Universidad de las Ciencias Informáticas | FACULTAD 4
Centro de Entornos Interactivos 3D (VERTEX)

Universidad de las Ciencias Informáticas

Universidad de las Ciencias Informáticas

Vertex, Facultad 4

Módulo de inteligencia artificial para agentes inteligentes para Unity 3D.

Trabajo de diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Autor: Ernesto Andrés González Perdomo

Tutores: Ing. Rodobaldo Livan Saroza Ramírez

Ing. Roberto Ross Aguirre

Ing. Nelson Manuel González Martínez

Proyecto: VERTEX

Rol: Desarrollador, Analista

La Habana, Julio 2018

"Año 60 de la Revolución"

DECLARACIÓN DE AUTORÍA

Declaro ser autor de la presente tesis y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo. Para que así conste firmo la presente a los ____ días del mes de _____ del año_____.

_____ _____

Autor: Ernesto Andrés González Perdomo **Tutor:** Ing. Rodobaldo Livan Saroza Ramírez

_____ _____

Tutor: Ing. Roberto Ross Aguirre **Tutor:** Ing. Nelson Manuel González Martínez

DEDICATORIA

Dedico este trabajo a mi madre que se encuentra de misión y aunque no esté en estos momentos presente, siempre ha estado pendiente de mí cuando la he necesitado y cuando no.

A mi abuela por cuidarme y preocuparse por mí desde que tengo uso de razón.

A mi padre por entenderme y apoyarme en mis decisiones. Mejores padres que ustedes ni en sueños.

A mi hermano por estar constantemente ayudándome y aunque se haga el duro, yo sé que se preocupa por mí. A ver, una risita Carli, a ver.

A mis tías, Maritza, Haysi, Niurka, Gladys y por supuesto a mis lindas primas, Tahirí, Vero Vero y Vale. Al primi que siempre está cayéndome arriba para explicarle programación, pasa la universidad vago. A mi sobrinito Eduard y su hermanita o hermanito por nacer.

A una persona muy especial para mí que, aunque entró en mi vida hace poco más de 4 meses, la ha cambiado por completo. Elizabeth, esa eres tú por si no te enteras, gracias por permanecer a mi lado, gracias por tu paciencia, comprensión, amor y por darme a una super suegra.

A Bolilla, Lourdes, María y mi amigo Guelmis que más que un amigo es otro hermano, y aunque no está aquí hoy porque es un completo inútil que se le olvida todo, yo sí me acuerdo de él. Para mi son parte de mi familia y no me lloren ok, no me lloren.

AGRADECIMIENTOS

Quiero agradecer a todas esas personas que a lo largo de la carrera me han apoyado y ayudado.
A mis amigos y todos mis compañeros de carrera, sin sus lecciones minutos antes de las pruebas no hubiese aprobado unas cuantas asignaturas.
A mis profesores durante estos cinco años, por su empeño, paciencia y dedicación.
A mis tutores por su ayuda, en especial al profesor Livan por ayudarme en los momentos finales, por su tutorización y paciencia. A la profesora Alina por preocuparse por mí constantemente.
Al comité de turbeo que prestó solidaridad con la causa suspendiendo las actividades hasta nuevo aviso.
A la Dra. Isa y su esposo Ramoncito por su ayuda y preocupación.
A mi familia por estar pendiente de mi y apoyarme en todas las decisiones que he tomado. Gracias por su apoyo, comprensión y fuerza a lo largo de todos estos años.
En fin a todas las personas que me han ayudado en lo más mínimo, saben que los aprecio a todos.
¡Muchas gracias a todos!

RESUMEN

En la actualidad informática, muchos han sido los avances que han tenido lugar en las diferentes materias y ramas de la misma, no quedando exenta la Inteligencia Artificial (IA), y dentro de ella, uno de sus principales ejes: los sistemas multiagentes. Los mismos se encargan de estudiar el modelo y comportamiento de varios agentes en un entorno, y ofrecen un sinnúmero de prestaciones de manera intuitiva y segura en aplicaciones que se encuentran dentro de las siguientes ramas: educación, salud y entretenimiento. Dentro de la sociedad del entretenimiento los videojuegos son un ejemplo de estas aplicaciones. La IA dentro de un videojuego permite crear comportamientos inteligentes a entidades que lo necesiten, pero su desarrollo resulta, en la mayoría de los casos, difícil de implementar debido a la falta de herramientas que faciliten su integración y conectividad entre comportamientos. Este trabajo propone analizar los comportamientos de los agentes inteligentes en un sistema multiagentes. Se realizó una investigación sobre las diversas técnicas de comunicación virtual y de toma de decisiones existentes. Se obtuvo un módulo que brinda las funcionalidades necesarias para crear agentes, además de crear estados que podrán ser asociados a los agentes mediante una arquitectura de máquina de estados finitos que permite configurar los distintos estados que puede tener un agente en un videojuego y sus transiciones. Con el uso del módulo desarrollado se facilita la programación de la parte inteligente de los juegos, reduciendo, por lo tanto, el tiempo de desarrollo de estas aplicaciones.

Palabras clave: sistemas multiagentes, agentes inteligentes, inteligencia artificial, toma de decisiones, comunicación virtual.

Índice

Índice de figuras

Índice de tablas

Introducción

Un videojuego es un software creado para el entretenimiento en general y basado en la interacción entre una o varias personas y un aparato electrónico que lo ejecuta; estos dispositivos electrónicos pueden ser una computadora, una máquina arcade, una videoconsola, un teléfono móvil, y son conocidos como "plataformas" [1]. En los últimos años los videojuegos se han convertido en una gran industria que ha experimentado altas tasas de crecimiento, debido al desarrollo de la computación y la capacidad de procesamiento. Las razones de su popularidad están dadas por el nivel de realismo en la representación de los escenarios y por el poder de interacción con los usuarios. Este último desempeña un papel preponderante en el nivel de aceptación de un producto, ya que los usuarios finales no sólo necesitan de un sistema gráfico con un alto nivel de detalles y similitud al mundo real, sino además que sea capaz de retar su inteligencia.

Cuando se habla de la inteligencia de un videojuego se hace referencia a uno de los principales componentes en el desarrollo del mismo: la inteligencia artificial, la cual es la encargada de la simulación de comportamientos de los personajes no manejados por el jugador, por sus siglas en inglés (NPC). La inteligencia artificial o IA es uno de los puntos más importantes a la hora de estudiar, criticar y desarrollar un videojuego. Es especialmente valiosa en el mundo de los videojuegos, ya que la experiencia del juego depende exclusivamente de la calidad.

Ya es común que los videojuegos incorporen nuevas características en sus sistemas de inteligencia artificial. Por ejemplo, existen videojuegos como el caso de “GTA V” y “Sombras de Mordor”, que están compuestos por entidades que actúan sin la intervención directa del jugador y que además son capaces de tomar sus propias decisiones, comunicarse y reaccionar ante los diferentes cambios que puedan ocurrir en el entorno. A estas entidades se les conoce en el campo de la inteligencia artificial como agentes inteligentes.

Para que un agente se comporte de manera racional, debe conocer las características del entorno que lo rodea y poseer la capacidad de sentir y de ver a los elementos que se encuentren dentro de su campo de visión. Cuando existen múltiples agentes inteligentes que interactúan entre sí en un mismo entorno se está en presencia de un sistema multiagente (SMA). Estos sistemas necesitan de un sistema de percepcion para adquirir la información necesaria del medio que rodea a los agentes. Mediante la obtención de estos datos, los agentes pueden tomar las decisiones necesarias para poder resolver un problema determinado; para ello es preciso que puedan intercambiar información con otros agentes del entorno.

En Cuba el desarrollo de la informática se ha incrementado en los últimos años producto de la necesidad de lograr un crecimiento tecnológico. En este sentido, Cuba ha identificado desde temprano la necesidad de dominar e introducir en la práctica social las Tecnologías de la Información y las Comunicaciones (TICs) y lograr una cultura digital como una de las características

imprescindibles del hombre nuevo, lo que facilitaría a la sociedad cubana acercarse más hacia el objetivo de un desarrollo justo, equitativo, sostenible y alcanzable. Cuba está consciente de que una sociedad para ser más eficaz, eficiente y competitiva debe aplicar la informatización en todas sus esferas y procesos.

En la esfera del entretenimiento el país no está alejado del desarrollo de videojuegos y un ejemplo de ello es el centro Vertex de la Facultad 4 de la Universidad de Ciencias Informáticas fundado en el año 2014; el cual tiene como uno de sus objetivos crear videojuegos con fines educativos y medicinales. Este centro ha desarrollado varios videojuegos utilizando como motor gráfico Unity3D, ejemplos son: La Chivichana, Súper Claria, Aventuras en la Manigua, La Neurona, Villa Tesoro, Especies Invasoras, Caos Numérico, entre otros. Cada uno de estos videojuegos tienen inteligencia artificial incorporada, ya sea utilizando técnicas de locomoción, de comportamiento predefinidos o técnicas de búsqueda de caminos (*pathfinding)*, pero carecen de un marco de trabajo que se adapte a sus necesidades.

A la hora de desarrollar un videojuego, hay que tener en cuenta que existen diferentes tipos, géneros y subgéneros de videojuegos, donde cada producto muestra sus propias características y necesidades. En la mayoría de los casos no podemos comparar la IA de un juego de estrategia por turnos con la de un juego de acción en primera persona, pero existen técnicas que si se pueden emplear e incluso modificar para el uso de un videojuego en específico.

Hasta el momento, en los videojuegos desarrollados por el centro, no se ha implementado una técnica de inteligencia artificial en la que los agentes del videojuego interactúen y se comuniquen no solamente con el jugador, sino también con personajes no jugadores (NPC) del sistema. Implementar una de las técnicas de IA puede resultar un proceso muy complejo si se tienen que desarrollar en cada *software* por separado. Unity brinda módulos para el desarrollo de técnicas de inteligencia artificial, pero estas son privativas y de alto costo para la Universidad. Por lo que, realizar un módulo genérico, estable y reutilizable, que incluya técnicas de toma de decisiones y comunicación virtual puede ahorrar a los equipos de desarrollo de videojuegos gran parte del trabajo y fomentar la inclusión de estas funcionalidades en los videojuegos que están por hacer.

Después de analizar la situación se identifica el siguiente **problema científico:**

¿Cómo incorporarle técnicas de toma de decisiones y comunicación virtual a los agentes empleados en los videojuegos que utilizan sistemas multiagentes?

El cual delimita como **objeto de estudio:** Las técnicas de toma de decisiones y comunicación virtual en un sistema multiagente.

El **objetivo general** de este trabajo es: Desarrollar un *plugin* en Unity3D que facilite las funcionalidades de comunicación virtual y de toma de decisiones para un sistema multiagente.

Lo que determina que se proponga como **campo de acción:** Las características funcionales

relacionadas con la comunicación virtual y la toma de decisiones en los sistemas de inteligencia artificial para videojuegos.

Luego, para darle cumplimiento al objetivo general se plantean como **tareas investigativas:**

- Elaborar el marco teórico de la investigación a partir del estado del arte existente actualmente sobre el tema de investigación.
- Indagar sobre los sistemas multiagentes y la inteligencia artificial en los videojuegos.
- Investigar técnicas existentes para la comunicación entre agentes y toma de decisiones.
- Estudiar y desarrollar *plugins* en Unity3D.
- Diseñar e implementar el módulo haciendo uso de la(s) técnicas investigadas.
- Validar las funcionalidades del software mediante el uso de un Demo.

Métodos Científicos:

Método Teórico:

Se emplea el método **analítico – sintético** como método teórico, ya que se analizan las teorías y documentos, posibilitando la extracción de los elementos más importantes que se relacionan con el objeto de estudio. Se emplea este método porque se va a desarrollar una investigación sobre las diferentes técnicas de comunicación y de toma de decisiones en un sistema multiagente, además se realizará un análisis de grandes volúmenes de documentos que hacen referencia a la comunicación entre agentes y la toma de decisiones, así como las informaciones y estudios relacionados con estos temas.

Histórico-lógico:

Permite estudiar la evolución y desarrollo de las técnicas que existen para lograr la toma de decisiones y el intercambio de información en los agentes, pudiendo conocer el estado actual de las mismas.

Métodos empíricos:

- **La consulta de fuentes de información:** Permite realizar un estudio actual de las distintas bibliografías que existen del tema tratado.
- **Observación:** permite adquirir información necesaria en cualquiera de las fases de la investigación, el acercamiento a la realidad y la determinación de la posible solución del problema desde diferentes ángulos.

Posibles resultados:

Como resultado de este trabajo se pretende la creación de un *plugin* que permita la comunicación entre agentes y toma de decisiones en videojuegos. Posibilitando que los proyectos productivos de la Facultad 4 interesados en aplicar estas funcionalidades puedan basar su desarrollo a partir de este *plugin*.

El contenido de este documento está estructurado en tres capítulos, así como las correspondientes secciones de las recomendaciones, bibliografía, el glosario de términos, glosario de abreviaturas y los anexos; organizados de la siguiente forma:

Capítulo 1 : Fundamentación teórica

Se exponen los principales conceptos relacionados con los sistemas multiagentes, sistemas de percepción, con la comunicación virtual y la toma de decisiones entre agentes inteligentes.

Capítulo 2: Propuesta y descripción de la solución

Se ofrece una descripción de las características que presentará el sistema como solución al problema científico planteado y de temas específicos relacionados con la propuesta de solución. Serán definidas, la modelación del negocio y la captura de los requisitos funcionales y requisitos no funcionales con la generación de sus respectivos artefactos.

Capítulo 3: Resultado de la investigación

Se muestran los resultados de la investigación y se abordan los temas del diseño y de la implementación del sistema. Se valida la propuesta de diseño mediante un prototipo de características seleccionadas. Se describen dichas características y su proceso de desarrollo. Al concluir el capítulo se tendrá un *plugin* validado y funcional.

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

Introducción

En el presente capítulo se realiza una evaluación del estado del arte referente al estudio de los sistemas multiagentes y de algunas de las técnicas que existen para poder lograr el intercambio de información y la toma de decisiones entre los agentes inteligentes en un videojuego.

1.1 Inteligencia Artificial

En 1956, John McCarthy define la Inteligencia Artificial (IA) como “La ciencia e ingenio de construir máquinas inteligentes, especialmente programas de cómputo inteligentes” [2]. Hoy, casi 60 años después, esta definición sigue siendo tan válida como entonces. Existen otras definiciones, unas más inclinadas hacia alguna rama en especial de la IA que otras, pero en resumen se puede decir que la IA trata de: “Desarrollar sistemas que piensen y actúen racionalmente”. Cuando se habla de IA en videojuegos, se suele referir al conjunto de algoritmos y sistemas que gobiernan a los enemigos controlados por la máquina [3].

En los últimos años la IA ha ido evolucionando, quizás con mayor celeridad que otras disciplinas, motivado probablemente por su propia inmadurez. Todo esto ha llevado a que la IA actualmente abarque una gran cantidad de áreas, desde algunas muy generales como razonamiento y búsqueda, a otras más específicas como los sistemas expertos, sistemas de diagnóstico, sistemas multiagentes, etc. Se puede afirmar que la IA puede ser aplicada hoy en día a infinidad de disciplinas científicas y es que la IA es susceptible de aparecer allí donde se requiera el intelecto humano [4].

En la actualidad, dentro de la IA ha surgido un nuevo paradigma, es nada más y nada menos que el “paradigma de agentes”, que esta teniendo un gran auge entre los investigadores. Este paradigma se centra en el desarrollo de entidades que puedan actuar de forma autónoma y razonada. Si se ve a la IA desde un punto que la presenta como el medio para desarrollar sistemas que piensen y actúen racionalmente, se puede decir que la IA en conjunto trata de construir a estos agentes autónomos e inteligentes. Estos, a su vez, permiten abordar, de una manera mas apropiada la construcción de sistemas inteligentes mas complejos aplicados a diversos campos.

