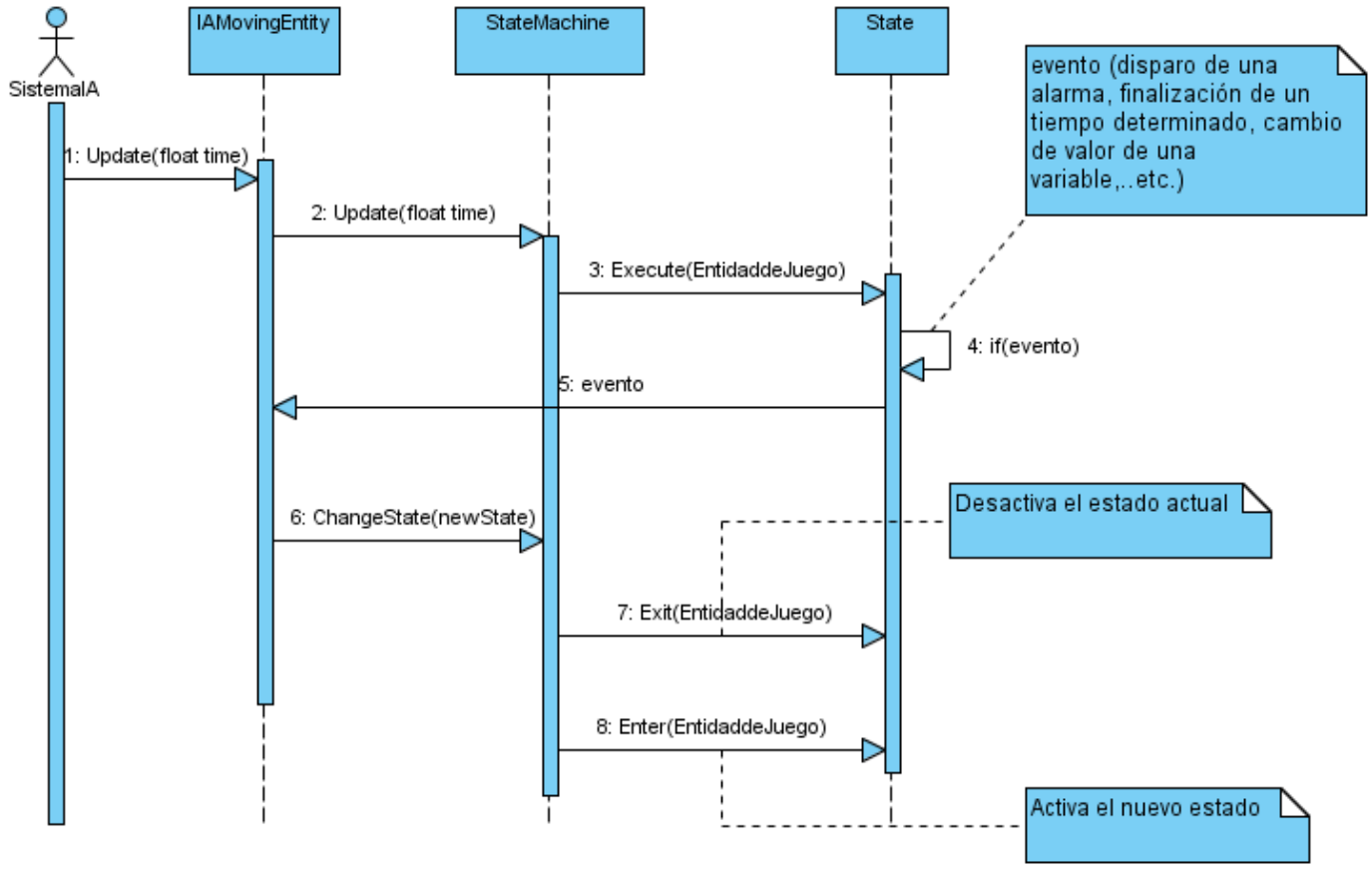


*Figura 18:* Cambiar estados.

## 4.3 DESCRIPCIÓN DE LAS CLASES DEL DISEÑO.

Aquí se describen las clases principales clases que forman parte del sistema con sus principales atributos y funcionalidades.

| **Nombre**: IAController | |
|---|---|
| **Tipo de Clase**: Controladora | |
| **Atributo** | **Tipo** |
| TMovingEntityVector | std::vector<IAMovingEntity*> |
| mk_path | Path* |
| mf_TimeScale | float |
| **Para cada responsabilidad**: | |

| | |
|---|---|
| Nombre: | GetInstance() |
| Descripción: | Retorna instancia de IAController |
| Nombre: | Update(const float) |
| Descripción: | Actualiza la lista de entidades en cada secuencia de juego. |
| Nombre: | Clear() |
| Descripción: | Limpia la lista de entidades. |
| Nombre: | InitIAEntities(const std::vector<IAMovingEntity*>&) |
| Descripción: | Inicializa la lista de entidades. |
| Nombre: | GetPath() |
| Descripción: | Retorna el camino. |
| Nombre: | SetPath(Path* &) |
| Descripción: | Actualiza el camino. |
| Nombre: | TagIAEntitiesWithinViewRange(IAMovingEntity*, double) |
| Descripción: | Marca todas las entidades que están dentro de un rango de visión pasado por parámetro, de la entidad pasada por parámetro. |
| Nombre: | GetAllIAEntities() |
| Descripción: | Retorna lista de entidades. |
| Nombre: | TagNeighbors(T& arg_entity, conT& arg_containerOfEntities, double arg_radio) |
| Descripción: | Función genérica para ejecutar el marcaje de todas las entidades (que están en el contenedor arg_containerOfEntities) dentro del rango de visión (arg_radio) de la entidad arg_entity. |

