Diseño e implementación de una plataforma gráfica e interactiva para evaluar estrategias de navegación de agentes en problemas de gran escala.

Karley Parada Haquin

Departamento de Ingeniería Informática y Ciencias de la Computación
Universidad de Concepción

Profesor Guía : Julio Godoy Del Campo
Departamento de Ingeniería Informática y Ciencias de la Computación.

Informe de Memoria de Título
para optar al título de
Ingeniero Civil Informático

Septiembre 2019
Concepción, Chile

Índice

Índice de figuras

AGRADECIMIENTOS

Se agradece a la Vicerrectoría de Investigación y Desarrollo de la Universidad de Concepción, por el apoyo económico, a través del proyecto de Iniciación VRID (código 218.093.018-1.0IN), para la realización de esta memoria de título.

Resumen

Los sistemas multiagente se utilizan en un amplio rango de aplicaciones, y pueden ser una forma muy conveniente para la comprensión, modelado, diseño e implementación de diferentes tipos de sistemas distribuidos [2][4].

En la actualidad, existen diversos métodos de navegación para sistemas multiagentes que pueden ser categorizados en los principios en los cuales se basa la interacción entre agentes y con obstáculos. Cada método tiene sus ventajas y desventajas, por lo que no existe un único método que sea mejor en cada una de las métricas manejadas.

Quienes estudian nuevos métodos de navegación o buscan mejorar los existentes deben generar una visualización para poder analizar de mejor manera el comportamiento de dicho método. Esto implica un gasto innecesario de recursos humanos (y tiempo) que pudiese dedicarse en mejorar las técnicas existentes, y con estas, el estado del arte en métodos de navegación. En este contexto es donde se propone la implementación de una plataforma que facilite el proceso de análisis y concluir con mayor rapidez por medio de un modelado de escenario.

En esta memoria de título, se describe el proceso de desarrollo de dicha plataforma, la cual incluye algunos métodos de navegación. En la plataforma se describen dos opciones de simulación en la cual se puede modelar un escenario a partir de un plano vacío, o comparar métodos en un escenario preestablecido.

La plataforma se desarrolló con el motor de videojuegos Unity, el cual ofrece un entorno de calidad con gráficas tanto en 2D como 3D adecuada a las exigencias actuales del usuario. Unity permite crear un juego (o, en este caso, simulación) en primera y tercera persona, una amplia visualización del plano, además gracias a su amplia variedad de objetos, elementos y herramientas en su tienda online, permite que el desarrollo sea más rápido y la visualización más atractiva.

La plataforma resultante permite al usuario simular un sistema multiagente de manera gráfica para evaluar estrategias de navegación de agentes y visualizar el comportamiento de los agentes simulados.

1. Introducción

Un sistema multiagente (SMA) es un sistema computacional compuesto por múltiples agentes[1] inteligentes, autónomos, que interactúan entre ellos y con el entorno, en donde cada agente tiene sus propios objetivos y motivaciones [1][3].

Uno de los ámbitos donde los sistemas multiagente se han mostrado como una herramienta efectiva, es para la investigación de métodos de navegación, donde existen diversos métodos que pueden ser categorizados en los principios en los que se basa la interacción entre un agente y el ambiente. Cada uno de estos métodos tiene sus ventajas y desventajas, por lo que no existe un único método que sea mejor en cualquier circunstancia.

El objetivo de la navegación de múltiples agentes es explorar sistemáticamente un entorno, adquiriendo información, por medio de receptores. Esto permite simular acciones e interacciones de individuos autónomos dentro de un entorno y así determinar qué efectos producen en el sistema.

1.1. Problema

En la actualidad, los investigadores que trabajan en mejorar las técnicas de navegación, destinan gran parte de su tiempo en la búsqueda de métodos implementados, entendiendo cada uno de los códigos (que pueden encontrarse en distintos lenguajes) y potencialmente implementando alguna forma de visualizar las trayectorias generadas por los agentes simulados.

Esto implica un gasto innecesario de recursos que pudiesen dedicarse en mejorar las técnicas existentes, y con estas, el estado del arte en métodos de navegación. Otro factor importante se relaciona con la complejidad del proceso de depuración a través de la simulación de múltiples agentes, debido a la gran cantidad de variables que observar, resultando en un desgaste mayor durante el proceso.

Para solucionar este problema, se diseñó e implementó una herramienta gráfica que permite evaluar de manera rápida el desempeño de navegación de agentes utilizando diferentes métodos de navegación. La herramienta permite establecer de manera gráfica los obstáculos, puntos de inicio y destino para cada agente ubicado, tanto al comienzo como durante la simulación.

La implementación de esta herramienta será de utilidad a los investigadores que trabajen en mejorar las técnicas de navegación de agentes, quienes podrán enfocarse en los aspectos algorítmicos de los métodos.

[1] Agente: Sistema computacional capaz de realizar acciones de manera autónoma en algún entorno, con el propósito de alcanzar una serie de objetivos designados.

Para lograr esto se estableció una comunicación entre el motor de videojuegos Unity, con los códigos de los métodos de navegación incluidos por medio de una librería dinámica generada.

1.2. Objetivos

Como objetivo para el desarrollo de la plataforma propuesta se tiene los siguientes:

1.2.1. Objetivo General

Implementar y diseñar una plataforma gráfica para evaluar estrategias de navegación de agentes en problemas de gran escala, que permita indicar los parámetros de la simulación por medio de una interfaz.

1.2.2. Objetivos Específicos

- Crear una estandarización de los códigos con los métodos de navegación.
- Comparar diferentes herramientas que permitan crear una plataforma gráfica.
- Diseñar e implementar simulación gráficas para los métodos de navegación.
- Evaluar y reportar el comportamiento de los agentes finalizada la simulación.

2. Discusión bibliográfica

La implementación de la plataforma gráfica para evaluar estrategias de navegación de agentes mediante una interfaz interactiva, permite al usuario establecer el escenario y los puntos de inicio y objetivo de cada agente, además de mapear un escenario por medio de obstáculos que este debe evitar.

Esto permitirá un mejor estudio de los sistemas multiagentes y su comportamiento con el entorno simulado.

Previo a la implementación, se realiza un estudio de los métodos de navegación a incluir en la plataforma, para así conocer el comportamiento de cada uno frente a distintas circunstancias, junto con esto, se realizó una comparación entre varios motores de videojuegos para conocer el más apto para la implementación de la plataforma. En esta sección se describe lo estudiado junto con los trabajos que se relacionan con este desarrollo.

2.1. Métodos integrados a la plataforma

Los métodos de navegación[2] corresponden a los diferentes algoritmos para realizar el proceso de navegación de agentes. En la actualidad, los métodos existentes calculan movimientos óptimos de manera local, esto se realiza analizando el comportamiento de los agentes vecinos y reaccionando según la información entregada por el entorno.

La plataforma implementada tiene integrado tres métodos de navegación que pueden ser asignados a cada agente. La plataforma permite contar con agentes que utilicen diferentes métodos de navegación en una misma simulación.

A continuación, se describen los métodos de navegación con los que cuenta la plataforma.

2.1.1. Optimal Reciprocal Collision Avoidance (ORCA)

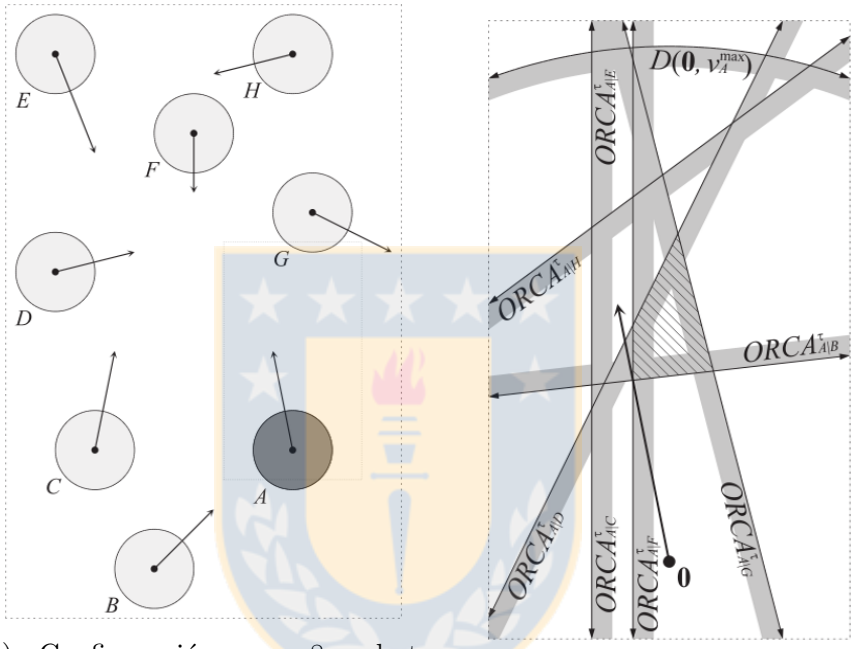
El método ORCA asume que la forma de los agentes (o robots) es simple: circular o un polígono convexo moviéndose en dos dimensiones en un espacio común. En ORCA, cada agente toma en cuenta la velocidad observada de otros agentes de tal forma que permita evitar las colisiones entre ellos en un tiempo futuro acotado.

El enfoque general de este método es el siguiente: Cada robot A realiza un ciclo continuo de percepción y movimiento en un rango de tiempo determinado. Durante

[2]Navegación: Bajo este contexto, la navegación hace referencia al estudio del proceso de monitoreo y control del movimiento de un agente mediante la determinación de la posición y orientación de este.

el proceso, el robot percibe el radio, la posición actual y velocidad actual de cada otro robot y de sí mismo (Figura 1a). Con esta información, el robot A infiere el semiplano de velocidades permitidas ($ORCA^{\tau}_{A|B}$) con respecto a cada otro robot, en este caso, un robot B (Figura 1b).

Finalmente, el conjunto de velocidades que se permiten para A con respecto a todos los robots es la intersección de los semiplanos de velocidades permitidas inducidas por cada otro robot ($ORCA^{\tau}_{A}$), correspondiente al área sombreada de la Figura 1b [6].



(a) Configuración con 8 robots (agentes) con sus velocidades actuales.

(b) Semiplanos de velocidades permitidas para robot A.

Figura 1: Enfoque general del método ORCA [6].

2.1.2. Social Forces

El modelo de Social Forces [7] se basa en el comportamiento peatonal, utilizando como sus variables la aceleración, la velocidad de movimiento deseada, la distancia que mantiene un peatón frente a otros, los bordes con los obstáculos y los efectos de atracción o repulsión de los mismos.

Un enfoque del modelo explica que para situaciones relativamente simples, se pueden desarrollar modelos de comportamiento estocásticos si se restringe a las probabilidades de comportamiento que se pueden encontrar en una gran población. Otro enfoque para modelar los cambios de comportamiento, es que estos son guiados por "fuerzas sociales"(social forces).

De acuerdo a esto, se plantea un proceso (Figura 2) que conduce a cambios de comportamiento a través de estímulos sensoriales, los cuales por medio de un proceso causan una reacción de comportamiento que depende de los objetivos personales. Esta reacción se elige de un conjunto de alternativas, de tal forma que permita maximizar la utilidad y cumplir el objetivo.

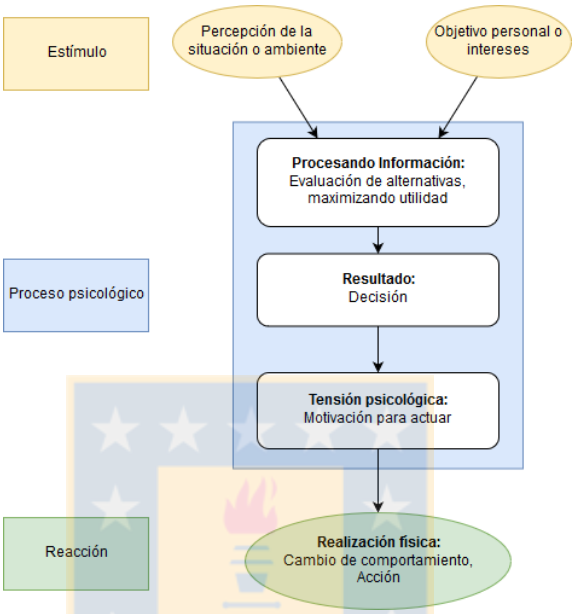


Figura 2: Representación esquemática del proceso que conduce a cambios de comportamiento.[7].