En muchos de los casos, los agentes no son desarrollados de forma independiente sino como entidades que constituyen un sistema multiagente. Algunos autores definen a los agentes como los nodos inteligentes por los cuales están compuestos los SMA [4].

1.2 Sistemas Multiagentes

Los agentes se pueden ver de manera aislada, incluso hay varias aplicaciones ya realizadas donde interviene un agente resolviendo determinado problema, sin embargo la potencialidad de este paradigma está en los sistemas multiagentes, o sea, en la interrelación entre agentes dentro de un

entorno específico [5].

Los sistemas multiagentes (SMA) son sistemas compuestos por nodos o elementos inteligentes de IA donde la conducta combinada de dichos elementos, produce un resultado en conjunto inteligente, comunicándose por medio de mecanismos basados en el envío y recepción de mensajes [6].

Los SMA están formados por varios aspectos fundamentales (Ferber, 1999):

1. **Un entorno.**
2. **Un conjunto de objetos.** Se encuentran integrados con el entorno, es posible en un momento dado asociar uno de estos objetos con un lugar en el entorno. Estos objetos son pasivos, pueden ser percibidos, creados, destruidos y modifiados por agentes.
3. **Un conjunto de agentes.** Se consideran objetos especiales que representan las entidades activas del sistema. (Cumpliendo las características antes expuestas).
4. **Un conjunto de relaciones.** Unen a los objetos, y, por lo tanto, agentes.
5. **Un conjunto de operaciones.** Hacen posible que los agentes perciban, produzcan, consuman, transformen y manipulen objetos.
6. **Operadores.** Representan la aplicación de operaciones sobre el mundo y la reacción de éste al ser alterado. Estos operadores se pueden entender como las leyes del universo.

1.2.1 Agentes

Los nodos o elementos inteligentes son los llamados agentes. Los cuales son la base de la construcción de los SMA y son vistos como entidades inteligentes que pueden percibir su ambiente a través de sensores. Éstos son capaces de evaluar tales percepciones y tomar decisiones por medio de mecanismos de razonamiento sencillo o complejo. (**Figura 1**)

Muchos expertos han dado su criterio en relación con dicho término y de acuerdo a la situación en que se encuentran; algunos de los cuales se mencionan a continuación:

- “Un agente es una entidad que percibe y actúa sobre un entorno” (Russell, 1996),.
- “Los agentes son sistemas computacionales que habitan en entornos dinámicos complejos, perciben y actúan de forma autónoma en ese entorno, realizando un conjunto de tareas y cumpliendo objetivos para los cuales fueron diseñados” [7].
- “Un agente es un sistema situado en alguna parte de un entorno que percibe dicho entorno y actúa en el en beneficio de su propia agenda, el efecto de su actuación se nota en el entorno” [6].

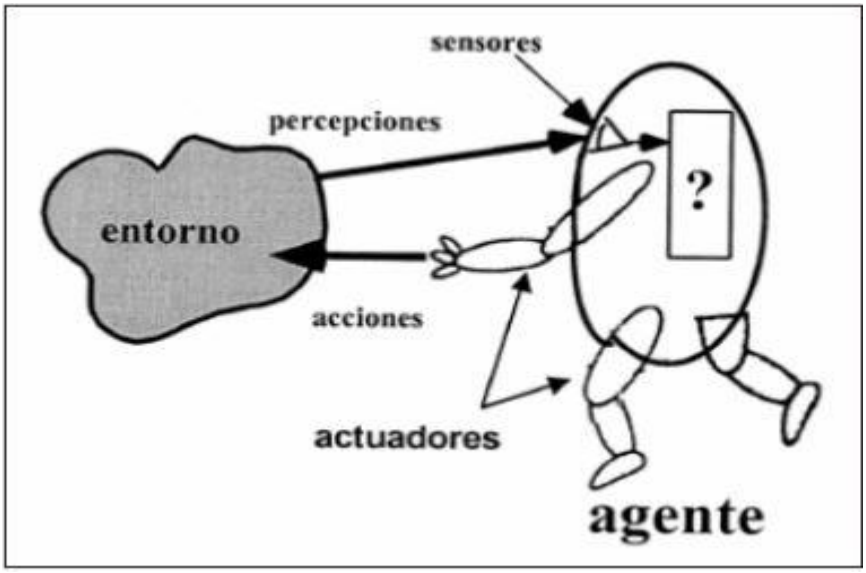

Figura 1 Visión esquemática de un agente [8]

Pero todos concuerdan en que presentan determinadas características que son importantes destacar.

1.2.2 Características de un agente

En [9] se explican las características de los agentes, que vienen dadas por una serie de calificativos, los cuales denotan ciertas propiedades a cumplir por los mismos:

- **Reactivo:** Los agentes deben ser capaces de percibir el ambiente que los rodea y responder a tiempo a los cambios en él, a través de sus acciones.
- **Pro-activos:** Deben exhibir un comportamiento orientado por sus metas, tomando la iniciativa para satisfacer sus objetivos.
- **Social:** Deben ser capaces de interactuar con otros agentes con miras a lograr la satisfacción de sus objetivos.

Es por estas razones que en el desarrollo de esta investigación, se considerará a un agente como: una entidad inteligente, capaz de percibir el entorno que lo rodea por medio de mecanismos basados en el envío y recepción de mensajes; interactuar con él y con otros agentes y tomar las decisiones que sean necesarias para darle solución a los problemas u objetivos que se le puedan presenta [9]. Estos agentes, para poder llevar a cabo el cumplimiento de sus tareas y objetivos, necesitan poder intercambiar información entre sí, dándole a este proceso el nombre de comunicación virtual. Sin la cual no llegarían a resolver los problemas y objetivos que puedan tener definidos en el medio que los rodea [9].

Según lo planteado anteriormente, para que el resultado de esta investigación cumpla con su objetivo se hace necesario realizar un análisis sobre las diversas técnicas de comunicación entre

agentes que existen y seleccionar cual o cuales de ellas responden a los resultados que se pretenden alcanzar con esta investigación.

1.2.3 Técnicas de comunicación entre agentes

En la actualidad son variadas las formas de lograr una comunicación entre varias personas gracias a la utilización de los medios informáticos, tales como los ordenadores. Al intercambio de información mediante esta vía se le llama comunicación virtual, pues se emite una gran cantidad de información mediante el uso de la red sin la participación directa de seres humanos.
Hoy día existen diferentes formas de comunicación virtual, dentro de ellas se pueden mencionar:

- Correo electrónico.
- Mensajería instantánea.
- Videoconferencia.
- Chats.
- Foros.
- Blogs.

La comunicación virtual, dentro de la IA, permite que los elementos que se encuentran dentro de un entorno determinado puedan comunicarse entre sí sin la necesidad de que intervenga algún ser humano [4]. Esta comunicación es definida, en este contexto, como la transmisión de cierto contenido informativo a un receptor mediante el uso de una señal. Forma base de la coordinación entre los agentes, pues puede darse el caso de que surja algún conflicto entre dos o más agentes, y es por eso que se hace necesaria la comunicación entre los mismos.
Dentro de las técnicas de comunicación que se describirán y permiten que dos o más agentes se comuniquen están:

- Sistemas de Pizarra (*BlackBoard*)
- Sistemas de mensaje/diálogo
- Sistemas Basados en Disparadores (*Trigger*)
- Máquinas de Estado Finitos (MEF)
- Sistemas Basados en Reglas (SBR)
- Lógica Difusa

1.2.3.1 Sistemas de Pizarra (BlackBoard)

Es uno de los sistemas más usados en la IA. La pizarra es considerada una estructura de datos que es utilizada como mecanismo general de comunicación entre las múltiples fuentes de conocimientos, siendo gestionada y arbitrada por un controlador. Esta pizarra es manejada como un área de trabajo común, donde los agentes pueden compartir cualquier tipo de información con

los demás agentes del entorno. Es un dispositivo para compartir información con múltiples lectores y escritores (**Figura 2**), pues permite que cada agente trabaje en su problema, escoja de la misma lo que necesite para resolverlo y lo publique una vez encontrada su solución para que sirva de retroalimentación a los demás agentes del entorno.

En este tipo de pizarras no existen áreas privadas, los agentes pueden escribir la información que deseen y acceder a la misma cuando estimen necesario, trayendo como consecuencia que pueda ocurrir una sobrecarga a la pizarra según la cantidad de agentes que intenten acceder a la misma vez.

Para lograr la solución a este problema, se haría necesario una nueva pizarra donde cada uno de los agentes del entorno tendría asignado su propio espacio para publicar información en dicha pizarra y contarían de permisos totales de acceder, pero sin modificar la información que los demás hayan publicado. [9]

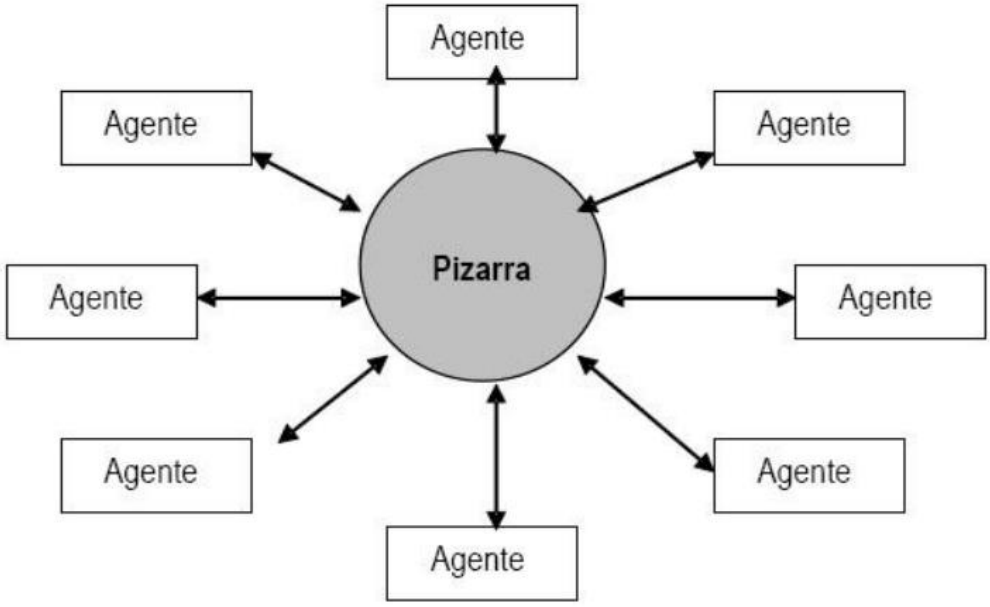

Figura 2 Estructura de un sistema de pizarra [8]

Este sistema tiene como ventajas:

- Es muy útil cuando el problema a resolver es extremadamente grande o no se tiene un sistema completo del sistema a resolver.
- No existe garantía, en la mayoría de los casos, de que se alcanzará una solución, pero si ésta se encuentra; es la mejor de todas.
- Cada agente tiene sus propios objetivos, aunque desconoce los objetivos de los demás y la solución del problema, pero puede llegar a obtener la información que necesita por el intercambio que pueda realizar con sus demás compañeros [9, p. 9].

Presenta dentro de sus desventajas:

- Es una técnica ineficiente, puesto que no existe una cota exacta respecto al tiempo de cómputo necesario para resolver el problema, pero se puede adaptar a problemas en los que se conocen bien sus características. Así que esta desventaja se puede convertir en una gran ventaja en dependencia de las características del sistema en cuestión.
- Es difícil obtener una traza de los pasos que llevaron a la solución, es decir, no ofrece explicaciones a la hora de llevar a cabo la realización del proceso [10].

1.2.3.2 Sistemas de mensaje/diálogo

En las técnicas basadas en mensajes, los mensajes que los agentes intercambian con otros agentes pueden ser utilizados para establecer comunicaciones y mecanismos de cooperación utilizando protocolos definidos. Este sistema empieza a funcionar cuando un agente, conocido como emisor, transfiere un mensaje específico a otro agente, denominado receptor (**Figura 3**). Al contrario de los sistemas de pizarra, aquí los mensajes son intercambiados directamente entre dos agentes, donde los demás agentes son incapaces de leer dicho mensaje si no va específicamente dirigido a ellos, siendo ésta su principal ventaja o desventaja desde el punto de vista que sea analizada la situación. [9, p. 9]

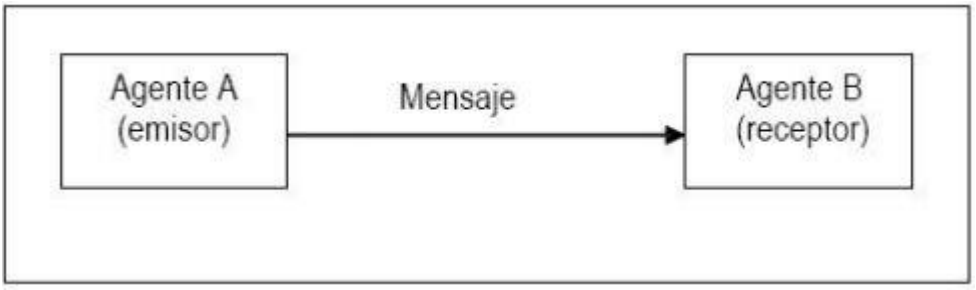

Figura 3 Principio de la transmisión de un mensaje [8]

1.2.3.3 Sistemas Basados en Disparadores (Trigger)

Estos sistemas constan de dos efectos esenciales en el mundo virtual:

- ✓ Realizar un seguimiento de los acontecimientos que los agentes puedan responder en el mundo.
- ✓ Minimizar la cantidad de agentes en proceso para responder a dichos eventos [10].

Un disparador o *trigger* puede ser cualquier estímulo que requiera de un agente para responder. En un entorno determinado, los factores desencadenados podrían ser cualquier elemento audible o visible, que afecta el comportamiento de los agentes, tales como: disparos de armas de fuego, explosiones, enemigos cercanos, ruidos de las puertas al cerrarse o los cadáveres [9, p. 10].
Los factores desencadenantes también pueden emanar de objetos inanimados que los agentes necesitan conocer, como palancas y paneles de control, para ello es necesario definirle a cada

agente qué tipo de factores desencadenantes son de interés para ellos.

Un disparador se define por un conjunto de banderas enumeradas del mismo tipo, y un conjunto de variables que describen sus parámetros (**Figura 4**). La combinación de banderas y de bits, permite a un agente especificar cuál es la acción que va a realizar. Un agente puede alternar y desactivar banderas e ignorar temporalmente determinados tipos de activación [9, p. 10].

```
enum TipoDisparador
{
        K_Ninguno  = 0,
        K_Explosión = (1 << 0),
        K_EnemigoCerca = (1 << 1),
        K_Disparo = (1 << 2),
        ...
};
```

Figura 4 Ejemplo de los estados de las banderas [8]

El beneficio de un sistema centralizado de activación, como también se les conoce, es la verificación de la prioridad antes de entregar la información al agente, de esta forma; el agente sólo responde a los procesos de mayor prioridad dentro del entorno. En un sistema centralizado los factores desencadenantes son registrados cuando se producen los acontecimientos. Para cada agente, el sistema utiliza una serie de pruebas para determinar si el agente está interesado en cualquier factor desencadenante existente. Estos factores son ordenados por prioridad, a fin de que el agente pueda responder a la acción más importante en cualquier instante. Ejemplo: en un juego, si un enemigo está de pie delante de un agente, éste no se preocupa por los sonidos que aparezcan en la distancia, por lo que su prioridad número uno pasa a ser el enemigo que está delante de él.