*Tabla 11:* Descripción de clases "*IAController*".

| **Nombre**: IABaseEntity | |
|---|---|
| **Tipo de Clase**: Entidad | |
| **Atributo** | **Tipo** |
| mi_ID | int |
| mi_Type | int |
| mb_Tag | bool |
| mst_Position | StVector2D |
| md_Radio | double |
| **Para cada responsabilidad**: | |
| Nombre: | IABaseEntity() |
| Descripción: | Constructor vacio. |
| Nombre: | IABaseEntity(int, StVector2D, double) |
| Descripción: | Constructor por parámetro, se le especifica el tipo de entidad, vector posición y radio de la entidad. |
| Nombre: | GetNextValidID() |
| Descripción: | Se ejecuta cada vez que se crea una entidad, se encarga de asignarle id a las entidades. Estos id no se repiten, este método se encarga de incrementar el id de la |

| | |
|---|---|
| | siguiente entidad a crear. |
| Nombre: | GetID() |
| Descripción: | Retorna el id de una determinada entidad. |
| Nombre: | GetPos() |
| Descripción: | Retorna la posición de una determinada entidad. |
| Nombre: | SetPos(StVector2D) |
| Descripción: | Actualiza la posición de una entidad dado un vector. |
| Nombre: | SetPos(const float, const float) |
| Descripción: | Actualiza la posición de una entidad dada las coordenadas. |
| Nombre: | GetRadio() |
| Descripción: | Retorna el radio de la entidad. |
| Nombre: | SetRadio(const double&) |
| Descripción: | Actualiza en radio de una entidad determinada. |
| Nombre: | IsTagged() |
| Descripción: | Retorna true si el valor de la variable mb_Tag es true. |
| Nombre: | Tag() |
| Descripción: | Pone en true la variable mb_Tag. |
| Nombre: | UnTag() |
| Descripción: | Pone en false la variable mb_Tag. |
| Nombre: | GetEntityType() |
| Descripción: | Retorna el tipo de una entidad determinada. |
| Nombre: | SetEntityType(const int) |
| Descripción: | Actualiza el tipo de una entidad determinada. |

*Tabla 12:* Descripción de clases "*IABaseEntity*".

| **Nombre**: IAMovingEntity | |
|---|---|
| **Tipo de Clase**: Entidad | |
| **Atributo** | **Tipo** |
| mst_Velocity | StVector2D |
| mst_Heading | StVector2D |
| mst_Side | StVector2D |
| md_masa | double |
| md_MaxSpeed | double |
| md_MaxForce | double |
| md_MaxTurnRate | double |
| mk_SteeringContoller | SteeringController* |
| **Para cada responsabilidad**: | |
| Nombre: | IAMovingEntity() |
| Descripción: | Constructor vacio. |
| Nombre: | IAMovingEntity(StVector2D, StVector2D, StVector2D, double, double, double, double, double) |

| | |
|---|---|
| Descripción: | Constructor por parámetro. Se le especifica los vectores posición, velocidad, heading (vector normal ubicado hacia el frente de la entidad, en el ciclo de juego se debe mantener normalizado con el vector velocidad o el vector posición), el valor del radio, la velocidad máxima, la masa, la fuerza máxima y la velocidad máxima de rotación. |
| Nombre: | Update(const float) |
| Descripción: | Se actualiza en cada actualización de la entidad. Es un método virtual que debe ser implementado en las entidades de juego que hereden de esta clase. |
| Nombre: | GetSteeringController() |
| Descripción: | Retorna el puntero a objeto de tipo SteeringController, que es el encargado del control y gestión de los steering behaviors. |
| Nombre: | GetVelocity() |
| Descripción: | Retorna el valor de la velocidad. |
| Nombre: | SetVelocity(const StVector2D&) |
| Descripción: | Actualiza el valor de la velocidad. |
| Nombre: | GetMasa() |
| Descripción: | Retorna el valor de la variable md_masa. |
| Nombre: | GetSide() |
| Descripción: | Retorna el valor de la variable mst_Side. |
| Nombre: | GetHeading() |
| Descripción: | Retorna el valor de la variable mst_Heading. |
| Nombre: | GetMaxSpeed() |
| Descripción: | Retorna el valor de la velocidad máxima. |
| Nombre: | SetMaxSpeed(double) |
| Descripción: | Actualiza el valor de la velocidad máxima. |
| Nombre: | GetMaxForce() |
| Descripción: | Retorna el valor de la velocidad máxima. |
| Nombre: | SetMaxForce(double) |
| Descripción: | Actualiza el valor de la fuerza máxima. |
| Nombre: | IsSpeedMaxedOut() |
| Descripción: | Retorna true si el valor de la velocidad máxima es mayor que la longitud de la velocidad. |
| Nombre: | GetSpeed() |
| Descripción: | Retorna la longitud de la velocidad. |
| Nombre: | GetSpeedSq() |
| Descripción: | Retorna la longitud de la velocidad al cuadrado. |
| Nombre: | GetMaxTurnRate() |
| Descripción: | Retorna el valor de la variable md_MaxTurnRate que indica la velocidad de rotación. |
| Nombre: | SetMaxTurnRate(double) |
| Descripción: | Actualiza el valor de la variable md_MaxTurnRate |

*Tabla 13:* Descripción de clases "*IAMovingEntity*".

