Universidad de las Ciencias Informáticas
Vertex, Entornos Interactivos 3D, Facultad 4

# Inteligencia Artificial para Enemigos en Comportamientos Grupales

**Trabajo de diploma para optar por el título de Ingeniero en Ciencias Informáticas**

Autor: Ariel Duque de Estrada Santiago

Tutor: Ing. Roberto Elías Pérez Ozete

Co-tutor: Msc. Marvyn Amado Márquez Rodríguez

**La Habana, 2017**

*Cada vez que decimos: «No sé », nos cerramos la puerta de nuestra propia fuente de sabiduría, que es infinita.*
*Louise L. Hay.*

# Dedicatoria

*A mis abuelos.*

# Agradecimientos

*A toda mi familia y amigos que nunca dejaron que me rindiera, a todos los que me apoyaron y me dieron fuerzas para seguir adelante cada día.*
*En estos años han sido muchos los contratiempos, pero siempre hubo alguien que me extendió su mano, siempre estuvo alguien a mi lado, un buen amigo, un buen amor, alguna que otra decepción pero que siempre me ayudó a superarme a mí mismo, a brincar esos obstáculos que aparecían en el camino.*
*A todas esas personas, aunque haya muchos que no puedan estar presentes, o no los pueda ver nunca más, a todos ellos, les estoy muy agradecido por ayudarme a estar aquí hoy.*
*Quiero agradecer además a todos mis amigos que dedicaron horas extras y esfuerzos para ayudarme a tener este trabajo listo, en especial a Ernesto Leyva, a Carlos A. Gavilla, a mis tutores, Roberto y Marvyn que en más de una ocasión me extendiendieron su apoyo. No quiero que se me quede nadie sin mencionar, por eso agradezco en general, porque se que son muchos, a todos,*
*Gracias.*

## Declaración de autoría

Declaramos ser autores de la presente tesis y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales sobre esta, con carácter exclusivo.

Para que así conste firmamos la presente a los ___ días del mes de _____ del año _____.

Ariel Duque de Estrada Santiago
Autor

Ing. Roberto Elías Pérez Ozete
Tutor

Msc. Marvyn Amado Márquez Rodríguez
Tutor

# Resumen

El presente trabajo está dirigido al desarrollo de un *plugin*[1] que se incorpore a la herramienta *Unity* para crear personajes. De esta manera se conformarían grupos con estos que incluyan la Inteligencia Artificial (IA) en comportamientos grupales.

Para la realización de este trabajo se hizo un estudio sobre los principales estilos de videojuegos, en específico los del género de aventura y acción. Además se analizaron los comportamientos grupales de los personajes enemigos en los videojuegos. Siendo esto el principal objetivo del *plugin* una vez estén conformados los grupos de personajes. Para el desarrollo del mismo, se empleó el motor de videojuegos *Unity*. Para guiar el proceso de creación de la solución propuesta se utilizó la metodología ágil de desarrollo de *software* Programación Extrema.

Como resultado de este trabajo, se obtuvo un *plugin* que permitirá crear personajes y conformar grupos con estos, sobre una base de comportamiento inteligente. Con el propósito de incluirlos en videojuegos de aventura y acción para combates en tiempo real. Además se realizó un escenario demo en el cual se podrá ver el comportamiento de los grupos de personajes, creados por la propuesta de solución, frente a un ejemplo de jugador. Esto sirvió para validar el funcionamiento del módulo implementado.

**Palabras clave:** comportamientos grupales, IA, personajes, *plugin*.

[1] Un *plugin* es un programa que puede anexarse a otro para aumentar sus funcionalidades (generalmente sin afectar otras funciones ni afectar la aplicación principal). No se trata de un parche ni de una actualización, es un módulo aparte que se incluye opcionalmente en una aplicación (ALEGSA, 2014).

# Índice general

# Índice de figuras

# Índice de tablas

# Introducción

La informática es una ciencia en continua evolución que con el transcurso de los años se ha aplicado a casi todos los sectores sociales. El aumento del conocimiento acumulado sobre esta y el creciente desarrollo tecnológico de las Tecnologías de la Informática y las Comunicaciones (TIC), le ha dado un nuevo giro al mundo, proporcionando nuevas oportunidades al desarrollo científico.

El creciente avance de las TIC y desarrollo en la creación de equipos informáticos y el estudio e investigación de las técnicas de gráficos generados por computadoras, dieron surgimiento a la realidad virtual y a la Inteligencia Artificial (IA). De esta manera se dio un giro a la forma de entretenimiento de las personas, permitiéndoles conocer lugares a los que nunca habían ido antes e interactuar con otras personas, surgiendo así disímiles géneros de videojuegos.

Estos videojuegos cada vez se hacen más complejos, en cuanto a gráficos, alcance y dificultad, entre otros. De esta manera, el usuario adquiere mayor experiencia de juego. Esta dificultad se complejiza a medida que se hacen mejoras, donde se perfecciona el comportamiento de los personajes que aparecen en estos. La IA en un videojuego, se refiere a las técnicas utilizadas en computadoras para producir la ilusión de inteligencia en el comportamiento de los personajes no jugadores. (Schreiner, 2009)

Es apropiado pensar en la IA como el comportamiento inteligente exhibido por el equipo que ha sido creado, o tal vez los cerebros artificiales detrás de ese comportamiento inteligente. Su estudio no es necesariamente para el propósito de crear máquinas inteligentes, sino con el propósito de obtener un mejor conocimiento de la naturaleza de la inteligencia humana. Una IA capaz de resolver un problema que requiera de razonamiento, si llegara a resolverse por un ser humano, no es suficiente, también debe aprender y adaptarse para ser considerada inteligente. Todo lo que da la ilusión de inteligencia a un nivel adecuado hace los videojuegos mas inmersivos, desafiantes y, lo más importante, más divertidos. (Marrero, 2010)

La IA es uno de los puntos más importantes a la hora de estudiar y criticar un videojuego. Desde hace algunos años, las compañías han comenzado a crearlas de forma grupal en los enemigos para mejorar la experiencia del usuario en los combates en tiempo real. Estas técnicas se han ido perfeccionando con los años hasta llegar a crear grupos inteligentes de enemigos que se ayudan entre sí para eliminar al personaje principal y sus aliados. (Rodriguez, 2016)

Cuba en estos últimos años ha estado inmersa en el desarrollo de videojuegos. Como pionera

en este campo, la Universidad de las Ciencias Informáticas (UCI) ha creado un grupo de videojuegos en colaboración con los estudios de animación del Instituto Cubano del Arte e Industria Cinematográficos (ICAIC), lo cual ha sido los primeros pasos de una industria de videojuegos naciente. Actualmente el centro que se encarga de este desarrollo es el centro Vertex, el cual emplea la plataforma *Unity* en la realización de la mayoría de sus productos con fines lúdicos. Algunos de estos videojuegos realizados sobre esta plataforma son: “Aventuras en la Manigua”, “La Neurona”, “Especies Invasoras”, entre otros. De estos, el videojuego que posee la IA más interesante es “Especies Invasoras”, la cual emplea la toma de decisiones en un grafo conexo para verificar sus posibles movimientos y sus tácticas de ataque. Este estilo de IA no es el que normalmente se utiliza en juegos que poseen combates en tiempo real, además que no se han realizado mecánicas, según fuentes de información del centro, que puedan satisfacer un videojuego de acción, por lo que a la hora de desarrollarlo el proceso tomaría más tiempo, sobretodo en la creación de un sistema de combate entre enemigos y jugadores. No obstante, Vertex posee un grupo de *Assets* que se han utilizado anteriormente, pero no para videojuegos en 3D con este estilo de combate, sobre la plataforma *Unity*.

Teniendo en cuenta la situación antes descrita se plantea como **problema de investigación**: ¿Cómo agilizar el proceso de creación de comportamientos grupales enemigos para videojuegos de acción y aventura para combates en tiempo real sobre la plataforma *Unity*?

Se define como **objeto de estudio** la IA para personajes enemigos en los videojuegos y como **campo de acción**: El comportamiento grupal en enemigos para combates en tiempo real.

Para dar solución al problema científico de la presente investigación se plantea el siguiente **objetivo**: Desarrollar un componente para *Unity* que permita crear grupos de personajes enemigos sobre la base de un comportamiento inteligente.

**Posibles resultados:**

- Componente con interfaz en *Unity* que permita crear IA grupales de enemigos para combates en tiempo real.
- Escenario Demo con el funcionamientos del componente.
- Manual de funcionamiento.

Para dar solución a estos resultados, se proponen las siguientes **tareas de investigación:**

- Elaboración del marco teórico de la investigación.
- Estudio de las características de los sistemas de IA grupales para combates en tiempo real.
- Diseño e Implementación de un escenario demo para la validación.
- Diseño e implementación del *plugin*.

- Comprobación del funcionamiento *plugin* propuesto mediante las pruebas de caja blanca y de aceptación.

Para dar cumplimiento a estas tareas de investigación se emplearon **métodos científicos teóricos y empíricos**.

**Métodos Teóricos**:

- Histórico-Lógico: Este método teórico se utilizará para realizar el estudio del estado del arte. Además, para conocer los tipos de comportamiento grupales de los personajes enemigos en los videojuegos.
- Analítico-Sintético: Se utilizará para estudiar cómo se puede vincular el comportamiento inteligente con los personajes creados por el *plugin*.
- Modelación: Este método se utilizará para crear un prototipo funcional del *plugin*.

**Métodos Empíricos**:

- Análisis documental: Para seleccionar la información necesaria para la construcción del marco teórico.
- Pruebas: Para comprobar el desempeño del *plugin* elaborado.

El presente documento está estructurado en 3 capítulos. En el Capítulo 1 se presentan los elementos teóricos que sirven de base a la investigación del problema planteado. En el Capítulo 2 se propone un método de solución basado en la creación de personajes inteligentes. Finalmente en el Capítulo 3 se muestran los resultados del *plugin* desarrollado, y se exponen las pruebas que se le realizaron.

CAPÍTULO 1

# Fundamentación Teórica

## Introducción

En este capítulo se abordarán temas referentes a los videojuegos, profundizando en los diferentes géneros, así como la IA presente en los mismos. Se verá el comportamiento inteligente en enemigos en los videojuegos de aventura y acción para combates en tiempo real. Se definirán las metodologías y herramientas necesarias para el desarrollo de una propuesta de solución.

## 1.1. Formas de Caracterizar un Videojuego.

Al crear un videojuego, las compañías desarrolladoras toman en cuenta las tendencias a nivel mundial referentes a los gustos de los usuarios, técnicas empleadas y estilos de videojuego. La mayoría de los videojuegos son cada vez más complejos, ya sea por su dificultad, nivel gráfico, historia e incluso por la IA presente en ellos.

Esto les permite ser clasificados en diferentes géneros teniendo en cuenta factores como las reglas de juego, el tipo de interactividad con el jugador, sus objetivos, etc. A medida que han ido evolucionando los videojuegos, ha surgido una variedad de géneros en relación con los avances que la tecnología permite. Entre los géneros de videojuegos más populares están los de acción, estrategia, rol, aventura, rompecabezas, simulación y deporte; cada uno de ellos con varios subgéneros. Por otro lado, hay videojuegos que combinan elementos de más de un género, dando lugar a géneros mixtos (por ejemplo, rol - acción, aventura - acción, entre otros) (Mojica Ortega, 2014).

Existen otras formas de caracterizar los videojuegos como pueden ser por su temática (fantástico-medieval, futurista, de guerra), su complejidad (videojuegos AAA, videojuegos casuales), o su finalidad (educativos, promocionales, artísticos).

Por otra parte, también se diferencian unos de otros, incluso dentro de un mismo género, por

la perspectiva visual que adoptan (la posición de la cámara). Así, hay videojuegos con perspectiva 2D (ya sea con proyección paralela, vista lateral o vista cenital), 2.5D (mediante proyección isométrica, oblicua, entre otras), y 3D (en perspectiva fija, en primera persona, o en tercera persona) (Universidad de Palermo, 2013).

Cada videojuego tiene su propia característica que lo diferencia de los demás, pero todos presentan ciertas estructuras básicas al crearse que intervienen en su caracterización. Entre ellas se encuentran las reglas de juego, la interconectividad y los géneros que abarca.

### 1.1.1. Características de los Videojuegos de Aventura y Acción.

A diario se crean videojuegos o se mejoran los ya existentes para complacer los gustos de los millones de usuarios alrededor del planeta. Naturalmente, los videojuegos simulan entornos y situaciones virtuales en los que el jugador controla a uno o varios personajes (o cualquier otro elemento de dicho entorno), para superar una o varias metas u objetivos definidos por las reglas del videojuego.

Teniendo en cuenta el tipo de juego en cuestión, una partida puede ser efectuada entre el usuario y la máquina, dos o más personas en la misma, e incluso múltiples jugadores conectados a una red *Local Area Network* (*LAN*) o en línea por Internet; compitiendo entre sí o de forma cooperativa contra el ordenador. Para satisfacer los gustos de los usuarios y ganar seguidores, los realizadores han ido creando una amplia gama de géneros dirigidos a todo tipo de público.

Es por esto que el término “género” de un videojuego, se refiere a la manera en cómo se pueden clasificar los mismos, dependiendo fundamentalmente a la mecánica de juego, aunque existen otros factores como son la temática que representa, la estética visual, los cuales influyen al definir un género en específico. Algunos de los géneros más representativos son los videojuegos de acción, rol, estrategia, simulación, deportes y aventura (ElOtroLado, 2017).

El presente trabajo se enfoca solamente en los videojuegos de los géneros de acción y de aventura. Por esta razón se define como videojuego de acción al tipo de videojuego en que el jugador tiene que combinar ciertas habilidades como la agilidad, destreza, velocidad y tiempo de reacción. Este género es el más amplio de todos, abarcando subgéneros como videojuegos de lucha, videojuegos de disparos en primera persona y videojuegos de plataformas. Estos subgéneros se caracterizan por usar la violencia como principal atractivo o rasgo de interactividad, específicamente en combates con armas de fuego o cuerpo a cuerpo (ibíd.).

Por otro lado, los videojuegos de aventura fueron, en cierto modo, los primeros que se vendieron en el mercado, empezando por *Colossal Cave Adventure* en los años 1970. Este tipo de videjuego se hizo famoso con los videojuegos de la serie *Zork* y consiguió alcanzar cierto nivel de popularidad en los años 80 que duró hasta mediados de los 90. Los primeros videojuegos de aventura eran textuales (aventuras textuales, aventura conversacional o ficción interactiva) (Universidad de Palermo, 2013).

En estos, el jugador utiliza el teclado para introducir órdenes como “coger la cuerda” o “ir hacia el oeste” y el ordenador describe lo que pasa. Cuando el uso de gráficos se generalizó, los videojuegos de aventura textuales dejaron paso a los visuales (por ejemplo, con imágenes del lugar presente) que sustituyeron de este modo, las descripciones por texto que se habían vuelto casi obsoletas. Estos videjuegos de aventura con gráficos seguían, no obstante, sirviéndose de la introducción de texto. Además, sigue existiendo una comunidad de autores y jugadores activos de ficción interactiva (ibíd.).

Como subgénero de los videojuegos de aventura se encuentran los de aventura de rol. Estos se caracterizan por la interacción con el personaje, una historia profunda y una evolución del personaje a medida que la historia avanza. Para lograr la evolución generalmente se hace que el jugador se enfrasque en una aventura donde irá conociendo nuevos personajes, explorando el mundo para ir juntando armas, experiencia, aliados e incluso magia. La inclusión del *CD-ROM* permitió contar la historia más detallada, utilizando videos de duración media que hacen que el jugador se sienta como dentro de una película (ibíd.).

Aunque la mayoría de videojuegos de aventura incluyen una dosis baja de Juegos de Rol o (*Role Playing Games* (*RPG*), por sus siglas en inglés). Los videojuegos de rol puros se enfocan específicamente en subir experiencia y customización del personaje; en juegos como la saga *The Elder Scrolls*, el crear y personalizar un personaje puede llevar hasta 30 horas. Los *RPG* clásicos, inspirados en los juegos de tablero, realizan las batallas por turnos, es decir, el jugador usa su equipo y habilidades aprendidas para atacar mediante una serie de comandos y después debe quedar estático y esperar a recibir el ataque del otro jugador o de la IA. El mejor ejemplo de esto es *Final Fantasy*, y *Dungeons & Dragons* (ibíd.).

Entre la mayoria de videojuegos de rol clásicos que aún existe, se usa el combate en tiempo real, es decir, no hay pausas y ambos atacan al mismo tiempo. Aunque muchos de estos videojuegos están basados en mundos medievales fantásticos, se consideran videojuegos de rol sólo aquellos que tuviesen relación con este tipo de realidades. Mientras que otros usan otro tipo de temática como *Mass Effect*, que está basado en un mundo posterior al viaje extraplanetario o *Fallout* que está basado en un universo postnuclear (ibíd.).

En otra subcategoría de los videojuegos de rol se encuentran los *RPG* en línea o *Massive Multiplayer Online RPG* (*MMORPG*). Cada jugador crea un personaje y mediante una conexión a internet, entra a un mundo donde miles de jugadores se unen a la aventura, exploran, intercambian y evolucionan juntos. El juego más conocido y jugado de este subgénero es *World of Warcraft*, basado en el mundo creado por *Blizzard* para sus juegos de estrategia de la saga *Warcraft*. También existen otros videojuegos conocidos de este género como *FF XI, Warhammer online, EVE* o *RuneScape*. Estos son conocidos por lo adictivos que acostumbran ser, además de la gran cantidad de cultura popular que suelen generar a su alrededor (ibíd.).

El objetivo principal en estos géneros de videojuegos es superar una serie de niveles hasta

llegar al último con el jefe principal, para ello, el jugador debe ir superando oleadas de enemigos, incluyendo mini jefes. Un mini jefe es quien da paso al final de un nivel o serie de niveles. Para vencer a los jefes se utiliza el reconocimiento de patrones y la velocidad de reacción física. En la mayoría de los videojuegos, los jefes se programan mediante un patrón de ataques simple o con movimientos que el jugador aprende a través de la experiencia. Estos patrones de ataque regularmente contienen movimientos especiales que requieren de las habilidades del jugador para que este sea capaz de saltar, esquivar o bloquear ataques para luego golpear en ciertos puntos claves, siempre teniéndose un control sobre el manejo de los patrones para atacar (CreativeCommons, 2017).

De esta manera, los jugadores buscan sentir nuevas emociones y desafíos. Los desarrolladores que conocen sus necesidades deciden emplear las técnicas de IA para obtener personajes capaces de actuar por sí solos y con un alto grado de dificultad; llevando a que el usuario encuentre como un reto enfrentarse a ellos.