Los factores desencadenantes pueden ser utilizados para alertar a los agentes de los sucesos del entorno, como pueden ser: disparos de armas de fuego y explosiones que ocurran. Un agente herido puede usar una forma dinámica de colocar una señal de activación cuando necesita ayuda de sus aliados. Un agente puede, incluso, detectar con lo que el jugador ha interactuado a través de objetos interactivos en el mundo virtual, tales como puertas y otras acciones que se disparan en el momento que son llamadas. [9, p. 11]

Este sistema presenta como ventajas [9, pp. 11-12]:

- Verifica la prioridad antes de entregar la información al agente.
- Pueden ser generados a partir de una variedad de fuentes, incluyendo códigos de juego, *scripts*, comandos de consola, animación y fotogramas clave.
- Permiten implementar programas basados en paradigma lógico (ejemplo: sistemas expertos).
- Son sistemas muy eficientes y rápidos, por lo que se garantiza encontrar siempre una solución.

Desventajas [9, p. 12]:

- No pueden ser llamados directamente.
- Dada su complejidad son difíciles, pero no imposibles de programar.

1.2.3.4 Máquinas de Estados Finitos

Una Máquina de Estado Finito (MEF) o también conocidas como Autómatas de Estados Finitos, son modelos de comportamiento de un sistema, con un número limitado de estados o condiciones predefinidas y de transiciones entre dichos estados. Además, pueden definir un conjunto de condiciones que determinan cuándo el estado debe cambiar. Son simples y lo suficientemente poderosas para manejar una amplia variedad de situaciones.

Las MEF son las técnicas más usadas en la implementación de los juegos en la IA, permitiendo una mejor comunicación entre los agentes que interactúan en dichos juegos. Esta técnica de cooperación representa una descripción no lineal de cómo los objetos pueden cambiar de estado con el tiempo, posiblemente en respuesta a los eventos en su entorno. Además, se prestan para ser esbozadas con rapidez durante el diseño y prototipo, y mejor aún, pueden ser implementadas de manera fácil y eficientemente [10].

Las MEF están compuestas por 4 elementos principales [8, p. 30], estos son:

1. **Estados:** definen el comportamiento y pueden producir acciones.
2. **Transiciones de estado:** son movimientos de un estado a otro.
3. **Reglas o condiciones:** deben cumplirse para permitir un cambio de estado.
4. **Eventos de entrada:** son externos o generados internamente, que permiten el lanzamiento de las reglas y las transiciones.

La **Figura 5** muestra un comportamiento basado en MEF donde los estados describen cómo los agentes van a actuar, y las transiciones entre estados representan las decisiones que tomarán éstos en el próximo paso.

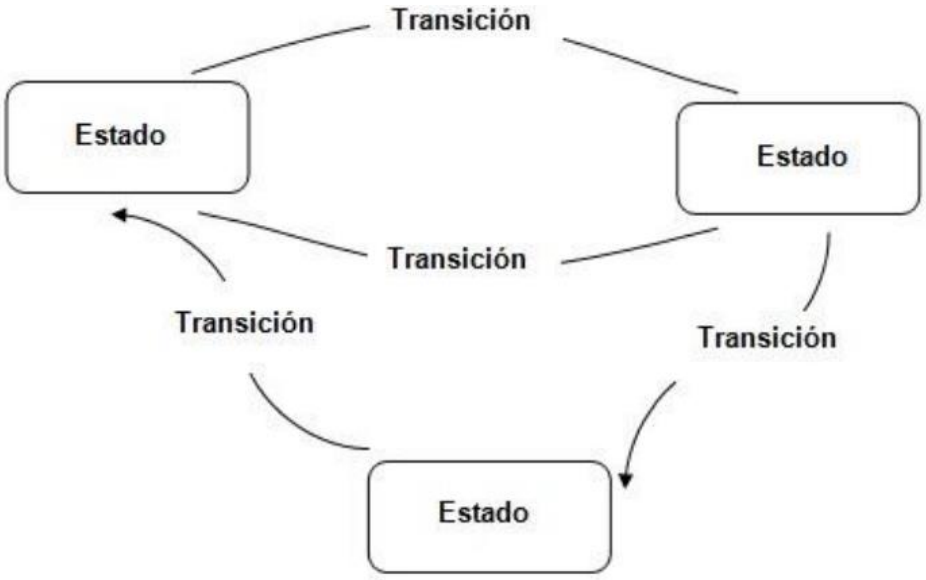

Figura 5 Ejemplo de una MEF como diagrama [8]

Ventajas [9, pp. 13-14]:

- Su simplicidad hace fácil, para los desarrolladores sin experiencia, realizar la implementación con poco o nada de conocimiento extra (fácil entrada).
- Predictibilidad, dado un grupo de entradas y un estado actual conocido, puede predecirse la transición de estados, facilitando la tarea de verificación.
- Dada su simplicidad, son rápidos de diseñar, rápidos de implementar y rápidos de ejecutar.
- Son relativamente flexibles.
- Pueden manejar una amplia variedad de situaciones.
- Presentan varios modos de implementarse partiendo de su topología.
- La transferencia desde una representación abstracta del conocimiento a una implementación es fácil.
- Bajo uso del procesador; apropiado para dominios donde el tiempo de ejecución está compartido entre varios módulos o subsistemas. Sólo el código del estado actual ha de ser ejecutado, además de requerir de un poco de lógica para determinar el estado actual.
- Es fácil determinar si se puede llegar o no a un estado, en las representaciones abstractas, resulta obvio si se puede o no llegar a un estado desde otro, y qué requerimientos existen para hacerlo.

Desventajas [9, p. 14]:

- La naturaleza predecible de las MEF puede no resultar conveniente en algunos dominios, como los juegos por ordenador.
- Si se implementa un sistema grande, usando esta técnica, puede ser difícil de administrar y mantener sin un buen diseño.

- No es apropiado para todos los dominios de problema, sólo debe ser usado cuando el comportamiento de un sistema puede ser descompuesto en estados separados con condiciones bien definidas para las transiciones, lo que significa que todos los estados, transiciones y condiciones deben ser conocidos y estar bien definidos con anterioridad.
- Las condiciones para las transiciones entre estados son rígidas, significando que están fijadas.

1.2.3.5 Sistemas Basados en Reglas

Los Sistemas Basados en Reglas (SBR) es una técnica de comunicación que trabajan mediante la aplicación de reglas, comparación de resultados y aplicación de las nuevas reglas. También pueden trabajar por inferencia de lógica dirigida, empezando con una evidencia inicial en una determinada situación y dirigiéndose hacia la obtención de una solución, o bien con hipótesis sobre las posibles soluciones y volviendo hacia atrás para encontrar una evidencia existente que apoye una hipótesis en particular.

La programación basada en las reglas es una de las técnicas más comúnmente usadas para crear bases de conocimiento. Las reglas se utilizan para representar conocimiento factual o heurístico, que especifican en un sistema las acciones que se realizarán para una situación dada. Una regla se compone de una parte "si" y una parte "entonces", donde la parte "si" es una serie de patrones que especifican los hechos (o los datos) que hacen que se aplique una regla. Juntos con los datos, los hechos también están presentes en la base de conocimientos. Estos hechos representan las declaraciones verdaderas que se utilizan para activar las reglas [8, p. 18].

A continuacion se mencionaran algunas de las ventajas y desventajas de estos sistemas:

Ventajas:

- Imitan a las personas de cierta manera, tienden a pensar y razonar dado un juego de hechos conocidos y su conocimiento sobre el dominio del problema particular.
- Son fácil de programar y manejar ya que pueden codificarse las reglas en cualquier orden.
- Son un modelo relativamente simple que puede adaptarse a cualquier número de problemas.

Desventajas:

- El conocimiento es difícil de extraer de los expertos humanos.
- Pueden existir decisiones que sólo son de competencia para un ser humano y no una máquina.
- Pueden producirse errores en la base de conocimientos, y conducir a decisiones equivocadas.
- No siempre puede explicar su lógica y razonamiento.

- Si hay demasiadas reglas el sistema puede ponerse difícil, o sea, puede ser difícil de mantener.

1.2.3.6 Lógica Difusa

La Lógica Difusa es un medio de mostrar los problemas de una manera similar a la forma en que los seres humanos pueden resolverlos, se deriva de la teoría de los conjuntos difusos, y su razonamiento se basa en la aproximación a la percepción humana.
La idea de esta técnica es resolver los problemas de una manera imprecisa como lo hacen los seres humanos. Permite plantear y resolver problemas lingüísticos utilizando términos similares a los que un ser humano podría usar.
Esta técnica pertenece a conjuntos cuyos valores varían entre 0 y 1 (ambos incluidos), y permite usar conceptos imprecisos como ligeramente, bastante y mucho. También posibilita la inclusión parcial en un conjunto.
Todos los términos de la Lógica Difusa pueden ser cierto, pero en diversos grados. Si se dice que algo es verdad a grado 1, es absolutamente cierto. Una verdad a grado 0 es absolutamente falsa. Por lo que, se puede tener algo que sea absolutamente cierto, o absolutamente falso, pero también se puede tener algo que esté entre 0 y 1. Incluye también, el uso de términos lingüísticos intuitivos por ejemplo, cerca, lejos, muy lejos, y así sucesivamente. Esto hace que el sistema sea más legible, fácil de entender y mantener [8, p. 20].
A continuacion se mencionarán algunas de las ventajas y desventajas de usar la lógica difusa [8]:
Ventajas:

- Facilidad de implementación y la velocidad en obtener una salida con una gran fiabilidad.
- Permiten solucionar gran parte de los problemas de control automático de una manera sencilla sin necesidad de conocer un modelo matemático que lo pueda controlar.
- Son sistemas que han dado buenos resultados en procesos no lineales y de difícil modelización.
- El modo de funcionamiento es similar al comportamiento humano.
- Es una forma rápida y económica de resolver un problema.

Desventajas:

- En las redes neuronales se precisa de un tiempo de aprendizaje para obtener los mejores resultados en la salida. (Al igual que ocurre con los humanos).
- Ante un problema que tiene solución mediante un modelo matemático, obtenemos peores resultados empleando esta técnica.

Las aplicaciones de la Lógica Difusa no se limitan a los sistemas de control. Esta técnica se puede utilizar para la toma de decisiones así como las aplicaciones. Un ejemplo típico incluye el análisis

de la carpeta de valores o de gestión, en virtud del cual, se puede utilizar este sistema para comprar o vender decisiones. La mayoría de los problemas que implica la toma de decisiones sobre la base es subjetiva, vaga o imprecisa [8].

1.2.3.5 Análisis de las técnicas a partir de sus ventajas e inconvenientes

Las técnicas mencionadas en el acápite anterior responden a dos funcionalidades principales las cuales a su vez son los dos conceptos esenciales de esta investigación: la toma de decisiones y la comunicación virtual en agentes inteligentes. El funcionamiento de estas técnicas permite conocer el comportamiento, el cambio de estado, la posición actual, y otras posibles informaciones que pueden presentar los agentes que interactúan dentro de un entorno virtual.
Todas estas técnicas tienen sus beneficios e inconvenientes, pero se hace necesario analizarlas detalladamente para saber cuál sería la más eficiente, y proponer una para su uso en esta investigación [8, pp. 41- 42]:

- Los Sistemas Pizarra son útiles cuando el problema es extremadamente grande, pero no garantizan que se alcance una solución, por lo que no son confiables.
- Los Disparadores son eficientes y rápidos en ejecutarse, pero son complejos en su programación.
- El uso de las Máquinas de Estados Finitos se hace efectivo, pues son sistemas simples por lo que son rápidos de diseñar, implementar y ejecutar. A pesar que para ser aplicadas, se tiene que verificar primero si el problema puede ser descompuesto, es decir, que todos los estados, transiciones y condiciones deben ser conocidos y estar bien definidos.
- La Lógica Difusa a pesar de sus facilidades para la implementación, no es recomendable, ya que contiene múltiples definiciones de operadores y reglas de inferencia difusas.
- Los Sistemas Basados en Reglas no son muy eficientes para emplearlos a problemas grandes, ya que cuando el número de reglas es demasiado grande pudieran surgir complicaciones aunque el proyecto ya esté bastante avanzado y esto traería como consecuencia mucha perdida de recursos.

A partir del análisis de las ventajas y desventajas de estas técnicas se obtuvo la siguiente información (**Figura 6**) que establece una comparación entre estas técnicas, teniendo en cuenta los siguientes parámetros:

- Eficiencia respecto al tiempo de cómputo necesario para resolver el problema.
- Nivel de dificultad en la implementación.
- Rapidez en la ejecución.
- Garantía de una solución final factible.

Esta comparación se realiza con el objetivo de aportar los elementos necesarios para la selección

de la técnica más idónea para realizar la propuesta.

Técnicas de comunicación	Eficiencia	Implementación	Rapidez en la ejecución	Garantía de una solución final
Lógica Difusa	Si	Fácil	Si	Si
Sistemas Basados en Reglas	No	Fácil	Si	No
Disparadores	Si	Difícil	Si	Si
Sistemas Pizarra	No	Difícil	Si	No
Máquinas de Estados Finitos	Si	Fácil	Si	Si

Figura 6 Comparación entre las técnicas descritas [8, p. 43]

- Primeramente se analizó la eficiencia donde las técnicas: Lógica Difusa, Disparadores y Máquinas de Estados Finitos cumplen con este parámetro, lo que implica una mejor optimización en el resultado final.
- En el otro parámetro sólo los Sistemas Basados en Reglas, Máquinas de Estados Finitos y Lógica Difusa presentan un alto nivel de facilidad en la implementación que le permitirá a los desarrolladores una mayor simplicidad en el trabajo con técnicas de este tipo.
- Todas estas técnicas son rápidas en su ejecución.
- La Lógica Difusa, Disparadores y Máquinas de Estados Finitos, sí garantizan una solución final factible, sin embargo los Sistemas Pizarra y Sistemas Basados en Reglas cuando son aplicadas a problemas grandes, se desconoce la solución final hasta que se termina el problema, por lo que no se puede saber si los resultados son favorables.

Según la evidencia anteriormente presentada las técnicas candidatas para la propuesta de solución de la presente investigación son: Máquinas de Estados Finitos y Lógica Difusa. Ello no significa que las técnicas restantes no brinden facilidades de uso para los desarrolladores; sólo que no responden positivamente a todos los criterios analizados para la solución a proponer en el presente trabajo.
En [8, p. 44] se explica cómo seleccionar una entre las dos técnicas propuestas utilizando un método donde se consultan expertos con experiencia y conocimiento elevado de la materia llamado método Delphi. Como resultado de la aplicación de este método se obtuvo que la técnica que se adapta correctamente a los proyectos de la facultad y que se utilizará en esta investigación es: Máquina de

Estados Finitos, debido a que puede ser implementada de varias formas de manera sencilla. Cabe destacar que esta es una de las técnicas más utilizadas para la toma de decisiones.

1.2.4 Toma de decisiones

En IA, la toma de decisiones es la capacidad de un personaje / agente para decidir qué hacer. El agente procesa un conjunto de información que utiliza para generar una acción que desea llevar a cabo.

- **Entrada:** conocimiento del agente sobre el mundo;
- **Salida:** una solicitud de acción;

El conocimiento se puede dividir en conocimiento externo e interno.

- **Conocimiento externo:** información sobre el entorno del juego (por ejemplo, las posiciones de los personajes, el diseño del nivel, la dirección del ruido).
- **Conocimiento interno:** información sobre el estado interno del personaje (por ejemplo, salud, objetivos, últimas acciones).

Por lo general, los personajes del juego tienen un conjunto limitado de comportamientos posibles. Siguen haciendo lo mismo hasta que algún evento o influencia los haga cambiar. Por ejemplo: un guardia se parará en su puesto hasta que note al jugador, luego cambiará al modo de ataque, cubriéndose y disparando.