| **Nombre**: State<class Entity_type> | |
|---|---|
| **Tipo de Clase**: Interfaz | |
| **Atributo** | **Tipo** |
| **Para cada responsabilidad**: | |
| Nombre: | Enter(Entity_type *) |
| Descripción: | Se ejecuta cuando se entra o inicia el estado. |
| Nombre: | Execute(Entity_type *) |
| Descripción: | Se ejecuta en cada actualización del estado. |
| Nombre: | Exit(Entity_type *) |
| Descripción: | Se ejecuta cuando se sale del estado. |

*Tabla 14:* Descripción de clases "*State*".

| **Nombre**: StateMachine<class Entity_type> | |
|---|---|
| **Tipo de Clase**: Controladora | |
| **Atributo** | **Tipo** |
| mk_Entity | Entity_type* |
| mk_CurrentState | State<Entity_type>* |
| mk_PreviousState | State<Entity_type>* |
| mk_GlobalState | State<Entity_type>* |
| **Para cada responsabilidad**: | |
| Nombre: | StateMachine(Entity_type*) |
| Descripción: | Constructor de la clase, se le pasa como parámetro la entidad a la que se le controlaran los estados. |
| Nombre: | SetCurrentState(State<Entity_type>*) |
| Descripción: | Se utiliza para inicializar el estado de la entidad. |
| Nombre: | SetGlobalState(State<Entity_type>*) |
| Descripción: | Se utiliza para inicializar el estado global. |
| Nombre: | SetPreviousState(State<Entity_type>*) |
| Descripción: | Se utiliza para especificar el estado anterior |
| Nombre: | Update() |
| Descripción: | Este método se ejecuta en cada actualización de la máquina de estado. Aquí es donde se ejecuta en caso de que exista el estado global y el estado en el que se encuentra la entidad. |
| Nombre: | ChangeState(State<Entity_type>*) |
| Descripción: | Cambia al nuevo estado pasado como parámetro. |
| Nombre: | RevertToPreviousState() |
| Descripción: | Cambia el estado al estado anterior. |
| Nombre: | isInState(const State<Entity_type>&) |
| Descripción: | Retorna true si el estado entrado por parámetro es igual al estado en que se encuentra la entidad. |

| | |
|---|---|
| Nombre: | GetCurrentState() |
| Descripción: | Retorna el estado actual de la entidad. |
| Nombre: | GetGlobalState() |
| Descripción: | Retorna el estado global de la entidad. |
| Nombre: | GetPreviousState() |
| Descripción: | Retorna el estado anterior de la entidad. |
| Nombre: | GetNameOfCurrentState() |
| Descripción: | Retorna el nombre del estado en el que se encuentra la entidad. |

*Tabla 15:* Descripción de clases "*StateMachine*".

| **Nombre**: SteeringController | |
|---|---|
| **Tipo de Clase**: Controladora | |
| **Atributo** | **Tipo** |
| eDeceleration | enum |
| mk_CurrentMovintEntity | IAMovingEntity* |
| mk_currentBehavior | SteeringBehavior* |
| md_waypointseekdistSq | double |
| md_SeekWeight | double |
| md_ArriveWeight | double |
| md_SeparationWeight | double |
| md_CohesionWeight | double |
| md_AllignmentWeight | double |
| md_FollowPathWeight | double |
| md_ViewDistance | double |
| mst_SteeringForce | StVector2D |
| mst_Target | StVector2D |
| mk_TargetAgent1 | IAMovingEntity* |
| mk_TargetAgent2 | IAMovingEntity* |
| mi_Flags | int |
| eBehavior_type | enum |
| **Para cada responsabilidad**: | |
| Nombre: | On(ebehavior_type) |
| Descripción: | Este metodo testea si algun bit de la bandera mi_Flags esta activo. |
| Nombre: | AccumulateForce(Vector2D &, Vector2D) |
| Descripción: | Metodo que se encarga de gestionar el incremento de la fuerza (mst_SteeringForce), dado la fuerza de cada behavior activo. |
| Nombre: | CalculatePrioritized() |
| Descripción: | Se calculan y suman las fuerzas para cualquier behavior active, se impone la que mayor peso tenga. |
| Nombre: | SteeringController() |

| | |
|---|---|
| Descripción: | Constructor de la clase. |
| Nombre: | Calculate() |
| Descripción: | Calcula las fuerzas para cada uno de los behaviors activos |
| Nombre: | Init(IAMovingEntity*) |
| Descripción: | Inicializa mk_CurrentMovintEntity que es la entidad a la que se le controlara el movimiento. |
| Nombre: | GetForce() |
| Descripción: | Retorna la fuerza de dirección(mst_SteeringForce) |
| Nombre: | GetSteering() |
| Descripción: | Retorna el behavior actual. |
| Nombre: | CreateSteering(ebehavior_type) |
| Descripción: | se encarga de crear un behavior de acuerdo al tipo pasado por parametro, y retorna el behavior creado |
| Nombre: | SetTarget(const Vector2D arg_target) |
| Descripción: | Actualiza el objetivo. |
| Nombre: | SeekOn() |
| Descripción: | Activa el behavior seek. |
| Nombre: | ArriveOn() |
| Descripción: | Activa el behavior arrive. |
| Nombre: | SeparationOn() |
| Descripción: | Activa el behavior separation |
| Nombre: | FollowPathOn() |
| Descripción: | Activa el behavior pathfollowing |
| Nombre: | CrowdPathFollowingOn() |
| Descripción: | Este comportamiento permite a las entidades del juego seguir un camino manteniendo la separación con las entidades vecinas. Combina el behavior PathFollowing y Separation. |
| Nombre: | SeekOff() |
| Descripción: | Desactiva el behavior seek. |
| Nombre: | ArriveOff() |
| Descripción: | Desactiva el behavior arrive. |
| Nombre: | SeparationOff() |
| Descripción: | Desactiva el behavior separation. |
| Nombre: | FollowPathOff() |
| Descripción: | Desactiva el behavior pathfollowing. |
| Nombre: | CrowdPathFollowingOff() |
| Descripción: | Desactiva el behavior CrowdPathFollowing. |