## 1.2. Inteligencia Artificial para los Videojuegos.

La industria de los videojuegos ha servido como base para probar y desarrollar muchas técnicas de IA; beneficiándose a la vez con modelos de inteligencia más realistas en sus productos de entretenimiento, que los hacen más atractivos para los usuarios (Trujillo Rivero y Márquez Rodríguez, 2008).

Debido a la variedad de videojuegos que existen en la actualidad, tras una investigación profunda realizada por el autor sobre el impacto de las diferentes tendencias de videojuegos, es posible llegar a la conclusión de que la clave del éxito se encuentra precisamente en la capacidad de entretener al usuario, presentándole rivales o retos capaces de poner a prueba sus habilidades con un adecuado nivel de inteligencia.

La calidad de la IA del videojuego se controla muchas veces mediante diferentes niveles de dificultad, que básicamente indican hasta qué punto se debe dotar de inteligencia a los elementos y cuántas probabilidades existen de que el comportamiento que se calcule como el mejor se lleve a cabo efectivamente.

Las técnicas a utilizar dependen mucho del tipo de videojuego que se esté diseñando, de la importancia que se le desee dar a la IA dentro del mismo, así como de la cantidad de recursos que se encuentren disponibles para ella. Estas se pueden clasificar en:

- **Deterministas:** El comportamiento del agente inteligente es especificado por completo; o sea, los programadores tienen que codificar todas las acciones explícitamente, dificultando y retrasando el proceso de desarrollo del videojuego. Las técnicas más usadas para coordinar estas acciones son las máquinas de estados finitos y los árboles de decisión (Trujillo Rivero y Márquez Rodríguez, 2008).

- **No deterministas:** Es todo lo contrario, el grado de incertidumbre es mayor, por lo que no se puede predecir el comportamiento que seguirá el agente. Generalmente usan técnicas como redes neuronales, algoritmos evolutivos o redes bayesianas, que facilitan el aprendizaje y la adaptación al entorno de los elementos inteligentes. No hay que codificar explícitamente todas las posibles situaciones en el videojuego, ya que los elementos inteligentes pueden incluso extrapolar sus comportamientos y desarrollar otros nuevos o emergentes (ibíd.).

### 1.2.1. Técnicas Deterministas.

Los desarrolladores de videojuegos han experimentado con la mayoría de las técnicas de IA; pero sin dudas las más sencillas, eficientes, fáciles de implementar, entender y depurar son las deterministas, por lo que han sido las más usadas en este campo. Además, el tiempo del que disponen las compañías para producir un videojuego es insuficiente para que los desarrolladores se decidan a experimentar con técnicas no deterministas que son muy difíciles de entender, implementar y probar.

De este grupo de técnicas se utilizan frecuentemente las máquinas de estado finitos, que definen una serie de estados en los que puede permanecer un elemento, así como las condiciones para que se produzcan transiciones entre ellos. Muchas veces se combinan con lógica difusa, para representar en lenguaje computacional conceptos imprecisos o nociones subjetivas como "cercano/lejano" (ibíd.).

En el videojuego "*The Sims*", la IA es implementada con máquinas de estados finitos difusas, además utilizan técnicas específicas de *A-Life2* para simular el comportamiento de organismos vivos (en este caso, personas que viven en familia). La base de la inteligencia en este videojuego es su motor de comportamiento, el cual asocia acciones posibles a cada objeto. Un *NPC* debe ser capaz de reconocer y comprender el espacio que le rodea, empleando para ello técnicas de percepción. Además debe ser capaz de buscar caminos para navegar, por lo que se le presta una especial atención a los algoritmos de búsqueda como *Dijkstra* o A*.

Los videojuegos de tablero como el ajedrez y el *backgammon* han usado árboles de búsqueda heurística con formidables resultados. Tradicionalmente en los videojuegos se han utilizado técnicas de planificación o guiones (del inglés *script*[1]) para definir la conducta de los agentes, al igual que los sistemas basados en reglas y los comportamientos grupales o de manadas. Los sistemas expertos son un tipo de sistema basado en reglas que definen el conocimiento para que los agentes autónomos se comporten de manera similar a un jugador experto (Trujillo Rivero y Márquez Rodríguez, 2008).

[1]Los *scripts* son programas, usualmente pequeños o simples, para realizar generalmente tareas muy específicas. Son un conjunto de instrucciones generalmente almacenadas en un archivo de texto que deben ser interpretados línea a línea en tiempo real para su ejecución (ALEGSA, 2014).

**Máquinas de Estados Finitos.**

Las máquinas de estados finitos (en inglés, *MFS*) permiten modelar el comportamiento de un sistema, especificando una serie de estados y condiciones que deben cumplirse para realizar las transiciones entre ellos. El sistema se encuentra siempre en un estado activo y, a partir de la información recibida en el sistema y los condicionales de las transiciones, se decide si se debe cambiar hacia otro estado o no (Duch i Gavalda, 2012).

Ventajas de las *MFS*:

- Su simplicidad las hace perfectas para desarrolladores con poca experiencia, ya que se diseñan e implementan de manera rápida.
- En el caso de las *MFS* deterministas se pueden predecir fácilmente las situaciones que provocan la transición, lo cual permite un mayor control de la depuración de la IA.
- Proporcionan una representación gráfica del comportamiento que permite determinar fácilmente cómo ir de un estado a otro y qué condiciones son necesarias para la transición (ibíd.).

Desventajas de las *MFS*:

- Generalmente son sistemas muy predecibles, sobre todo en el caso de una *MFS* determinista.
- Sin un buen diseño pueden ocurrir problemas durante la implementación, sobre todo si se trata de una máquina con muchos estados diferentes.
- Sólo sirven para aquellos casos en los que se puede separar el comportamiento en diferentes estados independientes y se pueden definir transiciones claras entre estados (ibíd.).

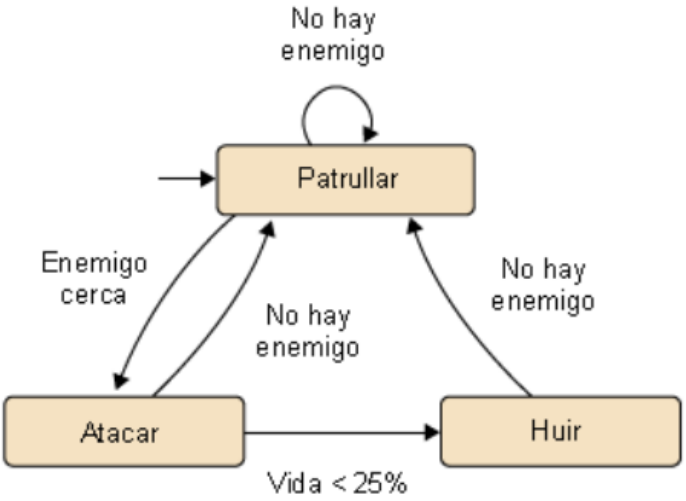



Figura 1.1. Ejemplo de una *MFS*. Tomado de (Duch i Gavalda, 2012)

**Árboles de Decisión.**

Un árbol de decisión es un sistema utilizado para analizar las diferentes opciones que los agentes pueden llevar a cabo. Se trata de una estructura en forma de árbol donde se colocan todas las decisiones que se pueden realizar y sus posibles consecuencias. La decisión en este tipo de árboles se lleva a cabo recorriéndolo desde la raíz hasta las hojas, para alcanzar así una decisión (ibíd.).

El árbol de decisión suele contener cuatro tipos de elementos diferentes:

- Nodos internos deterministas: contienen un test sobre la información de la situación actual que permitirá decidir qué rama elegir.
- Nodos internos probabilísticos: dependiendo de un evento aleatorio, se decidirá la siguiente rama.
- Hojas del árbol: representan el resultado que devolverá el árbol de decisión.
- Ramas del árbol: describen los posibles caminos que se extienden de acuerdo con la decisión tomada en los nodos internos (ibíd.).

Ventajas de los árboles de decisión:

- Son mucho más fáciles de entender e interpretar gracias a la representación gráfica de los otros elementos.
- Se puede seleccionar el nivel de profundidad de la decisión de una manera más clara, con lo que se puede ajustar a la capacidad computacional.

- Se puede combinar con otras técnicas de decisión para responder a cada una de las preguntas de los nodos intermedios (ibíd.).

Implementación de los árboles de decisión:

- Fase de diseño: durante la creación de un juego se deben introducir en el árbol todas las condiciones o reglas a tratar y las acciones que se encuentran en las hojas. Estos elementos se guardan en una estructura de datos en forma de árbol.
- Fase de análisis: en segundo lugar, se debe programar un algoritmo de análisis del árbol. El algoritmo descendie por este, evaluando las diferentes condiciones que se encuentre hasta que llegue a una hoja. Se deberá tener en cuenta que la evaluación puede ser determinista o probabilística, dependiendo del tipo de nodo interno utilizado.

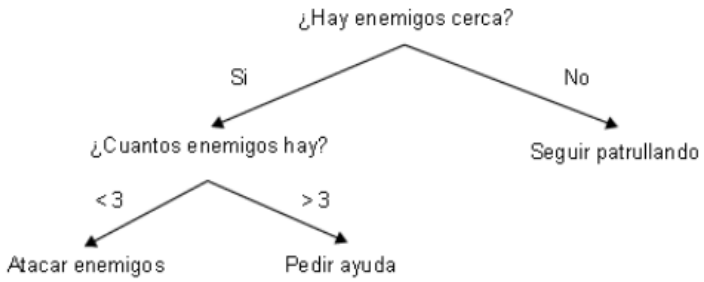



Figura 1.2. Ejemplo de un Árbol de Decisión. Tomado de (Duch i Gavalda, 2012).

### 1.2.2. Técnicas no Deterministas.

Con el avance de la industria tecnológica, el poder computacional se ha incrementado notablemente. Existen potentes tarjetas gráficas que liberan al procesador de la máquina para que pueda encargarse de otras tareas dentro del videojuego, por ejemplo la física y la IA. Esto, unido a la demanda de los usuarios de videojuegos más desafiantes, al deseo de los desarrolladores de lanzar un videojuego que marque la diferencia, así como al mayor interés de los "académicos" de la IA en los videojuegos como área de experimentación relativamente barata y sin riesgos, ha posibilitado que se comiencen a aplicar con mayor frecuencia técnicas no deterministas.

El reto actual de los desarrolladores es crear videojuegos novedosos, divertidos, realistas y con mayor vida útil. Un buen paso en ese sentido es lograr que los *NPC* aprendan, evolucionen, se adapten a nuevas situaciones y exhiban un comportamiento genuino. Estos resultados son posibles de alcanzar con métodos no deterministas que algunos desarrolladores investigan con gran interés a pesar de su complejidad.

Se han alcanzado excelentes resultados en videojuegos que usan redes neuronales, algoritmos genéticos o métodos probabilísticos como *Dirt Track Racing*, *Creatures*, *Black & White*, *Battlecruiser 3000AD*, entre otros. Generalmente se combinan con métodos deterministas más tradicionales y estudiados, conformando una especie de sistemas híbridos (Trujillo Rivero y Márquez Rodríguez, 2008).

Utilizando estas técnicas se pueden crear comportamientos inteligentes que controlen a los *NPC*. Esto resulta que puedan valerse por sí mismos e incluso coordinarse y actuar en colectivo.

### 1.2.3. Comportamientos Colectivos.

En algunas ocasiones, los enemigos en un videojuego forman parte de un colectivo que intenta conseguir un objetivo global. Para poder conseguir este objetivo, es necesario que los individuos cooperen entre sí para mejorar sus probabilidades de éxito.

Dentro de la IA han existido desde el principio varios métodos para implementar inteligencias colectivas o distribuidas. En el caso de los videojuegos, el más usado es el de los sistemas multi-agente (Duch i Gavalda, 2012).

#### Sistemas Multi-Agentes.

Un sistema multi-agente es una de las técnicas de IA más utilizadas para modelar el comportamiento de un conjunto de agentes, enfocándose más en el objetivo global de todo el sistema que en el objetivo individual de cada uno de los agentes que lo componen.

Un agente es visto como una entidad inteligente, que existe dentro de cierto contexto o ambiente, y que se puede comunicar por medio de un mecanismo de comunicación inter-proceso, usualmente un sistema de red, utilizando protocolos de comunicación (ibíd.).

Por esta razón, un sistema multi-agente necesita tres componentes claves:

- Una inteligencia local básica implementada en cada agente que le permite actuar de manera autónoma. Esta explica cómo ha de reaccionar el agente a partir de las percepciones que recibe del sistema.
- Un sistema de comunicación que permite enviar mensajes entre los agentes. Para esto se deben tener en cuenta dos elementos, primeramente, un gestor de mensajes; es decir, un componente que se encargue de que el mensaje se distribuya entre agente origen y destino de una manera rápida y eficiente. Y segundo, los tipos de mensajes a utilizar y sus posibles contenidos.
- Un sistema que permita a los agentes reaccionar cada vez que reciban un mensaje. Esta reacción provocará que este deje de comportarse de manera individual y pase a actuar

en favor del grupo. Ya sea para ayudar a un compañero o para atacar simultáneamente al jugador (ibíd.).

El objetivo principal de los sistemas multi-agente es que no debería existir un agente central que coordine las acciones de todos, sino que todos deberían encontrarse en el mismo nivel. No obstante, dado que en un videojuego se busca conseguir una sensación de realismo óptima utilizando el mínimo número de recursos, sería útil emplear un coordinador central de agentes (ibíd.). De esta manera, los personajes tendrían un comportamiento inteligente más realista.

## 1.3. Inteligencia Artificial en Videojuegos de Aventura y Acción.

Los desarrolladores han usado durante años la IA para dar vida a personajes aparentemente inteligentes en innumerables videojuegos, principalmente los personajes enemigos. En los videojuegos no siempre se da a los personajes no jugadores inteligencia de nivel humano. Tal vez se escribe código para controlar las criaturas no humanas, tales como dragones, robots, o incluso roedores; hacer algunos personajes no inteligentes puede agregar al videojuego variedad y riqueza.

Aunque es cierto que a menudo la IA de los videojuegos está llamada a resolver problemas bastante complejos, esta se puede emplear para elementos más simples como tener personajes con diferentes personalidades o que actúen según las acciones del jugador. Es aquí donde juega un papel importante la inclusión de la IA en los enemigos *NPC*, logrando que estos actúen, ya sea, individuales o como equipo para atacar al jugador y causarle el mayor daño posible. Esto conlleva que el usuario gane interés en el videojuego y quiera superarse y vencer a estos enemigos.

Este tipo de *NPC* puede presentar comportamientos pasivos en algunos casos, aunque generalmente poseen comportamientos hostiles hacia el jugador, sirviendo esto como desencadenante de peleas entre jugador humano y personaje no jugador, ayudando a incrementar los puntos de habilidades y experiencia al personaje del usuario (Marrero, 2010).

### 1.3.1. Comportamiento de IA en Videojuegos de Aventuras y Acción.

En los combates llevados a cabo en los videojuegos de aventura y acción, la IA debe ser eficiente y más humana, o al menos parecerlo. Una de las características más importantes que debe poseer en este tipo de videojuegos es la habilidad para cazar. En un principio los *NPC* se comportaban de manera sencilla, por ejemplo, si el jugador se encontraba en un área específica, estos reaccionaban, ya sea de manera ofensiva o defensiva.

No obstante, a medida que avanza el desarrollo de los videojuegos y la inteligencia de los *NPC* se hace cada vez más fuerte y casi real, la idea de caza tomada auge. En esta, la IA de los *NPC* puede buscar marcadores, tales como los sonidos hechos por el personaje o huellas que

pudieron haber dejado atrás. De esta forma, el jugador puede tener en cuenta si se acerca o evita a un enemigo.

Recientemente se ha incorporado a los comportamientos inteligentes en estos tipos de videojuegos el llamado instinto de supervivencia. Así, el enemigo puede reconocer diferentes objetos en el área donde se encuentre y determinar si son útiles o no para su supervivencia. Al igual que el usuario, el *NPC* se cubre, gracias al comportamiento que se le ha implementado, chequea su estado de salud y se defiende. Este establece una serie de marcadores que le ayudan a reaccionar de una forma determinada.

Por ejemplo, si en la implementación del personaje se ejecuta un comando para comprobar los puntos de vida a través de un combate, a continuación, otros comandos se pueden activar de forma tal que este reaccione de manera específica a un cierto porcentaje de salud. Si la salud está por debajo de un nivel determinado, entonces el comportamiento del *NPC* se puede configurar para que este se aleje del jugador y lo evite hasta que otra función se active. Este comportamiento en los *NPC* hace que luzcan más realistas y humanos, aunque todavía existe la necesidad de continuar la mejora de estos. Trayendo como principal limitante al intentar sorprender al jugador, que la IA necesita ser programada para todos los escenarios posibles (Schreiner, 2009).

### 1.3.2. Comportamiento de IA en Enemigos para Ataques Cuerpo a Cuerpo.

Los *NPC* enemigos presentan una serie de movimientos y combinaciones al enfrentarse en un ataque cuerpo a cuerpo. En algunos videojuegos, estos tienen un solo ataque, mientras que en otros, poseen múltiples ataques realizando combos de golpes. Así se evidencian algunos factores importantes como el daño que son capaces de producir, la capacidad de bloquear o interrumpir los ataques del jugador y defenderse.

Muchos de los videojuegos ofrecen al usuario la ventaja de enseñarle cómo pelear contra los enemigos, pero hay casos de videojuegos que no presentan esto y resultan ser un tanto más difíciles de jugar, ganando de esta manera seguidores que gustan de los retos y de la experiencia de usuario. Existe una característica fundamental en la mayoría de los enemigos, la cual se conoce como “aviso”, esta se basa en hacer saber al jugador que están a punto de atacar, siendo esto importante al tratar de comprender la legibilidad del sistema de combate.

Mientras más fuerte sea el enemigo y mayor repertorio de ataques posea, más claro será el “aviso” posibilitando que el jugador responda, ya sea, evitando el ataque, bloqueándolo, o simplemente respondiendo y pelear contra el enemigo. Esta característica en algunos videojuegos, permite al jugador prepararse para el ataque, incluso le muestra cómo evitar o bloquear algunos golpes. En otros videojuegos, se hace más difícil de evidenciar, puesto que hay un mayor número de enemigos en un mismo ataque y el jugador debe estar a la expectativa, siendo esto un grado de dificultad en el videojuego, lo cual llama la atención de los jugadores (Vossen, 2015).