Las técnicas de toma de decisiones utilizadas en videojuegos suelen estar basadas bien en máquinas de estados o similares, por ejemplo, máquinas de estados jerárquicas, en planificadores automáticos, las redes de tareas jerárquicas o alguna mezcla de ambas categorías como los planificadores reactivos, entre los que se encuentran los árboles de comportamientos. Cada una de estas técnicas tiene sus ventajas e inconvenientes, pero esencialmente se mueven entre dos tipos de resultados (como se muestra en la **Figura 7**). Conseguir comportamientos bien definidos y conocidos por tanto por el desarrollador del juego como por el jugador (lo que normalmente ofrecen las MEF) o intentar obtener comportamientos emergentes (generalmente mediante el uso de planificadores) [3].

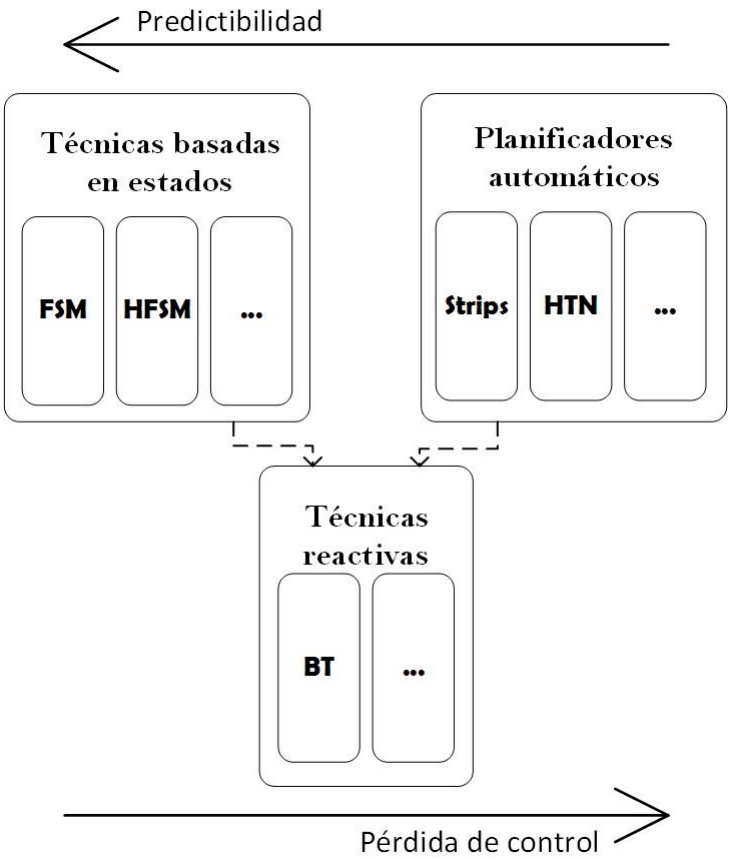

Figura 7 Ventajas o inconvenientes de las técnicas de control de decisiones [3]

La elección de una buena técnica o algoritmo es fundamental para el uso de un sistema de toma de decisiones y clave para la calidad del producto final. En el siguiente apartado se mencionarán las herramientas que se van a utilizar para el desarrollo del módulo de IA para agentes inteligentes.

1.3 Herramientas y tecnologías a utilizar

Para sentar las bases de la propuesta de solución realizada, se hizo necesario utilizar un conjunto de herramientas y tecnologías las cuales soportarán el desarrollo de la misma.

1.3.1 Metodología de desarrollo de software

La metodología de desarrollo de un software es la guía para realizar un software con calidad; definiendo, en un proyecto de software, quién debe hacer qué, cuándo y cómo debe hacerlo.
Se desarrolla con el objetivo de dar solución a los problemas que existen en la producción del software y cuenta con un conjunto de procedimientos, técnicas, herramientas y un soporte documental que ayuda a los desarrolladores a realizar un nuevo software [11].
Para regir el proceso de desarrollo de esta investigación, se seleccionó la metodología RUP, llamada así por sus siglas en inglés *Rational Unified Process* (Proceso Racional Unificado); porque

es aplicable a proyectos a largo plazo, intentando reducir la complejidad del software y la realización del mismo.

1.3.2 Herramienta de modelado

Se escoge a Visual Paradigm en su versión 8.0 como herramienta de modelado porque es multiplataforma y potente. Soporta el ciclo de vida completo del desarrollo de software, ayuda a una rápida construcción de aplicaciones de calidad y a obtener un menor coste en las mismas. Permite dibujar todos los tipos de diagramas de clases y generar diagramas desde código.
El trabajo con esta herramienta trae como ventajas [9] :

- Presenta capacidades de ingeniería directa e inversa.
- Es un producto de calidad y altamente usado en la población actual.
- Es un producto fácil de instalar, actualizar y manipular.

1.3.3 Lenguaje de modelado

Se selecciona como lenguaje de modelado UML (Lenguaje Unificado de Modelado) en su versión 2.5 porque es utilizado para visualizar, especificar, construir y documentar los artefactos de un sistema obtenidos a partir de una metodología determinada. Se pueden representar todas las fases de un proyecto informático.
Hacer uso del mismo trae como ventajas [9] :

- La trazabilidad y documentación del proyecto se realiza de una forma ordenada y guiada por los casos de uso.
- Es utilizado por Visual Paradigm como lenguaje de modelado.
- Es el más utilizado a nivel mundial para realizar el modelamiento de los artefactos creados durante el proceso de desarrollo de software para los sistemas orientados a objetos (OO).

1.3.4 Motor gráfico

Se define como motor gráfico al *framework* de software diseñado para crear y desarrollar videojuegos. Los desarrolladores de videojuegos pueden usar los motores para crear videojuegos para consola, móviles, ordenadores o dispositivos de Realidad Virtual.
Todo motor gráfico ha de ofrecer al programador una funcionalidad básica, proporcionando normalmente un motor de renderizado (“render”) para gráficos 2D y 3D, un motor que detecte la colisión física de objetos y la respuesta a dicha colisión, sonidos y música, animación, inteligencia artificial, comunicación con la red para juegos multijugador, posibilidad de ejecución en hilos, gestión de memoria o soporte para localización. Si bien en sus orígenes el desarrollo de videojuegos era muy dependiente de la plataforma, la evolución natural ha sido la de ofrecer un entorno de desarrollo que permitiera portar los desarrollos a distintas plataformas. Existen distintos tipos de motores

gráficos, siendo los más populares CryEngine, Unity 3D, Unreal Engine, Microsoft XNA, entre otros [12].

El motor gráfico que se utilizará para el desarrollo de la propuesta de solución será Unity 3D ya que es el utilizado por el centro Vertex para el desarrollo de videojuegos. La versión de Unity en la que se implementará el *plugin* será la 5.5.0f3 de 64 bit.

1.3.5 Unity 3D

Teniendo en cuenta que el módulo que se desarrollará debe de ser compatible con Unity 3D en este acápite se describirán algunas de las características de este motor gráfico.

Unity es un entorno de desarrollo de juegos multiplataforma, así como su *runtime* que permite desplegar los desarrollos a un gran número de plataformas incluyendo las móviles como Apple iOS, Google Android, Windows Phone, Blackberry 10, consolas (Microsoft Xbox 360, Xbox One, Sony Playstation 3 y 4, Nintendo Wii y Wii U) así como entornos de escritorio (Microsoft Windows, Apple Macintosh y Linux) o incluso un Webplayer para el despliegue en los principales navegadores.

El entorno unificado de desarrollo permite desde la creación, manipulación y visualización de *assets* y entidades, visualización tanto en el escritorio como en el hardware destino (móvil conectado al USB en modo depuración). Ofrece también herramientas para analizar el comportamiento en las distintas plataformas y permite la programación en lenguajes JavaScript, C# o Boo para facilitar la creación de animaciones [12].

Unity es un sistema de desarrollo único. Es enfocado en los *assets* y no en el código, el foco en los *assets* es similar al de una aplicación de modelado 3D. Unity hace el proceso de producción de juego simple dándole un set de pasos lógicos para construir cualquier panorama concebible de juego. Establece el uso del concepto **GameObject** (GO), donde se puede estudiar los componentes del juego en objetos dóciles, que está hecho de muchos componentes individuales. Haciendo objetos individuales dentro del juego e iniciando funcionabilidad en ellos con cada componente que se sume, se puede expandir el juego en una manera progresiva lógica. Los componentes a su vez tienen variables, esencialmente por los cuales serán controlados [13].

1.3.5.1 Herramientas que utiliza Unity

***Assets*:** Son los bloques constructivos de todo lo que el Unity posee en sus proyectos. Se guardan en forma de archivos de imagen, modelos del 3D y archivos de sonido, Unity se refiere a los archivos que se usarán para crear su juego como activos.

GameObjects: Cuando un activo es usado en una escena de juego, se convierte en un "GameObject". Todo *GameObjects* contiene al menos un componente con el que comenzar, es decir, el componente *Transform*. Transformación simple la cual le dice al motor de Unity la posición, rotación, y la escala de un objeto.

***Components*:** Los componentes vienen en formas diversas. Pueden ser para crear comportamiento, definiendo apariencia, e influenciando otros aspectos de la función de un objeto en el juego. Los componentes comunes de producción de juego vienen construidos dentro del Unity, desde el *Rigidbody*, hasta elementos más simples, como luces, las cámaras, los emisores de partículas, y más.

***Scripts*:** El *Scripting* es una parte esencial de Unity ya que define el comportamiento del juego. El lenguaje de programación recomendado para Unity es JavaScript, aunque C# puede ser igualmente usado.

***Prefabs*:** Almacena los objetos como activos para ser reusado en partes diferentes del juego, y luego creados o copiados en cualquier momento.

1.3.6 IDE de desarrollo

MonoDevelop en su versión 5.9.6 es el ambiente de desarrollo integrado (IDE) proporcionado con Unity que se utilizará en el proyecto. Un IDE combina la operación familiar de un editor de texto con características adicionales para depurar y gestionar otras tareas de proyecto [14]. MonoDevelop es un entorno de desarrollo integrado, libre, multiplataforma y gratuito, diseñado primordialmente para C# y otros lenguajes .NET como Nemerle, Boo, Java (vía IKVM.NET) y en su versión 2.2 Python.

1.3.7 Editor de texto y código fuente

La decision de utilizar Sublime Text en su version 3.0 compilación 3143 como editor de texto y código fuente fue tomada debido al aumento de rendimiento que proporcionará esta herramienta en el desarrollo del software.

Una de las características más apreciadas de Sublime Text es su simplicidad. Posee una interfaz limpia y sencilla que no muestra barras de herramientas y diálogos de configuración. Sin embargo, esto no quiere decir que carezca de funciones pues posee *plugins*, funciones de autocompletado, entre otras. Su interfaz se divide en pestañas, esto implica que puedes abrir muchos archivos.

Sublime Text es flexible y soporta muchos lenguajes de programación. Además, puedes realizar cambios en tu código de manera sencilla y tienes la opción de realizar selecciones múltiples. Su característica de Autocompletado hace que programar sea más sencillo y rápido. Te ofrece una lista de palabras posibles de acuerdo a palabras que hayas empleado anteriormente. Asimismo, Sublime Text posee una característica de resaltado para HTML, CSS, Java, PHP, C# y muchos otros lenguajes. En fin, que se está en presencia de un poderoso y completo editor de código con muchas otras características disponibles que mejorarán tu eficiencia y productividad [15].

1.4 Conclusiones del capítulo

Para definir una buena propuesta de solución es necesario tener conocimiento de las características de las técnicas que permiten la cooperación entre agentes. Por lo que en este capítulo se abordaron los principales conceptos relacionados con el desarrollo de agentes inteligentes. Se describieron las características que debe tener un agente y se mencionaron las técnicas que pueden ser utilizadas en el desarrollo del mismo. Además, se realizó una descripción y un análisis sobre las diversas técnicas de comunicación virtual y de toma de decisiones que son empleadas para desarrollar sistemas multiagentes llegando como acuerdo el uso de una técnica de MEF para la propuesta de solución. Finalmente se muestran las herramientas con las que se desarrollará la propuesta de solución.

CAPÍTULO 2: PROPUESTA Y DESCRIPCIÓN DE LA SOLUCIÓN

Introducción

En este capítulo se dará a conocer cuáles son los componentes fundamentales que debe tener un módulo de IA para que una entidad actúe de forma parecida a la humana en un videojuego. Se propondrá una estructura básica de cómo debe ser el sistema. Se realizará un mejor acercamiento sobre la técnica seleccionada en el capítulo anterior para usar en el módulo de IA para agentes inteligentes, especificando las distintas formas de implementación que tiene una máquina de estados finitos. Se especificarán los requisitos funcionales y no funcionales asi como los patrones de diseño necesarios para lograr satisfacer los objetivos de este trabajo.

2.1 Partes fundamentales que debe tener un sistema inteligente

Un sistema de software para proveer de inteligencia a los agentes dentro en un videojuego o simulador debe constar con tres componentes fundamentales: un componente encargado de la navegación, un componente encargado de la toma de decisiones y uno encargado de que el agente obtenga la percepción del mundo [16].

Componente de navegación: Es el encargado de calcular los vectores necesarios para que el NPC se mueva por el mundo virtual de acuerdo a la decisión tomada por el sistema de toma de decisiones. Para que el agente controlado por la computadora tenga una buena navegación debe conocer su ubicación así como la posición de las otras entidades.

Componente de toma de decisiones y comunicación virtual: Este componente es esencial para que el sistema que se está desarrollando parezca inteligente.

Supongamos que tenemos un sistema donde el agente se mueve de forma extraordinaria y posee una información elevada sobre el medio ambiente, pero si escoge mal la forma en que debe actuar; esto provocaría que fuese ineficiente para el cliente.

Puede diseñarse de dos formas fundamentales: para simuladores o juegos que requieran inteligencia de estrategia y para los que no. Aunque las técnicas utilizadas para el logro de cada uno de estos diseños son básicamente las mismas y tanto uno como otro incluirían las condiciones necesarias para la ejecución del juego según sus reglas.

Componente de percepción: Se encarga de todo lo que el agente es capaz de obtener del entorno virtual. Su estructura puede ir desde una clase que capture solo datos topológicos hasta un conjunto de clases complejas que capturen estos y además los datos medioambientales y las características tácticas del ambiente. Es muy importante para que una buena decisión sea tomada, que los datos incorporados sean los correctos y que estos se hagan con la rapidez necesaria. No obstante, existen *softwares* que trabajan con información ruidosa, para ello emplean lógica difusa.

Aquellos videojuegos donde las acciones del agente se realizan de forma lineal, generalmente

utilizan los comportamientos de locomoción para dirigir sus acciones de acuerdo a una situación previamente determinada. Por lo que el trabajo del componente de decisión se resumiría a identificar la situación en que se encuentra la entidad y enviar cual es el comportamiento ya predefinido.

2.2 Estructura básica

En el epígrafe anterior se vio cuales deben ser los componentes principales que debe tener un software para incluir inteligencia a una entidad o agente en un juego o simulador. ¿Ahora, cómo sería la relación entre ellos y que representaría cada uno dentro de un videojuego? La lógica de esta estructura quedaría de la siguiente manera (**Figura 8**), el sistema de percepción obtendría la información del medio donde la entidad se desarrolla, esta información se le daría al encargado de tomar la decisión y después que esta sea tomada pasaría al encargado de navegación y este a su vez determina cómo la entidad va a seguir el movimiento.