*Tabla 16:* Descripción de clases "*SteeringController*".

| **Nombre**: SteeringBehavior | |
|---|---|
| **Tipo de Clase**: Interfaz | |
| **Atributo** | **Tipo** |
| mk_MovingEntity | IAMovingEntity* |
| **Para cada responsabilidad**: | |
| Nombre: | SteeringBehavior(IAMovingEntity*) |
| Descripción: | Constructor de la clase. Recibe como parámetro la IAMovingEntity que se deberá mover con un comportamiento determinado. |
| Nombre: | Calculate() |
| Descripción: | Método virtual puro que deberá ser implementado por los diferentes comportamientos. |

*Tabla 17:* Descripción de clases "*SteeringBehavior*".

| **Nombre**: SeekBehavior | |
|---|---|
| **Tipo de Clase**: Entidad | |
| **Atributo** | **Tipo** |
| mst_Target | StVector2D |
| **Para cada responsabilidad**: | |
| Nombre: | SeekBehavior(IAMovingEntity*, StVector2D) |
| Descripción: | Constructor de la clase. Recibe como parámetro la IAMovingEntity que se moverá con este comportamiento y un vector indicando donde deberá moverse la entidad. |
| Nombre: | Calculate() |
| Descripción: | Retorna una fuerza que dirige a la entidad hacia el punto especificado. |

*Tabla 18:* Descripción de clases "*SeekBehavior*".

| **Nombre**: ArriveBehavior | |
|---|---|
| **Tipo de Clase**: Entidad | |
| **Atributo** | **Tipo** |
| mi_Deceleration | int |
| mst_Target | StVector2D |
| **Para cada responsabilidad**: | |
| Nombre: | ArriveBehavior(IAMovingEntity*, StVector2D, int) |
| Descripción: | Constructor de la clase. Recibe como parámetro la IAMovingEntity que se moverá con este comportamiento, un vector indicando el lugar a donde se moverá y la aceleración a la que se moverá. |
| Nombre: | Calculate() |
| Descripción: | Retorna una fuerza que desacelera a la entidad en movimiento mientras se acerca al punto indicado y logra que esta se detenga cuando se encuentra sobre el punto. |

*Tabla 19:* Descripción de clases "*ArriveBehavior*".

| **Nombre:** PathFollowingBehavior | |
|---|---|
| **Tipo de Clase:** Entidad | |
| **Atributo** | **Tipo** |
| mpk_path; | Path* |
| md_waypointseekdistSq | double |
| **Para cada responsabilidad:** | |
| Nombre: | PathFollowingBehavior(IAMovingEntity*, Path* &, double) |
| Descripción: | Constructor de la clase. Recibe como parámetro la IAMovingEntity que se moverá con este comportamiento, una referencia al camino a seguir y una distancia que indica la cercanía a un determinado *waypoint*. |
| Nombre: | Calculate() |
| Descripción: | Calcula la fuerza que le permitirá a la IAMovingEntity transitar por todo el camino dado. |
| Nombre: | Seek(StVector2D) |
| Descripción: | Método de apoyo, con este método la entidad puede moverse de un *waypoint* a otro del camino. Ver descripción de SeekBehavior. |
| Nombre: | Arrive(StVector2D, int) |
| Descripción: | Método de apoyo, con este método la entidad en movimiento puede detenerse en el *waypoint* que indica el final del camino. Ver descripción de ArriveBehavior. |

*Tabla 20:* Descripción de clases "*PathFollowingBehavior*".

| **Nombre**: SeparationBehavior | |
|---|---|
| **Tipo de Clase**: Entidad | |
| **Atributo** | **Tipo** |
| **Para cada responsabilidad**: | |
| Nombre: | SeparationBehavior(IAMovingEntity*) |
| Descripción: | Constructor de la clase. Recibe como parámetro la IAMovingEntity que se moverá con este comportamiento. |
| Nombre: | Calculate() |
| Descripcion: | Calcula la fuerza que permite alejar las entidades vecinas lejos del área de visión o radio de vecinos de la entidad en cuestión. |

*Tabla 21:* Descripción de clases "*SeparationBehavior*".

# CAPÍTULO 5: IMPLEMENTACIÓN

## INTRODUCCIÓN

Esta etapa del proyecto constituye el paso del diseño de clases a la creación de componentes físicos, que se traducen en ficheros .h y .cpp correspondiente a la implementación en C++.

## 5.1 ESTÁNDARES DE CODIFICACIÓN.

Las convenciones o estándares de codificación son pautas de programación que no están enfocadas a la lógica del programa, sino a su estructura y apariencia física para facilitar la lectura, comprensión y mantenimiento del código.

A continuación se presenta la propuesta de estándar de codificación a utilizar para el desarrollo de este trabajo.

### Comentarios:

Cada archivo cabecera comenzara con un comentario que incluya:

- Nombre.

Especifica el nombre del archivo.

- Descripción.

Se hace una pequeña descripción de las clases o funcionalidades que contiene el archivo.

- Autor.

Especifica el nombre del autor.

- Referencia.

La referencia es opcional, aquí se especifica la fuente de la idea original.