Es importante que, en este tipo de combate, la IA de los enemigos se diseñe con una serie

de propósitos, que posibilite que el *NPC* enemigo desafíe al jugador constantemente, esto es con el objetivo de que el usuario se vea cada vez más motivado por el videojuego. Existen cuatro categorías principales para clasificar y separar a los enemigos en los combates cuerpo a cuerpo, estas son:

- **Enfatizadores:** estos enemigos se enfocan en hacer que los jugadores utilicen contra ellos ataques y habilidades específicas para derrotarlos, no obstante, es posible vencerlos además de manera regular.
- **Ejecutores:** estos enemigos hacen que el jugador se vea obligado a usar un tipo específico de ataque y de armamento para derrotarlos, siendo los demás ataques inútiles contra ellos.
- **Destrozadores:** este tipo de enemigo es más fácil de derrotar y presenta la característica de permitir al jugador divertirse destrozándolos.
- **Retadores:** esta clase es la más temeraria, puesto que constantemente están desafiando al jugador y probando sus habilidades, esto conlleva que el usuario se esfuerce para derrotarlo.

Estas clases se combinan en los videojuegos de ataque cuerpo a cuerpo para dar una mayor experiencia de usuario, usualmente se presentan los enemigos que más fácil se pueden derrotar, brindando confianza al jugador, después se combinan los de dificultad media y fuerte, haciendo que el jugador pruebe una serie de ataques y armamentos específicos y aprenda cuáles debe usar para poder vencer a estos *NPC* (Vossen, 2015).

De esta manera, se busca llamar la atención de los usuarios, haciendo más atractivo el modo de juego, aumentando la dificultad y ganando seguidores a lo largo del mundo del videojuego.

### 1.3.3. Comportamiento de IA en Enemigos para Ataques a Distancia.

En este tipo de ataques, la IA en los enemigos cambia, llevándolos a no enfocar la atención en los ataques cuerpo a cuerpo. De esta forma se diferencian los enemigos unos de otros al darles un arma característica de cada uno, tanto para los enemigos en combates cuerpo a cuerpo como para los enemigos de combates a distancia.

En la mayoría de los videojuegos que incluyen ataques a distancia, el jugador pelea en primera persona. Mientras que está ocupado peleando con enemigos en un combate cuerpo a cuerpo, pueden aparecer enemigos en el ataque que no usen esta característica, sino que se centren en atacar desde lejos al jugador, con ataques esporádicos pero que combinados con los de otros enemigos pueden infligir bastante daño al personaje.

En estas combinaciones de ataque entre enemigos de uno o varios tipos, ya sea cuerpo a cuerpo o distancia, la IA juega un papel importante, ya que se encarga de controlar cómo serán los ataques y quiénes los realizaran, puesto que los enemigos están conscientes de que hay otros

atacando y estos pueden o esperar o atacar desde lejos. Dando paso así a los comportamientos grupales en los ataques combinando diferentes personajes (ibíd.).

## 1.4. Comportamientos Grupales en Ataques para Videojuegos de Aventura y Acción.

Existen videojuegos de acción y aventura en que los enemigos *NPC* se agrupan, forman equipos y se coordinan, resultando ser impredecibles en algunas ocasiones, dificultando su derrota por parte del jugador. Estos equipos pueden estar conformados entre 3 y 5 *NPC*, con ciertas características y habilidades que juntos, ponen resistencia al usuario. Entre estos personajes, se encuentran tres tipos básicos, los de ataques cuerpo a cuerpo, los de ataque a distancia y los de magia.

Un grupo conformado por estos tres tipos puede tener varias formas de coordinarse. Los *NPC* de ataque cuerpo a cuerpo pueden estar rodeando al personaje principal, atacar y moverse a su alrededor, de forma tal que el jugador se enfoque en ellos mientras que los *NPC* de ataque a distancia usan sus habilidades para atacar, ya sea con diferencia de algunos segundos entre los ataques de los enemigos de ataque cuerpo a cuerpo como simultáneamente y variar su posición o permanecer estáticos. Junto a estos, los magos pueden lanzar ataques que ralenticen al jugador, o lo debiliten, de manera que puedan curar a sus compañeros. Siendo esta una de las tácticas de ataque más empleada en la mayoría de los videojuegos, tales como *WorldOfWarcraft, DarkSould, DungeonSide*, entre otros.


Figura 1.3. Comportamiento Grupal en Ataques.

En videojuegos tales como *Reckoning*, como se muestra en la imagen, los grupos de enemigos son pequeños, no exceden los cinco integrantes. Estos pueden o no caracterizarse como un equipo de enemigos, es decir, pueden atacar por iniciativa propia y tener libre movimiento o tener

uno o varios jefes que digan cuándo y a quién atacar.

La mayoría de los ataques ocurren de manera grupal, caracterizándose por existir dos grupos, uno cercano al jugador y otro más distante. El grupo más cercano de enemigos, generalmente compuesto por algunos miembros, representa la amenaza más inmediata que debe enfrentar el jugador. Mientras que el otro grupo que se encuentra más allá del jugador, contiene el resto de los enemigos. De esta manera los enemigos se pueden coordinar entre sí para derrotar al jugador, ya sea realizando combos de ataques de de corto, mediano y largo alcance.

Estos ataques son coordinados mediante máquinas de estado o árboles de decisión. Estas son las principales técnicas de toma de decisión de IA encargadas de controlar el tiempo entre ataques y cómo serán las combinaciones usadas para vencer o intentar derrotar al jugador. Intercambiándose de esta forma los enemigos de ambos grupos, moviéndose alrededor del personaje, ajustándose al movimiento del mismo (ibíd.).

Una vez finalizado el estudio correspondiente a la investigación y analizados los puntos más escenciales para la misma, se procede a analizar las metodologías de desarrollo de *software* que guiarán la propuesta de solución.

## 1.5. Metodologías de Desarrollo de *Software*.

El desarrollo de un *software* de alta calidad, en el tiempo planificado, con los menores costos posibles y que además satisfaga las necesidades y expectativas del cliente, requiere que durante todo el ciclo de vida de este, se trabaje de forma organizada. Para esto se debe controlar y documentar toda la información relacionada con el proyecto, prever los posibles riesgos que se puedan presentar durante su ejecución y definir las medidas para mitigarlos. La correcta realización de estas tareas depende en gran medida del empleo de una metodología eficaz que se adapte a las características del producto deseado.

Es por ello, que un cambio en algún momento del desarrollo de un *software*, que no sea la fase inicial cuando se realiza la captura de requisitos, puede devenir en atrasos en los cronogramas trazados o pérdidas monetarias. En base a este problema existen metodologías que son en cierta medida más o menos sensibles a estos cambios. Las mismas son conocidas como **tradicionales**, para el caso en que estas son mucho menos adaptables a un cambio en los requerimientos y **ágiles**, en el caso que estas sean más abiertas al mitigar atrasos o pérdidas monetarias por un cambio en la concepción del producto por parte del cliente (Pressman, 2010).

Las metodologías tradicionales, están guiadas por una rigurosa planificación durante todo el proceso de desarrollo, en el cual se establecen estrictamente las actividades a realizar. Están orientadas a proyectos de gran envergadura, para los cuales se definen una gran cantidad de roles y artefactos a generar, contando con una detallada documentación (Canós, Letelier y Penadés, 2003; Letelier, 2006). Entre estas metodologías se destaca *Rational Unified Process* (*RUP*).

- *RUP*: Basada en componentes e interfaces bien definidas. Esta metodología define un marco de trabajo genérico, el cual puede especializarse para una gran variedad de sistemas de *software*, en diferentes áreas de aplicación, diferentes tipos de organizaciones, diferentes niveles de aptitud y diferentes tamaños de proyecto. Esta comprende tres principios claves: dirigido por casos de uso, centrado en la arquitectura e iterativo e incremental. Esto significa que los casos de uso describen los requisitos funcionales del sistema, desde la perspectiva del usuario. Estos no solo inician el proceso de desarrollo sino que también proporcionan un hilo conductor, en el que se verifica, luego de la implementación, que el producto implemente adecuadamente cada caso de uso. Por otra parte, la arquitectura de un sistema permite tener una visión común entre los desarrolladores y los usuarios, teniendo una vista del diseño completo del *software* con las características más importantes resaltadas. Además *RUP* propone un proceso iterativo e incremental, donde el trabajo se divide en partes más pequeñas. Cada una de estas partes se puede ver como una iteración de la cual se obtiene un incremento que produce un crecimiento en el producto (Jacobson, Booch y Rumbaugh, 1999).

Las **metodologías ágiles**, son aquellas que están orientadas a la producción de código, con ciclos de desarrollo cortos. Se dirigen por grupos pequeños, haciendo hincapié en aspectos humanos asociados al trabajo en equipo. Están orientadas a proyectos pequeños, los cuales pueden presentar cambios durante su realización e involucran activamente al cliente en el proceso de desarrollo (Canós, Letelier y Penadés, 2003; Letelier, 2006). Entre estas metodologías se encuentran:

- Programación Extrema o *Extreme Programming* (*XP*): Basada en la simplicidad, la comunicación y la reutilización de código. Esta metodología trata de darle al cliente el *software* que él necesita y cuándo lo necesita, lo que implica responder rápidamente a las necesidades de este. Además tiene como objetivo potenciar al máximo el trabajo en grupo, donde los clientes y los desarrolladores son parte del equipo de trabajo y están involucrados en el proceso de desarrollo del *software* durante todo su ciclo de vida. Esta metodología se recomienda para proyectos de corto plazo, poniendo más énfasis en la adaptabilidad que en la previsibilidad. Es definida especialmente como una metodología adecuada para proyectos con requisitos imprecisos y muy cambiantes. Esta metodología intenta reducir la complejidad del *software* por medio de un trabajo orientado directamente al objetivo, basado en las relaciones interpersonales y la velocidad de reacción (Calero, 2003; Escribano, 2002).

- SCRUM: Esta metodología requiere trabajo duro, puesto que no se basa en el seguimiento de un plan, sino en la adaptación continua a las circunstancias de la evolución del proyecto. Esta metodología además, está indicada especialmente para proyectos con un rápido cambio de requisitos. El desarrollo de *software* se realiza mediante iteraciones, denominadas

*sprints*, con una duración de 30 días. Donde el resultado de cada *sprint* es un incremento ejecutable que se muestra al cliente. Otro aspecto importante de esta metodología son las reuniones diarias de 15 minutos que esta propone. En estas se debate lo que se ha hecho desde la última reunión, los posibles obstáculos que atentan contra la productividad del equipo, así como las tareas que se realizarán antes de la siguiente reunión (Pressman, 2010).

No obstante, no existe una metodología universal que permita hacer frente con éxito a cualquier proyecto de desarrollo de *software*. Ya que toda metodología debe ser adaptada al contexto del proyecto, supóngase recursos técnicos y humanos, tiempo de desarrollo o tipo de sistema, entre otros (Bautista, 2014).

Se decide utilizar la metodología *XP*, ya que muchas de sus características son aplicables al contexto de realización del proyecto. Fundamentalmente, esta es ideal para equipos pequeños (en este caso, una persona), es de código sencillo y permite que el cliente sea parte del equipo de desarrollo. Se diseñan e implementan las pruebas antes de programar la funcionalidad y cada integrante tiene autoridad para cambiar cualquier parte del código fuente. Es una metodología que propone integraciones continuas y está diseñada para adaptarse a los cambios de requisitos que puedan surgir al comenzar el proyecto.

Al establecer la metodología a utilizar, se analizan las herramientas necesarias para dar solución al objetivo de este trabajo.

## 1.6. Herramientas para el Desarrollo.

*Unity* es un motor de videojuegos para *PC* (*Personal Computer* o Computadora Personal) y *Mac* que viene empaquetado como una herramienta para crear juegos, aplicaciones interactivas, visualizaciones y animaciones en 2D, 3D y en tiempo real. *Unity* puede implementar contenido para múltiples plataformas como *PC*, *Mac*, *Nintendo Wii* e *iPhone*. También puede publicar juegos basados en web usando el componente *Unity Web Player*.

El contenido del videojuego es construido desde el Editor del programa usando *scripts*. Esto significa que los desarrolladores no necesitan ser expertos en *C++* para crear videojuegos con *Unity*, ya que las mecánicas de juego son compiladas usando una versión de *JavaScript* o *C#*. *Unity* utiliza además mallas de navegación, estas son estructuras de dato abstractas, utilizadas en aplicaciones de IA para asistir a los agentes en la búsqueda de caminos a través de espacios complicados (Unity, 2016).

### 1.6.1. Plugin e Interfaces.

Un *plugin* es un programa que puede anexarse a otro para aumentar sus funcionalidades (generalmente sin afectar otras funciones ni afectar la aplicación principal). No se trata de un parche ni de una actualización, es un módulo aparte que se incluye opcionalmente en una aplicación (ALEGSA, 2014).

En *Unity*, para crear un *plugin*, se utiliza el Editor, de esta manera se podrá añadir una nueva funcionalidad a la barra de herramientas. Para ello se deben seguir tres pasos (Unity, 2016):

- Crear un *script* que se derive de la Ventana del Editor.
- Usar el código para activar la ventana y mostrarla.
- Implementar el código *GUI* para la herramienta que se está desarrollando.

#### *Script* que deriva de la Ventana del Editor.

Para crear una nueva ventana en el Editor, el *script* debe ser almacenado adentro de una carpeta llamada “Editor”. Luego se debe hacer una clase en este *script* que derive de la Ventana del Editor para establecer los controles *GUI* en el interior de la función *OnGUI* (ibíd.). De esta manera:

```
//C# Example

using UnityEngine;
using UnityEditor;
using System.Collections;

public class Example : EditorWindow

    {
        void OnGUI () {
            // The actual window code goes here
           }
    }
```

Figura 1.4. Ejemplo de Cómo Derivar la Ventana del Editor. Tomado de (Unity, 2016)

#### Activar y Mostrar la Ventana.

Con el fin de mostrar la ventana en la pantalla, se crea un elemento del menú que lo muestra. Esto se logra creando una función que es activada por las propiedades del elemento *MenuItem*.

El comportamiento por defecto en *Unity* es reutilizar ventanas, de esta manera, seleccionando el *item* del menú nuevamente va a mostrar las ventanas existentes. Esto es hace a través la función *EditorWindow.GetWindow* (ibíd.). De esta manera:

```
//C# Example

using UnityEngine;
using UnityEditor;
using System.Collections;

class MyWindow : EditorWindow {
    [MenuItem ("Window/My Window")]

    public static void  ShowWindow () {
        EditorWindow.GetWindow(typeof(MyWindow));
    }

    void OnGUI () {
        // The actual window code goes here
    }
}
```

Figura 1.5. Ejemplo de Cómo Activar y Mostrar la Ventana. Tomado de (Unity, 2016).

Esto creará una ventana de editor estándar acoplable, que guarda su posición entre las invocaciones, pudiendo ser utilizada en diseños personalizados.

**Implementar el *GUI* de la Ventana.**

Los contenidos de la ventana son renderizados al implementar la función *OnGUI*. Se pueden utilizar las mismas clases *UnityGUI* que usa el *GUI* dentro del par (*GUI* y *GUILayout*). Adicionalmente, se proporcionan algunos controles *GUI*, ubicados solamente en las clases del editor *EditorGUI* y *EditorGUILayout*. Estas clases se suman a los controles que ya están disponibles en las clases normales, por lo que se pueden mezclar y combinar a voluntad (ibíd.). De esta manera:

```
//C# Example
using UnityEditor;
using UnityEngine;

public class MyWindow : EditorWindow
{
    string myString = "Hello World";
    bool groupEnabled;
    bool myBool = true;
    float myFloat = 1.23f;

    // Add menu item named "My Window" to the Window menu
    [MenuItem("Window/My Window")]
    public static void ShowWindow()
    {
        //Show existing window instance. If one doesn't exist, make one.
        EditorWindow.GetWindow(typeof(MyWindow));
    }
    void OnGUI()
    {
        GUILayout.Label ("Base Settings", EditorStyles.boldLabel);
        myString = EditorGUILayout.TextField ("Text Field", myString);

        groupEnabled = EditorGUILayout.BeginToggleGroup ("Optional Settings", groupEnabled);
            myBool = EditorGUILayout.Toggle ("Toggle", myBool);
            myFloat = EditorGUILayout.Slider ("Slider", myFloat, -3, 3);
        EditorGUILayout.EndToggleGroup ();
    }
}
```

Figura 1.6. Ejemplo de Cómo Agregar Elementos *GUI* a la Ventana del Editor Personalizada. Tomado de (Unity, 2016).

Una vez concluida la implementación de la nueva ventana, esta quedaría así:

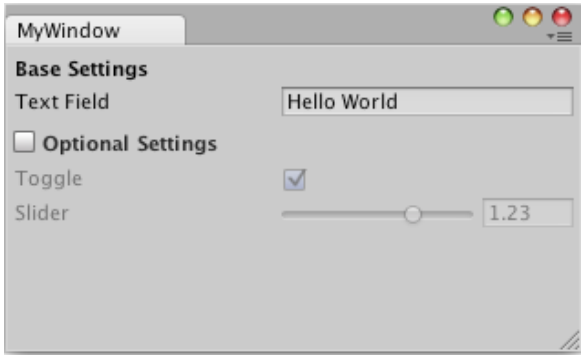



Figura 1.7. Ejemplo de Cómo Quedaría la Ventana. Tomado de (Unity, 2016).

### 1.6.2. Mallas de Navegación. Técnica y Algoritmos.

Una malla de navegación es una colección de polígonos convexos bidimensionales que define qué areas de un entorno son transitables por los agentes. En otras palabras, un caracter en un juego puede moverse libremente por aquellas áreas que no estén obstruidas por algún tipo de barrera que forme parte del ambiente. Estos polígonos están conectados entre sí en un grafo.

La representación de las áreas atravesables en forma de 2D simplifica los cálculos que de otra manera tendrían que hacerse en el entorno 3D, pero a diferencia de una cuadrícula 2D, permite que las áreas que se superponen lo hagan por encima y por debajo a diferentes alturas. Los polígonos de varios tamaños y formas en mallas de navegación pueden representar entornos arbitrarios con mayor precisión que las redes regulares (Golodetz, 2013; Unity, 2016).

Esta utiliza la Búsqueda de Caminos o *Pathfinding* como técnica para hallar el camino entre dos polígonos. El *Pathfinding* dentro de uno de estos polígonos se puede hacer trivialmente en una línea recta, debido a que el polígono es convexo y atravesable. Mientras que la búsqueda de caminos entre polígonos en la malla puede hacerse con uno de los algoritmos de búsqueda de grafos, como *A**. Los agentes en una malla de navegación pueden de esta forma, evitar chequeos de detección de colisión computacionalmente caros, con obstáculos que forman parte del ambiente. Además emplea otros algoritmos tales como Recorrido en Profundidad, Recorrido en Amplitud y *Dijkstra* (Duch i Gavalda, 2012).