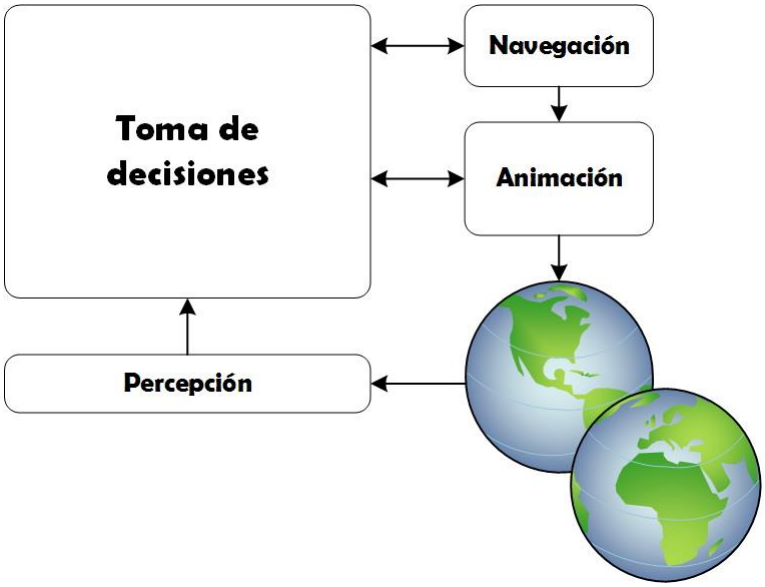

Figura 8 Arquitectura de un sistema de IA en un videojuego de Acción/Aventura [3]

2.2.1 Sistema de toma de decisiones

El sistema de toma de decisiones suele ser el punto central de la Inteligencia Artificial de un videojuego. Por lo general, este sistema es el punto de comunicación entre la lógica de alto nivel -o "inteligencia" de los agentes- y el propio mundo del juego. Esta interacción se realiza mediante la monitorización de cambios en el entorno (sistema de percepción) y la actuación sobre el mundo del juego, que suele ser el resultado de una animación [3].

2.2.2 Interacción con el mundo: navegación y animación

El sistema de animación se encarga de modificar el esqueleto de los personajes basándose en uno o más cuadros (*frames*) de animación -mediante técnicas como el *blending*-; en cuanto al

movimiento, en videojuegos, se denomina navegación(**Figura 9**) al sistema que engloba la búsqueda de caminos, o *path finding* (es decir, se encarga de seleccionar cuál es la mejor ruta para llegar del punto A al punto B) y la locomoción (*path following* o *locomotion*), que se encarga de seleccionar las animaciones adecuadas para que un personaje se mueva de forma realista por el mundo siguiendo el camino devuelto por el *path finder* [3].

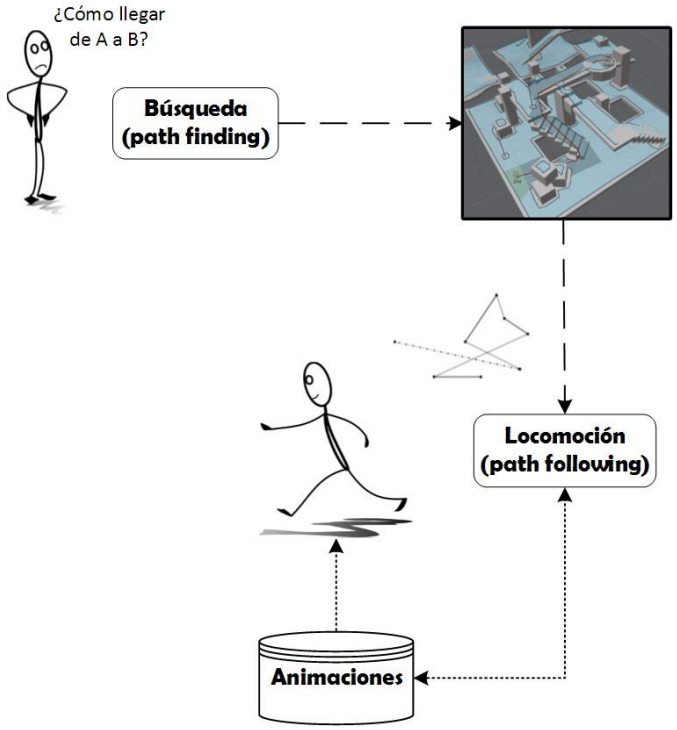

Figura 9 Solicitud de camino óptimo al path finder [3]

Los agentes de IA son capaces de ejecutar las decisiones tomadas por su lógica interna mediante el uso de animaciones. Una vez que la decisión está tomada, se suele estar ante dos casos diferenciados: o bien el agente puede realizar la acción en el punto en el que se encuentra o necesita modificar su posición antes de llevar a cabo la acción.

La búsqueda de caminos suele requerir dos elementos. Primero, se debe contar con una representación adecuada de las superficies navegables del mundo de juego. Esto suele conseguirse mediante el uso de grafos de navegación, que son un conjunto de nodos (posiciones) conectados mediante enlaces que indican el tipo de locomoción entre dichos puntos (por ejemplo, un agente puede llegar de A a B caminando, mientras que para llegar de A a C debe nadar), o mediante el uso de una **NavMesh** o representación simplificada de la geometría navegable del mundo de juego (generalmente, un conjunto de triángulos), más adelante se explica la utilidad de esta clase. Finalmente, un algoritmo de búsqueda como A* o similar (como algún algoritmo jerárquico) se encargará de producir un camino (o lista de puntos - *waypoints*) que el agente debe utilizar [3].

Entre las técnicas más utilizadas para la navegación y búsqueda de caminos se encuentran el uso de **NavMesh** y *Waypoints:*

NavMesh es una clase de unity responsable de calcular un camino viable y óptimo para los agentes. Es una superficie parametrizable desde el inspector de Unity que permite establecer valores como anchura máxima de los agentes, separación con las paredes o límites de la superficie y otros parámetros relacionados con el mismo. En el escenario, esta superficie se corresponde con el suelo del escenario.

Generalmente, el **NavMesh** debe realizar un *bake* antes de ejecutar la escena para tener cálculos previos. Sin embargo, en las últimas versiones de Unity se han incluido mejoras y ejemplos que permiten recalcular en tiempo real los nuevos caminos que pueden producirse cuando se introduce un cambio en la superficie, como una modificación del escenario o la inclusión de obstáculos

Los *Waypoints* son puntos marcados por el diseñador dentro de la geometría del juego o simulador que almacenan información, dependiendo para que se esté usando (Millington, 2006). La unión de más de un punto se puede emplear también para montar ataques, teniendo a las entidades en puntos estratégicos. Además los waypoints pueden representar lugares de refugio, o dónde se puede lograr mayor alcance en los disparos en un combate e incluso donde una entidad perdedora pueda escapar.

2.2.3 Sistema de percepción

El sistema de percepción se encarga de hacer llegar a la lógica de toma de decisiones el estado del mundo que rodea al agente. Esto lo consigue mediante el uso de sensores artificiales, como sistemas de visión o audición, sistemas de eventos y memoria.

Por ejemplo, un sistema de visión en un videojuego trata de comprobar si desde la posición en la que se encuentra el agente (probablemente, desde la posición de sus ojos) es posible ver algún elemento de importancia, como puede ser otro agente o algún tipo de objeto. La visión suele comenzar por algún sistema de conos o "cajas" de visión. Cualquier elemento dentro de dichas cajas, debe ser tomado en cuenta como un posible punto de interés. Tras ello, comenzará un proceso de **Prefab** o comprobación de línea de visión al punto de interés: el objetivo ahora es cerciorarse de que nada se interpone entre los ojos y el posible objeto que se está detectando, es decir, nada obstruye nuestra visión. Dependiendo del tipo de objeto que se está tratando, se puede necesitar una o varias de estas comprobaciones de línea [3].

2.2.3.1 Componentes de Unity para percepción

Unity brinda en su motor de físicas un conjunto de componentes útiles para recibir información del entorno. Siendo uno de los más utilizados los componentes de tipo Collider.

Los componentes de *Collider* definen la forma de un objeto para colisiones físicas. Un

colisionador, que es invisible, no necesita tener exactamente la misma forma que la malla del objeto y, de hecho, una aproximación aproximada suele ser más eficiente e indistinguible en el juego.
Los colisionadores más simples (y menos intensivos en el uso del procesador) son los denominados tipos de colisionador primitivo. En 3D, estos son *Box Collider*, *Sphere Collider* y *Capsule Collider*. En 2D, puede usar *Box Collider* 2D y *Circle Collider* 2D. Se puede agregar cualquier cantidad de estos a un solo objeto para crear colisionadores compuestos.
Con un posicionamiento y dimensionamiento cuidadosos, los colisionadores compuestos a menudo se pueden aproximar bastante bien a la forma de un objeto y, al mismo tiempo, mantienen una baja sobrecarga del procesador. Se puede obtener mayor flexibilidad al tener colisionadores adicionales en los objetos secundarios (por ejemplo, las cajas se pueden girar en relación con los ejes locales del objeto principal). Al crear un colisionador compuesto como este, solo debe haber un componente de **Rigidbody**, colocado en el objeto raíz en la jerarquía [17].
El sistema de *scripting* puede detectar cuándo ocurren las colisiones e iniciar acciones usando la función **OnCollisionEnter**. Sin embargo, también puede usar el motor de física simplemente para detectar cuándo un colisionador ingresa al espacio de otro sin crear una colisión. Un colisionador configurado como *Trigger* (usando la propiedad **Is Trigger**) no se comporta como un objeto sólido y simplemente permitirá el paso de otros colisionadores. Cuando un colisionador ingresa a su espacio, un disparador llamará a la función **OnTriggerEnter** en los scripts del objeto disparador [17]. Otro de los components utilizados para percibir el entorno es ***NavMeshAgent y RayCastHit***. El primero de ellos es un componente que se adjunta a un personaje móvil en el juego para permitirle navegar por la escena usando una malla de navegación conocida en Unity como **NavMesh**. El segundo es un método de detección de colisiones a lo largo de una línea recta. Dado un origen y una dirección, la llamada devolverá el primer objeto con el que se haya encontrado.

2.3 Formas de implementación de una MEF

Los componentes antes mencionados servirán como apoyo para la creación del sistema de toma de decisiones. Este sistema será implementado utilizando la técnica de máquina de estados finitos como ya se habia acordado anteriormente. Lo básico que debe tener una máquina de estados es que [18]:

- Las acciones o comportamientos están asociados con cada estado.
- Cada transición conduce de un estado a otro, y cada uno tiene un conjunto de condiciones asociadas.
- Cuando se cumplen las condiciones de una transición, el personaje cambia de estado al estado objetivo de la transición.
- Cada personaje está controlado por una máquina de estado y tienen un estado actual.

Existen distintas formas de implementacion de esta técnica que serán mencionadas a continuación.

2.3.1 MEF codificadas con switch

Fundamentalmente, una máquina de estados es una serie de sentencias condicionales, por lo que se puede codificar como un conjunto de sentencias if-then o como una instrucción **switch** (Ver **Figura 10**).

```
1 If CurrentState == STATE_1
2 {
3    State1Behaviour();
4    If CheckTrigger1to2()
5    {
6       SetState (STATE_2);
7    }
8 }
```

Hard coded switch

Figura 10 *Hard coded* **switch** [19]

Este es el código del controlador FSM; toda la lógica de transición está centralizada aquí. Observar que el comportamiento real de IA está contenido dentro de una función tal como **State1Behaviour**, y el algoritmo para decidir si una determinada transición se ha iniciado está contenida en una función como **CheckTrigger1to2**. La relación de comportamiento y el algoritmo desencadenante podrían incluirse en esta pieza de trabajo en lugar de en funciones separadas, pero darán como resultado una estructura de control de flujo compleja e incomprensible. Sin embargo, la codificación, incluso de una máquina de estados simples utilizando una sentencia **switch** (como el ejemplo de Pac-Man), se convertiría rápidamente en un código espagueti, lo que sería muy difícil de cambiar si el diseño evoluciona, o un nuevo tipo de IA debe ser introducido [19].

2.3.2 MEF con patrón de estado codificado

Un enfoque más extensible es establecer una clase de Estado básica y un conjunto de estados específicos que heredan su estructura de la clase de Estado básica. En cualquier momento que un agente de IA tiene un estado asociado, cuando se produce una transición, el estado se reemplaza con el nuevo estado. En este caso, cada estado es responsable de determinar cuándo debe activarse una transición, de modo que la lógica de FSM se distribuye a través de los estados, en lugar de existir dentro de una función de controlador centralizado.

Para hacer el código del controlador más centralizado, también se puede implementar una clase de FSM que se usa para la transición entre estados cuando se producen desencadenantes. En ese caso, es posible que se implemente un sistema de mensajería, por el cual se le indica al FSM que cambie los estados cuando se detecta un disparador dentro de un comportamiento de estado [19].

2.3.3 MEF con clase *StateMachineBehaviour*

StateMachineBehaviour es un componente que se puede agregar a un estado de máquina de estado. Es la clase base de la que se deriva cada *script* de un estado y hereda de la clase **ScriptableObject**.

Por defecto, el **Animator** instancia una nueva instancia de cada comportamiento definido en el controlador. El atributo de clase **SharedBetweenAnimatorsAttribute** controla cómo se crean las instancias de los comportamientos.

Funcionamiento del mecanismo del sistema de Unity **Animator** [20] :

- En Unity, puede crear un activo llamado **Animator Controller**. Esta es una plantilla de máquina de estado.
- Hay estados dentro de su máquina de estado. Se pueden asociar estados con animaciones, pero en realidad son opcionales en caso de querer crear una máquina de estado de lógica pura.
- Para ejecutar su máquina de estado, se agrega un componente llamado **Animator** a un **GameObject** y se configura con cualquier controlador **Animator** que se haya creado. Esta es ahora una instancia de la máquina de estado.
- Para ejecutar la lógica en cada estado, se necesita crear un script derivado de la clase **StateMachineBehaviour**. Una vez que se tiene este nuevo comportamiento, se puede agregar a cualquier estado dentro de la máquina de estado.

2.3.4 Activación de scripts

Unity al proporcionar un inspector que muestra todos los componentes que se le asignan al **GameObject**, brinda la posibilidad de deshabilitar los componentes y habilitarlos a conveniencia. Como Unity brinda la posibilidad de añadir una cantidad ilimitada de scripts como componentes a los **GameObjects** estos se pueden mantener desactivados hasta que se activen por un evento (cambio de estado). Esta forma de implementación requiere de una clase base para referenciar cada *script* añadido.

2.4 Analisis de las formas de implementación de una MEF

Aunque las MEF codificadas con **switch** son fáciles de escribir y son muy rápidas, son a su vez muy difíciles de mantener debido a que las máquinas complejas de estados finitos requieren miles de líneas de código y pueden ocasionar cuellos de botella.

Las MEF con patrón de estado codificado o enfoque basado en clases brinda mucha flexibilidad a las máquinas de estados finitos, pero reduce su rendimiento debido a la gran cantidad de llamadas a métodos.

La implementación utilizando activación de *scripts* supondría un elevado costo computacional debido a la gran cantidad de scripts que tendría cada **GameObject** y la cantidad de referencias que tienen que ser inicializadas aún sin estar en uso.

Hasta el momento la implementación usando Unity **Animator StateMachineBehaviour** sería una de las más eficaces ya que la clase **StateMachineBehaviour** al heredar de **ScriptableObject** supondría un menor costo computacional puesto que es una clase de la que se puede derivar si se quieren crear objetos que no necesiten unirse a los objetos del juego. La creación de instancias de **ScriptableObject** vinculadas a activos en su proyecto se realiza mediante **CreateAssetMenuAttribute**. En el siguiente acápite se explicará el uso de esta clase y sus beneficios. Luego se dedicará un apartado para explicar el uso de la clase **EditorWindow**, con la cual se pueden crear cualquier número de ventanas personalizadas en su aplicación.