*ej:*

```
/* ---------------------------------------------------------------------
*
*  Nombre:      ArchivoCabecera.h
*  ---------------------------------------------------------------------
*  Desc:        Descripción.
*
*  Autor:       Juan Perez.
*
*  Referencia: Thinking in C++.
*
*  ---------------------------------------------------------------------
*/
```

Cada variable debe contener una pequeña descripción del objetivo de la variable.

*ej:*

*// Almacena un entero.*

int entero;

Cada función debe contener una pequeña descripción del objetivo de la función.

*ej:*

*// Retorna un entero.*

int GetInt(){return entero;}

## Nombre de los ficheros:

Los nombres de los ficheros .h y .cpp comienzan con el prefijo IA.

*ej:* IANameOfUnits.cpp

| Tipo de dato | Abreviatura |
|---|---|
| integer | i |
| float | f |

| | |
|---|---|
| double | d |
| bool | b |
| void | v |
| enum | e |
| Arreglo | ar |
| char | c |
| Estructura | st |
| puntero | p |
| Instancia a objetos | k |

*Tabla 22:* Tipos de datos.

### Nombre de identificadores:

Se considera como identificador a los nombres de variables (arreglos, matrices, apuntadores), funciones, así como cualquier tipo de dato definido por el usuario (estructura, clase). Dichos identificadores deberán seguir las siguientes normas.

#### *Identificadores de variables:*

Los nombres de las variables comienzan con un identificador del tipo de dato al que correspondan, en caso de que sean variables miembros de una clase, se le antepone en identificador m en minúscula seguido del tipo de dato al que corresponde, ( “mi_” par una variable miembro de tipo integer). En caso de ser argumento de funciones se le antepone el prefijo “arg_” seguido del identificador en minúscula.

*ej:*

```
int mi_Variable;  //variable miembro de una clase
float mf_Variable; //variable miembro de una clase
```

### Instancias y tipos creados:

A las instancias y tipos creado se le antepone, como identificador, la letra k minúscula.

En caso de los tipos enumerativos se le antepone la letra e como identificador.

*ej:*

MyEnumerated **e**Name;  //tipo enumerativo
ClassName **k**ObjectName; // objeto de una clase
ClassName* **pk**Name; // puntero a objeto
ClassName* **mk**_Name; // variable miembro de una clase

### Lista e iteradores de la std:

Para los tipos de datos utilizados de la librería estándar de C++ (*vector*, *list*, etc.), se utiliza el indicador "**T**" antes del identificador, con los sufijos **List**, **Vector** según la estructura, se utiliza el identificador **iter** para los iteradores se le antecede el tipo de estructura:

*ej:*

std::vector<CNode*> **T**Node**Vector**;
TNodeVector::iterator **viter**;  //iterador de vector

std::list<CNode*> **T**Node**List**;
TNodeList::iterator **listiter**;  //iterador de una lista

### Métodos:

En el caso de los métodos, se les antepone el identificador del tipo de dato de devolución, y en caso de no tenerlo (void), no se les antepone nada. Los constructores y destructores, como lo exigen los compiladores, llevan el nombre de la clase.

Constructor y destructor:

*ej:*ClassName (bool arg_bVarName, float& arg_fVarName);
~ClassName ();

*Funciones:*

*ej:* bool bFunction1 (…); int* piFunction2 (…); CClassName* pkFunction3 (…);

*Métodos de acceso a miembros:*

Los métodos de acceso a los miembros de las clases se nombrarán “Get” seguido del identificador, para el caso de retorno de valores y “Set” en caso de la actualización de valores.

*ej:* Para la siguiente variable, los métodos de acceso serían:

int **mi**_MyVar; //variable

int GetMyVar(); //”Get”
void SetMyVar(int arg_myvar) //”Set”

## 5.2 DIAGRAMA DE DESPLIEGUE.

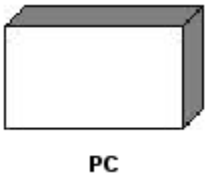


*Figura 19:* Diagrama de despliegue.