Es así como se plantea una clasificación de estímulo (Tabla 1), entre situación simple o estándar lo que corresponde a una situación que puede ser predecible, y una compleja que puede ser modelada con modelos probabilísticos. De acuerdo al estímulo es que se produce una reacción, en caso de que el estímulo sea predecible la reacción será automática o un acto reflejo, mientras que si el estímulo es a partir de nuevas situaciones la reacción dependerá de un proceso de decisión. [7].

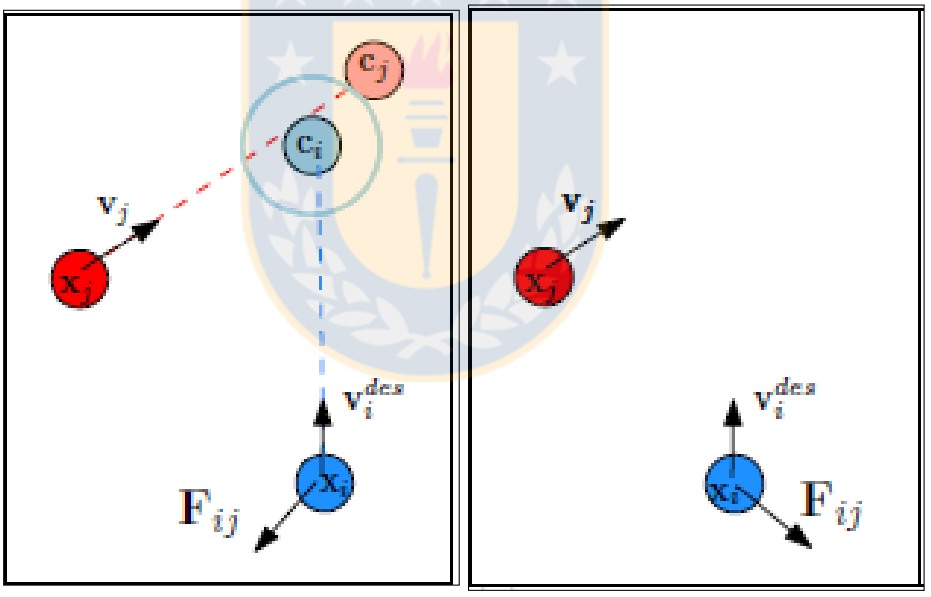
Estímulo	Simple/Situaciones estándar	Complejo/Nuevas situaciones
Reacción	▪ Reacción automática. ▪ Reflejo.	▪ Resultado de evaluación. ▪ Proceso de decisión.
Caracterización	Predecible.	Probabilístico.
Concepto de modelado	Modelo social force.	Modelo teórico de decisión.

Tabla 1: Clasificación de comportamiento de acuerdo a su complejidad.[7].

2.1.3. Predictive Collision Avoidance Model (PAM)

Según estudios de comportamiento de personas, existen dos procesos que gobiernan el comportamiento de evasión de colisiones entre peatones: la externalización y el escaneo. En la externalización el peatón usa su lenguaje corporal para informar a los demás su curso planificado y al mismo tiempo, escanea el entorno recopilando señales de otros peatones, y seleccionando la acción que requiere menos esfuerzo, reduciendo la cantidad de movimiento y esfuerzo de giro, y así llevar a cabo una coordinación voluntaria y resolver una colisión.

El principal enfoque de este modelo es la fuerza evasiva que selecciona un peatón **P** con el fin de evitar una colisión y colisiones cercanas con otros peatones (Figura 3). En primer lugar, el modelo establece un conjunto de peatones que están en curso de colisión con el peatón **P** estimando su velocidad actual (no puede estimar la velocidad deseada debido a que no conoce su posición de meta). Según el método se produce una colisión cuando un peatón se encuentra dentro o se cruza con el espacio personal de otro.



(a) Enfoque de método PAM. (b) Modelo basado en partículas.

Figura 3: Ejemplo de fuerza utilizada para evitar una colisión futura [9].

Del conjunto seleccionado de peatones en curso de colisión, estos se clasifican en orden con el objetivo de aumentar el tiempo previo a la colisión. Esto reduce el tiempo de ejecución del algoritmo y, además, refleja el comportamiento humano natural. De este modo, el peatón **P** intenta evadir a los **N** peatones con los que colisionará primero. Con el fin de evitar la colisión se establece el momento de colisión para obtener la ubicación de los peatones en ese instante. En base a esto, el peatón P determina una fuerza evasiva tal que corresponda al promedio de las fuerzas evasiva del conjunto seleccionado, para evitar sin problema a los otros peatones [9].

2.2. Motores de videojuego

Para el desarrollo de la plataforma se optó por utilizar un motor de videojuegos, debido a que estos proporcionan herramientas de desarrollo visual y componentes de software que pueden ser reutilizables, lo cual ayuda a reducir los costos, complejidades y tiempos de desarrollo.

Un motor de videojuego se subdivide en dos categorías; Motor gráfico y motor físico. El motor gráfico permite generar imágenes integrando información visual y espacial ya sea en 2D o 3D. El motor físico integra las leyes de la física, permitiendo simular acciones reales por medio de variables como la gravedad, masa, fricción, fuerzas ejercidas, etc.

Hoy en día, y gracias al desarrollo de nuevas tecnologías, existen variados motores de videojuegos, los cuales entregan al desarrollador una amplia gama de opciones según el tipo de videojuego o plataforma a implementar.

La siguiente tabla (Tabla 2) corresponde a una comparación entre los motores de videojuegos analizados previo a la implementación.

	Yoyo Game	Godot	Cryengine	Unreal Engine	Unity
Diseño de videojuego	2D	2D -3D	3D	3D	2D - 3D
Lenguaje de programación	Lenguaje propio	Lenguaje propio	C++	C++	C#
Licencia	Requiere licencia pagada	Gratuito	Requiere licencia pagada	Gratuito	Gratuito
Documentación	Inglés	Inglés	Inglés	Inglés, Chino, Japonés, Coreano	Inglés, Japonés, Español, Coreano, Ruso

Tabla 2: Comparación de motores de videojuegos—

Debido a que se busca que la plataforma permita al usuario una visualización completa del escenario, dándole el control de la cámara por medio de un desarrollo en primera persona. Además, se desea que la plataforma se adapte a cualquier sistema operativo, es decir, sea multiplataforma y gracias a su extensa y detallada documentación, es que se decide desarrollar la plataforma en Unity. Otro factor influyente corresponde a los requerimientos de hardware, debido a que Unity tiene bajos requerimientos de tarjeta gráfica.

2.3. Trabajos relacionados

Existe un trabajo relacionado con este proyecto, desarrollado por un estudiante de Doctorado en Ciencias de la Computación de la Universidad de Concepción[5]. Su trabajo consiste en una visualización de los métodos ORCA y Social Forces. Este trabajo fue implementado en C++, donde por medio de la pantalla de comandos se ingresaba una selección según lo que se quisiera visualizar, para luego ejecutar la visualización.

La principal diferencia entre el trabajo anteriormente descrito y lo desarrollado en esta memoria de título, corresponde a la forma de crear escenarios, debido a que en el trabajo existente [5] se simulan escenarios predefinidos indicando las posiciones de los agentes y obstáculos, lo cual no permite que el usuario visualice dicho escenario hasta que se ejecute la simulación. Por el contrario, la plataforma desarrollada en esta oportunidad permite modelar el escenario con facilidad, de manera visual y además, entrega métricas de evaluación del comportamiento del agente.

Otros trabajos relacionados, corresponden a implementaciones de visualizaciones para un método de navegación particular, con un escenario predefinido, y agentes preestablecidos, como los son las pruebas de los métodos anteriormente mencionados. El desafío es lograr ampliar esto, teniendo un plataforma que además de visualizar, permita mezclar métodos de navegación y realizar comparaciones de comportamiento.

3. Conceptos Previos

Previo a la implementación, se realizaron pruebas para conocer a fondo las opciones y funcionalidades entregadas por Unity, como lo son las fuerzas físicas aplicadas, movimiento armonioso del agente, áreas de navegación y opciones de interfaz.

3.1. Entorno Unity

Para un mejor entendimiento de lo que se realizó en la implementación de la plataforma se debe manejar el entorno Unity, el cual se compone por una serie de layouts que permiten el desarrollo de la plataforma. En la Figura 4 se pueden ver las herramientas utilizadas para la implementación de la plataforma, las cuales se describirán más adelante.

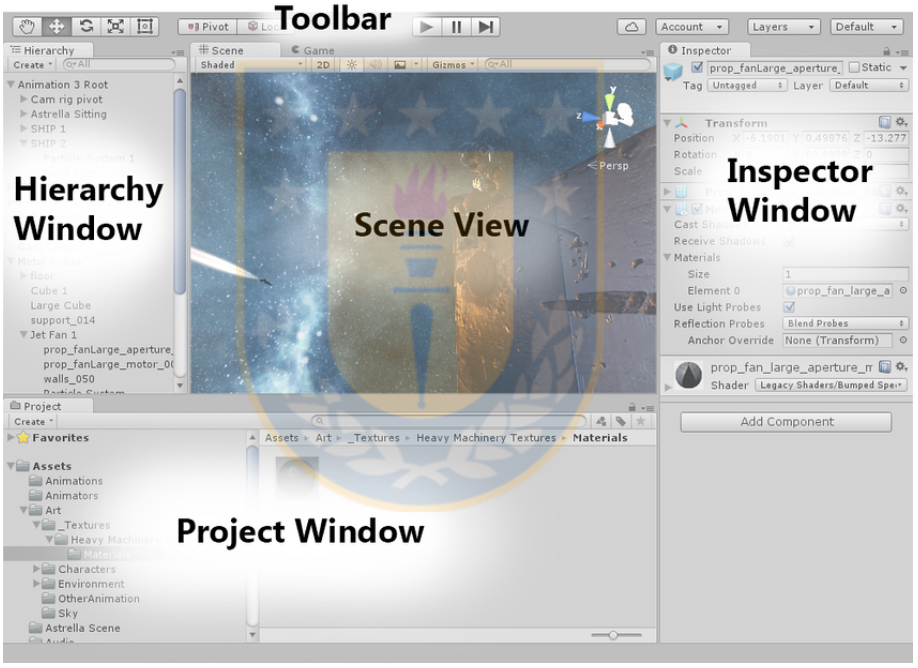


Figura 4: Layout de Unity [10].

Los principales layout utilizados son `Hierarchy`, `Scene`, `Game`, `Inspector` para la simulación, `Project` para la organización del proyecto y `Console` para las pruebas en el momento del desarrollo. A continuación, se describirá brevemente cada uno de estos.

3.1.1. Scene View

Los juegos o simulaciones en Unity se conforman de escenas, es por esto que esta ventana es donde se modela o crea el juego o simulación. Además se puede escoger el tipo de simulación, ya sea 2D o 3D y a partir de esto se puede ver los ejes coordenados en la esquina superior derecha. También se puede ver las diferentes

opciones de iluminación, terreno entre otros, junto con la ubicación de los objetos del juego entre otros (ver Figura 5).

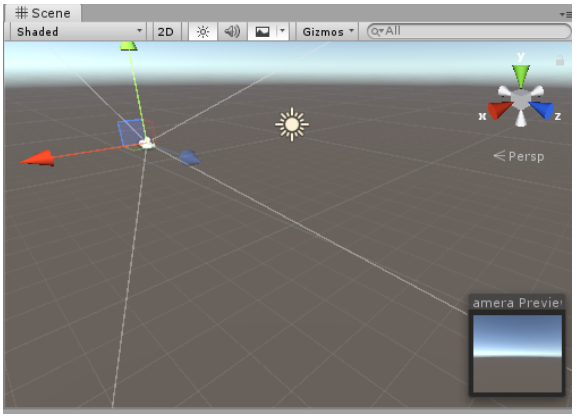


Figura 5: Scene

3.1.2. Game Window

En esta ventana se muestra cuál y cómo será el resultado en el juego según la cámara seleccionada, permitiendo realizar una simulación de este sin la necesidad de construir la aplicación como un ejecutable, y hacer pruebas durante el desarrollo. La visualización del juego dependerá plenamente de lo que muestra la cámara, un ejemplo de como se verá la ventana de `Game` es el cuadro de la esquina inferior derecha de la Figura 5, este cuadro aparece una vez de que se selecciona la cámara.