2.5 La clase ScriptableObject

El **ScriptableObject** es una clase que permite el almacenamiento de grandes cantidades de datos compartidos independientes de instancias de *script*. No ha de confundirse esta clase con la muy similar **SerializableObject**, la cual es una clase del editor y llena un propósito diferente. Considere por ejemplo que se ha hecho un *prefab* con un script que tiene un arreglo de millones de enteros. El arreglo ocupa 4MB de memoria y su dueño es el *prefab*. Cada vez que se instancie el *prefab*, se va a obtener una copia de ese arreglo. Si se ha creado 10 **GameObjects**, entonces se terminaría con 40MB de datos del arreglo para las 10 instancias.

Unity serializa todos los tipos primitivos, *strings*, arreglos, listas, tipos específicos para Unity tal como *Vector3* y sus clases personalizadas con el atributo **Serializable** como copias que pertenecen al objeto en el que se declararon Esto significa que, si se ha creado un **ScriptableObject** y almacena millones de enteros en un arreglo, entonces declara que el arreglo será almacenado con la instancia. Las instancias son pensadas en sus propios datos individuales. Los campos **ScriptableObject**, o cualquier campo **UnityEngine.Object**, tal como **MonoBehaviour**, **Mesh**, **GameObject** y así, son almacenados por referencia opuesto a un valor. Si se tiene un *script* con una referencia al **ScriptableObject** con un millón de enteros, Unity va a solamente guardar una referencia al **ScriptableObject** en los datos de script. El **ScriptableObject**, por el contrario, almacena el arreglo. Diez instancias de un *prefab* que tiene una referencia **ScriptableObject**, que mantiene 4MB de datos, va a pesar en total 4MB y no 40MB como se discutió en el otro ejemplo [21].

El caso intencionado para utilizar el **ScriptableObject** es reducir el uso de memoria al evitar las copias de valores, pero también se puede utilizar para definir conjuntos de datos conectables. Un ejemplo de esto sería imaginar una tienda NPC en un juego RPG. Se pueden crear múltiples *assets* de **ScriptableObject ShopContents** personalizado, cada uno definiendo un conjunto de *items* que están disponibles para compra. En un escenario dónde el juego tiene tres zonas, cada zona puede

ofrecer diferentes tipos de *items*. El *script* de la tienda podría referenciar un objeto **ShopContents** que define qué *items* están disponibles [21].

2.6 La clase EditorWindow

Esta clase se usa para crear una ventana de edición. Esta ventana personalizada puede flotar libremente o ser acoplada como una pestaña, al igual que las ventanas nativas en la interfaz de Unity. Las ventanas del editor normalmente se abren usando un elemento de menú. Mediante el uso de esta clase se pueden crear cualquier número de ventanas personalizadas en su aplicación. Estas se comportan igual que el Inspector, Escena o cualquier otra integrada. Heredar de **EditorWindow** es una gran manera de agregar una interfaz de usuario a un sub-sistema para su juego.

Hacer una ventana del editor personalizada implica los siguientes pasos sencillos:

1. Crear un script que se derive de la clase **EditorWindow**.
2. Use el código para activar la ventana para mostrarse a sí misma.
3. Implemente el código **GUI** para su herramienta.

2.7 Patrón de diseño Delegación

El patron de diseño Delegación o *Delegate Pattern* en inglés, se trata de una técnica en la que un objeto permite mostrar cierto método al exterior, pero internamente la implementación o las acciones desencadenadas por el llamado de este método se delega a otro objeto de una clase distinta pero asociado. La delegación es utilizada como un mecanismo para centralizar en torno a un solo objeto los compartimentos(métodos) de varios objetos donde dichos comportamientos mantienen cierto nivel de relación.

Este patrón se usa comúnmente en casos donde:

- Se desea reducir el acoplamiento de métodos para una clase.
- Existen componentes que tienen comportamientos similares, pero que en un futuro es posible que se realicen cambios.
- Se desea remplazar la herencia, generalmente se usa la delegación como alternativa a la herencia.

Hay que tener en cuenta que la herencia a pesar de ser una buena estrategia para ser utilizada cuando existe una estrecha relación entre las clases involucradas, no es recomendada cuando la relación no es tan estrecha, ya que se crea un lazo de dependencia directa entre las clases; por otra parte la delegación es la forma más flexible para expresar una relación entre clases [22].

2.8 Solución a implementar

En los acápites anteriores de este capítulo se abordaron temas relacionados con la propuesta de solución que a continuación se presentará. Es válido aclarar, que cuando se va utilizar la estructura básica mencionada anteriormente en el acápite **2.2** (**Figura 8**) debe pensarse de antemano cómo se va a incluir en el sistema que estamos diseñando, por lo que debe hacerse un trabajo de diseño antes de ir directo al código, con el fin de permitir la reutilización del mismo y evitar los errores costosos en la etapa de desarrollo. Para ello la inclusión de patrones de diseño como el de delegación descrito con anterioridad sería una buena opción para la realización del módulo.

En la delegación, un objeto maneja una solicitud al delegar en un segundo objeto (el delegado). Se delegará el comportamiento a los recursos de **ScriptableObject** y en el se creará la conexión entre estados de el sistema (**Figura 11**). Recordar que la clase **ScriptableObject** es similar a la clase **MonoBehaviour** pero no se adjunta a los **GameObjects**. Además esta clase puede ser usada para crear *assets* que almacenen información o ejecuten código [23]. En la propuesta de solución se empleará para ambas funciones.

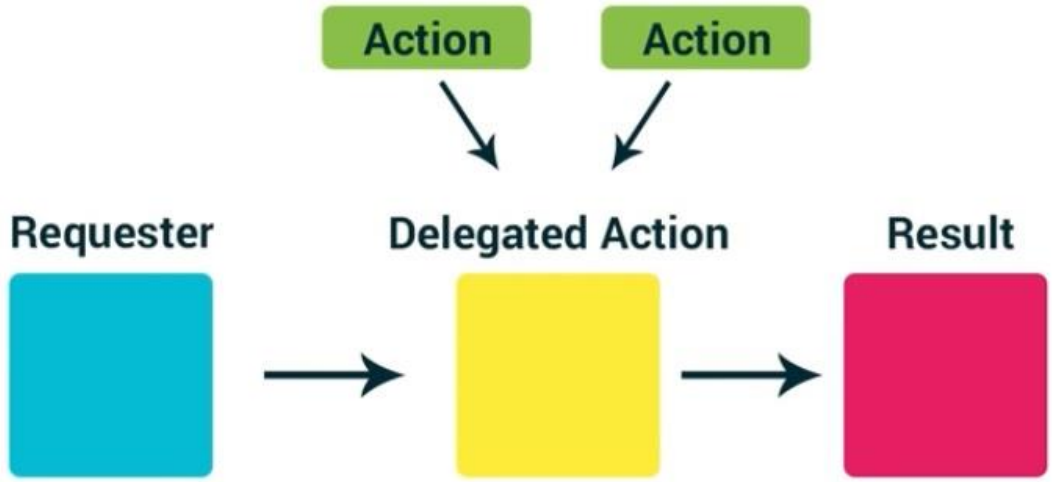

Figura 11 Funcionamiento del patrón delegación

En la **Figura 11** se muestra como será el funcionamiento del patrón delegación en el software a desarrollar. Se tiene una solicitud de querer hacer algo. Se envía la solicitud a la acción delegada y no necesita saber qué acción está llamando, solo pide que se haga una acción, y luego se puede conectar y desconectar diferentes acciones en la ranura amarilla que se muestra en el gráfico, posteriormente dará salida a una base de resultados en cualquier acción que se haya delegado actualmente.

En el listado de *assets* se van a tener cuatro elementos necesarios para el módulo de IA a implementar. Estos son las acciones, decisiones, transiciones y estados. Además de un *script* que hereda de la clase **MonoBehaviour** que se va a llamar **StateController** y será adjuntado a los agentes que requieran IA, dentro de esta clase serán definidos el estado actual, los datos del agente

y un campo llamado ojos para implementar un **RayCasting** como elemento de percepción visual. Las acciones serán los comportamientos que el agente realiza como por ejemplo patrullar o alertarse, estas no cambiarán el estado de la MEF. Las decisiones en este sistema serán los eventos que un agente hace que tendrán el potencial de cambiar el estado de la MEF, como por ejemplo buscar un objetivo. Los estados contendrán una colección de acciones y decisiones y se repetirán hasta que la decisión de cambiar de estado sea tomada. A través de la delegación **StateController** solo deberá saber que estará trabajando con un estado o, una acción, no cual estado o acción estará controlando. Esto dará como resultado que luego, si se quiere añadir un nuevo estado o acción en el sistema se puede hacer fácilmente y los accesos entre las distintas partes serán creados por el usuario en el inspector de Unity, no directamente en el código. Además se utilizará la clase EditorWindow para otras funcionalidades como crear agentes y estados.

2.9 Modelo de Dominio

Se realiza un modelo de dominio para comprender mejor la estructura y los conceptos asociados al funcionamiento del sistema. El modelo de domino es un diagrama de UML, que permite mostrar los principales conceptos y relaciones que intervienen en el dominio del problema. Captura los tipos más importantes de objetos que existen o los eventos que suceden en el entorno donde se encuentra el sistema. En la **Figura 12** se muestra el diagrama de clases del Modelo de dominio.

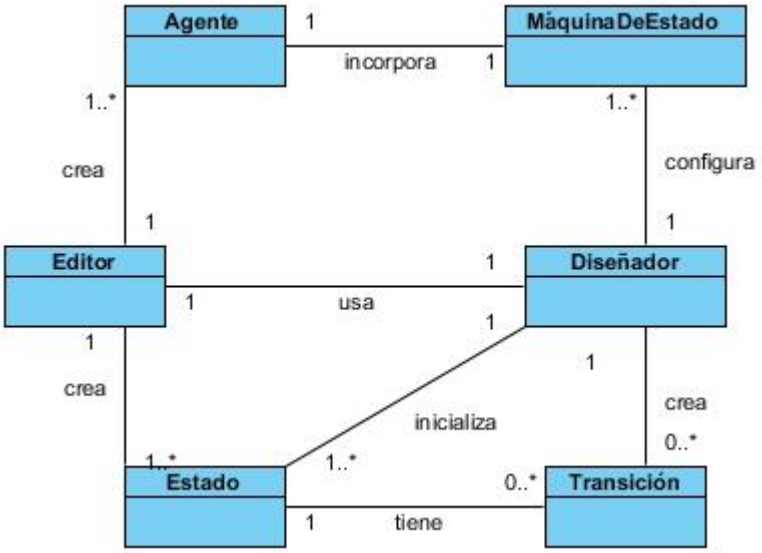

Figura 12 Modelo de Dominio

El diseñador es el encargado de usar el Editor para crear los agentes y estados. Este inicializa los estados y crea las transiciones entre ellos. Además realiza la configuracion y diseño de las máquinas de estado de cada agente.

2.10 Especificación de los requisitos de software

2.10.1 Requisitos funcionales

Una vez conocidos los conceptos asociados al objeto de estudio, se puede empezar a analizar ¿Qué debe hacer el sistema para que se cumpla el objetivo planteado al inicio de este trabajo?, para ello se enumera a través de los requisitos funcionales las funciones que el sistema deberá ser capaz de realizar.

A continuación se muestra una descripción detallada, para lograr un mejor entendimiento, de los requisitos funcionales (RF).

Tabla 1: Especificación de los requisitos funcionales

№	Nombre	Descripción	Complejidad	Prioridad para el cliente
RF1	Crear agentes	El sistema debe ser capaz de crear agentes de forma manual, de acuerdo a los datos de interés que tengan los mismos especificados y añadiendo materiales de tipo de prefab solamente.	Alta	Alta
RF2	Actualizar agentes	El sistema debe ser capaz de actualizar los agentes, de forma que los datos que éstos reciban del entorno sean proporcionados por un sólo agente.	Alta	Alta
RF3	Eliminar agentes	El sistema debe ser capaz de eliminar a los agentes.	Media	Alta
RF4	Crear estados	El sistema debe permitir la creación de nuevos comportamientos (estados).	Alta	Alta
RF5	Eliminar estados	El sistema debe ser capaz de eliminar los estados	Media	Baja
RF6	Asociar estados	El sistema deber permitir la inclusion de nuevos estados	Alta	Alta

		en la MEF y definir cuando va a cambiar el mismo hacia otro estado.		
RF7	Percepción visual	El sistema debe contar con un sistema de percepción visual que permita a los agentes encontrar enemigos a la vista.	Baja	Alta

2.10.2 Requisitos no funcionales

Los requisitos no funcionales son propiedades o cualidades que el producto debe tener. Debe pensarse en estas propiedades como las características que hacen al producto usable, rápido o confiable. A continuación se muestran las especificaciones de cada uno de los requisitos no funcionales :

a) RnF1. <Requisito de Usabilidad>

- **Finalidad:** Los usuarios que trabajen con el módulo deben tener conocimientos básicos programación en C# y noción de trabajo sobre cómo utilizar el inspector de Unity3D y la función crear de la ventana de proyecto. El módulo deberá ser lo suficientemente flexible como para que los futuros usuarios puedan a partir de los modelos generales implementados, adaptarlo a su problema en particular con bajo costo de desarrollo.

b) RnF2. <Requisito de Soporte>

- **Soporte:** El sistema que se desarrollará, en su finalidad, deberá tener compatibilidad con las versiones mas recientes de Unity3D.
- ***Hardware*:** Se necesita que la computadora, en que se vaya a utilizar la aplicación, conste de un Microprocesador Intel Pentium 4 a 1.4GHz. Y que tenga una memoria RAM de 1Gb, como mínimo.

c) RnF3. <Requisito de Restricciones del Diseño>

- **Lenguaje de programación:** Se utilizará el lenguaje de programación C# bajo el paradigma de programación OO.

2.11 Modelo del caso de uso del sistema

En esta sección se reconocen los actores del sistema a desarrollar, y se definen los casos de uso (CU) del sistema. Además, se seleccionan los CU correspondientes al primer ciclo de desarrollo creando sus especificaciones textuales en formato expandido.

2.11.1 Actores del sistema

Los actores de un sistema son agentes externos o roles que las personas o usuarios desempeñan cuando interactúan con el software que se realizará [27]. En el caso particular de esta investigación, quien inicializará el sistema será la misma aplicación en sí, que como actor del sistema será llamado Aplicación.

Tabla 2: Actor del sistema

Actor	Descripción
Aplicación	Es el encargado de inicializar el sistema permitiendo que sean ejecutadas las funcionalidades: crear agentes, crear estados, asociar estados y percepción visual.

2.11.2 Diagrama de CU del sistema

El diagrama de CU del sistema define las relaciones entre los actores y los diferentes CU existentes. Para la realización de esta investigación fueron definidas un conjunto de acciones que deben ser ejecutadas por el actor del sistema y que desencadenan un conjunto de operaciones. Las interrelaciones entre las acciones y el actor del sistema son agrupadas en el diagrama de CU del sistema que se muestra a continuación (**Figura 13)**.

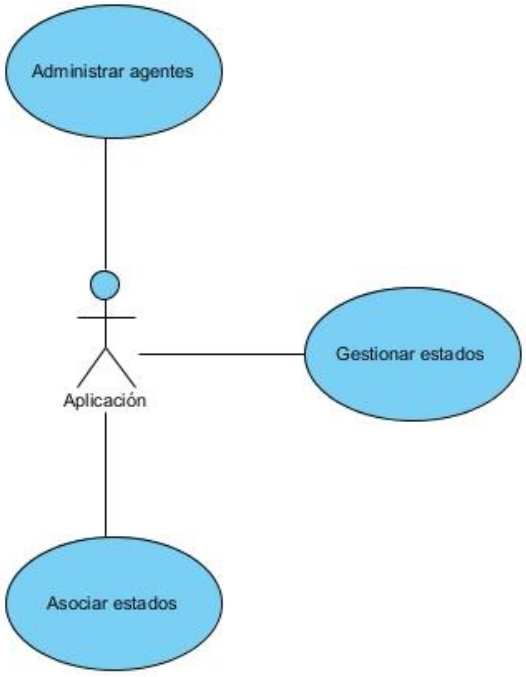

Figura 13 Diagrama de CU del sistema

2.11.3 Descripción de los CU del sistema

Los CU son artefactos narrativos que describen, bajo la forma de acciones y reacciones, el comportamiento del sistema desde el punto de vista del usuario, estableciendo un acuerdo entre clientes y desarrolladores sobre las condiciones y posibilidades (requisitos) que debe cumplir el sistema [24] .