#### Algoritmos

En cierto momento se necesita saber el camino que debe recorrer un elemento para ir de un punto A hasta un punto B. Para ellos se emplean estos algoritmos:

- Recorrido en Profundidad: Este es el algoritmo más sencillo para buscar un camino en un grafo y consiste en visitar todos los nodos posibles a partir del inicial. En este caso, se cuenta con una lista de nodos chequeados y, a medida que se vayan tratando, se introducirán aquellos vecinos de cada uno de los nodos. Se debe tener además, una lista de nodos visitados para evitar ciclos. Si se toma el grafo como un árbol, se visitarán antes los nodos hijos que los hermanos (ibíd.).

- Recorrido en Amplitud: Este algoritmo es similar al anterior. Aquí la prioridad de búsqueda a partir de un nodo del árbol son primero sus vecinos y después sus hijos. En el caso de que el coste entre nodos sea fijo, este algoritmo asegura encontrar el camino mínimo (ibíd.).

- *Dijkstra*: Este algoritmo consiste en generar un árbol en el que la raíz es el vértice de inicio y va ramificándose por todos los nodos del grafo sin crear ciclos. El algoritmo asegura que, utilizando estas ramificaciones, el camino entre un vértice y cualquier otro sea mínimo (Duch i Gavalda, 2012).

- *A**: Este algoritmo permite buscar un camino entre dos nodos de un grafo. Se basa en la búsqueda de caminos en profundidad y en amplitud. Mientras que los dos primeros presentan algunas reglas fijas sobre qué elementos analizar primero (hijos o hermanos), el algoritmo *A** deduce en todo momento qué nodo es el que ha de visitarse mediante una función heurística. Básicamente, se posee una lista de nodos que se han visitado y los nodos a los que se sabe cómo llegar (a partir de los visitados). De los nodos a los que se puede llegar, se elige aquel que la función de heurística proporcione el de menor coste y que, en teoría, es el camino más probable. Si, a medida que se profundiza por un camino, éste se vuelve más costoso que sus alternativas, el algoritmo de selección desechará el camino y continuará por otro nuevo y más prometedor (ibíd.).

Al emplear la malla de navegación para permitir que los personajes se muevan por el escenario y utilizar el *Pathfinding* como técnica para la búsqueda de caminos; se usa el algoritmo *A** para el cálculo de los caminos mínimos. De esta forma se garantiza una solución óptima ya que es un algoritmo completo, o sea, en caso de existir una solución, siempre dará con ella.

### 1.6.3. Entorno de Desarrollo.

*Unity* integra *MonoDevelop* como entorno de desarrollo ya que es libre y gratuito y este fue diseñado primordialmente para *C#* y otros lenguajes *.NET*. Además este es el entorno de desarrollo incluido por defecto en *Unity* (Unity, 2016). Características de *MonoDevelop* que potencian su uso:

- Ambiente sumamente intuitivo y simple.
- La ayuda es muy completa e incluye una amplia gama de ejemplos.
- Posee autocompletado de sintaxis.
- Posee un navegador incorporado.
- Es multiplataforma.

La herramienta *Unity*, descrita anteriormente, es utilizada en el centro Vertex para el desarrollo de videojuegos. El *plugin* se realizará sobre esta y una vez terminado podrá ser integrado a la misma sin presentar problemas de compatibilidad. Utilizando las mallas de navegación y sus propiedades para que los personajes creados puedan moverse en la escena.

## 1.7. Lenguajes de Programación.

El motor de videojuegos *Unity* brinda a los programadores la posibilidad de contar con varios lenguajes de programación para el desarrollo de aplicaciones (ibíd.). Estos lenguajes son:

- *C# (C Sharp)*: Es el lenguaje de mayor confianza que utiliza *Unity* para el desarrollo de componentes en el Editor. Es un lenguaje de propósito general diseñado por *Microsoft* para su plataforma *.NET*. La sintaxis y estructuración de este lenguaje es muy similar a la de *C++*, ya que la intención de *Microsoft* era facilitar la migración de códigos escritos en este lenguaje y su aprendizaje por parte de los programadores habituados a *C++*. Además, el lenguaje de programación *C#* combina las mejores características de lenguajes preexistentes como *Visual Basic, Java* y *C++* (ibíd.).

- *JavaScript*: Es un lenguaje personalizado inspirado en la sintaxis *ECMAScript*. Es un lenguaje ligero e interpretado, orientado a objetos con funciones de primera clase, más conocido como el lenguaje de *script* para páginas web, pero también usado en muchos entornos sin navegador, tales como *node.js* o *Apache CouchDB*. Es un lenguaje *script* multi-paradigma, basado en prototipos, dinámico, soporta estilos de programación funcional, orientado a objetos e imperativo (MozillaDeveloperNetwork, 2017).

Los lenguajes de programación mencionados anteriormente brindan al programador potencialidades similares al trabajar con *Unity*, estos permiten exportar aplicaciones a las plataformas seleccionadas. Para desarrollar la aplicación se seleccionó el lenguaje de programación *C#*, ya que se cuenta con experiencia desarrollando en este lenguaje y se posee un mayor dominio de este que del resto.

## Conclusiones parciales

El análisis realizado en este capítulo de las principales técnicas de IA para la toma de decisiones y los comportamientos colectivos que controlan a los *NPC*, permitió seleccionar aquellas necesarias a emplear en la propuesta de solución. Al hacerse énfasis en los videojuegos de acción y aventura, se establece que el módulo a realizar es para los comportamientos grupales en combates de tiempo real en los mismos sobre la plataforma *Unity*. Además con los distintos tipos de IA presentes en los estilos de combates, ya sean individuales como grupales, se pueden definir los tipos de personajes que este módulo creará. Por otra parte, se seleccionó con la herramienta y el lenguaje de programación a utilizar en el desarrollo de la solución, guiado este proceso por la metodología ágil *XP*.

CAPÍTULO 2

# Propuesta de solución

## Introducción

El presente capítulo tiene como objetivo hacer una descripción de las principales características del *plugin* a desarrollar. Se hará mención de las primeras fases de la metodología de desarrollo a utilizar para la implementación de la solución que se propone y se expondrán los artefactos generados durante el transcurso de estas.

## 2.1. Propuesta de Solución.

El proceso de creación de un videojuego es bastante amplio, desde que se establecen la historia y la trama sobre las que se desarrollará, hasta que se incluyen los personajes. Esto normalmente se hace de forma manual, resultando tedioso para los desarrolladores en algunas ocasiones. Cuando se añaden los *NPC*, ya sea, de forma individual o grupal, hay que hacerlo de uno a uno, demorando el trabajo. Es por esta razón que para dar cumplimiento al objetivo de la investigación se propone crear el *plugin Artificial Intelligence for Groupal Behavior (A.I.G.B.)* para la herramienta *Unity*, así como el comportamiento inteligente que controlará a los personajes creados por este. Además se presentará un escenario demo en el cual se podrá ver el comportamiento de los grupos de personajes, creados por el módulo a desarrollar, al encontrarse con un ejemplo de jugador. Esto servirá para validar la propuesta de solución.

El *plugin* estará compuesto por tres elementos fundamentales:

- Crear Enemigo.
- Conformar Grupo.
- Lista de *NPC*.

Estos elementos permitirán al desarrollador crear una serie de enemigos *NPC* con sus características, habilidades y comportamiento de IA específicos, crear grupos con estos personajes de manera que se comporten como un equipo y se coordinen como tal. Hechos estos grupos, podrán ser incorporados a cualquier tipo de videojuego de acción y aventura para combates en tiempo real.

La opción Crear Enemigo estará activada por defecto, en esta existirán tres tipos de enemigos, Ataque Cuerpo a Cuerpo, Distancia y Magia, estos enemigos poseerán atributos en común, pero además, otros que los distingan entre sí. Una vez que se establezcan los valores para cada enemigo, este se creará y se le añadirá la malla del personaje modelado en dependencia del tipo que sea. Cada *NPC* se guardará en una base de datos para poder trabajar con ella en la funcionalidad Lista de *NPC*; de esta forma se garantiza el acceso a todos los personajes.


Figura 2.1. Prototipo Crear Enemigo

Una vez creado el *NPC*, en la opción Conformar Grupo, se podrán conformar equipos de 3 a 5 enemigos, donde se les asignará un comportamiento Activo o Variado. El comportamiento Activo consistirá en que todos los personajes que compongan el grupo creado, serán asignados como jefes. Esto traerá consigo que cuando el jugador se encuentre con cualquier *NPC*, este podrá llamar al resto de sus compañeros hasta la posición del jugador y comenzarán a atacarlo. Por otra parte, el comportamiento Variado, consistirá en que el desarrollador podrá seleccionar al crear un grupo, los personajes que desee como jefes. Esto traerá consigo que cuando el jugador se encuentre con los *NPC* jefes, estos llamarán al resto de sus compañeros hasta la posición de él y comenzarán a atacarlo. En caso de que se encuentre con un *NPC* que no es asignado como jefe, este solo lo perseguirá y lo atacará mientras se encuentre en su rango de acción.


Figura 2.2. Prototipo Conformar Grupo

Mientras tanto, en el componente Lista de *NPC*, se cargará la base de datos con todos los *NPC* que hayan sido creados, ya sea, en la ejecución del *plugin* como los ya creados en otras ocasiones. Esto permitirá modificar o eliminar los personajes, de manera que los modificados puedan ser reutilizados.

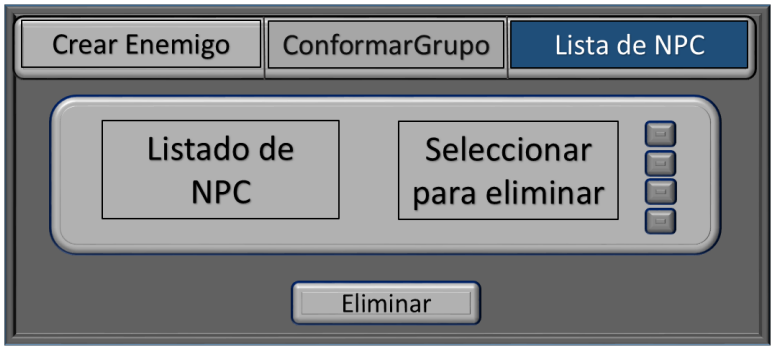



Figura 2.3. Prototipo Lista de *NPC*

Además del *plugin* a realizar, existirá una clase encargada de controlar las acciones de los *NPC*. Esta permitirá que los personajes, mientras no se hayan encontrado con el jugador, patrullen el mapa de forma que no se alejen mucho de su jefe. Esto lo harán seleccionando puntos aleatorios cercanos a la posición del mismo, una vez seleccionado el punto y haberse dirigido a su destino, esperarán un intervalo de tiempo y volverán a seleccionar otro punto. Una vez que se encuentren con el jugador, estos serán capaces de perseguirlo y atacarlo, moviéndose alrededor de él. Dependiendo del tipo definido, al atacar, se posicionarán a una distancia específica con respecto al jugador. Además, tendrán su propio tipo de ataque, lo cual los diferenciará. En caso de que algún compañero tenga a otro en su línea de fuego, este corregirá su pocisión y continuará

atacando. Si el jugador huye, los enemigos lo perseguirán, si este llega a una distancia lo suficientemente lejos, entonces, lo habrán perdido y volverán al estado de patrullar. Además permitirá que al encontrarse el jugador con el jefe de los enemigos, este pueda llamar a sus compañeros para que lo persigan y ataquen.

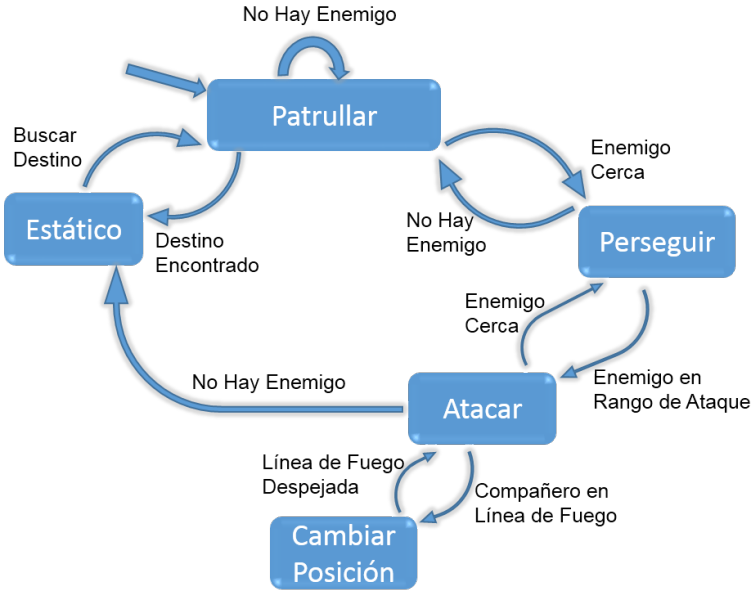



Figura 2.4. Prototipo de *MFS* para el Comportamiento de los *NPC*.

Para controlar estas acciones, se utilizarán como técnicas de IA, las *MFS* para los comportamientos individuales de los *NPC* y para los comportamientos grupales, los Sistemas Multi-Agentes. Para posibilitar que los *NPC* se muevan libremente en el mapa y no colisionen entre ellos, se utilizarán las mallas de navegación con el *Pathfinding* como técnica para la búsqueda de caminos, empleando el algoritmo *A**.

## 2.2. Fase de Exploración.

En esta fase los clientes plantean a grandes rasgos las Historias de Usuario (HU) que son de interés para la primera entrega del producto. Asimismo, el equipo de desarrollo se familiariza con las herramientas, tecnologías y prácticas que se utilizarán en el proyecto. Además, se prueba la tecnología y se exploran las posibilidades de la arquitectura del sistema construyendo un prototipo. La fase de exploración toma de pocas semanas a pocos meses, dependiendo del tamaño y familiaridad que tengan los programadores con la tecnología (Letelier, 2006).

### 2.2.1. Historias de Usuario

Las HU son utilizadas como herramienta para dar a conocer los requerimientos del sistema al equipo de desarrollo. Son pequeños textos en los que el cliente describe una actividad que realizará el sistema. Estos se redactan utilizando la terminología del cliente, de forma que sea clara y sencilla, sin profundizar en detalles. Se puede considerar que las HU en *XP* juegan un papel similar a los casos de uso en otras metodologías, pero en realidad son muy diferentes. Las HU solo muestran la silueta de una tarea a realizarse.

Por esta razón es fundamental que el usuario o un representante suyo se encuentre disponible en todo momento para dar respuesta a las dudas que puedan surgir, ya que estas no proporcionan información detallada acerca de una actividad específica. Las HU también son utilizadas para estimar el tiempo que el equipo de desarrollo tomará para realizar las entregas. En una entrega se puede desarrollar una o varias HU, esto depende del tiempo que demore la implementación de cada una de ellas (Echeverry Tobón y Delgado Carmona, 2007). Principales aspectos de una HU:

- **Número:** Número asignado a la HU.
- **Nombre de HU:** Atributo que contiene el nombre de la HU.
- **Fecha:** Fecha en la cual fue redactada la HU.
- **Usuario:** El usuario del sistema que utiliza o protagoniza la HU.
- **Prioridad en el negocio:** Contiene el nivel de prioridad de la HU en el negocio. Es Alta en caso de que la HU sea indispensable en el negocio, Media en caso de que su realización o no afecte el negocio y Baja cuando no se considera una prioridad.
- **Riesgo de desarrollo:** Contiene el nivel de riesgo en caso de no realizarse la HU. Es Alta, si el riesgo de no realizarla incide en el funcionamiento de la plataforma, Media si el riesgo de no realizarla es medianamente importante, y Baja en caso de que no se considere un riesgo tardar en su realización y no incida en el funcionamiento de la plataforma.
- **Puntos estimados:** Este atributo es una estimación hecha por el equipo de desarrollo sobre el tiempo de duración de la HU. Cuando el valor es 1 equivale a una semana ideal de trabajo, y un día equivale a 0.2 puntos.
- **Iteración asignada:** Especifica la iteración a la que pertenece la HU correspondiente.
- **Descripción:** Posee una breve descripción de lo que realizará la HU.
- **Observaciones:** Aquellos detalles relevantes que serán resueltos tras la conversación del equipo desarrollador con el cliente.