3.1.3. Hierarchy Window

Este layout permite crear la jerarquía de los objetos del juego o `Game Object`, agregando estos mismo en él. Cada escena por defecto trae 2 Game Object, uno correspondiente a la cámara principal (`Main Camera`) y el otro es la luz de la escena, la que por defecto es una luz directa (`Directional Light`).

Existen diferentes tipos de Game Objects, objetos 2D, 3D, objetos para la interfaz de usuario, cámara, iluminación, audio, vídeo, efectos y objetos vacíos. Para la implementación de la plataforma se utilizaron principalmente objetos 3D y de interfaz de usuario, además de los objetos por defecto como la cámara y la iluminación y objetos para almacenar los script.

3.1.4. Inspector Window

En este layout se definen los componentes que conforman cada Game Object, para la implementación de la plataforma se utilizan componentes tales como las componentes de `Renderer` que da el color y diseño del objeto, componente `Transform` que tiene la posición, escala y rotación del objeto, componente `Rigidbody` que permite reaccionar a las fuerzas físicas definidas, y componente `Collider` que da solidez al objeto.

3.1.5. Project Window

En esta ventana se muestra las carpetas que permiten la organización del proyecto, los cuales en Unity se llaman `Assets`. Para la implementación de la plataforma se organizaron carpetas para los scripts, modelos, materiales, texturas, escenas y animaciones.

3.1.6. Assets Store

En esta ventana se da acceso a la tienda de Unity, de donde se obtienen los modelos pre-diseñados por otros usuarios. Para la implementación de la plataforma se obtuvo de esta tienda el modelo de los agentes el cual consiste en el diseño de este.

4. Solución Propuesta

Para lograr los objetivos propuestos y llegar a la solución, además de la investigación previa de los métodos de navegación y el motor de videojuegos, se trabajó en base a 3 hitos críticos:

a. **Integración de código:** Proceso de integrar el desarrollo de los métodos de navegación con el entorno Unity mediante scripts.

b. **Desarrollo de plataforma:** Proceso de diseño y desarrollo de la plataforma e integración de los scripts a esta.

c. **Evaluación de escenarios:** Diseño de escenarios y sus resultados junto a la evaluación entregada por la plataforma.

4.1. Integración de códigos

Para lograr la comunicación entre los métodos de navegación mencionados en la Sección 2.1) y los script del entorno Unity se realiza el proceso *Marshalling*, el cual corresponde al proceso de transformar la representación de la memoria de un objeto en un formato de datos adecuado para el almacenamiento o la transmisión. Mediante este proceso se crea una librería dinámica (Dynamic-link library, DLL) encargada de transmitir los datos.

La Figura 6, muestra la arquitectura que representa la integración entre los códigos de los métodos ORCA, Social Forces y PAM (descritos en la sección 2.1) con el entorno Unity, donde el cuadro A muestra la generación de la librería dinámica (DLL), el cuadro ampliado B muestra la comunicación entre la librería generada y el script de C#, mientras que en C representa la simulación en el entorno Unity.

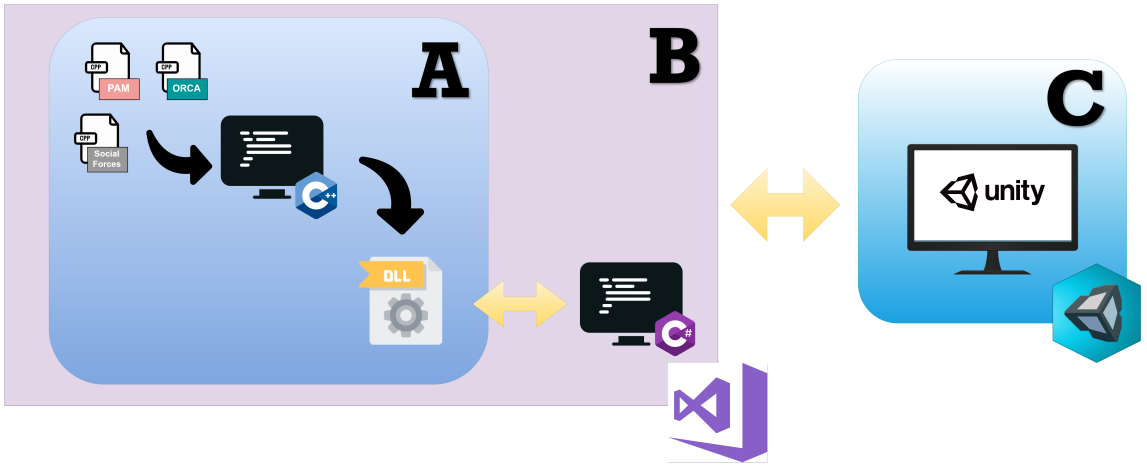


Figura 6: Comunicación entre scripts.

4.1.1. Adaptación de métodos de navegación.

A partir de los métodos de navegación desarrollados en C++, que serán incluidos en la plataforma, se creó una librería dinámica (Dynamic-link library, DLL) la cual almacena los métodos y permite hacer uso de estos en el script de Unity en C#.

Antes de la creación de la librería, se estudió cada uno de los métodos de navegación para conocer la estructuras utilizadas por cada uno y establecer un formato estándar de lectura y escritura de los datos correspondientes a los parámetros de agentes, obstáculos y destino. Para esto, se realizaron una serie de modificaciones que permitirán mapear el escenario creado en el simulador, calcular la próxima posición de cada agente, entre otros. Esto se logró mediante la creación de una estructura de datos estándar (para cada método de navegación) que permite identificar la posición de los agentes.

```
struct Point{
    public float x;
    public float y;
    public Point(float ejex, float ejey){
        x = ejex;
        y = ejey;
    }
}
```

Además, se creó una función para crear el escenario simulado en la plataforma, entregando la posición de cada objeto sobre el plano de visualización, e identificando el tipo de objeto.

Una vez finalizadas las modificaciones necesarias, y la inclusión de los métodos se construye la librería dinámica (DLL), como se muestra en el cuadro A de la Figura 6.

4.1.2. Comunicación con script en Unity

A partir de la librería generada (DLL), se da acceso a las principales funciones: crear el objeto que almacene los datos de la simulación, configuración del escenario, con las posiciones de los agentes, objetivos y obstáculos, y el movimiento actualizado de cada agente dada su posición en el instante del cálculo según el método de navegación seleccionado.

El script principal (que se encuentra adjunto en el Anexo 7.2.1) utilizado para el cálculo de las nuevas posiciones del agente, se forma por las funciones `Play()`, `Stop()`, `Update()` y funciones para obtener la posición de los agentes, objetivos y obtáculos, las funciones `Play()` y `Stop()` se encargan de la visualización de la simulación según corresponda, iniciando y pausando esta misma. En la función `Play()` se da la acción de crear el objeto que representa el escenario, para luego mapear

(o configurar el mismo), mientras que la función `Update()` permite actualizar las posiciones de cada agente.

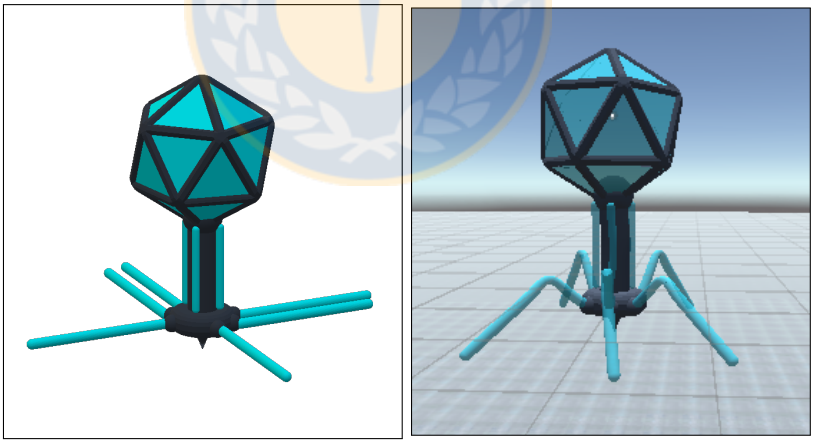
La actualización de las posiciones se realiza mediante una función incluida en la librería anteriormente generada, donde por cada objeto en la simulación (correspondiente a un *GameObject* según Unity), su posición inicial o "actual" y el método de navegación seleccionado, se calcula la nueva posición del agente y se mueve en la visualización, esto se realiza si es que no ha llegado a su objetivo aún, en caso de encontrarse con el objetivo se detiene el agente.

4.2. Desarrollo de plataforma

4.2.1. Definición de Agente

Se definió el modelo gráfico del agente, obtenido a partir del *Assets store* que provee Unity, de donde se importó un modelo de agente (ver Figura 7). Además se definieron variables tales como las fuerzas físicas actuantes en los agentes, las cuales para este caso fue solo la fuerza gravitacional, junto con el movimiento armonioso de las extremidades del agente.

Sumado a las definiciones físicas del agente, se agregó un script que permite mover el agente a gusto para así ubicarlo en el lugar deseado sólo arrastrando el modelo.



(a) Modelo de agente. (b) Vista en simulación.

Figura 7: Definición de agente

Para poder identificar de mejor forma el método de navegación utilizado por cada agente, la plataforma cuenta con tres versiones del modelo de agente, el cual difiere por los colores de cada uno, en la Figura 6 se puede ver las diferencias entre modelos y cuál es el método de navegación que estos utilizan.

(a) Modelo de agente ORCA. (b) Modelo de agente Social Forces. (c) Modelo de agente PAM.

Figura 8: Diferencias entre modelos de agentes

4.2.2. Definición de escenarios

La plataforma cuenta con dos escenas según la opción escogida en la pantalla de inicio, uno de los escenarios corresponde a un escenario preestablecido con un total de 10 agentes y un plano más reducido que permite al usuario lograr una visualización completa de la escena. La otra escena corresponde a un escenario de modelamiento, donde el usuario mueve los agentes a libre elección dentro de un plano más amplio.

Para ambos escenarios, se entrega la opción de movimiento de la cámara, para lograr una visión detallada o ampliada según sea requerido.

4.2.3. Diseño de escenarios.

Se diseñó una interfaz intuitiva y de fácil uso para el usuario. Para el desarrollo, se consideraron tres escenas de Unity, las cuales se describen a continuación.

- Home: Pantalla de inicio formada por los botones de opciones que da acceso a las diferentes escenarios.
- CreateScene: Pantalla que permite el modelado de la escena a elección del usuario.
- StaticScene: Pantalla pre-establecida, que muestra la navegación de 8 agentes.

A continuación, se detalla la formación de cada escena y las componentes Unity utilizadas.

4.2.4. Escena Inicio: Home

La escena de inicio esta principalmente formada por una cámara principal, luz y un `Canvas`. El Canvas es un tipo de Game Object el cual corresponde al área donde deben estar todos los elementos de la interfaz de usuario. Esta escena tiene como objetivo dar acceso a las opciones de escenarios desarrolladas.

Figura 9: Visión de juego.

La Figura 9 muestra el resultado final, la vista del juego ya en ejecución. A continuación se describe (a partir de la siguiente figura) la jerarquía de objetos para la creación de esta escena.

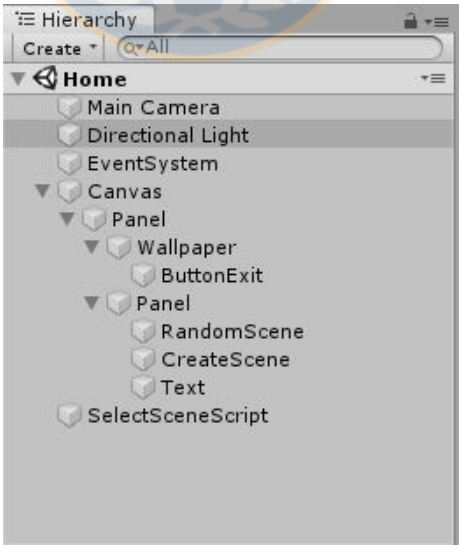


Figura 10: Jerarquía de Game Object para escena de inicio.