## 5.3 DIAGRAMA DE COMPONENTES.

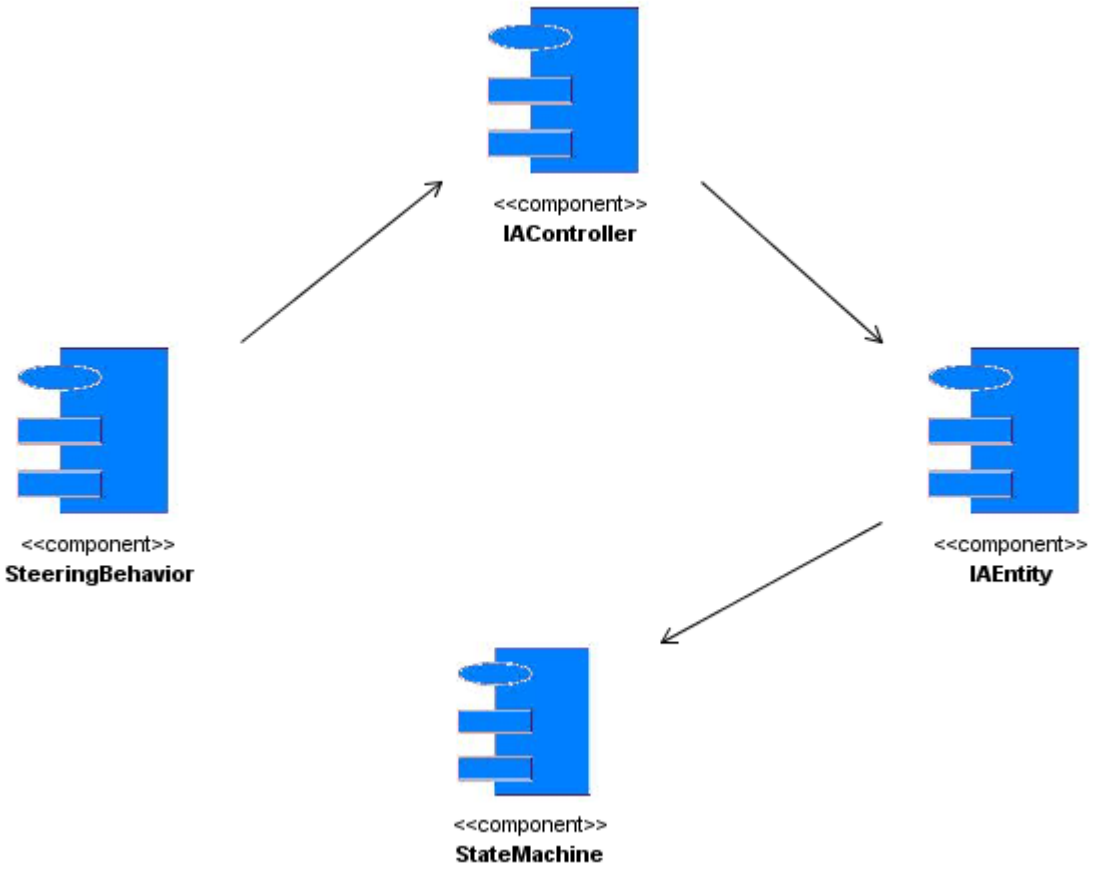


*Figura 20:* Subpaquetes de Componentes.

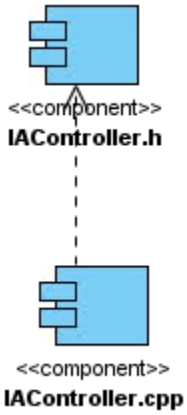


*Figura 21:* Diagrama de Componentes “IAController”.

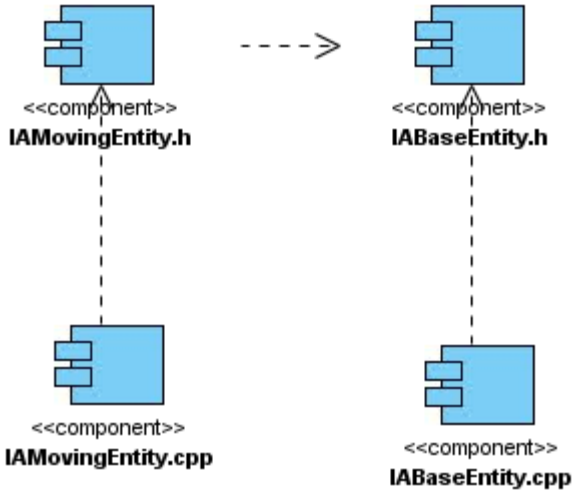


*Figura 22:* Diagrama de Componentes “IAEntity”.

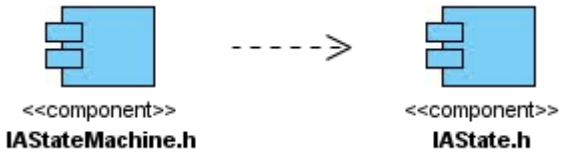


*Figura 23:* Diagrama de Componentes “StateMachine”.