Tabla 2.1. Historia de usuario # 1

| **Historia de usuario** | |
|---|---|
| **Número:** 1 | **Nombre:** Crear Enemigo. |
| **Usuario:** Especialista | |
| **Prioridad en negocio:** Alta | **Riesgo en desarrollo:** Alta |
| **Puntos estimados:** 3.0 | **Iteración asignada:** 1 |
| **Programador responsable:** Ariel Duque de Estrada Santiago | |
| **Descripción:** El *plugin* será capaz de conformar el *NPC* con los parámetros que defina el usuario creando un objeto *prefab*[1]. | |
| **Observaciones:** | |

Tabla 2.2. Historia de usuario # 2

| **Historia de usuario** | |
|---|---|
| **Número:** 2 | **Nombre:** Crear Grupo. |
| **Usuario:** Especialista | |
| **Prioridad en negocio:** Alta | **Riesgo en desarrollo:** Alta |
| **Puntos estimados:** 3.0 | **Iteración asignada:** 2 |
| **Programador responsable:** Ariel Duque de Estrada Santiago | |
| **Descripción:** Con los personajes creados, el *plugin* podrá conformar grupos de 3 hasta 5 integrantes. A cada grupo se le podrá definir el comportamiento que tendrá, mediante el cual, sus miembros reaccionarán de una forma u otra ante el jugador. | |
| **Observaciones:** Para conformar un grupo, se pueden elegir personajes creados en el momento de ejecución del *plugin* o personajes creados en otras ocasiones y se encuentren guardados en la carpeta que los contendrá. | |

Tabla 2.3. Historia de usuario # 3

| **Historia de usuario** | |
|---|---|
| **Número:** 3 | **Nombre:** Listar *NPC*. |
| **Usuario:** Especialista | |
| **Prioridad en negocio:** Alta | **Riesgo en desarrollo:** Media |
| **Puntos estimados:** 1.5 | **Iteración asignada:** 2 |
| **Programador responsable:** Ariel Duque de Estrada Santiago | |
| **Descripción:** El *plugin* deberá mostrar en una lista todos los personajes que se hayan creado y brindará la opción de eliminar o modificar aquellos que el usuario considere necesario. | |
| **Observaciones:** | |

Tabla 2.4. Historia de usuario # 4

| **Historia de usuario** | |
|---|---|
| **Número:** 4 | **Nombre:** ComportamientoIA. |
| **Usuario:** Especialista | |
| **Prioridad en negocio:** Alta | **Riesgo en desarrollo:** Alta |
| **Puntos estimados:** 3.0 | **Iteración asignada:** 3 |
| **Programador responsable:** Ariel Duque de Estrada Santiago | |

Continúa en la próxima página

[1]Un *Prefab* permite almacenar un objeto *GameObject* completamente con componentes y propiedades. Este actúa como una plantilla a partir de la cual se pueden crear nuevas instancias del objeto en la escena. Cualquier edición hecha a un *prefab* será inmediatamente reflejada en todas las instancias producidas por él, pero, también se pueden anular componentes y ajustes para cada instancia individualmente(Unity, 2016)

Tabla 2.4. Continuación de la página anterior

| |
|---|
| **Descripción:** Se implementa la IA que estará presente en los personajes que se crearán con la herramienta desarrollada. Esta controlará el tipo de *NPC* que haya definido el usuario, podrá seleccionar si es jefe o no, en dependencia de esto podrá cumplir un papel específico. Permitirá que el personaje mientras no haya encontrado al jugador, pueda estar patrullando por el mapa, siendo este su estado inicial. Dependiendo del tipo definido, al atacar, se posicionarán a una distancia específica con respecto al jugador. Además, tendrán su propio tipo de ataque, lo cual los diferenciará. Si es jefe, llamará a sus compañeros a atacar y perseguir al jugador, si no lo es, entonces solo este perseguirá y atacará al jugador; en caso de que el jugador se aleje mucho, entonces el *NPC* volverá a su estado inicial. En caso de que cada *NPC* esté atacando al jugador, no permanecerán estáticos, sino que se moverán alrededor de él y si tienen a un compañero en su línea de fuego, ajustarán su posición y seguirán atacando. |
| **Observaciones:** |

## 2.3. Fase de planificación

En esta fase el cliente establece la prioridad de cada historia de usuario y se acuerda el alcance del producto. Los programadores estiman cuánto esfuerzo requiere cada historia y a partir de esto se define el cronograma en conjunto con el cliente. Una entrega debería obtenerse en no más de tres meses. El equipo de desarrollo mantiene un registro de la velocidad de desarrollo, totalizando el número de historias de usuario realizadas en una iteración. Esta medida ayuda a determinar la cantidad de historias que se pueden implementar en las siguientes iteraciones, aunque no de manera exacta. La revisión continua de esta métrica en el transcurso del proyecto se hace necesaria, ya que las historias varían según su grado de dificultad, haciendo inestable la velocidad de la realización del sistema (Anaya Villegas, 2007; Echeverry Tobón y Delgado Carmona, 2007).

### 2.3.1. Plan de Estimación.

El cliente es quien establece la prioridad de cada HU y los programadores realizan una estimación del esfuerzo necesario de cada una de ellas. Las estimaciones de esfuerzo asociadas a la implementación de las historias son establecidas por los programadores utilizando como medida el punto, lo que equivale a una semana ideal de programación.

Las historias generalmente valen de 1 a 3 puntos. La planificación se puede realizar basándose en el tiempo o el alcance. Esto es utilizado para estimar el tiempo que el equipo de desarrollo necesitará para realizar las entregas. En una entrega pueden desarrollarse una o varias historias de usuario, dependiendo del tiempo que demore la implementación de cada una de ellas (Escribano, 2002).

A continuación se muestra mediante una tabla la planificación de las diferentes HU para cada iteración teniendo en cuenta su prioridad.

Tabla 2.5. Estimación de esfuerzo por historia de usuario

| Iteración | Historias de usuario | | Puntos estimados (semanas) |
|---|---|---|---|
| 1 | 1 | Crear Enemigo | 3.0 |

Continúa en la próxima página

Tabla 2.5. Continuación de la página anterior

| | | | |
|---|---|---|---|
| 2 | 2 | Crear Grupo | 3.0 |
| | 3 | Listar *NPC* | 1.5 |
| 3 | 4 | ComportamientoIA | 3.0 |
| **Total** | | | **10.5** |

### 2.3.2. Plan de Iteraciones.

Como parte del ciclo de vida de un proyecto guiado por la metodología de desarrollo *XP*, se crea el plan de duración de las iteraciones que se llevarán a cabo durante el desarrollo del proyecto. Este plan tiene como objetivo fundamental mostrar la duración de cada una de las iteraciones en las que está dividida la fase de desarrollo del proyecto, así como el orden en que serán implementadas las HU en cada iteración según la prioridad asignada por el cliente. El plan de entrega está compuesto por iteraciones de no más de tres semanas. En la primera iteración se establece una arquitectura del sistema que pueda ser utilizada durante el resto del proyecto. Esto se logra escogiendo las historias que fuercen la creación de esta arquitectura, sin embargo, esto no siempre es posible ya que es el cliente quien decide qué historias se implementarán en cada iteración (Echeverry Tobón y Delgado Carmona, 2007).

Una vez definidas las HU y estimado el esfuerzo propuesto para la realización de cada una de ellas, se decide por parte del equipo de desarrollo realizar el sistema en 3 iteraciones, las cuales se describen de manera detallada a continuación:

**Iteración 1:** Esta iteración tiene como propósito dar cumplimiento a la HU 1. Esta compone la base del desarrollo del *plugin*. Una vez concluida la implementación de esta HU, el sistema podrá crear los personajes *NPC*. De esta iteración pueden derivarse nuevos requerimientos funcionales. Por último se realizan las pruebas pertinentes a la funcionalidad implementada. Finalizado esto se efectúa la primera entrega del *plugin*, para mostrar al cliente lo realizado y recibir sus valoraciones en relación con esta entrega.

**Iteración 2:** El objetivo de esta iteración es dar cumplimiento a las HU 2 y 3. Al finalizar la implementación de dichas HU el sistema deberá ser capaz de conformar los grupos y listar todos los *NPC* creados. Luego se pasará a solucionar los errores y no conformidades que fueron detectadas en esta iteración. Una vez concluida la misma, el *plugin* contará con sus principales funcionalidades implementadas y se realizará la segunda entrega a los clientes.

**Iteración 3:** En esta iteración se implementa la HU 4. Una vez finalizada la implementación de esta, el sistema tendrá incorporada una clase encargada de controlar la IA correspondiente a cada *NPC*. Luego de probar las funcionalidades implementadas y corregir los errores detectados en esta iteración, se contará con una versión estable del producto lista para entregar la primera versión completa al cliente.

### 2.3.3. Plan de Duración de las Iteraciones.

Como parte del ciclo de vida de un proyecto guiado por la metodología de desarrollo *XP*, se crea el plan de duración de las iteraciones que se llevarán a cabo durante el desarrollo del proyecto. Este plan tiene como objetivo fundamental mostrar la duración de cada una de las iteraciones en las que está dividida la fase de desarrollo del proyecto, así como el orden en que serán implementadas las HU en cada una según la prioridad asignada por el cliente.

Tabla 2.6. Plan de duración de las iteraciones

| Iteración | Historias de usuario | | Duración (semanas) |
|---|---|---|---|
| 1 | 1 | Crear Enemigo | 3.0 |
| 2 | 2 | Crear Grupo | 4.5 |
| | 3 | Listar *NPC* | |
| 3 | 4 | ComportamientoIA | 3.0 |
| **Total** | | | **10.5** |

### 2.3.4. Plan de Entrega.

Como resultado de la fase de planificación se genera el plan de entrega que plantea una fecha para la entrega de cada iteración, tomando dos días como holgura para completar la misma.

Tabla 2.7. Plan de entregas

| Iteración | Entrega | Fecha |
|---|---|---|
| 1 | Versión 0.3 | 24 de marzo del 2017 |
| 2 | Versión 0.8 | 28 de abril del 2017 |
| 3 | Versión 1.0 | 26 de mayo del 2017 |

### 2.3.5. Tareas de Ingeniería.

Las historias de usuario son descompuestas en tareas de ingeniería y asignadas a los programadores para ser implementadas en cada iteración (Canós, Letelier y Penadés, 2003).

Tabla 2.8. Tarea de ingeniería # 1

| **Tarea** | |
|---|---|
| **Número de tarea:** 1 | **Número de historia de usuario:** 1 |
| **Nombre de la tarea:** Implementación del componente para visualizar los datos de los personajes a crear. | |
| **Tipo de tarea:** Desarrollo | **Puntos estimados:** 1.5 |
| **Fecha de inicio:** 1 de marzo de 2017 | **Fecha de fin:** 10 de marzo de 2017 |
| **Programador responsable:** Ariel Duque de Estrada Santiago | |
| **Descripción:** Se implementa un componente que conforma la interfaz con la que interactúa el desarrollador al crear un personaje. | |

Tabla 2.9. Tarea de ingeniería # 2

| **Tarea** | |
|---|---|
| **Número de tarea:** 2 | **Número de historia de usuario:** 1 |
| **Nombre de la tarea:** Implementación de la clase para procesar los datos del personaje y crearlo. | |
| **Tipo de tarea:** Desarrollo | **Puntos estimados:** 1.5 |
| **Fecha de inicio:** 13 de marzo de 2017 | **Fecha de fin:** 24 de marzo de 2017 |
| **Programador responsable:** Ariel Duque de Estrada Santiago | |
| **Descripción:** Se encarga de procesar la información y crear el objeto personaje con los datos entrados por parámetros. | |

Tabla 2.10. Tarea de ingeniería # 3

| **Tarea** | |
|---|---|
| **Número de tarea:** 3 | **Número de historia de usuario:** 2 |
| **Nombre de la tarea:** Implementación del componente para conformar grupos con los personajes creados. | |
| **Tipo de tarea:** Desarrollo | **Puntos estimados:** 1.0 |
| **Fecha de inicio:** 27 de marzo de 2017 | **Fecha de fin:** 31 de marzo de 2017 |
| **Programador responsable:** Ariel Duque de Estrada Santiago | |
| **Descripción:** Se implementa un componente que conforma la interfaz con la que interactúa el desarrollador al crear grupos de personajes. | |

Tabla 2.11. Tarea de ingeniería # 4

| **Tarea** | |
|---|---|
| **Número de tarea:** 4 | **Número de historia de usuario:** 2 |
| **Nombre de la tarea:** Implementación de la clase para procesar los personajes creados y conformar grupos con ellos. | |
| **Tipo de tarea:** Desarrollo | **Puntos estimados:** 1.5 |
| **Fecha de inicio:** 3 de abril de 2017 | **Fecha de fin:** 12 de abril de 2017 |
| **Programador responsable:** Ariel Duque de Estrada Santiago | |
| **Descripción:** Se encarga de visualizar los personajes creados y de conformar grupos con estos asignándoles su comportamiento. | |

Tabla 2.12. Tarea de ingeniería # 5

| **Tarea** | |
|---|---|
| **Número de tarea:** 5 | **Número de historia de usuario:** 3 |
| **Nombre de la tarea:** Implementación de un componente para visualizar, modificar y eliminar los *NPC* creados. | |
| **Tipo de tarea:** Desarrollo | **Puntos estimados:** 1.0 |
| **Fecha de inicio:** 13 de abril de 2017 | **Fecha de fin:** 19 de abril de 2017 |
| **Programador responsable:** Ariel Duque de Estrada Santiago | |
| **Descripción:** Se implementa un componente que permita la visualización de todos los *NPC* creados por el desarrollador, así como modificarlos y eliminarlos. | |

Tabla 2.13. Tarea de ingeniería # 6

| **Tarea** | |
|---|---|
| **Número de tarea:** 6 | **Número de historia de usuario:** 3 |
| **Nombre de la tarea:** Implementación de una clase para mostrar la información de los personajes creados y guardar cada cambio realizado a estos. | |
| **Tipo de tarea:** Desarrollo | **Puntos estimados:** 1.0 |
| **Fecha de inicio:** 20 de abril de 2017 | **Fecha de fin:** 28 de abril de 2017 |
| **Programador responsable:** Ariel Duque de Estrada Santiago | |
| **Descripción:** Se encarga de mostrar en una lista cada *NPC* creado con su información referente, posibilitando guardar cada cambio realizado sobre estos. | |

Tabla 2.14. Tarea de ingeniería # 7

| **Tarea** | |
|---|---|
| **Número de tarea:** 7 | **Número de historia de usuario:** 4 |
| **Nombre de la tarea:** Implementación de una clase para crear la IA referente a los *NPC*. | |
| **Tipo de tarea:** Desarrollo | **Puntos estimados:** 3.0 |
| **Fecha de inicio:** 1 de mayo de 2017 | **Fecha de fin:** 26 de mayo de 2017 |
| **Programador responsable:** Ariel Duque de Estrada Santiago | |
| **Descripción:** Esta clase es la encargada de crear la IA que controlará los *NPC*. Dependiendo del tipo definido, al atacar, se posicionarán a una distancia específica con respecto al jugador. Además, tendrán su propio tipo de ataque, lo cual los diferenciará. | |

## 2.4. Fase de Diseño.

En *XP* se considera que no es posible tener un diseño completo y sin errores del sistema desde el principio, dada la naturaleza cambiante del proyecto. Por lo tanto, hacer un diseño muy extenso en las fases iniciales del proyecto para luego modificarlo, se considera un malgasto de tiempo. Es por ello, que esta tarea es permanente durante la vida del proyecto, partiendo de un diseño inicial el cual va siendo corregido y mejorado durante el proceso de desarrollo (Echeverry Tobón y Delgado Carmona, 2007).

### 2.4.1. Arquitectura.

La arquitectura de *software* es la estructura de un sistema, la cual comprende los componentes por los que está formado, las propiedades de estos componentes visibles externamente, y las relaciones entre ellos. En el contexto del diseño arquitectónico, un componente de *software* puede ser tan simple como un *plugin*, pero también puede ser algo tan complicado como incluir bases de datos que permita la configuración de una red de clientes y servidores. Las propiedades de los componentes son aquellas características necesarias para entender cómo los componentes interactúan con otros componentes, dejando de lado las propiedades internas específicas de cada uno. Las relaciones entre los componentes pueden ser tan sencillas como una llamada de

procedimiento de un *plugin* a otro, o tan complicadas como el protocolo de acceso a bases de datos.

La arquitectura constituye un modelo relativamente pequeño y comprensible de cómo está estructurado el sistema y de cómo trabajan sus componentes en conjunto. Esta brinda la posibilidad de analizar la efectividad del diseño para darle cumplimiento a los requisitos fijados. De esta forma, el diseño arquitectónico y los patrones arquitectónicos pueden ser aplicados en el diseño de otros sistemas y representados a través de un conjunto de abstracciones que facilitan la descripción de la arquitectura de un modo predecible (Pressman, 2010).

**Patrón Arquitectónico por Capas.**

En una arquitectura basada en capas el sistema se organiza jerárquicamente, de manera tal que cada capa brinde servicios a la capa superior y se sirva de las prestaciones que le brinda la inferior. En esta arquitectura los componentes de la capa externa se encargan de la interacción con el usuario mediante una interfaz. En la capa interna, los componentes realizan operaciones exclusivas al ámbito del *software*, como la obtención de datos, la interacción con el sistema operativo o el control de un dispositivo. Mientras, las capas intermedias proporcionan mecanismos necesarios para que la capa externa muestre los datos obtenidos por la capa inferior, controlando la lógica del sistema.

Esta arquitectura permite la creación de un diseño basado en varios niveles de abstracción, en el cual un problema complejo puede dividirse en partes más pequeñas para darle solución. A cada nivel se le confía una misión simple, lo que permite el diseño de una arquitectura escalable, la cual puede ampliarse con facilidad en caso que las necesidades aumenten. En una arquitectura basada en capas, cada nivel puede ser modificado de forma transparente. Conociendo la interfaz de la capa inferior y manteniendo la utilizada por la capa exterior, puede ser modificado el funcionamiento de la capa intermedia con el fin de optimizar un algoritmo o modificar el tratamiento de los datos (Pressman, 2010). A continuación se muestra la arquitectura por capas del componente para *Unity* a desarrollar.

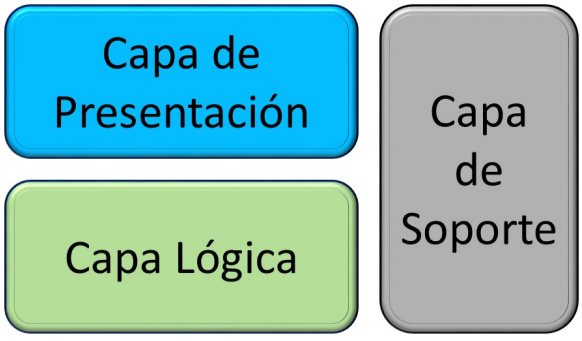



Figura 2.5. Arquitectura del *plugin* Grupo*NPC*.

**Capa de Presentación.**

Esta es la capa con la que interactúa el usuario. En esta se encuentran las clases que permiten la interacción con el *plugin*, así como las estructuras a través de las cuales son devueltos los datos capturados.

**Capa lógica.**

En esta capa se controla el proceso de recolección de datos entrados por el usuario para la generación de personajes. Además se encuentra en esta la clase encargada de conformar y controlar la IA de los *NPC* creados.

**Capa de soporte.**

En esta capa se encuentran las clases que provee el motor de videojuegos *Unity*, las cuales son utilizadas para almacenar la información que provee el *plugin*.

### 2.4.2. Tarjetas Clase - Responsabilidad - Colaboración.

En la metodología *XP* no es necesaria la descripción del sistema a través de diagramas de clase utilizando notación *Unified Modeling Language* (*UML*), sino que se guía por técnicas como las tarjetas Clase - Responsabilidad - Colaboración (CRC). Las tarjetas CRC son fichas en las que se escriben brevemente las responsabilidades de la clase y una lista de los objetos con los que colabora para llevar a cabo estas responsabilidades.