La composición de una escena y creación de nuevos objetos en Unity, siguen una composición jerárquica. La Figura 10 muestra la composición de la escena `Home`.

Debido a que la escena es para representar la interfaz de usuario, el principal objeto corresponde al `Canvas`, a partir de este se desglosan los otros objetos como el botón de salida y el `Panel` secundario que almacena los botones que dan acceso a cada escena. Además de los objetos visibles en la escena, existe un Game Object vacío (`SelectSceneScript`) donde se incluye el script que contiene las funciones de acción de los botones, en este caso los botones de selección de escenario.

4.2.5. Escena Interactiva: CreateScene

La escena interactiva está formada por los objetos principales (cámara y luz), el plano donde se disponen los agentes y un objeto canvas, en esta situación el canvas cumple la función contenedor de botón de salida (`ButtonExit`), botón para volver atrás (`ButtonBack`) y del panel donde se encuentran los botones de creación o eliminación de objetos (agentes, obstáculos), dar inicio o pausar la simulación, salir de la plataforma y volver a la escena anterior.

Además de las funciones ejecutadas al presionar un botón, la plataforma da la opción al usuario de realizar movimiento con la cámara presionando una tecla, por ejemplo, al presionar la tecla `Z` la cámara realiza un acercamiento que permite que el usuario tenga un mayor detalle, al presionar la tecla `X` la cámara realiza un alejamiento para permitir al usuario tener una visión panorámica, además las teclas de cursor permiten libre movimiento de la cámara considerando las limitaciones de espacio.

Figura 11: Jerarquía de Game Object en escena de creación de escenario.

El detalle de la formación de la escena en base a su jerarquía de Game Object (según la Figura 11) se basa en el `Canvas` del cual se desglosa un `Panel` y a partir de este los botones. Cabe destacar que para la creación de una escena interactiva no se crearon objetos agentes, sino que el botón de creación es el que le da la orden de agregar a la jerarquía. Para esta escena al igual que la anterior, se necesitaron de 2 `Game Object` vacíos, para almacenar el script correspondientes a las funciones de los botones (`ButtonGameObject`, Anexo 7.2.4) y para el cálculo de las posiciones y movimientos generales a realizar por cada agente (`GeneralMoveGameObject`, Anexo 7.2.1).

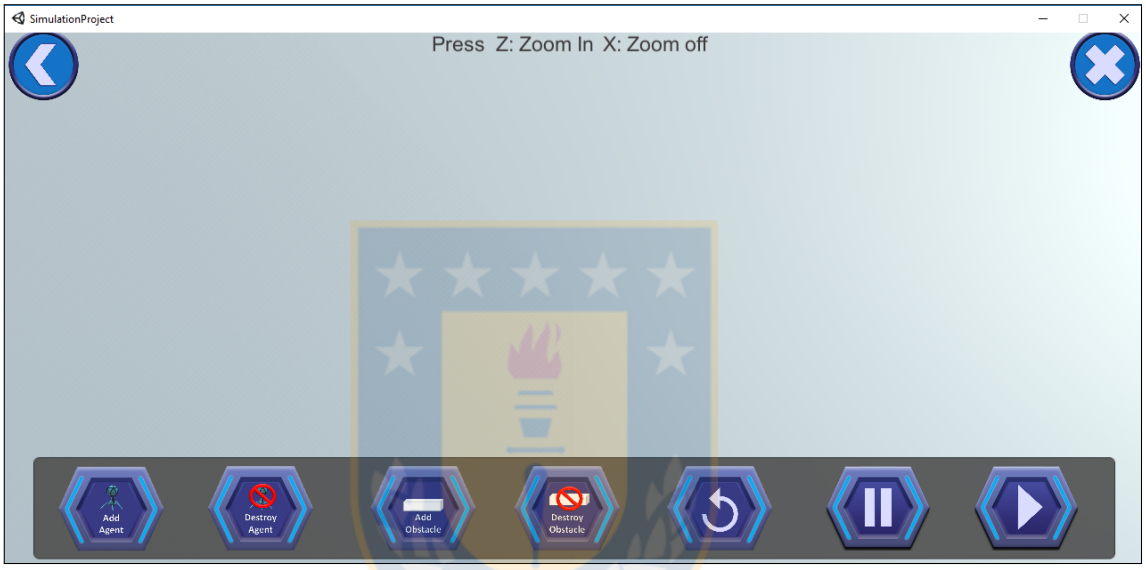


Figura 12: Vista para crear escenario.

El proceso para la agregar un agente y crear un escenario es el que se ve en las Figuras 12 a 15, en estas se ve en detalle la vista inicial al seleccionar la opción de crear escena (Figura 12), las opciones presentes para la creación de un agente, donde el modelo de agente que se mostrará será dependiendo del método de navegación que este seguirá (Figura 13).

Una vez seleccionado el método de navegación que seguirá el agente, este será agregado en el centro de la pantalla, (Figura 14) junto con un circulo negro, este último corresponde al objetivo que debe alcanzar el agente, ambos objetos se pueden mover a decisión del usuario, haciendo click sobre él y moviendo el cursor asignando al agente una nueva posición, como se ve en el ejemplo de la Figura 15.

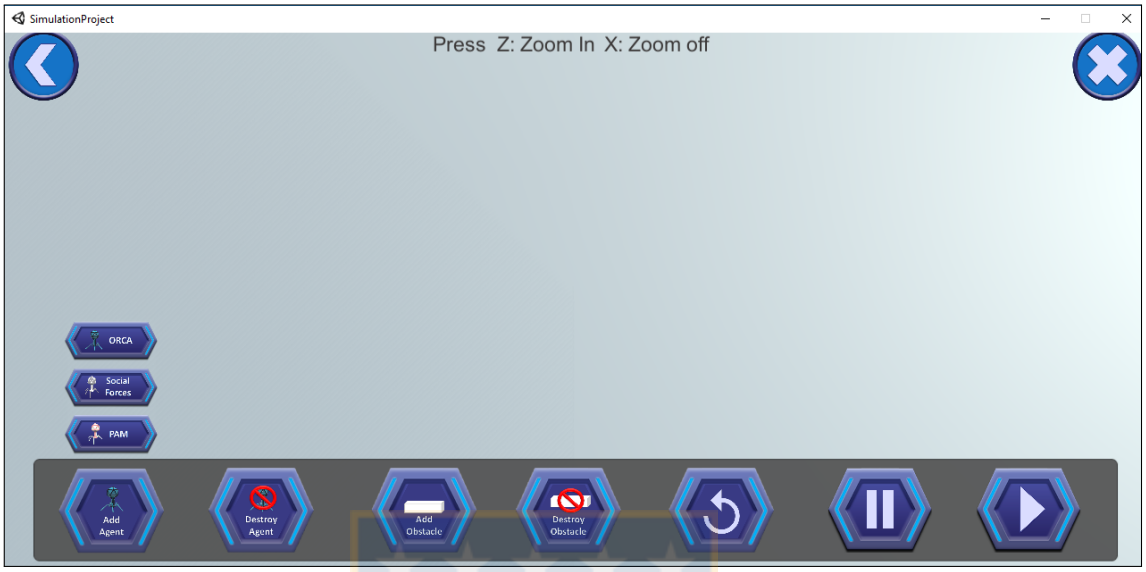


Figura 13: Opciones de creación de agentes.

Figura 14: Creación de agente y objetivo.

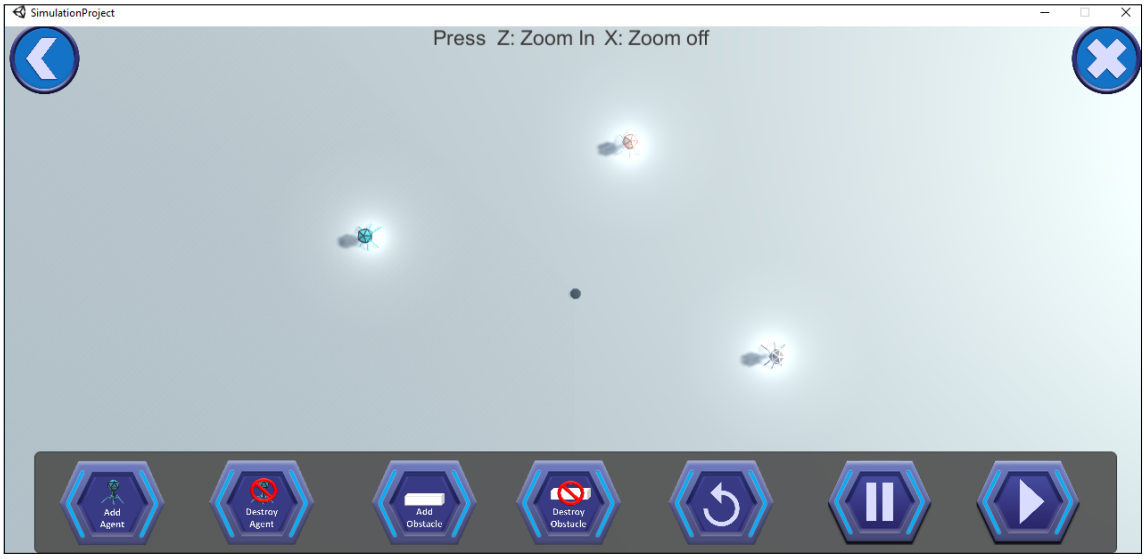


Figura 15: Movimiento de objetos.

4.2.6. Escena Estática: StaticScene

La escena estática está formada por los mismos componentes que la escena interactiva, la diferencia se encuentra en los tipos de botones: para el escenario estático se cuenta con botones para iniciar la simulación y pausar, además de los comunes de salir y volver a la escena anterior. La Figura 16 muestra los `Game Object` que componen esta escena. Al igual que la escena interactiva, esta escena permite que la cámara realice movimientos en todo el plano, para una mayor comprensión del escenario.

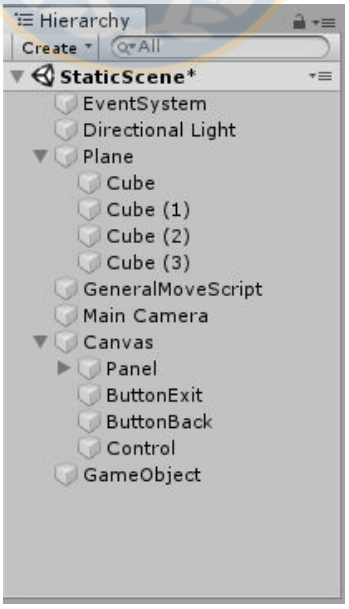


Figura 16: Jerarquía de Game Object en escena de estático.

En la escena estática, se incluyeron un total de 6 agentes con diferentes métodos de navegación y diferentes objetivos. El objetivo de esta pantalla es, a partir de

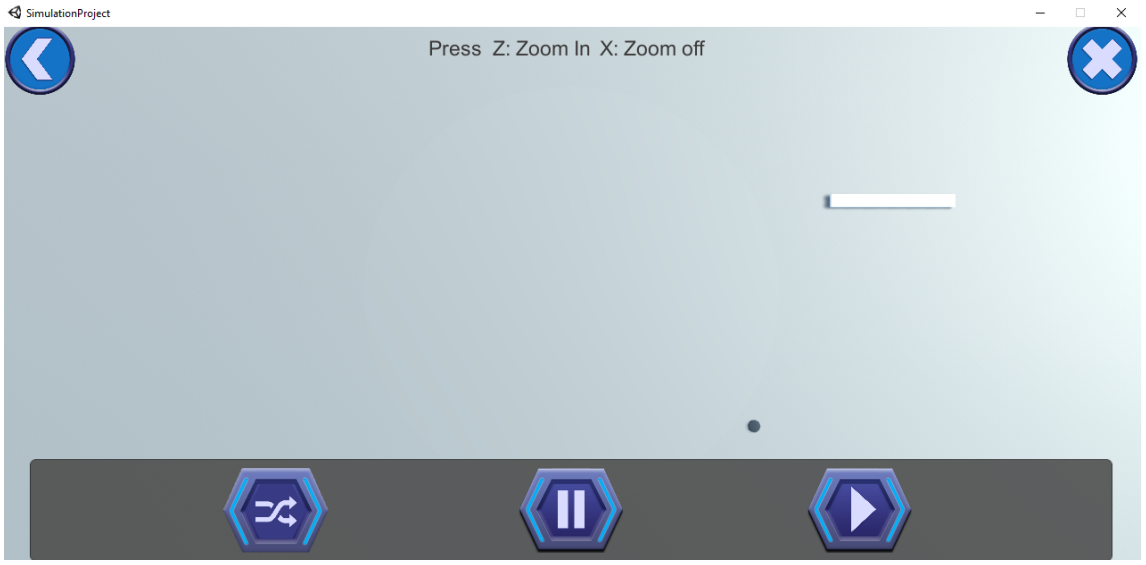


Figura 17: Vista de escena estática.

una escena preestablecida, obtener diferentes resultados dependiendo del método de navegación que utilice cada uno de los agentes.