Cada CU tiene una descripción de las funcionalidades que ejecutará el sistema propuesto como respuesta a las acciones del usuario. Las descripciones presentadas a continuación muestran detalladamente los flujos operacionales de estos.

Tabla 3: Administrar agentes

CU 1	Administrar agentes
Actores	Aplicación
Resumen	El CU inicia cuando la aplicación necesita crear un agente con sus datos específicos, permitiendo que el mismo sea creado, actualizado o eliminado.
Precondiciones	Debe existir un material prefabricado para su creación e ingresar un nombre en el formulario
Referencias	RF1, RF2, RF3
Prioridad	Crítico
Flujo Normal de Eventos	
Acción del Actor	**Respuesta del Negocio**
1. La Aplicación llama a las diferentes funcionalidades: a) Crear de agentes. b) Actualizar agentes. c) Eliminar agentes.	1.1 El sistema inicializa las secciones “Crear agentes”, “Actualizar agentes” y “Eliminar agentes”.
Sección “Crear agentes”	
Acción del Actor	**Respuesta del Negocio**
2. La Aplicación pasa los siguientes datos: a) Datos de interés del agente. b) Material de prefabricado	2.1 El sistema crea el agente de acuerdo a los datos recibidos y finaliza el CU.
Flujo Alterno de Eventos	
Sección “Crear agentes”	

Acción del Actor	**Respuesta del Negocio**
2. La Aplicación pasa los siguientes datos: a) Datos de interés del agente. b) Material de prefabricado	4.1 El sistema verifica que el agente no haya sido creado. 4.2 Si el agente ya está creado muestra un error y finaliza el CU.
Sección "Actualizar agentes"	
Acción del Actor	**Respuesta del Negocio**
3. La Aplicación manda a actualizar al agente.	3.1 El agente toma los datos que el entorno le ofrece y finaliza el CU.
Sección "Eliminar agentes"	
Acción del Actor	**Respuesta del Negocio**
4. La Aplicación selecciona el agente que desea eliminar.	4.1 El agente es eliminado y finaliza el CU.
Poscondiciones	Quedan creados, actualizados o eliminados los agentes.

Tabla 4: Gestionar estados

CU 2	Gestionar estados
Actores	Aplicación
Resumen	El CU inicia cuando la aplicación necesita crear un estado, permitiendo que el mismo sea instanciado o eliminado.
Precondiciones	Debe ingresar el nombre del estado en su formulario para que se muestre el botón de crear
Referencias	RF4, RF5, RF6
Prioridad	Crítico
Flujo Normal de Eventos	
Acción del Actor	**Respuesta del Negocio**
1. La Aplicación llama a la funcionalidad: a) Crear estado. b) Actualizar estado. c) Eliminar estado.	1.1 El sistema inicializa las secciones "Crear estado", "Actualizar estado" y "Eliminar estado".
Sección "Crear estado"	

Acción del Actor	**Respuesta del Negocio**
2. La Aplicación pasa los siguientes datos: a) Nombre de estado	2.1 El sistema crea el estado de acuerdo a los datos recibidos y finaliza el CU.
Flujo Alterno de Eventos	
Sección "Crear estado"	
Acción del Actor	**Respuesta del Negocio**
2. La Aplicación pasa los siguientes datos: a) Nombre de estado	4.1 El sistema verifica que el estado no haya sido creado. 4.2 Si el estado ya está creado muestra un error y finaliza el CU.
Sección "Actualizar estado"	
Acción del Actor	**Respuesta del Negocio**
3. La Aplicación manda a actualizar estado.	3.1 Se crea una instancia del estado. 3.2 Se actualiza el estado agregando o eliminando la cantidad de transiciones que va a tener el esto y finaliza el CU
Sección "Eliminar estado"	
Acción del Actor	**Respuesta del Negocio**
4. La Aplicación selecciona el estado que desea eliminar.	4.1 El estado es eliminado junto con sus transiciones y finaliza el CU.
Poscondiciones	Quedan creados, actualizados o eliminados los estados.

Tabla 5: Asociar estados

CU 3	Asociar estados
Actores	Aplicación
Resumen	El CU inicia cuando la aplicación necesita asociar un estado con otro creando de esta forma las transiciones entre estados.
Precondiciones	Debe existir un conjunto de estados creados y actualizados.
Referencias	RF4, RF5, RF6
Prioridad	Crítico
Flujo Normal de Eventos	
Acción del Actor	**Respuesta del Negocio**
1. La Aplicación llama a la funcionalidad: a) Asociar estados.	1.1 El sistema inicializa la sección "Asociar estados".

Sección "Asociar estados"	
Acción del Actor	**Respuesta del Negocio**
2. La Aplicación pasa los siguientes datos al estado ya actualizado: a)Decisión. b)Estado en caso de decisión ser verdadera. b)Estado en caso de decisión ser falsa.	2.1 El sistema configura la transición entre estados de acuerdo a los dato recibidos y finaliza el CU.
Flujo Alterno de Eventos	
Sección "Asociar estados"	
Acción del Actor	**Respuesta del Negocio**
2. La Aplicación pasa los siguientes datos al estado ya actualizado: a)Decisión. b)Estado en caso de decisión ser verdadera. b)Estado en caso de decisión ser falsa.	4.1 El sistema no puede configurar la transición entre estados debido a que el estado no ha sido actualizado 4.2 Se requiere de forma manual actualizar el estado y crear la cantidad de transiciones que pueda tener.
Poscondiciones	Quedan asociados los estados y configuradas sus transiciones.

2.12 Conclusiones del capítulo

En el presente capítulo se detalló la solución propuesta, y quedan creadas las bases sobre las cuales se podrá iniciar la implementación del sistema haciendo uso de las herramientas seleccionadas. Además quedaron establecidos los requisitos funcionales y no funcionales de la aplicación final.

CAPÍTULO 3: Resultado de la investigación

En este capítulo se abordan los temas del diseño y de la implementación del sistema, basado en todo el trabajo acumulado a lo largo del desarrollo de los capítulos anteriores y mostrando la realización de los distintos artefactos correspondientes a cada uno de los flujos de trabajo analizados; así como el análisis de los resultados obtenidos al desarrollar la aplicación que dará cumplimiento a los objetivos planteados con anterioridad.

3.1 Diagrama de clases del diseño

Un diagrama de clases describe la estructura de un sistema mostrando las clases del sistema, sus atributos, operaciones (o métodos), y las relaciones entre los objetos. En la **Figura 14** se muestra el diagrama de clases del diseño del módulo de inteligencia artificial para agentes inteligentes.

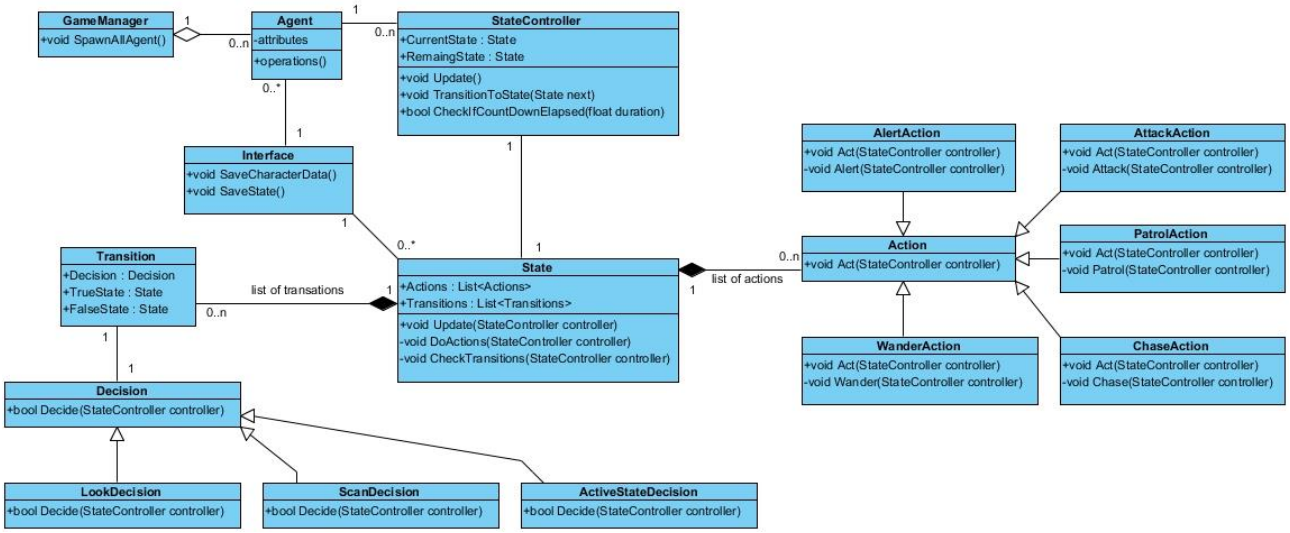

Figura 14 Diagrama de clases del diseño

3.1.1 Descripción de la estructura del sistema

La clase **StateController** es la encargada de ejecutar el estado inicial y actualizar el mismo mediante el método **Update**(). Esta función realiza una llamada al método **UpdateState** (**StateController** controller) de la clase **State**, es importante mencionar que con el uso de esta estructura hay que continuamente, pasar una referencia a la clase **MonoBehaviour** que esté llamando los métodos de la clase **State** (en este caso **StateController**). La clase **State** a su vez invoca a las demás funcionalidades de las clases del diagrama mediante los métodos **DoAction** (**StateController** controller) y **CheckTransition** (**StateController** controller). Este es el funcionamiento interno de las clases, pero en realidad mediante el uso de la clase **ScriptableObject** (la cual se utiliza en este módulo para crear objetos (*assets*) que sirven tanto para ejecutar código como para almacenar datos y no necesitan ser adjuntados a *GameObjects*), la clase

StateController lo que va a llamar es a los *assets* de tipo **State** que contienen una referencia para ejecutar código, en este caso el código de la clase **State**. Dentro de estos *assets* es que se van a crear las acciones, transiciones y las decisiones para realizar el cambio de estado. Cada uno de estos elementos serán al igual, creados como *assets* y configurados en el inspector, en los *assets* de tipo **State** creados. Esto significa que si se quiere añadir nuevos estados o acciones al sistema se puede hacer fácilmente y la conectividad entre las diferentes clases son creadas por el usuario en el inspector mediante el uso de *assets* y no directamente en el código. En la **Figura 15** se muestra la conectividad entre dos estados.

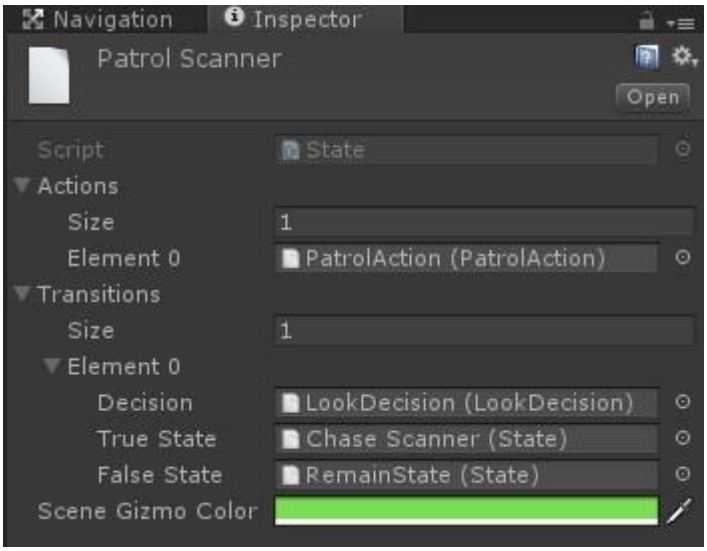

Figura 15 Conectividad entre clases

El *asset* **Patrol Scanner** de tipo **State** fue creado a partir de la clase **State** mediante el uso de la propiedad **CreateAssetMenu**, luego se añaden la cantidad de acciones (comportamientos del agente, como en el ejemplo: **PatrolAction**) que este estado realiza y las transiciones hacia otros estados. En este caso el estado **Patrol Scanner** va a tener un solo comportamiento que es el de patrullar y una sola transición hacia el estado **Chase Scanner**, está transición dependerá de la decisión **LookDecision**, esta decisión va a retornar un booleano que evalúa si el agente encuentra un enemigo en su rango de visión. En caso de que se encuentre con algún enemigo retorna *true* y se realiza la transición hacia el otro estado, sino, retorna *false* y se queda en el estado de patrulla.

3.2 Validación con prototipos

Una de las formas más habituales y convenientes de analizar el sistema consiste en construir un prototipo, pues es una versión operativa preliminar del sistema para fines de demostración y evaluación. “Una ventaja fundamental que presenta la construcción de prototipos desde el punto de vista de la validación radica en que estos modelos, una vez construidos, pueden ser evaluados directamente por los usuarios o expertos en el dominio para validar sobre ellos el análisis y el diseño

del sistema" [25].

3.2.1 Clases de prototipo

A continuación se exponen los tipos de prototipos que existen y sus características:

- **Prototipo Corregido:** Es la construcción de un sistema que funciona pero se corrige simultáneamente. En la ingeniería a este enfoque se le llama elaboración de una tabla experimental.
- **Prototipo No Funcional:** Modelo no funcional creado para probar ciertos aspectos del diseño.
- **Primer Prototipo de una serie:** Creación de un primer modelo a escala completa de un sistema llamado piloto.
- **Prototipo de características seleccionadas:** Creación de un modelo funcional que incluya algunas, pero no todas las características del sistema final.

Para validar la propuesta de solución se utilizará el prototipo de características seleccionadas. Mediante este modelo se pueden validar las funcionalidades del módulo desarrollado en diferentes proyectos de videojuegos.

3.3 Prototipo de características seleccionadas

En el presente epígrafe se presentan imágenes del prototipo construido teniendo en cuenta las funcionalidades definidas para el sistema.

Se utilizaron un conjunto de *assets* para simular un videojuego de tanques llamado Tank, en donde se utilizó la propuesta de solución, creando un conjunto de estados y sus transiciones. Para ello se diseño una máquina de estados finitos como la mostrada en la **Figura 16**.

Figura 16 Diseño de MEF

En la **Figura 17** la ventana que se muestra contiene los datos a ser llenados para crear un agente

inteligente con la máquina de estados finitos ya incluida como componente, y contiene para la validación de la propuesta tres estados mencionados en la **Figura 16**. En la **Figura 18** se muestra la asociación de una transición entre dos estados, la cantidad de transiciones que tendrá un estado la establece el diseñador de la MEF.