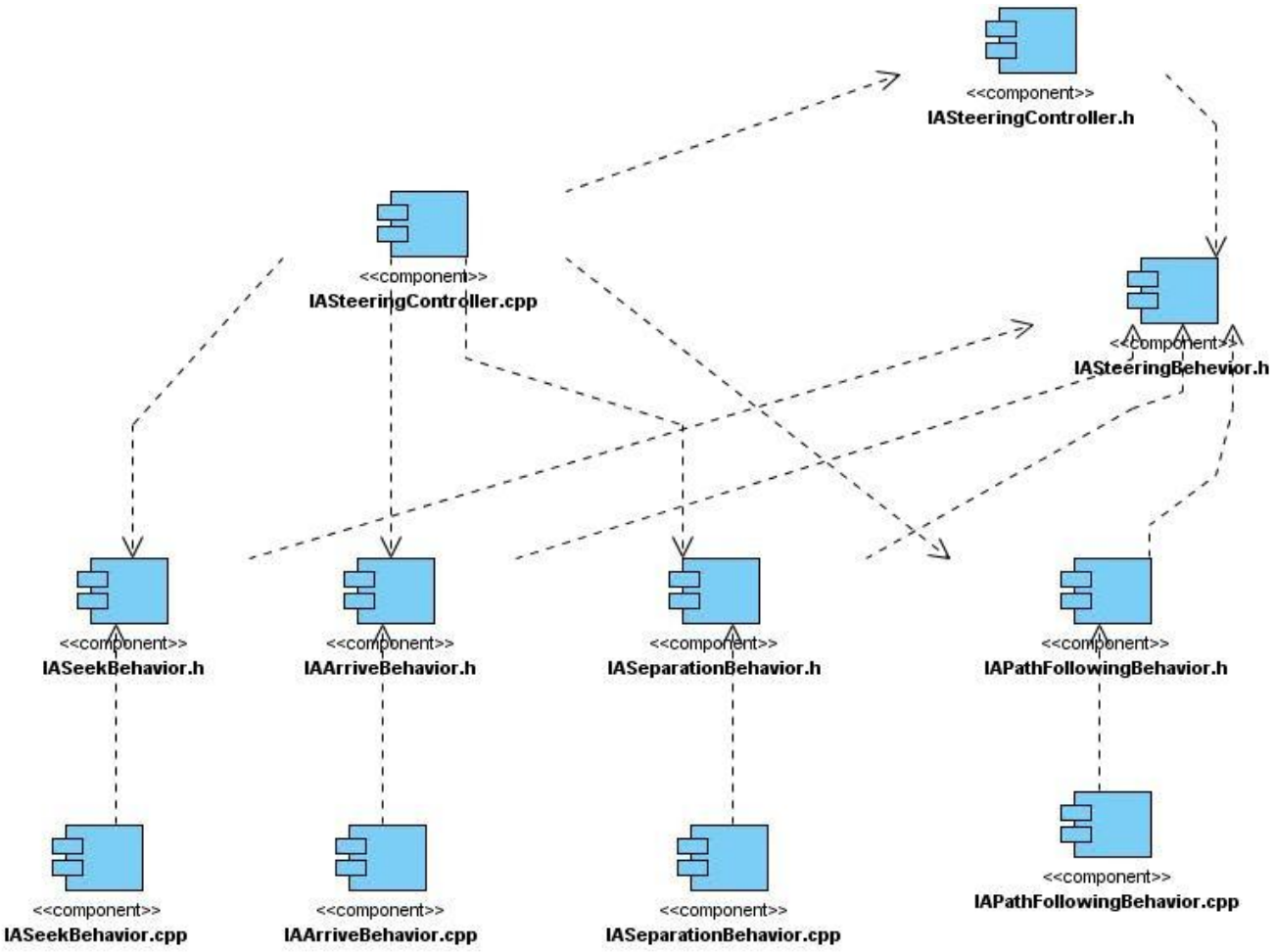


*Figura 24:* Diagrama de Componentes “Steering Behavior”.

# CONCLUSIONES

Para el cumplimiento de los objetivos y en concordancia con las exigencias hechas por el cliente, se requirió hacer un estudio de la situación actual en el desarrollo de IA para video-juegos así como las técnicas que se utilizan para este fin; para escoger cuales de estas se emplearían para el desarrollo de la solución propuesta y cumplir con las tareas planteadas.

Por lo tanto se puede concluir que con el desarrollo de este trabajo se le da cumplimiento a los objetivos planteados:

- Se realizó un sistema que cumple con las exigencias hechas por el cliente.
- Se elaboró un diagrama de clases modular que permite que se le agreguen otras técnicas y comportamientos de dirección.
- Se implementaron y se acoplaron al juego simulador de indicios aduaneros comportamientos grupales que se pueden observar en el primer nivel del juego.

# RECOMENDACIONES

En el desarrollo de este trabajo fueron tomadas en consideración muchas ideas, pero dadas las necesidades de atender aspectos de mayor importancia no todas fueron contempladas.

Por lo que se recomienda para el desarrollo futuro del sistema:

- Desarrollar un módulo de comunicación entre agente para que lo agentes tengan interacción social a través del intercambio de mensajes entre sí.
- Hacer uso de un algoritmo de búsqueda de caminos.
- Implementar alguna técnica para el aprendizaje para los agentes en el juego.
- Implementarles más comportamientos grupales e individuales.

# BIBLIOGRAFÍA


1. **Bourg, D. M y Seemann, G. 2004.** *AI for Game Developers.* s.l. : O'Reilly, 2004.
2. **Buckland, Mat. 2005.** *Programming Game AI by Example.* s.l. : Wordware Publishing, 2005.
3. Character Studio . [En línea] www.discreet.com.
4. **Duchi, Jordi y Tejedor, Heliodoro.** *Diseño y Programación de Videojuegos. Lógica de un videojuego.* s.l. : Universitat Oberta de Catalunya.
5. **Dybsand, Eric. 2003.** http://www.gamasutra.com/. [En línea] 22 de July de 2003. http://www.gamasutra.com/view/feature/2824/ai_middleware_getting_into_.php.
6. Game::AI++. [En línea] www.iagamedev.com.
7. IA-Implant . [En línea] www.ia-implant.
8. **Millington, Ian. 2006.** *Artificial Intelligence for Games.* s.l. : Academic Press, 2006.
9. Renderware A.I . [En línea] www.renderware.com.
10. **Reynolds, Craig. 1986.** *http://www.red3d.com/cwr/boids/.* [En línea] 1986. http://www.red3d.com/cwr/boids/.
11. **—. 1999.** Steering Behaviors For Autonomous Characters. *http://www.red3d.com/cwr/steer/gdc99/.* [En línea] 1999. http://www.red3d.com/cwr/steer/gdc99/.
12. **Russell, S y Norvig, P. 1995.** *Artificial Inteligence: A Modern Approach.* s.l. : Prentice Hall, 1995.
13. **Salvi, Jorge Luis. 2006.** *Desarrollo de una biblioteca de clases para el control de NPCs.* Rio de Janeiro : s.n., 2006.
14. **Schwab, B. 2004.** *AI Game Engine Programint.* s.l. : Charles River Media, 2004.
15. **Woodcock, Steven. 2001.** http://www.gameai.com/. [En línea] 2001.

# GLOSARIO

**IA**: Inteligencia Artificial

**NPC**: Carácter no personal. Hace referencia a los caracteres que no son controlados por el jugador sino por la computadora.

**FSM**: Máquina de estados Finitos.

**Movimiento autónomo**: Se refiere a un movimiento que se realiza con cierta autonomía.

**Steering Behavior**: Comportamiento de dirección.

**Comportamientos grupales**: Se refiere a ciertos comportamientos de dirección que toman como objetivo uno o más caracteres presentes en el juego para su ejecución.