En la Figura 17 se muestra la vista inicial de la escena estática con los agentes preestablecidos, debido a la extensión del plano en esta escena no es posible ver el escenario completo (para mayor detalle ver Figura 20). En la parte inferior de la ventana se muestran un panel con botones con diferentes acciones:

- Reubicar los agentes de manera aleatoria.
- Pausar la simulación.
- Comenzar la simulación.

4.3. Evaluación de escenarios

Para permitir una mejor evaluación de escenarios se incluye, en las escenas de creación de escenario y en la escena estática, un panel con los resultados de la simulación. Estos resultados consideran parámetros individuales (por agente), agrupados (por método de navegación) y generales de la simulación como se ve en la Figura 18.

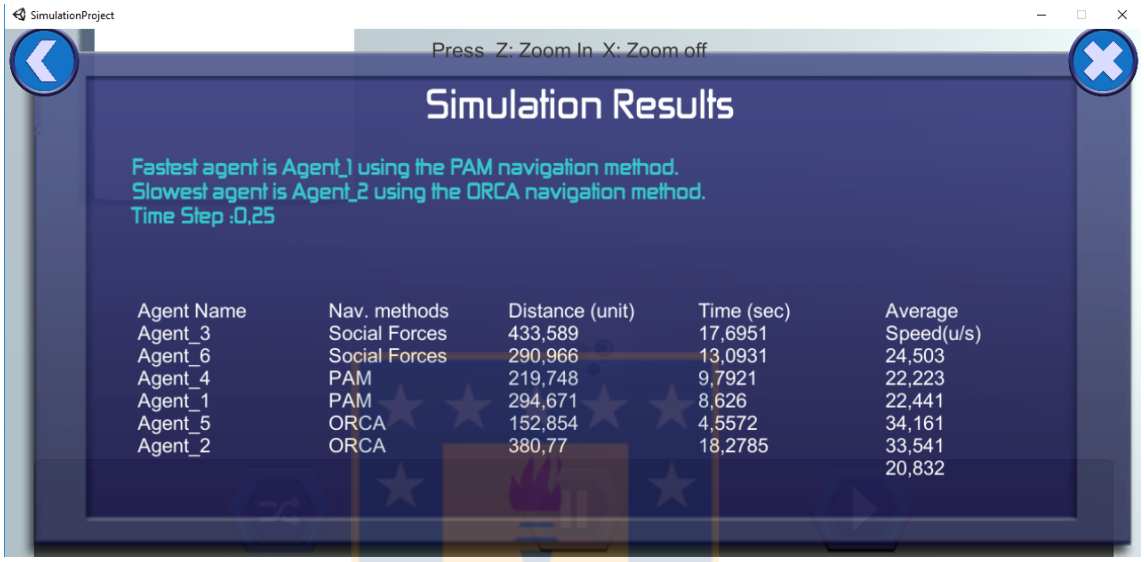


Figura 18: Vista de los resultados de una simulación.

A la siguiente sección, se describen los escenarios simulados y los resultados de cada uno, en mayor detalle.

5. Experimentación y Resultados

En esta sección se describen los escenarios estudiados y los resultados obtenidos por cada uno. Se simula desde un escenario simple a más complejo, incluyendo paulatinamente más agentes y con diferentes métodos de navegación. El objetivo de esto es analizar la escalabilidad de la plataforma y sus limitaciones.

El equipo donde se realizaron las pruebas corresponde a un notebook con procesador Intel(R) Core(TM) i5-7200 CPU @ 2.5GHz, con una memoria RAM de 8,00 GB, sistema operativo de 64 bits y procesador gráfico Intel(R) HD Graphics 620. Esto último es importante debido a que la experimentación corresponde a una simulación 3D, y por lo tanto posee una mayor cantidad de vértices geométricos (correspondientes a las esquinas que forman el modelo) con respecto a una simulación en 2D, lo cual aumenta la utilización de la GPU del sistema.

5.1. Evaluación de comportamiento de agente individual.

Se analizará el comportamiento de un agente en un escenario sin obstáculos y otro con obstáculo en medio de su recorrido. Se considera una distancia similar entre el escenario A (Anexo 7.1.1, Figuras 27, 29 y 31) sin obstáculos y el escenario B (Anexo 7.1.1, Figuras 28, 30 y 32), con obstáculos.

Para ambos escenarios (Anexo 7.1.1) se simuló un escenario sencillo donde el agente se encuentra en libertad de moverse directamente hacia su objetivo, debido a que no hay obstáculos en su camino, mientras que en el otro, el agente se encuentra con un bloque que le impide seguir avanzando por lo que debe rodearlo. Los resultados obtenidos se muestran en la Tabla 3.

Escenario	Tiempo[3] [s]	Velocidad Media [unit/s]
ORCA sin obstáculos	3,5841	28,369
ORCA con obstáculos	3,7829	27,906
Social Forces sin obstáculos	3,2383	29,954
Social Forces con obstáculos	3,2787	29,561
PAM sin obstáculos	3,211	40,183
PAM con obstáculos	3,2448	46,165

Tabla 3: Resultados de simulación de 1 agente.

De la Tabla 3 se puede concluir que debido a los obstáculos, el tiempo de ejecución aumenta y la velocidad disminuye, esto es producto de que el agente al encontrarse con un obstáculo disminuye su velocidad mientras lo rodea, por lo tanto tarda más en terminar la simulación.

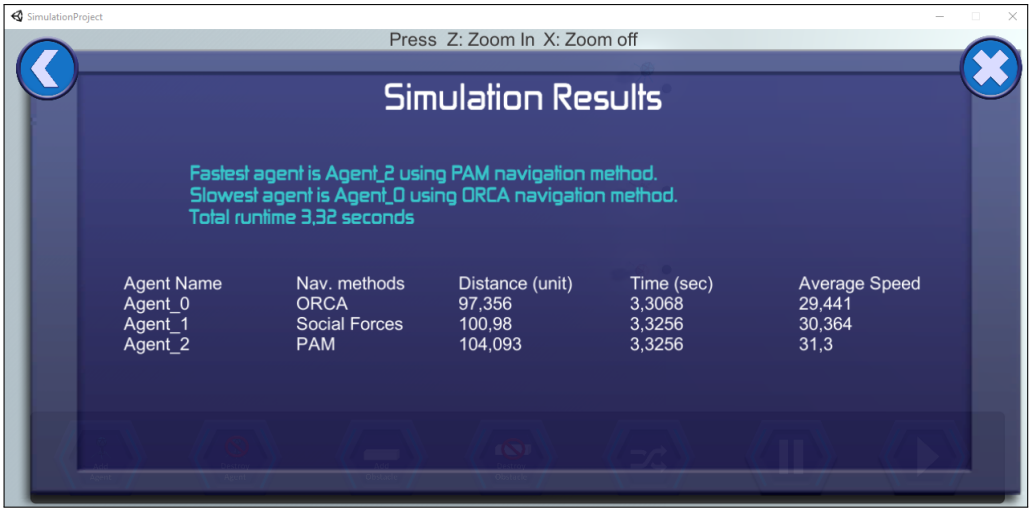
5.2. Evaluación de comportamiento de múltiples agentes.

Se realizaron pruebas con múltiples agentes en diferentes escenarios, entre los cuales se encontraban múltiples agentes utilizando el mismo método de navegación cuyos resultados se encuentran en el Anexo 7.1.2 y múltiples agentes con distintos métodos, los cuales se detallan a continuación.

5.2.1. Evaluación de 3 agentes con diferentes métodos de navegación.

En esta evaluación se ubicaron 3 agentes cada uno por un tipo de método de navegación, los cuales se ubican a una distancia similar a sus objetivos lo que permite llegar a una conclusión. La Figura 19 un ejemplo de los escenarios simulados descritos anteriormente.

(a) Escenario simulado.



(b) Resultado de simulación.

Figura 19: Escenario de 3 agente con distintos métodos de navegación.

En la Tabla 4, se muestran los resultados de la simulación descrita, indicando el tiempo que tarda el agente en llegar a su destino. A partir de los resultados, y para el escenario simulado se puede ver que el método más rápido corresponde al método PAM. Esto muestra cómo la plataforma permite, además de simular los escenarios, obtener estadísticas por agente.

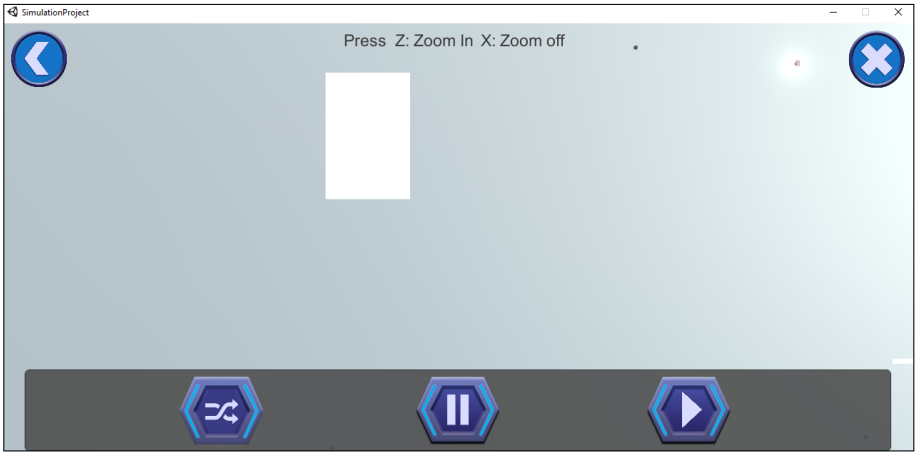
Evaluación	Método de navegación	Distancia[m]	Tiempo[s]	Velocidad Promedio [m/s]
Simulación 1	ORCA	173,949	3,2855	52,944
Simulación 1	Social Forces	178,285	3,3685	52,927
Simulación 1	PAM	181,139	3,383	53,544
Simulación 2	ORCA	97,356	3,30698	29,441
Simulación 2	Social Forces	100,98	3,3256	30,364
Simulación 2	PAM	104,093	3,3256	31,3

Tabla 4: Resultados de evaluaciones.

5.2.2. Evaluación de 6 agentes con diferentes métodos de navegación.

Para esta evaluación se utilizó la escena estática, donde se encuentran 6 agentes (2 agentes ORCA, 2 agentes Social Forces y 2 agentes PAM) en un escenario con obstáculos. Para evaluar de mejor manera se establecen distancias aleatorias, del mismo escenario, permitiendo analizar el comportamiento de un agente a distintas distancias de su objetivo.

La Figura 20 muestra parte escenario estático donde se realiza la evaluación. Debido a la extensión del plano, no es posible mostrar el escenario completo. Mediante el primer botón del panel inferior se reubican los agentes de forma aleatoria, esto implica que en cada evaluación la distancia y las posiciones iniciales no serán la misma, sin embargo, el escenario (obstáculos y objetivos) se mantienen en su ubicación establecida.