Tabla 2.15. Tarjeta CRC # 1

| **Tarjeta CRC** | |
|---|---|
| **Clase:** Grupo*NPC* | |
| **Responsabilidad** | **Colaboración** |
| • Crear la interfaz del *plugin* a desarrollar, con las funcionalidades que permitan al usuario crear los *NPC* y conformar grupos con estos.<br>• Almacenar en una base de datos los personajes creados para poder listarlos en caso que se deseen modificar o eliminar. | • ComportamientoIA<br>• *PrefabUtility*<br>• *Enemy_DB* |

Tabla 2.16. Tarjeta CRC # 2

| **Tarjeta CRC** | |
|---|---|
| **Clase:** *Enemy_DB* | |
| **Responsabilidad** | **Colaboración** |

Continúa en la próxima página

Tabla 2.16. Continuación de la página anterior

| | |
|---|---|
| • Crear el personaje que se mostrará en la base de datos, el cual estará vinculado con el personaje creado por el *plugin*.<br>• Crear la base de datos que contendrá los personajes creados por el *plugin*. | |

Tabla 2.17. Tarjeta CRC # 3

| **Tarjeta CRC** | |
|---|---|
| **Clase:** ComportamientoIA | |
| **Responsabilidad** | **Colaboración** |
| • Crear el comportamiento de IA que controlará los *NPC*.<br>• Permitirá a los *NPC* patrullar en busca del jugador, siendo este su estado inicial, seleccionando puntos aleatorios cerca del jefe y moviéndose por intervalos de tiempo.<br>• Dependiendo del tipo definido, al atacar, se posicionarán a una distancia específica con respecto al jugador. Además, tendrán su propio tipo de ataque, lo cual los diferenciará.<br>• Una vez encontrado el objetivo, lo perseguirán y atacarán mientras se encuentre en su rango de persecusión y de ataque. Además corregirán su posición al tener a un compañero en su línea de fuego y se moverán por intervalos de tiempo alrededor del jugador mientras atacan; de esta manera no permanecerán estáticos.<br>• En caso de perder al jugador en la persecusión o haberlo derrotado, los *NPC* esperarán unos segundos y volverán a su estado inicial. | • *NavMeshAgent*<br>• *Damage*<br>• *Player* |

Tabla 2.18. Tarjeta CRC # 4

| **Tarjeta CRC** | |
|---|---|
| **Clase:** *Damage* | |
| **Responsabilidad** | **Colaboración** |
| • Calcular el daño que pueden inflingir los *NPC* en el jugador en cada ataque.<br>• Aplicar el daño correspondiente a cada tipo de *NPC* sobre el jugador en cada ataque. | |

Tabla 2.19. Tarjeta CRC # 5

| **Tarjeta CRC** | |
|---|---|
| **Clase:** *Player* | |
| **Responsabilidad** | **Colaboración** |
| • Crear un personaje para simular a un jugador, el cual servirá para validar el comportamiento que tendrán los *NPC* al encontrarse con este en un escenario demo.<br>• Controlar las propiedades de movimiento para moverse en la escena.<br>• Establecer los puntos de vida y activar el daño que recibirá por parte de los enemigos.<br>• Aplicar el daño correspondiente a cada tipo de *NPC* sobre el jugador en cada ataque.<br>• Finalizar la escena una vez este haya sido derrotado. | • *Damage* |

### 2.4.3. Elementos del Diseño.

Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de *software* y otros ámbitos referentes al diseño de interacción o interfaces. Son reusables ya que pueden ser aplicados en otros diseños o problemas (Gamma et al., 1995).

Para el diseño del componente de *software* para la creación de IA de *NPC* en comportamientos grupales, se tuvieron en cuenta los siguientes patrones *General Responsibility Assignment Software Patterns* (*GRASP*): Experto, Bajo acoplamiento, Alta cohesión y Controlador.

Se hizo uso del patrón Experto, esto se pone de manifiesto en la clase Grupo*NPC* ya que esta clase es la encargada de todo lo referente al componente. Esta contiene toda la información necesaria para el trabajo con el mismo.

Se evidencia el uso del patrón Bajo Acoplamiento y Alta Cohesión, debido a que las clases fueron diseñadas de tal forma que existiera la menor relación entre ellas, a la vez logrando que la información almacenada en cada clase fuera coherente. Con este diseño se logra que de existir una modificación tenga la mínima repercusión posible en el resto de las clases, se garantiza la reutilización de código y disminuye la dependencia entre las mismas.

Se utiliza el patrón Controlador, su uso se evidencia en la clase Grupo*NPC* la cual es la encargada de la capa lógica del negocio, aumentando así la reutilización de código y un mayor control. Esta clase es la encargada de controlar todo el proceso de negocio asignando las responsabilidades a las clases correspondientes.

## Conclusiones parciales

Finalizado este capítulo, se llega a la conclusión de que las bases de la solución propuesta quedaron bien definidas, gracias al análisis de las primeras fases de la metodología empleada y de los artefactos generados por ellas. Esto facilitó la creación del diseño del *plugin*, mostrando los elementos más importantes a tener en cuenta durante su desarrollo. La planificación estimada permitió establecer el tiempo necesario para desarrollar cada HU, de esta manera, se evita realizar nuevas iteraciones. Además, la descripción de las clases de la solución y sus relaciones mediante las tarjetas CRC, brinda un mayor entendimiento de su estructura, de manera que contribuya a reducir el tiempo de desarrollo.

CAPÍTULO 3

# Resultados

## Introducción

En este capítulo se analiza la propuesta de solución desde el punto de vista de la calidad, para evaluar cómo se le da cumplimiento al problema de la presente investigación y verificar si fueron implementadas de forma correcta cada una de las HU. Se procede a diseñar un conjunto de pruebas correspondientes a la metodología *XP*, las cuales se dividirán en dos partes: unitarias y de aceptación. Por otro lado, se describe de forma detallada todo el proceso de pruebas y se muestra un resultado final, que explica y muestra al usuario que el sistema está implementado correctamente y que cuenta con las funcionalidades descritas en el capítulo anterior.

## 3.1. Fase de Implementación.

En esta fase se realiza la implementación de las HU correspondientes a cada iteración, se chequea el plan de iteraciones por si es necesario realizar modificaciones y se crean las tareas de programación para implementar exitosamente cada HU.

Se realizó el diseño e implementación del *plugin*, el cual está almacenado en una carpeta principal con el nombre *Assets*, la misma contiene la herramienta desarrollada, distribuida en varias subcarpetas. La clase encargada de crear y controlar la interfaz del *plugin* se encuentra en la subcarpeta *Editor*, mientras que los *script* que controlan el comportamiento de IA, los eventos y animaciones, además de un ejemplo de *player* para el escenario demo, se encuentran distribuidos en subcarpetas específicas dentro de la subcarpeta *Script*. En esta misma subcarpeta se encuentra el *script* encargado de crear la base de datos que contendrá los personajes creados por el módulo. Los materiales con los que estarán compuestos los personajes, ya sean, texturas, modelos o animaciones, están almacenados en subcarpetas específicas de cada uno en la subcarpeta *Art*. A su vez, los *prefab* de cada personaje creado mediante la herramienta, así como los

*prefab* de los grupos y de los ataques, están en subcarpetas específicas dentro de la subcarpeta *Prefabs*. El escenario demo en el cual se mostrarán los personajes o grupos creados con su comportamiento de IA y un ejemplo de *player*, se encuentra en la subcarpeta *Scene*.

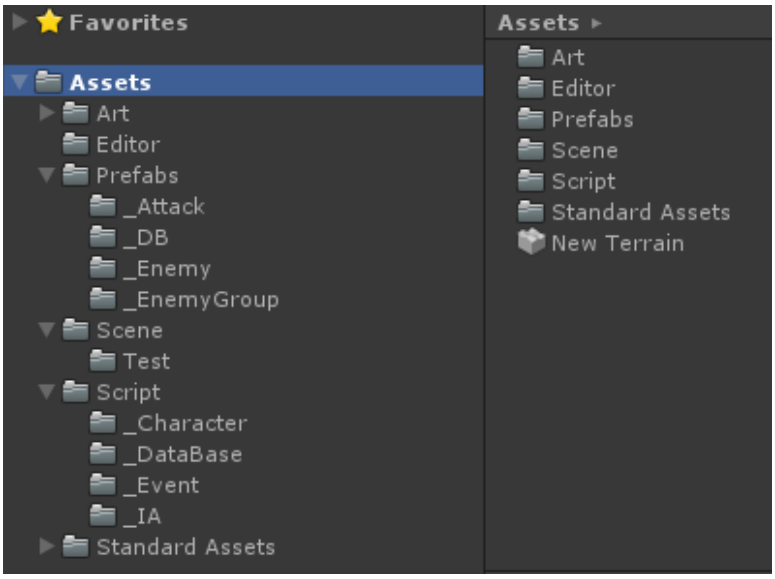



Figura 3.1. Distribución del *plugin*

Al quedar estructurado y dsitribuido el *plugin* en las carpetas que lo conformarán, se procede a la implementación de las clases. La interfaz con la que se encontrará el desarrollador está compuesta por tres funcionalidades, Crear Enemigo, Conformar Grupo y Lista de *NPC*. Estas se implementaron en el *script GrupoNPC.cs* en la clase GrupoNPC. Esta clase contiene los métodos referentes a cada funcionalidad del *plugin*, así como los que permitirán la visualización del mismo y las variables que recogerán los datos que el usuario entre por parámetro. Una vez finalizada la implementación de esta interfaz, se obtiene la herramienta funcional.

Cuando el desarrollador despliegue la herramienta se encontrará con una ventana que permitirá seleccionar la funcionalidad con la que desee trabajar. En la funcionalidad Crear Enemigo, se carga la malla 3D que conformará el caracter, se selecciona el tipo de enemigo y se establece un nombre para este. Hecho esto, se procede a configurar los demás atributos de ataque, tales como, puntos de vida, puntos de ataque, distancia de ataque, tiempo entre ataques y cargar el prefab que contendrá el mismo. Dando paso a configurar los atributos de movimiento tales como la velocidad, el tiempo de inactividad y la distancia de persecución. Establecidos todos los parámetros se puede crear entonces el personaje presioando el botón “Crear Enemigo”. Finalizado este proceso, el *plugin* crea el objeto *prefab* del *NPC* en la capeta *Prefabs* y limpia los parámetros de la ventana posibilitando crear nuevos personajes. Cada personaje es guardado en una base de datos, la cual se utilizará para mostrarlos, modificarlos o eliminarlos en la funcionalidad Lista de *NPC*.

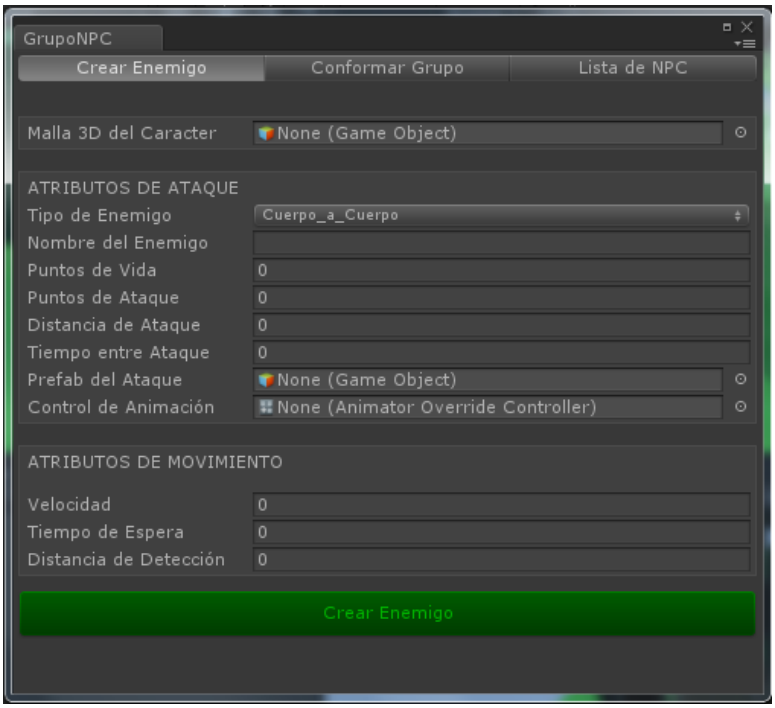



Figura 3.2. Funcionalidad Crear Enemigo

En caso que el desarrollador seleccione la pestaña de la funcionalidad Conformar Grupo, se asigna un nombre para cada grupo a crear y se procede a configurar el mismo. Para esto se implementó un componente que permite seleccionar la cantidad de personajes que conformarán un grupo. Este puede ser de 3 a 5 miembros, una vez seleccionada esta opción se despliega una lista de campos en los que se pueden cargar los objetos generados por la función Crear Enemigo. Una vez cargados los personajes, se asigna un comportamiento al grupo, siendo este Activo o Variado. Para esto se implementó un componente que permite seleccionar qué comportamiento tendrá el grupo a crear; además de mostrar una información referente a cada uno. Si se selecciona el comportamiento Activo, todos los integrantes del grupo serán jefes; mientras que al seleccionar el comportamiento Variado, se muestra junto a cada *NPC* que conforme el grupo, la posibilidad de seleccionar quiénes serán jefes. Establecidos estos parámetros se puede crear el grupo presionando el botón “Crear Grupo”. Finalizado este proceso, el *plugin* crea el objeto *prefab* del grupo en la carpeta *Prefabs* y limpia los parámetros de la ventana posibilitando crear nuevos grupos.

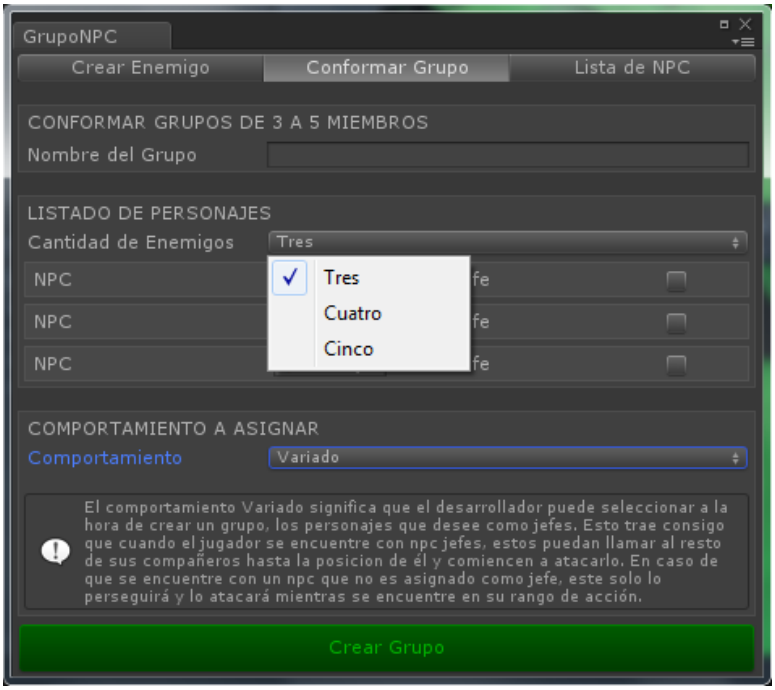



Figura 3.3. Funcionalidad Conformar Grupo

Mientras que en la pestaña de la funcionalidad Lista de *NPC*, se carga una base de datos que contiene un listado de todos los personajes creados, permitiendo modificar o eliminar estos. Si se desea modificar un personaje, en la lista se muestra junto a cada *NPC*, el set de parámetros que lo conforman, se modifican estos y se presiona el botón “Modificar”. Este actualiza el *prefab* del personaje seleccioando guardado en la carpeta *Prefabs*. Para eliminar el personaje o personajes, se seleccionan estos y se presiona el botón “Eliminar”, el cual borra de la carpeta *Prefabs* el objeto referente al *NPC* seleccionado. Para esto se implementó una función que permite modificar o destruir el objeto seleccionado.


Figura 3.4. Funcionalidad Lista de *NPC*

Concluida la implementación de la interfaz del *plugin* se procedió a la implementación de las clases que controlan la IA de los *NPC* y sus animaciones así como el daño que pueden ocasionarle al jugador. Además de la clase encargada de controlar el ejemplo de *player* para comprobar el funcionamiento de la IA en el patrullaje, persecución y ataque de los *NPC* hacia el mismo. Para ello se implementaron tres métodos principales, encargados del patrullaje, persecución y ataque de los *NPC*, sincronizados por el método *Update()*, el cual contiene el código que se encargará de actualizar en tiempo de ejecución el *GameObject*(Unity, 2016); estos son: *doSomething()*, encargado del patrullaje, *Chase()*, encargado de la persecución y *Attack()*, encargado de atacar. Existen otros métodos que interactúan con estos, tales como: *FriendTarget()*, encargado de devolver verdadero si un *NPC* tiene a un compañero en la línea de fuego; el método *RandomPoint()*, encargado de calcular los puntos aleatorios a los cuales se moverá el *NPC*; el método *AttackEvent()*, encargado de controlar el *prefab* del ataque y activar el daño que recibirá el *player* y el método *OnTriggerEnter()*, encargado de controlar si el *NPC* se encontró con el *player*, en este caso se activará una variable *booleana* que indicará lo ocurrido y definirá la posición del mismo.

El método *Update()* sincroniza los métodos principales, en este caso, primero comprueba si la variable *booleana* que controla el método *OnTriggerEnter()* es verdadera, si es así, ordena al *NPC* a moverse hacia la posición del *player* y activa el método *Attack()*. Para esto se comprueba que en el rango de ataque se encuentre un compañero con el método *FriendTarget()*, si es así, se ordena a quien esté atacando a moverse hacia la derecha o a la izquierda y continuar atacando al *player*. Si el *player* se aleja, entonces activa el método *Chase()*. Además, se comprueba si el *player* se encuentra con un *NPC* jefe y que los demás *NPC* no conozcan la posición del enemigo; este hará un llamado de alerta al resto de sus compañeros hasta la posición del *player* y les ordenará perseguirlo. Si la variable es falsa, entonces se activará el método *doSomething()* encargado del

patrullaje.

En el método *doSomething()*, los *NPC* patrullan por el mapa, eligiendo puntos aleatorios a los cuales moverse, esperando un tiempo entre cada punto al que van a caminar.

El método *Chase()* comprueba que la distancia entre la posición del *NPC* y el *player* sea menor que el valor que contiene la variable que controla la distancia de persecución para cada *NPC*. Si esta condición se cumple, el *NPC* se moverá hacia la posición del enemigo, conservando la distancia de parada con respecto al *player*. En caso de no cumplirse la condición anterior, el *NPC* pasará al estado de patrullar, pues el enemigo logró huir.

El método *Attack()* se encarga de hacer que el *NPC*, una vez que tenga al *player* en su rango de ataque, se mueva hasta su pocisión y se detenga en su distancia de ataque. Una vez pocisionado, ataca y se mueve alrededor del *player*, combinando de esta manera, ataque y cambio de pocisión. En caso de haber derrotado a su oponente, este pasa al estado de patrullar.

Finalizada la implementación de la clase encargada de la IA de los personajes, se conforma un escenario demo en el cual se procede a comprobar el funcionamiento del *plugin* al crear los personajes y los grupos, vinculados con la IA asociada a estos. Esto se puede ver en la siguiente imagen que refleja la interfaz del *plugin* y un escenario donde hay algunos *NPC* atacando al *player*.