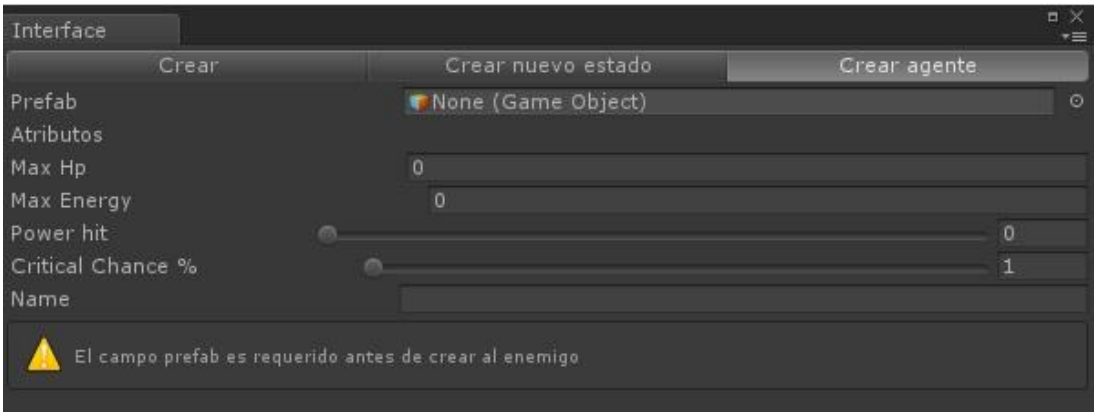

Figura 17: RF1 Crear agentes

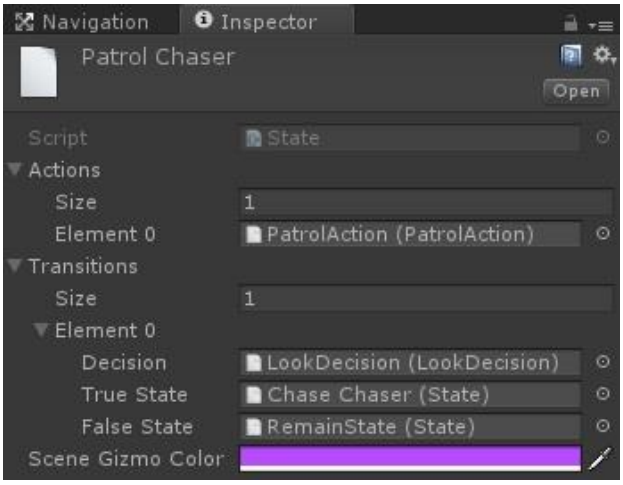

Figura 18: RF6 Asociar estados

La creación de comportamientos se realiza mediante la funcionalidad crear estados de la interfaz de *plugin*. Una vez creado el comportamiento, el mismo puede ser instanciado como un *asset* (**Figura 19**).

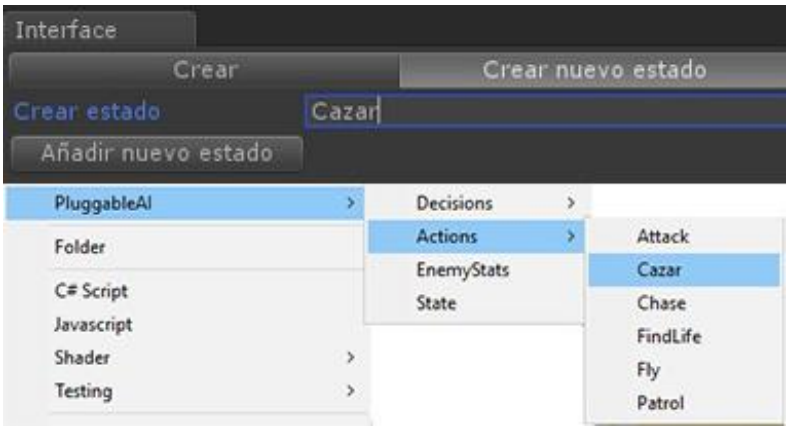

Figura 19: RF4 Crear estados

En la **Figura 20** y **Figura 21** se muestra como prueba final de validación mediante dos prototipos de videojuego en el que se integró exitosamente el módulo realizado de la propuesta de solución.

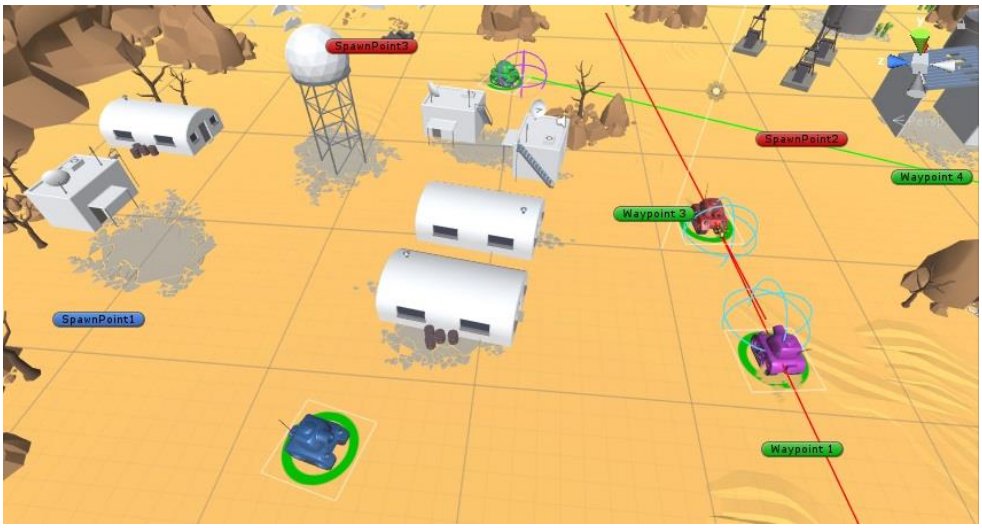

Figura 20: Primer prototipo donde se incorporó el *plugin* desarrollado

Figura 21: Segundo prototipo donde se incorporó el plugin desarrollado

3.4 Pasos para integrar el *plugin*

Para integrar el *plugin* realizado es necesario establecer una estructura de carpetas como la que se muestra en la **Figura 22** en el proyecto donde se quiera utilizar y copiar los scripts que se muestran en cada carpeta para que las funcionalidades trabajen correctamente. Se muestran un conjunto de scripts que sirven como guía para el desarrollador como por ejemplo: OrcoData, ActiveStateDecision, ScanDecision, EnemyStats y AttackAction. En el prototipo donde se integró la solución hay otros comportamientos definidos que pueden servir como una guía al igual.

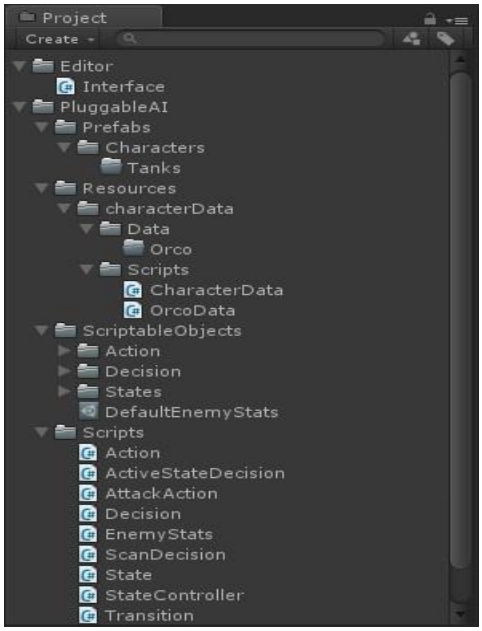

Figura 22: Estructura de carpetas

3.5 Conclusiones del capítulo

En este capítulo se explican los detalles de funcionamiento del plugin desarrollado y se valida la propuesta de solución mediante la elaboración de un prototipo de características seleccionadas. Los prototipos de videojuegos utilizados se acoplaron correctamente a las funcionalidades que brinda el *plugin*. Además se explica como integrar la herramienta a los proyectos y como utilizarla

CONCLUSIONES

Para el cumplimiento de los objetivos de esta investigación, fue necesario realizar un estudio de las técnicas de toma de decisiones y comunicación virtual de inteligencia artificial y cómo podrían ser incluidas en videojuegos. El estudio realizado permitió crear una arquitectura vinculada a una interfaz, que los desarrolladores podrán utilizar en sus videojuegos para facilitar la creación de máquinas de estados finitos e incorporarla a los agentes inteligentes, o usar la misma para controlar la transición entre escenas o niveles de los videojuegos. Además de permitir la creación de agentes en un entorno y permitir la toma de decisiones y comunicación virtual. La arquitectura elaborada posibilita también el trabajo tanto con agentes individuales como con un grupo de ellos, pudiéndosele incorporar todas las funcionalidades que poseen los sistemas multiagentes.

RECOMENDACIONES

- Utilizar el plugin desarrollado para la incorporación de técnicas de inteligencia artificial en los videojuegos del centro Vertex.
- Continuar con el desarrollo del *plugin* para agregarle nuevas técnicas como por ejemplo un sistema de pizarra y sería útil la incorporación de una máquina de estados jerárquicas.
- Agregar al *plugin* la funcionalidad de crear agentes con rasgos visuales que permitan diferenciar a los mismos.

GLOSARIO DE TÉRMINOS

***Bake*:** Permite preparar el área de navegación por la que caminan las entidades de un entorno.

Entorno virtual: Es el mundo por el cual se mueven las entidades inteligentes.

***Framework*:** Es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.

***Frames*:** También conocido como Tasa de Refrescamiento, es la velocidad (tasa) a la cual un dispositivo de imagen muestra imágenes llamadas cuadros o fotogramas. El término aplica igualmente a películas y cámaras de video, gráficos computacionales y sistemas de captura de movimiento. La tasa de refrescamiento se expresa en **Frames Per Second** (FPS) en Inglés.

Mesh: Una clase que permite crear o modificar las mallas de las escrituras.

Módulo: En programación un módulo es una porción de un programa de ordenador.

Módulo de inteligencia artificial: Término que se refiere a la parte de un sistema de software que le incorpora al mismo Inteligencia Artificial.

MonoBehaviour: Es una de las clases que forman el núcleo principal de las **API** de Unity. Contiene diversos eventos y funciones muy útiles a la hora de desarrollar un proyecto.

Plataforma: En informática, una plataforma es un sistema que sirve como base para hacer funcionar determinados módulos de *hardware* o de *software* con los que es compatible.

***Plugin*:** En informática, un *plugin* o complemento es una aplicación (o programa informático) que se relaciona con otra para agregarle una función nueva y generalmente muy específica.

Realidad virtual: Es una simulación generada por computadora de imágenes o ambientes tridimensionales interactivos con cierto grado de realismo físico o visual.

Renderizado: Es un término usado en informática para referirse al proceso de generar una imagen o vídeo mediante el cálculo de iluminación partiendo de un modelo en 3D. Este término técnico es utilizado por los animadores o productores audiovisuales y en programas de diseño en 3D como por ejemplo CINEMA 4D, 3DMax, Maya, Blender, Unity3D, etc.

RigidBody: Permite a los **GameObjects** actuar bajo el control de la física. El **Rigidbody** puede recibir fuerza y torque para hacer que sus objetos se muevan en una manera realista. Cualquier **GameObject** debe contener un Rigidbody para ser influenciado por gravedad, actué debajo fuerzas agregadas vía *scripting*, o interactuar con otros objetos a través del motor de física NVIDIA Physx.

***Runtime*:** Se denomina *runtime* (tiempo de ejecución) al intervalo de tiempo en el que un programa de computadora se ejecuta en un sistema operativo. Este tiempo se inicia con la puesta en memoria principal del programa, por lo que el sistema operativo comienza a ejecutar sus instrucciones. El intervalo finaliza en el momento en que éste envía al sistema operativo la señal de terminación, sea ésta una terminación normal, en que el programa tuvo la posibilidad de concluir sus instrucciones

satisfactoriamente, o una terminación anormal, en el que el programa produjo algún error y el sistema debió forzar su finalización.

***Scripts*:** Son documentos que contienen instrucciones, escritas en códigos de programación. El **script** es un lenguaje de programación que ejecuta diversas funciones en el interior de un programa de computador.

Serializable: Es un atributo que permite incrustar una clase con sub propiedades en el inspector.

BIBLIOGRAFÍA

[1] EcuRed, «EcuRed,» [En línea]. Available: https://www.ecured.cu/Videojuego#Videojuego. [Último acceso: 18 6 2018].

[2] J. McCarthy, «http://www-formal.stanford.edu/jmc/whatisai/whatisai.html,» 12 11 2007. [En línea]. Available: http://www-formal.stanford.edu. [Último acceso: 28 5 2018].

[3] S. O. Barriales, «Inteligencia Artificial en la industria del videojuego AAA. ¿Puede el mundo académico contribuir a su éxito?,» Asociación de Enseñantes Universitarios de la Informática, 23 4 2014. [En línea]. Available: http://www.aenui.net. [Último acceso: 28 5 2018].

[4] V. Botti, Agentes Inteligentes: El siguiente paso en la inteligencia artificial, 2000.

[5] Y. Coca Bergolla, «Agentes inteligentes. Aplicación a la realidad virtual,» *Revista Cubana de Ciencias Informáticas,* vol. 3, nº 1-2, pp. 49-54, 2009.

[6] C. Llamas Bello, Introducción a los Agentes y Sistemas Multi-Agentes, Valladolid, 2000.

[7] A. Caglayan, Agent Sourcebook, 1997.

[8] H. y. V. P. L. Sánchez Bernillo, «Propuesta y técnica para la Comunicación entre Agentes Virtuales,» La Habana, 2008.

[9] S. H. Gracia, «Módulo de cooperación entre agentes virtuales para el sistema de percepción,» La Habana, 2011.

[10] L. Vallejo Díaz, Propuesta de arquitectura para el sistema de gestión automatizado de recursos humanos GESTAPro., 2009.

[11] M. A. Mendoza Sánchez, Metodología de Desarrollo de Software, 2004.

[12] G. B. M. S. G. L. R. S. M. D. González Muñoz C., TecsMedia: Análisis Motores gráficos y su aplicación en la industria, 2015.

[13] EcuRed, «EcuRed,» [En línea]. Available: https://www.ecured.cu/Unity3D. [Último acceso: 1 06 2018].

[14] U. Technologies, «Unity Technologies,» 8 12 2015. [En línea]. Available: https://docs.unity3d.com/es/current/Manual/MonoDevelop.html. [Último acceso: 1 6 2018].

[15] A. F. O. P. Training, «Aula Formativa Online Professional Training,» 24 5 2014. [En línea]. Available: http://blog.aulaformativa.com. [Último acceso: 1 6 2018].

[16] L. M. V. Benítez., «Estructura para la incorporación de Inteligencia Artificial en videojuegos,» La Habana, 2011.

[17] U. Technologies, «Unity Technologies,» 2015. [En línea]. Available: https://docs.unity3d.com/es/current/Manual/PhysicsSection.html. [Último acceso: 4 6 2018].

[18] E. S. d. Lima, «Artificial Intelligence. Lecture 03 – Finite State Machines,» 2018.

[19] C. T. School of Computing Science, «Newcastle University,» 2016. [En línea]. Available: https://research.ncl.ac.uk/. [Último acceso: 4 6 2018].

[20] D. Tsung, «Don't Re-invent Finite State Machines: How to Repurpose Unity's Animator,» 31 7 2016. [En línea]. Available: https://medium.com/. [Último acceso: 4 6 2018].

[21] U. Documentation, «Unity Technologies,» 2016. [En línea]. Available: https://docs.unity3d.com. [Último acceso: 4 6 2018].

[22] M. G. Mejia, «Ingeniods,» 11 9 2013. [En línea]. Available: https://ingeniods.wordpress.com/. [Último acceso: 4 6 2018].

[23] R. Fine, «Overthrowing the MonoBehaviour Tyranny in a Glorious ScriptableObject Revolution,» 2016.

[24] I. y. o. Jaconson, El Proceso Unificado de Desarrollo de Software, 2000.

[25] J. E. K. A. N. KENNETH E. KENDALL, 2005. [En línea]. Available: http://es.scribd.com/doc/16126833/Analisis-ydiseno-de-sistemas-Kendall-Kendall. [Último acceso: 2018 6 20].