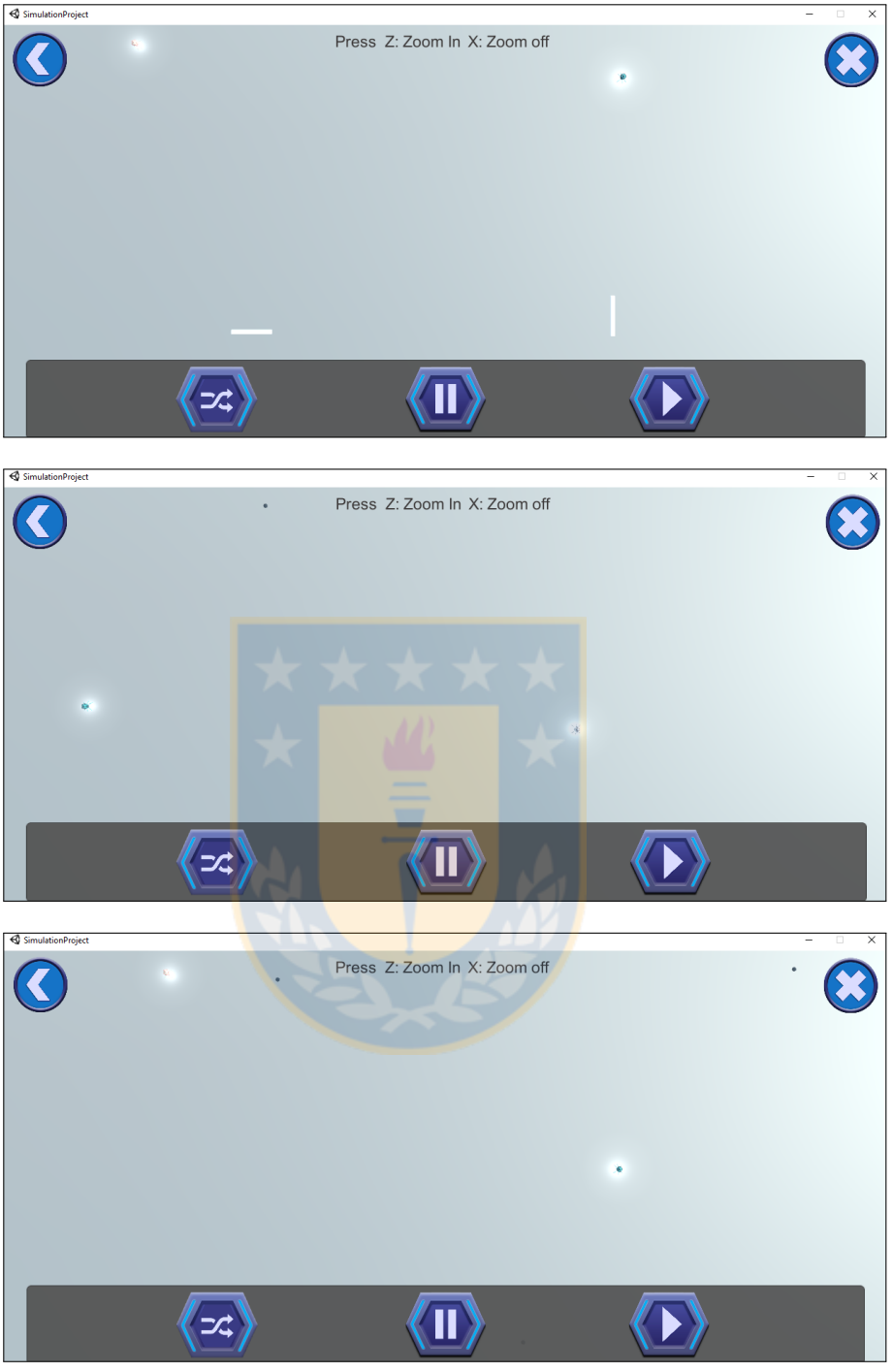


Figura 20: Simulación de escenario estático de 6 agente con distintos métodos de navegación y distancias aleatorias.

La Tabla 5, muestra la relación que existe entre la velocidad y la distancia, y cómo se desempeñan los agentes frente a grandes distancias junto a otros agentes que se pueden interponer en su camino. Los resultados de la evaluación **Sim1-Agente_N** (considerando N como número identificador del agente) corresponde al escenario por defecto que trae la plataforma. La evaluación **Sim2-Agente_N** y **Sim3-Agente_N** muestran los resultados de simular en el mismo escenario que el anterior, pero reubi-

cando de manera aleatoria los agentes. A partir de esto, se puede visualizar el efecto causado por los obstáculos en la velocidad de los agentes y por ende en el tiempo que tarde en llegar al objetivo, siendo el agente más restringido el que tenga menor velocidad.

Evaluación	Método de navegación	Distancia[m]	Tiempo[s]	Velocidad Promedio [m/s]
Sim1-Agente_3	Social Forces	432,589	17,4843	24,742
Sim1-Agente_6	Social Forces	289,966	12,8823	22,509
Sim1-Agente_4	PAM	218,748	9,5811	22,831
Sim1-Agente_1	PAM	293,671	8,4147	34,9
Sim1-Agente_5	ORCA	151,854	4,3297	35,073
Sim1-Agente_2	ORCA	379,77	18,068	21,019
Sim2-Agente_3	Social Forces	310,055	10,2477	30,256
Sim2-Agente_6	Social Forces	423,846	17,234	24,594
Sim2-Agente_4	PAM	468,565	18,3842	25,487
Sim2-Agente_1	PAM	257,327	7,2802	35,334
Sim2-Agente_5	ORCA	328,225	11,6156	28,257
Sim2-Agente_2	ORCA	242,189	6,3629	38,063
Sim3-Agente_3	Social Forces	98,366	5,6672	17,357
Sim3-Agente_6	Social Forces	520,665	20,1396	25,853
Sim3-Agente_4	PAM	356,132	15,3714	23,169
Sim3-Agente_1	PAM	179,362	10,02	17,9
Sim3-Agente_5	ORCA	317,848	11,2194	28,33
Sim3-Agente_2	ORCA	632,677	26,2086	24,14

Tabla 5: Resultados de simulación de 6 agente con distintos métodos de navegación en un escenario predeterminado y distancias aleatorias.

De las evaluaciones realizadas, se llega a la conclusión, que un buen determinante para la comparación de métodos de navegación, corresponde al grado de restricción de cada agente. Un agente más restringido, implica que es más lento, y por ende le llevará más tiempo llegar a su destino.

5.3. Pruebas de carga de la plataforma

Para conocer los alcances de la plataforma, se realizaron pruebas de carga aumentando la cantidad de agentes y monitoreando el rendimiento del equipo donde se ejecutan las simulaciones. Para realizar una mejor comparación, las pruebas se realizaron con un tipo de agente aumentando su cantidad desde 1, el mismo ejercicio se realiza con los tres métodos incluidos en la plataforma.

Al finalizar cada simulación, la plataforma entrega los resultados de la misma, con la descripción de los nombres de cada agente simulado, la distancia que recorrió, el tiempo que tardó en llegar a su objetivo y la velocidad promedio a lo largo del recorrido. Con estos datos se realizó un análisis comparativo entre los métodos en cada escenario.

5.3.1. Velocidad promedio de agentes

A partir de los datos de velocidad promedio de cada agente, se calculó el promedio por simulación, donde los resultados obtenidos son los que se muestran en el siguiente gráfico (Figura 21).

La Figura 21 muestra la velocidad promedio según la cantidad de agentes simulados en cada escenario, para realizar una comparación de los escenarios se simuló el mismo escenario para cada tipo de método de navegación. Esta gráfica permite conocer qué tan restringido es el movimiento de cada tipo de agente.

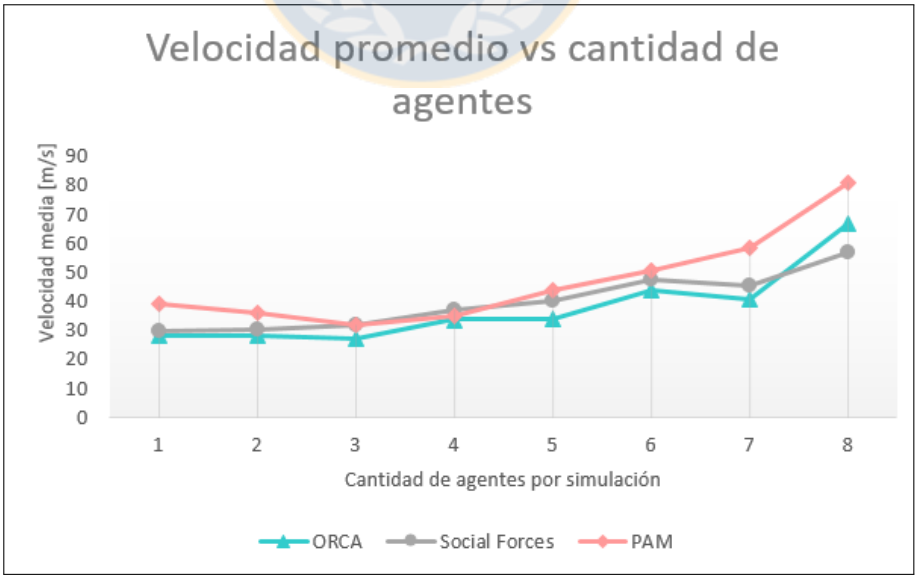


Figura 21: Velocidad promedio de simulación de cada método.

Mediante la comparación realizada se ve que los agentes que recorren con método PAM son más veloces que los agentes que recorren usando método Social Forces y este más veloz que los agentes que recorren con método ORCA, es decir un agente que

recorre utilizando método ORCA se encuentra más restringido que los otros agentes. Sin embargo, hay que considerar que las distancias recorridas por cada agente en cada escenario no son los mismos.

5.3.2. Tiempo de ejecución de la simulación.

A partir de los escenarios simulados (Anexo 7.1.2) se obtiene el tiempo de simulación, es decir, el tiempo trascurrido desde que se presiona Play hasta que finaliza la simulación y se muestra la ventana de resultados. Los resultados obtenidos se muestran en la siguiente gráfica (Figura 22).

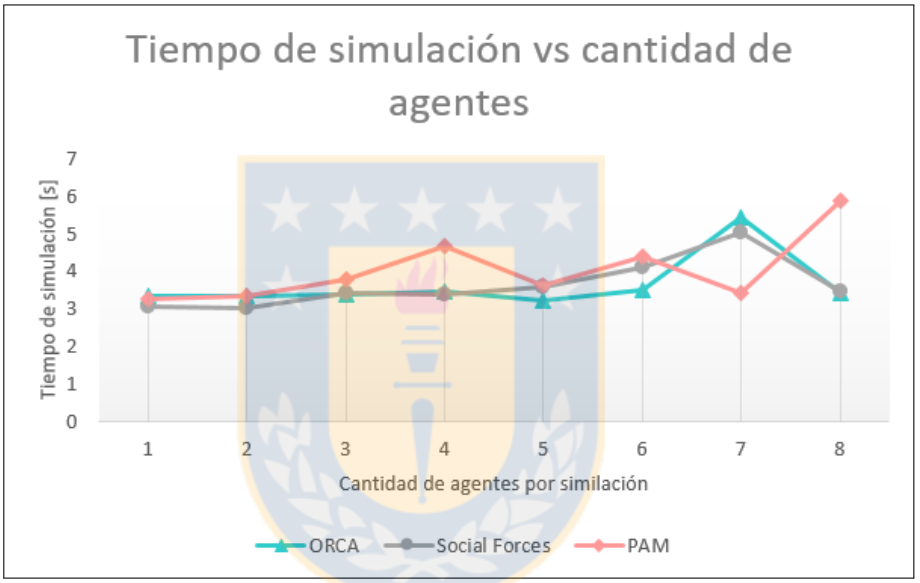


Figura 22: Tiempo de ejecución de simulación de cada método.

Debido a que de la gráfica anterior (Figura 22) no establece una relación entre el tiempo de ejecución y la cantidad de agentes simulados, se realizaron pruebas de congestión con diferentes cantidades de agentes.

Las pruebas de congestión consistieron en 3 simulaciones en un escenario común, la primera simulación se conforma de 7 agentes con objetivos que consideran una trayectoria de alta y baja congestión (Figura 23). La segunda simulación, se conforma de 3 agentes (1 por cada método incluido en la plataforma), donde estos tenían un recorrido de alta congestión, ya sea por posible colisión con obstáculos u otros agentes (Figura 24). La tercera simulación, se conforma de 3 agentes al igual que la anterior, pero en este caso de baja congestión, donde las trayectorias no llevan a colisionar (Figura 25).