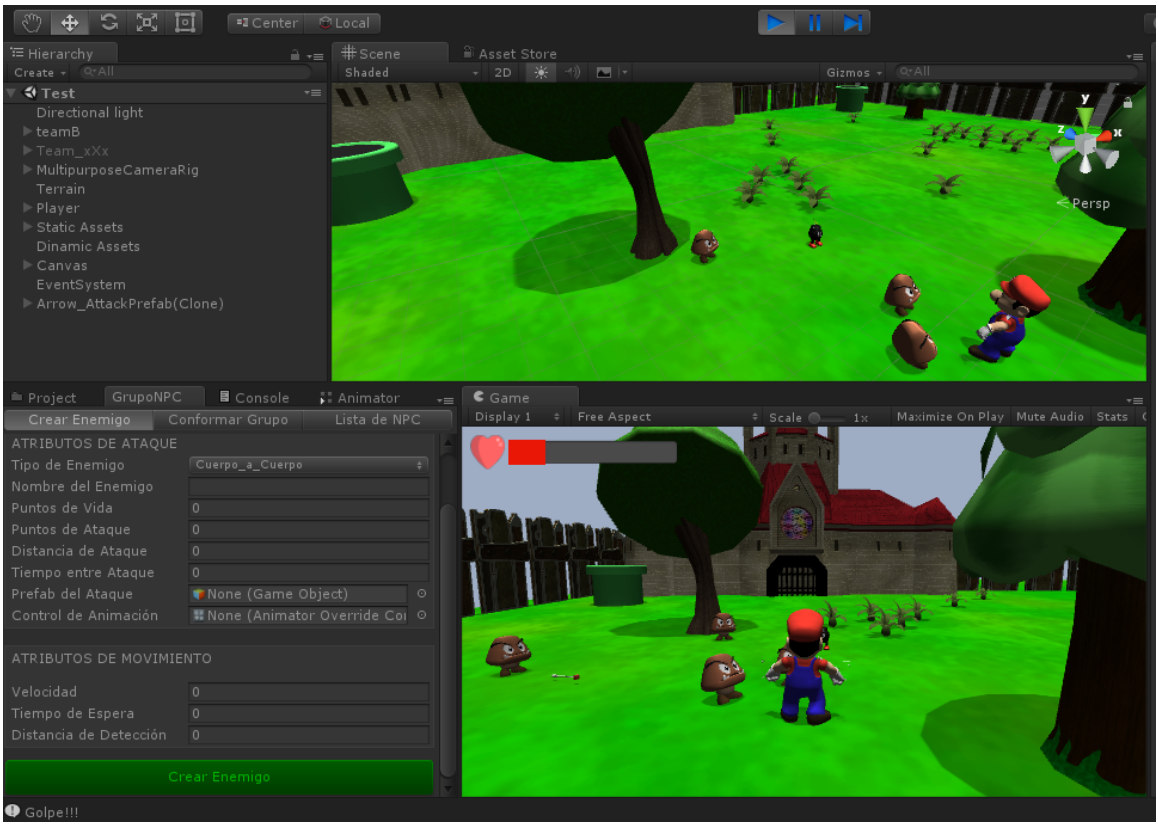



Figura 3.5. Escenario Demo

De esta manera se procede a realizar las pruebas al sistema implementado y corregir cualquier error detectado, así como ajustar la IA de los *NPC*.

## 3.2. Pruebas del sistema.

Las pruebas del sistema se realizan con el objetivo de comprobar la calidad requerida y de detectar tantos errores como sea posible en la aplicación una vez que se finaliza la fase de implementación. Estas son realizadas por los desarrolladores en colaboración con el cliente, buscando corregir cualquier no conformidad que pudiera surgir (Echeverry Tobón y Delgado Carmona, 2007).

**Pruebas de caja blanca:** Están centradas en el código y le permiten al desarrollador comprobar si las funciones o algoritmos implementados en el *software* tienen la calidad requerida o si tienen algún error. Para su aplicación se analizan los posibles caminos que puede tener una determinada función o algoritmo y luego es probada para comprobar si el resultado esperado es el obtenido (Pressman, 2010).

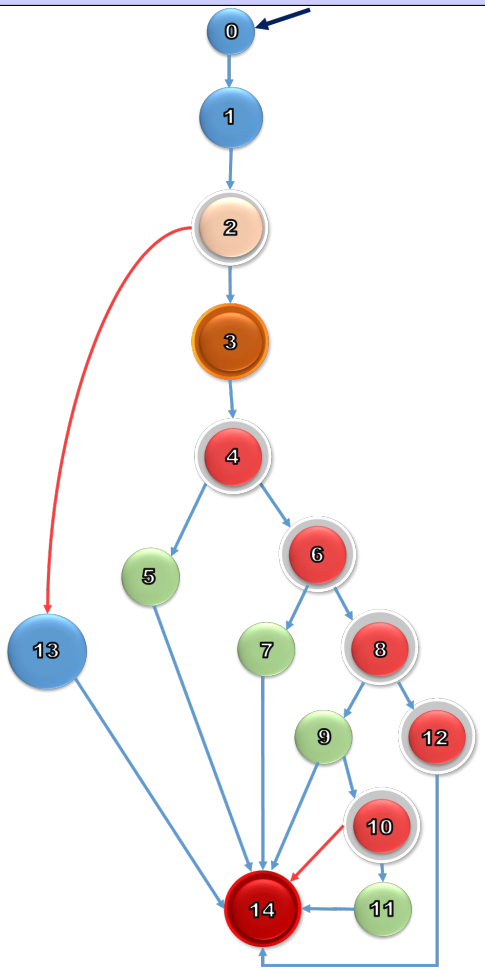
**Pruebas de aceptación:** Estas pruebas las realiza el cliente. Son básicamente pruebas funcionales, sobre el sistema completo, y buscan una cobertura de la especificación de requisitos y del manual del usuario, las mismas permiten al cliente dar el visto bueno al sistema desarrollado (Echeverry Tobón y Delgado Carmona, 2007).

### 3.2.1. Pruebas de caja blanca

A continuación se realizaron pruebas de caja blanca a las principales funcionalidades del sistema. Estas representan la mayor complejidad en la solución. Son las funcionalidades más utilizadas y las que marcan un hito en el desarrollo de la solución lo que trae consigo que un error en dichas funcionalidades conlleve a que la solución no funcione correctamente o no cumpla con los requisitos planteados por el cliente. Los métodos escogidos para las pruebas de caja blanca son: los métodos *Chase(), FriendTarget(), Attack()* y *RandomPoint()* de la clase ComportamientoIA.

Tabla 3.1. Prueba de Caja Blanca para el método Chase()

| **Prueba de Caja Blanca** | |
|---|---|
| **Método: Chase()** | |
| 0 | void Chase()<br>{ |
| 1 | bobeado = false; |
| 2 | if (Vector3.Distance(transform.position, agentDestination. position)<br>< ChaseDistance)<br>{ |
| 3 | switch (classType)<br>{ |
| 4 | case iaBehavior.Cuerpo_a_Cuerpo: |
| 5 | agent.Resume();<br>agent.speed = 3.5f;<br>agent.stoppingDistance = 2f;<br>agent.SetDestination(agentDestination.position);<br>break; |
| 6 | case iaBehavior.Magia: |
| 7 | state = “chase”;<br>agent.Resume();<br>agent.speed = 2f;<br>agent.stoppingDistance = 6f;<br>agent.SetDestination(agentDestination.position);<br>break; |
| 8 | case iaBehavior.Distancia: |
| 9 | state = “chase”;<br>agent.Resume();<br>agent.speed = 2f;<br>agent.stoppingDistance = 8f;<br>agent.SetDestination(agentDestination.position); |
| 10 | if(isBoss) { |
| 11 | state = “chase”;<br>agent.Resume();<br>agent.speed = 2f;<br>agent.stoppingDistance = 8f;<br>agent.SetDestination(agentDestination.position);<br>}<br>break; |
| 12 | default:<br>break;<br>}<br>} |
| 13 | else<br>{<br>CharacterFind = false;<br>bossAlarm = false;<br>randomPoint = EnemyFriends[bossPos].transform.position;<br>agent.stoppingDistance = 0;<br>agent.SetDestination(randomPoint);<br>bobeado = true;<br>} |
| 14 | } |




Continúa en la próxima página

Tabla 3.1. Continuación de la página anterior

| **Complejidad Ciclomática:** | |
|---|---|
| V (G)= # de regiones = 7<br>V (G)=A - N + 2 = 20 - 15 + 2 = 7<br>V (G)= P + 1 = 6 + 1 = 7 | {0, 1, 2, 13, 14};<br>{0, 1, 2, 3, 4, 5, 14};<br>{0, 1, 2, 3, 4, 6, 7, 14};<br>{0, 1, 2, 3, 4, 6, 8, 9, 14};<br>{0, 1, 2, 3, 4, 6, 8, 9, 10, 14};<br>{0, 1, 2, 3, 4, 6, 8, 9, 10, 11, 14};<br>{0, 1, 2, 3, 4, 6, 8, 12,14}; |

Tabla 3.2. Prueba de Caja Blanca para el método FriendTarget()

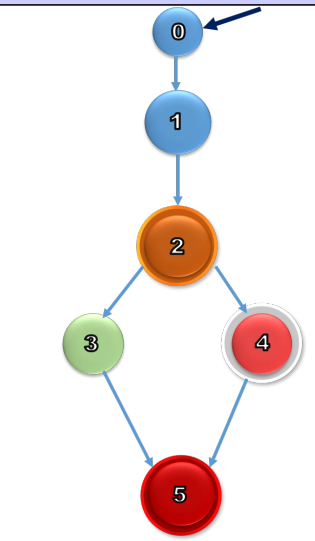
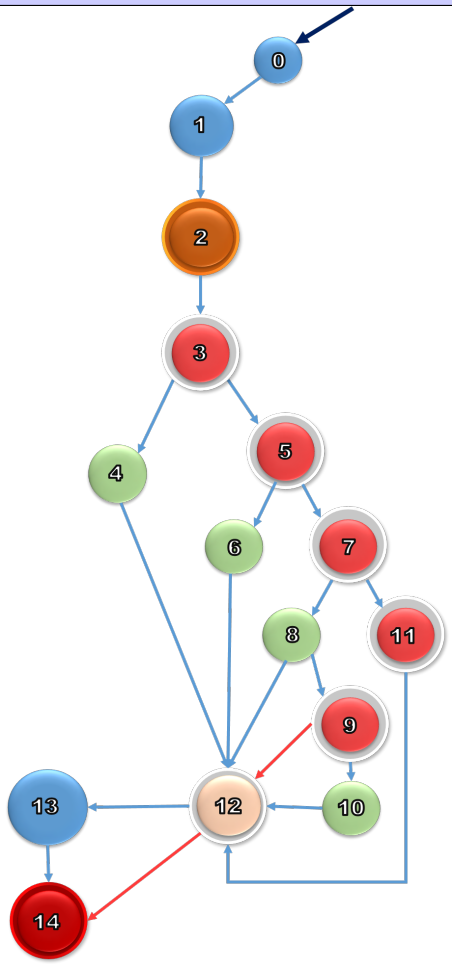
| **Prueba de Caja Blanca** | | |
|---|---|---|
| **Método: FriendTarget()** | | |
| 0 | bool FriendTarguet()<br>{ |  |
| 1 | Vector3 fwd = transform.TransformDirection(Vector3.forward);<br>RaycastHit hit; | |
| 2 | if (Physics.Raycast(transform.position, fwd, out hit, 10f))<br>{ | |
| 3 | return hit.collider.CompareTag("Friend");<br>} | |
| 4 | return false; | |
| 5 | } | |
| **Complejidad Ciclomática:** | | |
| V (G)= # de regiones = 2<br>V (G)=A - N + 2 = 6 - 6 + 2 = 2<br>V (G)= P + 1 = 1 + 1 = 2 | | {0, 1, 2, 3, 5};<br>{0, 1, 2, 4, 5}; |

Tabla 3.3. Prueba de Caja Blanca para el método Attack()

| **Prueba de Caja Blanca** | |
|---|---|
| **Método: Attack()** | |
| 0 | void Attack()<br>{<br>Anim.SetBool(“walk”, false);<br>agent.Stop();<br>transform.LookAt(agentDestination); |
| 1 | switch (classType)<br>{ |
| 2 | case iaBehavior.Cuerpo_a_Cuerpo: |
| 3 | agent.stoppingDistance = 0;<br>nextMove = Time.time + moveRate;<br>transform.Translate(Vector3.right * 0.005f);<br>break; |
| 4 | case iaBehavior.Magia: |
| 5 | agent.stoppingDistance = 0;<br>nextMove = Time.time + moveRate;<br>transform.Translate(Vector3.right * 0.003f);<br>break; |
| 6 | case iaBehavior.Distancia: |
| 7 | agent.stoppingDistance = 0;<br>nextMove = Time.time + moveRate;<br>agent.speed = 2f;<br>transform.Translate(Vector3.right * 0.006f);; |
| 8 | if(isBoss) { |
| 9 | transform.Translate(Vector3.right * 0.04f);<br>nextMove = Time.time + moveRate;<br>agent.stoppingDistance = 0f;<br>}<br>break; |
| 10 | default:<br>break;<br>}<br>} |
| 11 | if (agent.pathStatus == NavMeshPathStatus.PathComplete && Time.time >nextFire)<br>{ |
| 12 | anim.SetTrigger(“attack”);<br>nextFire = Time.time + fireRate;<br>} |
| 13 | } |
| **Complejidad Ciclomática:** | |

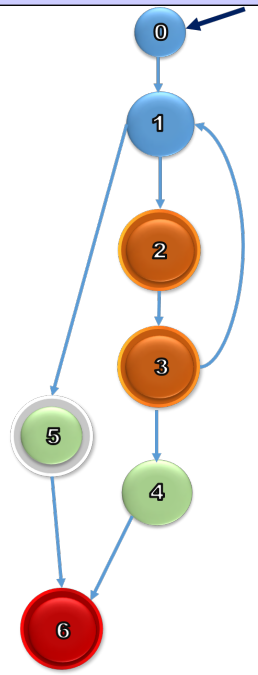



Continúa en la próxima página

Tabla 3.3. Continuación de la página anterior

| | |
|---|---|
| V (G)= # de regiones = 7<br>V (G)=A - N + 2 = 20 - 15 + 2 = 7<br>V (G)= P + 1 = 6 + 1 = 7 | {0, 1, 2, 3, 4, 12, 14};<br>{0, 1, 2, 3, 4, 12, 13, 14};<br>{0,1, 2, 3, 5, 6, 12, 14};<br>{0,1, 2, 3, 5, 6, 12, 13, 14};<br>{0,1, 2, 3, 5, 7, 8, 12, 14};<br>{0,1, 2, 3, 5, 7, 8, 12, 13, 14};<br>{0,1, 2, 3, 5, 7, 8, 9, 12, 14};<br>{0,1, 2, 3, 5, 7, 8, 9, 12, 13, 14};<br>{0,1, 2, 3, 5, 7, 8, 9, 10, 12, 14};<br>{0,1, 2, 3, 5, 7, 8, 9, 10, 12, 13, 14};<br>{0,1, 2, 3, 5, 7, 11, 12, 14};<br>{0,1, 2, 3, 5, 7, 11, 12, 13, 14}; |

Tabla 3.4. Prueba de Caja Blanca para el método RandomPoint()

| **Prueba de Caja Blanca** | | |
|---|---|---|
| **Método: RandomPoint()** | | |
| 0 | bool RandomPoint(Vector3 center, float range, out Vector3 result)<br>{ |  |
| 1 | for (int i = 0; i <30; i++)<br>{ | |
| 2 | Vector3 randomPoint = center + Random.insideUnitSphere * range;<br>NavMeshHit hit; | |
| 3 | if (NavMesh.SamplePosition(randomPoint, out hit, 1.0f, NavMesh.AllAreas))<br>{ | |
| 4 | result = hit.position;<br>return true;<br>} | |
| 5 | } | |
| 6 | result = Vector3.zero;<br>return false; | |
| 7 | } | |
| **Complejidad Ciclomática:** | | |
| V (G)= # de regiones = 3<br>V (G)=A - N + 2 = 8 - 7 + 2 = 3<br>V (G)= P + 1 = 2 + 1 = 3 | | {0, 1, 2, 5, 6};<br>{0, 1, 2, 3, 4, 6}; |

Una vez definidos los casos de prueba de Caja Blanca a los métodos seleccionados, se les realizaron las pruebas a cada camino perteneciente a cada caso de prueba. Para un total de 23 caminos, se aplicaron dos pruebas a cada uno, obteniéndose 46 pruebas en total. En una primera

iteración de un total de 46, 30 fueron satisfactorias y 16 no satisfactorias, esto hizo necesaria una segunda iteración en la cual de 46 pruebas, 40 fueron satisfactorias y 6 no satisfactorias. Se corrigieron los errores y se realizó una tercera iteración en la cual de 46 pruebas en total, 46 fueron satisfactorias, no necesitándose así más iteraciones.

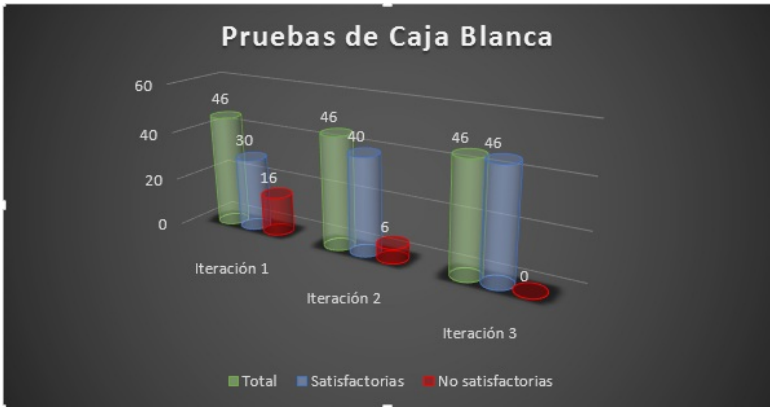



Figura 3.6. Estadísticas de las Pruebas de Caja Blanca.

### 3.2.2. Pruebas de aceptación.

A continuación se muestran las pruebas de aceptación realizadas a las principales funcionalidades del sistema. Para estas pruebas son escogidas todas las funcionalidades del sistema. Debido a que la prioridad de cada una de estas historias de usuario es alta es necesario comprobar que todas sean aceptadas por el cliente y de no ser así pasar a corregirlas inmediatamente.