**Waypoitns**: Puntos de camino. Estos son utilizados por los algoritmos de búsqueda.

**A***: Algoritmo de búsqueda muy utilizado por los desarrolladores de juego.

**A-Life**: Vida Artificial. Es una técnica de IA utilizada para simular el desarrollo social de algunos seres vivos.

# LISTA DE FIGURAS Y TABLAS

## Índice de Figuras

FIGURA 1: IMPORTANCIA DE LA IA EN LOS VIDEO JUEGOS. [SALVI. 2006] ..... 7
FIGURA 2: EJEMPLO DE UNA MÁQUINA DE ESTADO SIMPLE. ..... 10
FIGURA 3: ESCENA TOMADA DE STANLEY AND STELLA IN: BREAKING THE ICE. ..... 14
FIGURA 4: EJEMPLO DEL COMPORTAMIENTO SEPARACIÓN. ..... 17
FIGURA 5: EJEMPLO DEL COMPORTAMIENTO ALINEACIÓN. ..... 17
FIGURA 6: EJEMPLO DEL COMPORTAMIENTO COHESIÓN. ..... 18
FIGURA 7: MODELO DE DOMINIO. ..... 26
FIGURA 8: DIAGRAMA DE CASOS DE USOS. ..... 30
FIGURA 9: DIAGRAMA DE CLASES DEL DISEÑO. ..... 39
FIGURA 10: SUBSISTEMA STEERING BEHAVIOR. ..... 40
FIGURA 11: CU INICIALIZAR IA. ..... 41
FIGURA 12: CU ACTUALIZAR IA. ..... 41
FIGURA 13: CU CREAR MÁQUINA DE ESTADO. ..... 42
FIGURA 14: CU INICIALIZAR COMPORTAMIENTO DE DIRECCIÓN. ..... 42
FIGURA 15: CU ACTUALIZAR ENTIDAD DE JUEGO. ..... 43
FIGURA 16: ACTUALIZAR MÁQUINA DE ESTADO. ..... 43
FIGURA 17: CALCULAR FUERZA DE DIRECCIÓN. ..... 44
FIGURA 18: CAMBIAR ESTADOS. ..... 45
FIGURA 19: DIAGRAMA DE DESPLIEGUE. ..... 58
FIGURA 20: SUBPAQUETES DE COMPONENTES. ..... 59
FIGURA 21: DIAGRAMA DE COMPONENTES “IACONTROLLER”. ..... 60
FIGURA 22: DIAGRAMA DE COMPONENTES “IAENTITY”. ..... 60
FIGURA 23: DIAGRAMA DE COMPONENTES “STATEMACHINE”. ..... 60
FIGURA 24: DIAGRAMA DE COMPONENTES “STEERING BEHAVIOR”. ..... 61

# Índice de Tablas

TABLA 1: DESCRIPCIÓN DE LOS ACTORES DEL SISTEMA. ..... 29
TABLA 2: DESCRIPCIÓN DEL CU INICIALIZARIA EN FORMATO EXPANDIDO. ..... 31
TABLA 3: DESCRIPCIÓN DEL CU ACTUALIZARIA EN FORMATO EXPANDIDO. ..... 32
TABLA 4: DESCRIPCIÓN DEL CU FINALIZARIA EN FORMATO EXPANDIDO. ..... 32
TABLA 5: DESCRIPCIÓN DEL CU CREAR MÁQUINA DE ESTADOS EN FORMATO EXPANDIDO. ..... 33
TABLA 6: DESCRIPCIÓN DEL CU INICIALIZAR COMPORTAMIENTO DE DIRECCIÓN EN FORMATO EXPANDIDO. ..... 34
TABLA 7: DESCRIPCIÓN DEL CU ACTUALIZAR ENTIDADIA EN FORMATO EXPANDIDO. ..... 35
TABLA 8: DESCRIPCIÓN DEL CU ACTUALIZAR MÁQUINA DE ESTADOS EN FORMATO EXPANDIDO. ..... 35
TABLA 9: DESCRIPCIÓN DEL CU CALCULAR FUERZA DE DIRECCIÓN EN FORMATO EXPANDIDO ..... 36
TABLA 10: DESCRIPCIÓN DEL CU CAMBIAR ESTADOS EN FORMATO EXPANDIDO. ..... 37
TABLA 11: DESCRIPCIÓN DE CLASES “IACONTROLLER”. ..... 46
TABLA 12: DESCRIPCIÓN DE CLASES “IABASEENTITY”. ..... 47
TABLA 13: DESCRIPCIÓN DE CLASES “IAMOVINGENTITY”. ..... 48
TABLA 14: DESCRIPCIÓN DE CLASES “STATE”. ..... 49
TABLA 15: DESCRIPCIÓN DE CLASES “STATEMACHINE”. ..... 50
TABLA 16: DESCRIPCIÓN DE CLASES “STEERINGCONTROLLER”. ..... 51
TABLA 17: DESCRIPCIÓN DE CLASES “STEERINGBEHAVIOR”. ..... 52
TABLA 18: DESCRIPCIÓN DE CLASES “SEEKBEHAVIOR”. ..... 52
TABLA 19: DESCRIPCIÓN DE CLASES “ARRIVEBEHAVIOR”. ..... 52
TABLA 20: DESCRIPCIÓN DE CLASES “PATHFOLLOWINGBEHAVIOR”. ..... 53
TABLA 21: DESCRIPCIÓN DE CLASES “SEPARATIONBEHAVIOR”. ..... 53
TABLA 22: TIPOS DE DATOS. ..... 56