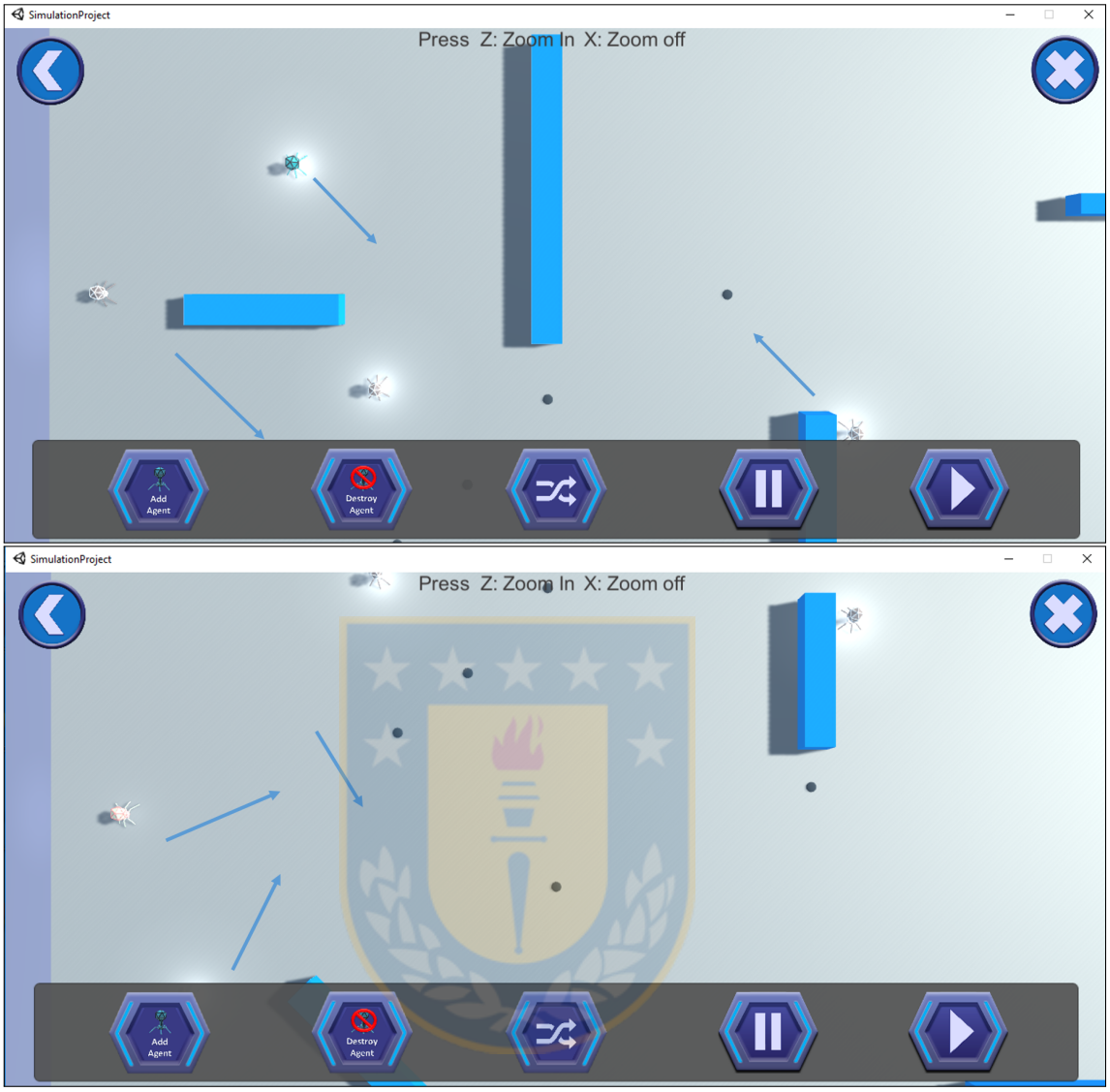


Figura 23: Simulación 1.

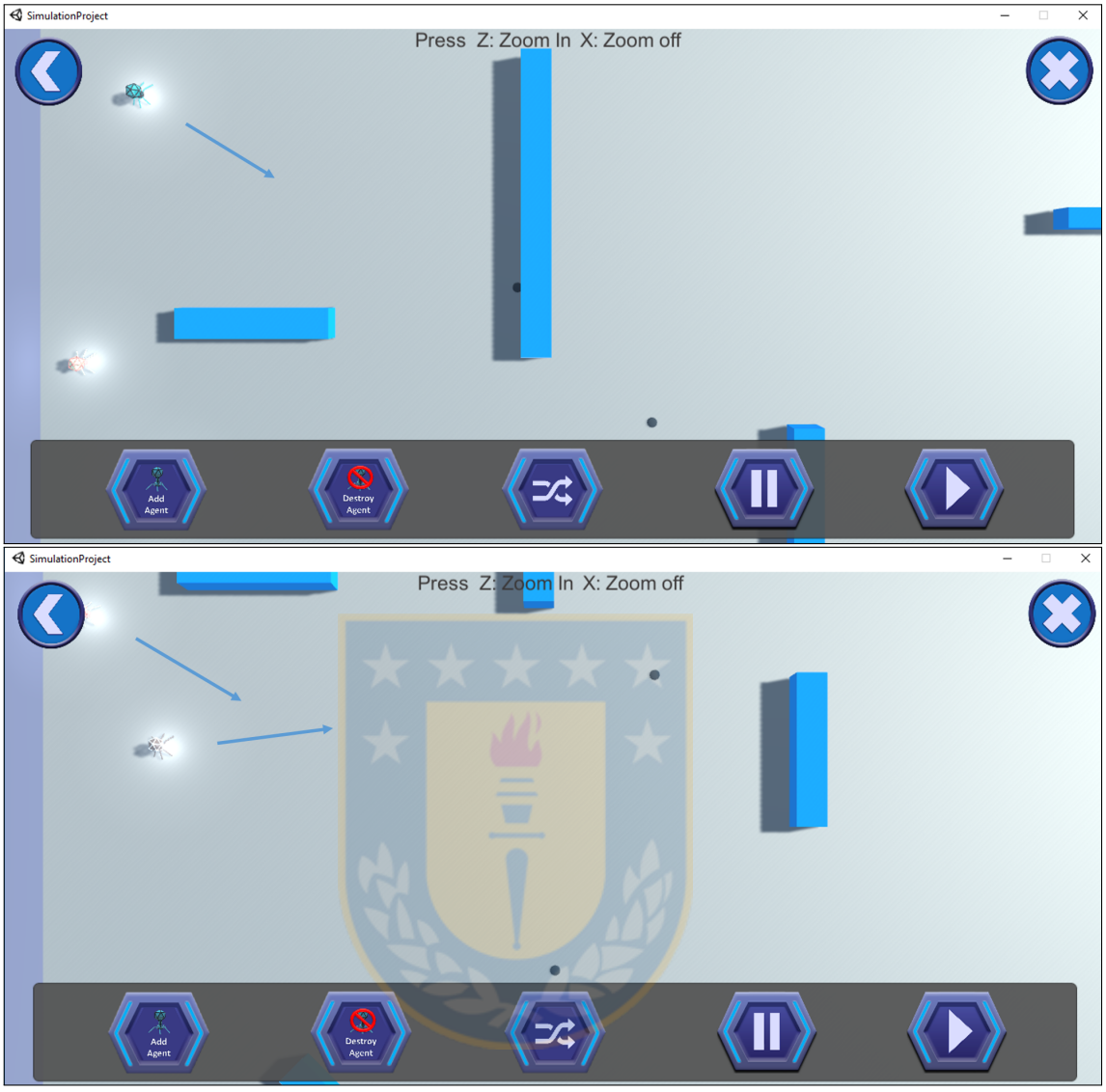


Figura 24: Simulación 2.

Figura 25: Simulación 3.

Las flechas de las Figuras 23, 24 y 25, indican el objetivo de cada uno de los agentes, debido al tamaño del escenario el plano fue separado en dos partes. Los resultados de estas simulaciones son los que se muestran en la Tabla 6.

Simulación	Método de navegación	Distancia[m]	Tiempo[s]	Velocidad Promedio [m/s]
Sim.1-Agente_1	ORCA	94,72	2,1568	43,916
Sim.1-Agente_2	Social Forces	111,931	2,623	42,673
Sim.1-Agente_3	Social Forces	29,657	1,1225	26,42
Sim.1-Agente_4	PAM	92,394	1,773	52,111
Sim.1-Agente_5	Social Forces	90,487	3,1068	29,129
Sim.1-Agente_6	PAM	100,365	2,4396	41,129
Sim.2-Agente_1	ORCA	120,05	2,8147	42,651
Sim.2-Agente_2	Social Forces	109,775	3,2151	34,144
Sim.2-Agente_3	PAM	129,267	3,8324	33,73
Sim.3-Agente_1	ORCA	105,347	2,6274	40,095
Sim.3-Agente_2	Social Forces	120,942	3,0945	39,082
Sim.3-Agente_3	PAM	38,409	1,6935	22,68

Tabla 6: Resultado de evaluaciones.

La Tabla 6, corresponde al detalle de las simulaciones anteriores, esta se resume en la siguiente tabla.

Simulación	Cantidad de agentes	Tipo de congestión	Tiempo de simulación [s]
Simulación 1	6	Alta y baja	3,11
Simulación 2	3	Alta	3,83
Simulación 3	3	Baja	3,09

Tabla 7: Resultados de tiempo de simulación.

De la Tabla 7 se concluye que el tiempo de simulación no depende de la cantidad de agentes sino de su congestión, visualmente mediante la plataforma se pude confirmar esto, debido que el agente al encontrarse con un obstáculo en su trayectoria, disminuye su velocidad para lograr rodearlo y no colisionar contra este.

5.3.3. Resultado de pruebas de carga.

Debido a las pruebas, se puede concluir que actualmente la plataforma se encuentra habilitada para una simulación de hasta 8 agentes, ya sean con el mismo método de navegación o diferentes, esto debido a que al simular un noveno agente, los recursos del equipo (CPU) se ven agotados y no permite comenzar la navegación de los agentes. Esto se debe a que al realizar acceso a memoria cada vez que se realiza el cálculo de la nueva posición por cada agente y por cada "paso"que este da para llegar a su objetivo. Además, al tener una visualización 3D, se consume una mayor cantidad de GPU.

La Figura 26 muestra el rendimiento del equipo durante una simulación de 9 agentes, donde se puede apreciar los altos peaks de consumo de CPU y GPU.

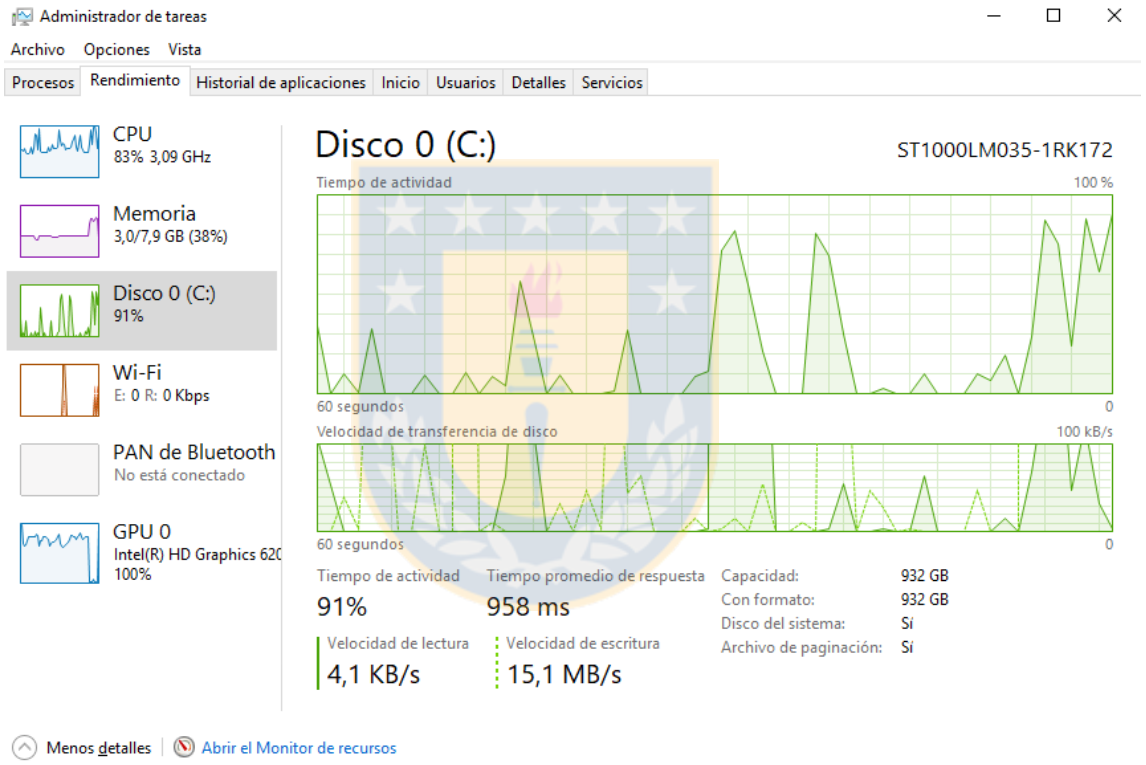


Figura 26: Resultado de prueba de 9 agentes.