Tabla 3.5. Prueba de aceptación # 1

| **Caso de prueba de aceptación** | |
|---|---|
| **Código:** HU1_P1 | **Historia de usuario:** 1 |
| **Nombre:** Crear Enemigo. | |
| **Descripción:** Es la primera opción con la que se encuentra el desarrollador al interactuar con el *plugin*. Permite crear el personaje que desee dependiendo del tipo que se seleccione. | |
| **Condiciones de ejecución:**<br>• No presenta ninguna condición para su ejecución. | |

Continúa en la próxima página

Tabla 3.5. Continuación de la página anterior

| |
|---|
| **Pasos de ejecución:**<br>1. Clic en el botón "*A.I.G.B.*" de la barra de tareas.<br>2. Seleccionar "Crear Grupo de Enemigos".<br>3. Sale en pantalla la ventana principal del *plugin* con la funcionalidad "Crear Enemigo" seleccionada por defecto.<br>4. Seleccionar la malla 3D del personaje.<br>5. Seleccionar el tipo de personaje a crear.<br>6. Llenar los demás datos que conformaran al personaje.<br>7. Clic en el botón "Crear Enemigo". |
| **Resultados esperados:**<br>• Se crea un objeto prefab, este representa el personaje que se acaba de crear. |
| **Evaluación:** Satisfactoria |

Tabla 3.6. Prueba de aceptación # 2

| **Caso de prueba de aceptación** | |
|---|---|
| **Código:** HU2_P2 | **Historia de usuario:** 2 |
| **Nombre:** Crear Grupo. | |
| **Descripción:** Se crean grupos de personajes y se asigna un comportamiento a cada grupo. | |
| **Condiciones de ejecución:**<br>• Los personajes deben haber sido creados anteriormente. | |
| **Pasos de ejecución:**<br>1. Si no se ha cerrado la ventana del *plugin*, dar clic en la funcionalidad "Crear Grupo". De lo contrario, seguir los dos primeros pasos de ejecución de la prueba anterior.<br>2. Visualiza todos los personajes creados, posibilitando seleccionar entre 3 y 5 miembros que se deseen para conformar cada grupo.<br>3. Se selecciona el tipo de comportamiento que tendrá el grupo.<br>4. Clic en el botón "Crear Grupo". | |
| **Resultados esperados:**<br>• Visualiza los personajes creados. Se seleccionan y se crean tantos grupos como se desee. | |
| **Evaluación:** Satisfactoria | |

Tabla 3.7. Prueba de aceptación # 3

| **Caso de prueba de aceptación** | |
|---|---|
| **Código:** HU3_P3 | **Historia de usuario:** 3 |
| **Nombre:** Listar *NPC*. | |
| **Descripción:** Lista todos los personajes creados, permitiendo eliminar o modificar los que el desarrollador desee. | |

Continúa en la próxima página

Tabla 3.7. Continuación de la página anterior

| |
|---|
| **Condiciones de ejecución:**<br>• Los personajes deben haber sido creados anteriormente. |
| **Pasos de ejecución:**<br>1. Si no se ha cerrado la ventana del *plugin*, dar clic en la funcionalidad “Listar *NPC*”. De lo contrario, seguir los dos primeros pasos de ejecución de la primera prueba.<br>2. Visualiza todos los personajes creados, posibilitando seleccionar tantos como se desee para eliminarlos.<br>3. Clic en el botón “Eliminar”.<br>4. Visualiza todos los personajes creados, posibilitando seleccionar tantos como se desee para modificarlos.<br>5. Clic en el botón “Modificar”. |
| **Resultados esperados:**<br>• Se eliminan los personajes seleccionados. |
| **Evaluación:** Satisfactoria |

Tabla 3.8. Prueba de aceptación # 4

| **Caso de prueba de aceptación** | |
|---|---|
| **Código:** HU4_P4 | **Historia de usuario:** 4 |
| **Nombre:** ComportamientoIA. | |
| **Descripción:** La IA asociada que controla cada personaje. | |
| **Condiciones de ejecución:**<br>• Los *NPC* deben haber sido creados. | |
| **Pasos de ejecución:**<br>1. El personaje se encuentra caminando aleatoriamente por el mapa.<br>2. El personaje encuentra al jugador.<br>3. Persigue al jugador.<br>4. Ataca al jugador.<br>5. Si en su rango de ataque se encuentra un compañero, modifica su posición para atacar al jugador.<br>6. El jugador se encuentra con el jefe del grupo.<br>7. Se lanza una alerta a los demás personajes para que vengan hasta él.<br>8. Llegan y comienzan a atacar.<br>9. El jugador huye, todos lo persiguen y lo atacan. | |
| **Resultados esperados:**<br>• El comportamiento de IA hace que el personaje persiga y ataque al jugador. | |
| **Evaluación:** Satisfactoria | |

Al realizar las pruebas de aceptación en una primera iteración de un total de 4 pruebas, 2 fueron satisfactorias y 2 no satisfactorias, estas fueron las pruebas realizadas a las historias de

usuario Listar NPC y ComportamientoIA . En Listar NPC no se lograba mostrar en la lista los personajes creados en otras ocaciones, solo los nuevos; mientras que en ComportamientoIA, los personajes no asociaban el comportamiento cuando este se les integraba. Por estas razones, fue necesaria una segunda iteración de pruebas. En esta se corrigieron los errores de la teración anterior y de un total de 4 pruebas, 4 fueron satisfactorias y ninguna no satisfactoria, por lo que no fueron necesarias más iteraciones. De esta manera, se lograron integrar de manera exitosa los objetos creados por el *plugin* con el comportamiento de IA que los controlaría.


Figura 3.7. Estadísticas de las Prueba de Aceptación.

## 3.3. Análisis de Resultados.

La solución propuesta se implementó de manera que cualquier desarrollador de videojuegos del centro Vertex pudiera facilmente crear personajes, agruparlos y asignarles un comportamiento de IA. de esta manera, cuando el usuario se encuentra frente a la aplicación tendrá tres opciones a realizar Crear Enemigo, Conformar Grupo, Listar *NPC*.

Para el análisis se tomó una muestra de cinco desarrolladores del centro Vertex para realizar una comparación teniendo en cuenta el tiempo que demoran estos para integrar el comportamiento de IA a los enemigos y crear los grupos de forma manual contra la utilización del *plugin*. De esta manera, se puede ver si la utilización del *plugin* puede agilizar o no el proceso de desarrollo.

### 3.3.1. Medición del Tiempo de Desarrollo.

A continuación se muestran las tablas que contienen las observaciones de los especialistas sobre el uso de la herramienta implementada.

Tabla 3.9. Validación del *plugin* # 1

| **Validación del *plugin*** | | | |
|---|---|---|---|
| **Desarrollador** | Roberto Elías Pérez Ozete | | |
| **Tiempo de implementaciín manual (min)** | 8 | **Observación** | "Se realiza sin ningún problema pero el proceso de hacer la composición es muy largo y tedioso." |
| **Tiempo de implementación asistido(min)** | 1 | **Observación** | "Es solo colocar el *mesh* del enemigo, hacer un balance con los atributos y la composición se realiza automáticamente." |
| **Conclusiones** | | | |
| "La utilización del *plugin* para componer enemigos y darle un comportamiento de IA es mucho más rápido que hacer el *setup* de forma manual, sobre todo por el trabajo con los componentes de *Unity*. Además al crear el grupo es más confiable la generación automática al incorporar los amigos.<br>De igual modo recomiendo continuar con el perfeccionamiento de esta herramienta para obtener resultados más completos para utilizar una IA más fuerte. " | | | |

Tabla 3.10. Validación del *plugin* # 2

| **Validación del *plugin*** | | | |
|---|---|---|---|
| **Desarrollador** | Ernesto A. Leyva Piñeda | | |
| **Tiempo de implementaciín manual (min)** | 10 | **Observación** | "Es muy complicado" |
| **Tiempo de implementación asistido(min)** | 2 | **Observación** | "Se simplifica el proceso y se hace mas fácil." |
| **Conclusiones** | | | |
| "Debería continuar su perfeccionamiento en trabajos futuros." | | | |

Tabla 3.11. Validación del *plugin* # 3

| **Validación del *plugin*** | | | |
|---|---|---|---|
| **Desarrollador** | Carlos Alberto Gavilla Cruz | | |
| **Tiempo de implementaciín manual (min)** | 30 | **Observación** | "Hacerlo manualmente es tedioso y largo sobre todo para aquellos que no tienen muchos conocimientos de IA." |
| **Tiempo de implementación asistido(min)** | 2 | **Observación** | "Solo necesito saber cómo trabajar con el plugin su configuración y saber montar el navmesh." |
| **Conclusiones** | | | |
| "La utilización de este *plugin* para crear enemigos y darle un comportamiento de IA es mucho más rápido y facil que hacerlo de forma manual, teniendo en cuenta todo el trabajo con los componentes de *Unity*.<br>Sería muy interesante que se continuara el desarrollo y expansión de esta herramienta para obtener resultados más completos para utilizar una IA más fuerte." | | | |

Tabla 3.12. Validación del *plugin* # 4

| **Validación del *plugin*** | | | |
|---|---|---|---|
| **Desarrollador** | Orlando Sánchez Palacios | | |
| **Tiempo de implementaciín manual (min)** | 25 | **Observación** | "Es un proceso generalmente tedioso y particular para cada tipo de videojuego. La existencia de muchas tareas repetitivas hace que sea fácil la existencia de errores por parte de los desarrolladores" |
| **Tiempo de implementación asistido(min)** | 3 | **Observación** | "Aceleró de manera significativa la configuración de la IA de los enemigos permitiendo que ello se realice de manera más simple" |
| **Conclusiones** | | | |
| "Simplifica y acelera el proceso de creación y configuración de la IA de enemigos durante el proceso de desarrollo del videojuego." | | | |

Tabla 3.13. Validación del *plugin* # 5

| **Validación del *plugin*** | | | |
|---|---|---|---|
| **Desarrollador** | Diosmel Yvonnet Guerra | | |
| **Tiempo de implementaciín manual (min)** | 10 | **Observación** | "Para un especialista promedio, este proceso se realiza en poco tiempo, pero resulta tedioso llevarlo al resultado deseado." |
| **Tiempo de implementación asistido(min)** | 1 | **Observación** | "Los valores se introducen de manera rápida y acertada quedando el balance de los parámetros a decisión del usuario." |
| **Conclusiones** | | | |
| "Con la utilización de este *plugin* se aprecia un ahorro significativo en el tiempo de configuración de enemigos grupales para tres tipos de enemigos con un comportamiento de IA básico. En el caso de la creación de los grupos, es bastante simple adicionar los miembros y definir el líder. Se recomienda continuar el desarrollo de esta aplicación incorporando más parámetros que correspondan con otros tipos de comportamientos grupales." | | | |

Con las pruebas realizadas se confirmó que el *plugin* creado puede ser utilizado por los desarrolladores del centro Vertex mediante la herramienta *Unity*. El mismo permite crear personajes con comportamiento de IA y realizar grupos con estos. De esta manera se agiliza el proceso de desarrollo de videojuegos, al incorporar estos grupos inteligentes en cualquier tipo de videojuego de aventura y acción para combates en tiempo real en fase de producción.

## Conclusiones parciales

En este capítulo se realizó la descripción de la solución y el análisis del resultado de las pruebas realizadas al *plugin* para *Unity*. Del análisis del resultado obtenido se llegó a la conclusión de que se cumplió con el objetivo de la investigación al lograr correctamente el desarrollo, tanto de la herramienta como de la IA que controlará cada tipo de personaje creado. Además, con la realización de estas pruebas se garantizó el funcionamiento óptimo del código, posibilitando que el cliente obtuviera resultados desde etapas tempranas del desarrollo de la solución. Finalmente, con el demo realizado se demuestra que el *plugin* está terminado y cumple con los requerimientos definidos; de esta manera se logra la aceptación por parte del cliente de la herramienta desarrollada.

# Conclusiones

Con la realización de este trabajo se arribó a las siguientes conclusiones:

- El *plugin* para *Unity* permite crear personajes enemigos y agruparlos sobre una base de comportamiento inteligente con el propósito de comportarse como un equipo.
- Con esta nueva herramienta, los desarrolladores del centro Vertex, podrán crear enemigos e incluirlos en grupos con comportamiento inteligente para ser insertados en videojuegos de aventura y acción en escenas de combate en tiempo real.
- Mediante el escenario demo creado con varios grupos de enemigos y un ejemplo de jugador se validó el funcionamiento del *plugin*. En este, los personajes creados se comportan como un grupo, ya sea Activo o Variado, realizando las funciones implementadas correctamente.

# Recomendaciones

El *plugin* creado es una herramienta que se incorpora a *Unity*, con esta los desarrolladores pueden crear videojuegos capaces de estar a la altura de las tendencias de IA grupales a nivel mundial. De esta manera, se recomienda que en futuros trabajos se desarrolle y perfeccione el mismo, con el propósito de agregar nuevas funcionalidades y comportamientos que puedan ser incorporados a los videojuegos que se creen.

# Acrónimos

***A.I.G.B.*** *Artificial Intelligence for Groupal Behavior*. 26, 54

***GRASP*** *General Responsibility Assignment Software Patterns*. 40

***GUI*** *Graphical User Interface*. 20–22

***LAN*** *Local Area Network*. 5

***MFS*** *Machine Finite State*. 9, 10, 29

***MMORPG*** *Massive Multiplayer Online RPG*. 6

***NPC*** *Non-Player Character*. 8, 11–16, 25–29, 31–40, 43–48, 54–56

***RPG*** *Role Playing Games*. 6

***RUP*** *Rational Unified Process*. 17, 18

***UML*** *Unified Modeling Language*. 38

***XP*** *Extreme Programming*. 18, 19, 25, 30, 33, 34, 36, 38, 42

**CRC** Clase - Responsabilidad - Colaboración. 38, 41

**HU** Historias de Usuario. 29, 30, 32–34, 41, 42

**IA** Inteligencia Artificial. 1, 2, 4, 6–9, 11–15, 17, 19, 25, 27, 29, 32, 33, 36, 38–40, 42, 43, 46–48, 55–59, 61

**ICAIC** Instituto Cubano del Arte e Industria Cinematográficos. 2

**TIC** Tecnologías de la Informática y las Comunicaciones. 1

**UCI** Universidad de las Ciencias Informáticas. 2

# Referencias bibliográficas


ALEGSA (2014). *Definición de script y plugin*. ALEGSA. URL: http://www.alegsa.com.ar (vid. págs. 8, 20).

Anaya Villegas, Adrian (2007). *A propósito de programación extrema XP* (vid. pág. 32).

Bautista, Jose M. (2014). «Ingeniería de Software». En: URL: http://ingenieriadesoftware.mex.tl/images/18149/PROGRAMACI%C3%83%C2%93N%20EXTREMA.pdf (vid. pág. 19).

Calero, Manuel (2003). «Una explicación de la programación extrema (XP)». En: *MADRID* (vid. pág. 18).

Canós, José H, Patricio Letelier y Mª Carmen Penadés (2003). «Metodologías Ágiles en el desarrollo de Software». En: *VIII Jornadas de Ingeniería de Software y Bases de Datos, JISBD* (vid. págs. 17, 18, 34).

CreativeCommons (2017). *Generos de Videojuegos*. GamerDic. URL: http://www.gamerdic.es/tema/generos (vid. pág. 7).

Duch i Gavalda Jordi; Tejedor Navarro, Heliodoro (2012). *Inteligencia Artificial*. Ed. por SL Eureca Media. Universitat Oberta de Catalunya (vid. págs. 9-13, 23, 24).

Echeverry Tobón, Luís Miguel y Luz Elena Delgado Carmona (2007). *Caso práctico de la metodología ágil XP al desarrollo de software* (vid. págs. 30, 32, 33, 36, 48).

ElOtroLado (2017). *GÃ©neros de los Videojuegos*. ElOtroLado. URL: http://www.elotrolado.net/wiki/G%C3%83%C2%A9neros_de_los_videojuegos (vid. pág. 5).

Escribano, Gerardo Fernández (2002). *Introducción a Extreme Programming* (vid. págs. 18, 32).

Gamma, Erich et al., (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley (vid. pág. 40).

Golodetz, Stuart (2013). «Automatic Navigation Mesh Generation in Configuration Space». En: URL: https://accu.org/index.php/journals/1838 (vid. pág. 23).

Jacobson, Ivar, Grady Booch y James Rumbaugh (1999). *The Unified Software Development Process*. Ed. por Addison Wesley (vid. pág. 18).

Letelier, Patricio (2006). *Metodologías ágiles para el desarrollo de software: Extreme Programming (XP)* (vid. págs. 17, 18, 29).

Marrero, Adrian Guerra (2010). *Introducción a la IA en los videojuegos*. Razon Artificial. URL: http://razonartificial.com/ (vid. págs. 1, 13).

Mojica Ortega, Irvin A. (2014). *Videojuegos*. Blogspot.com. URL: http://videojuegoslomejo.blogspot.com/ (vid. pág. 4).

MozillaDeveloperNetwork, MDN (2017). *JavaScript*. MDN MozillaDeveloperNetwork. URL: https://developer.mozilla.org/es/docs/Web/JavaScript (vid. pág. 25).

Pressman, Roger S. (2010). *Sotware Engineering. A Practitioner's Approach.* Ed. por McGraw Hill. Seventh Edition (vid. págs. 17, 19, 37, 48).

Rodriguez, Guillermo (2016). *La evolución de la inteligencia artificial en los videojuegos.* Vix. URL: www.vix.com/es/btg/gamer/4846/la-evolucion-de-la-inteligencia-artificial-en-los-videojuegos (vid. pág. 1).

Schreiner, Tim (2009). *Artificial Intelligence in Game Design*. URL: http://ai-depot.com/GameAI/Design.html (vid. págs. 1, 14).

Trujillo Rivero, Andy y Marvyn A. Márquez Rodríguez (2008). «Inteligencia Artificial en Videojuegos». En: (vid. págs. 7, 8, 12).

Unity (2016). *Unity3D*. URL: https://unity3d.com/es/unity (vid. págs. 19-25, 31, 46).

Universidad de Palermo, Facultad de Diseño y Comunicación (2013). *Clasificación Géneros de Videojuegos*. Facultad de Diseño y Comunicación. URL: http://fido.palermo.edu/servicios_dyc/blog/docentes/detalle_tp.php?id_docente=80946&id_blog=10680 (vid. págs. 5, 6).

Vossen, Bart (2015). *Enemy design and enemy AI for melee combat systems*. Gamasutra's community. URL: http://www.gamasutra.com/blogs/author/BartVossen/996035/ (vid. págs. 14-17).