6. Conclusión

La herramienta desarrollada es una plataforma que permite visualizar el comportamiento de múltiples agentes a partir de el método de navegación que utilicen, esto para simular un sistema multiagente con mayor facilidad. Esto se logró integrando los métodos de navegación PAM, Social Forces y ORCA con el motor de videojuegos Unity. La integración de nuevos métodos no es compleja, solo se requiere que siga la misma estandarización usada con los métodos anteriormente nombrados.

La plataforma desarrollada ayudará a que investigadores que trabajen en mejorar las técnicas de navegación existentes destinen una mayor parte de su tiempo en la investigación, desarrollo y optimización de las técnicas, permitiendo realizar comparaciones con mayor facilidad y visualizando el comportamiento de los agentes ante obstáculos del escenario, ya sean otros agentes u objetos inanimados.

Durante el proceso de desarrollo de la plataforma, y relacionado con los hitos críticos donde a partir de una estandarización de los códigos incluidos en la plataforma, se desarrolla la simulación gráfica, la cual permite la interacción del usuario, logrando los objetivos principales. Sin embargo, la interacción del usuario en ejecución se logra parcialmente, esto debido que la interacción se encuentra condicionada, es decir, se logra una vez pausada la simulación, y no mientras esta se ejecuta.

La correcta ejecución de la plataforma, se encuentra sujeta a las características y rendimiento del equipo, esto debido a que la simulación está desarrollada en un escenario 3D, por lo que la simulación de una gran cantidad de agente se ve limitada.

La plataforma tiene las cualidades tales para poder ser adaptada para cumplir otros objetivos, como por ejemplo simular un entorno para evaluar tiempos de evacuación en caso de emergencias.

6.1. Trabajo Futuro

Finalizada la implementación de la plataforma, se encontraron nuevas aristas a trabajar, alguno de estos trabajos futuros se encuentran los siguientes:

- Ampliar la cantidad de métodos incluidos en la plataforma.
- Implementar un sistema que permita incluir nuevos métodos con facilidad.
- Aplicar paralelismo en los códigos para el cálculo de las nuevas posiciones.
- Incluir nuevas métricas para comparar el comportamiento de los agentes.
- Ampliar los modelos de agentes y obstáculos para la simulación.

Referencias

[1] Sistemas Multiagente y Simulación. Fernando Sancho Caparrini, Obtenido de: http://www.cs.us.es/ fsancho/?e=57, 2019.

[2] Moreno Ruiz, Lorenzo & Hamilton, Alberto & Mendez, Juan Albino & Nicolás Marichal Plasencia, Graciliano & José González González, Evelio. (2006). Diseño e implementación de un sistema multiagente para la identificación y control de procesos. Ingeniería informática, ISSN 0717-4195, Nº. 12, 2006.

[3] EcuRed. Sistemas Multiagentes, Obtenido de: https://www.ecured.cu/Sistemas_Multiage ntes, 2019.

[4] Wikipedia. Sistemas Multiagentes, Obtenido de: https://es.wikipedia.org/wiki/Sistema_m ultiagente, 2019.

[5] Documento en desarrollo, José Luis Viloria Rodríguez, Ingeniero Informática, Master en Informática Aplicada, Universidad Tecnológica de La Habana José Antonio Echeverría. https://www.linkedin.com/in/jviloria/.

[6] Jur van den Berg, Stephen J. Guy, Ming Lin & Dinesh Monacha. Reciprocal n-body Collision Avoidance. International Symposium on Robotics Research, 2009.

[7] Dirk Helbing, Peter Molnar, Social Force Model for Pedestrian Dynamics. Physical Review E 51, 4282-4286, 1995.

[8] Ioannis Karamouzas, Brian Skinner & Stephen J. Guy. Universal power law governing pedestrian interactions. Phys. Rev. Lett. 113, 238701 (2014)

[9] Ioannis Karamouzas, Peter Heil, Pascal van Beek & Mark H. Overmars. A Predictive Collision Avoidance Model for Pedestrian Simulation.MIG, 2009.

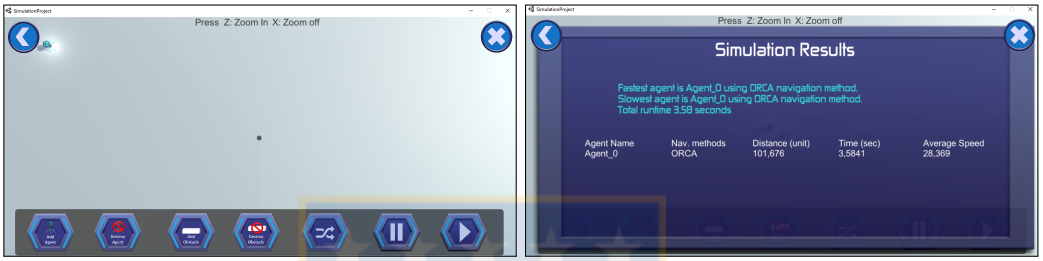
[10] Unity - Manual: Learning the interface (2018.3), https://docs.unity3d.com/2018.3/Documentation/Manual/LearningtheInterface.html

7. Anexos

7.1. Resultados de simulaciones.

7.1.1. Evaluación de comportamiento de agente individual

En las siguientes imágenes se muestra los resultados de estos escenarios, indicando información tal como velocidad, distancia recorrida, y tiempo del agente. Se debe mencionar que para estos casos el agente más rápido y el más lento será el mismo, debido a que corresponde a un escenario de prueba de solo un agente.

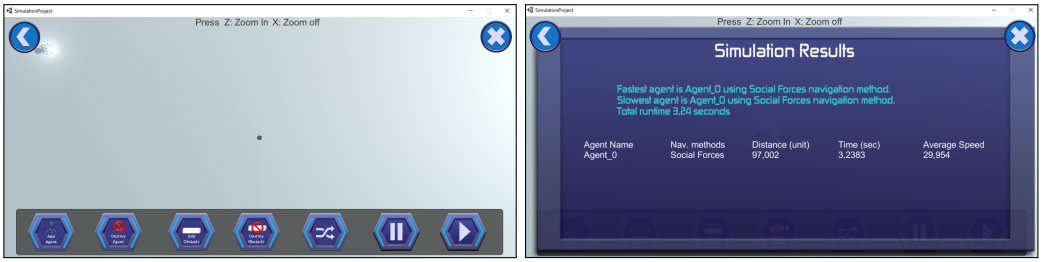


(a) Escenario sin obstáculos. (b) Resultados sin obstáculos.

Figura 27: Escenario de 1 agente con método de navegación ORCA sin obstáculos.

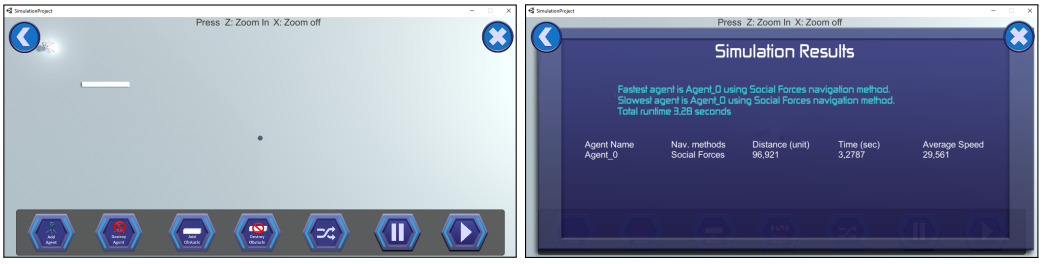
(a) Escenario con obstáculos. (b) Resultados con obstáculos.

Figura 28: Escenario de 1 agente con método de navegación ORCA con obstáculos.



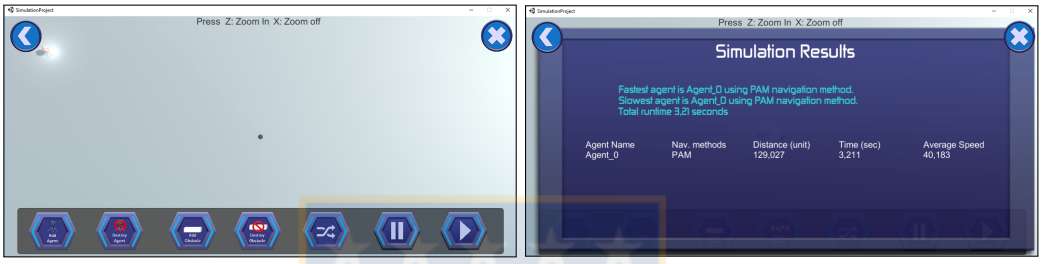
(a) Escenario sin obstáculos. (b) Resultados sin obstáculos.

Figura 29: Escenario de 1 agente con método de navegación Social Forces sin obstáculos.



(a) Escenario con obstáculos. (b) Resultados con obstáculos.

Figura 30: Escenario de 1 agente con método de navegación Social Forces con obstáculos.



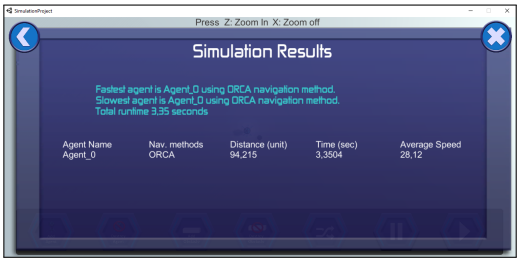
(a) Escenario sin obstáculos. (b) Resultados sin obstáculos.

Figura 31: Escenario de 1 agente con método de navegación PAM sin obstáculos.

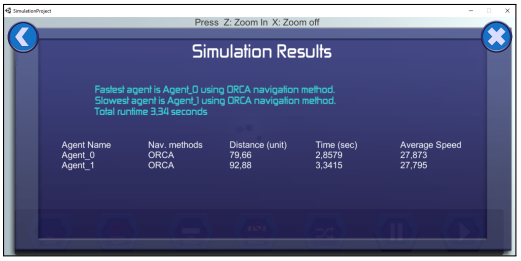
(a) Escenario con obstáculos. (b) Resultados con obstáculos.

Figura 32: Escenario de 1 agente con método de navegación PAM.

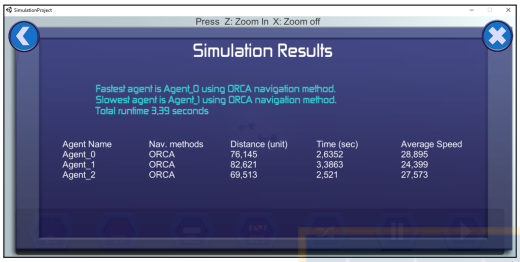
7.1.2. Evaluación de escalabilidad de plataforma.



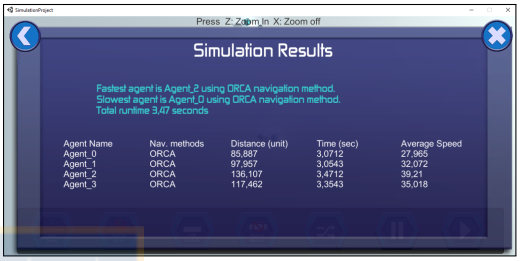
(a) Simulación de 1 agente.



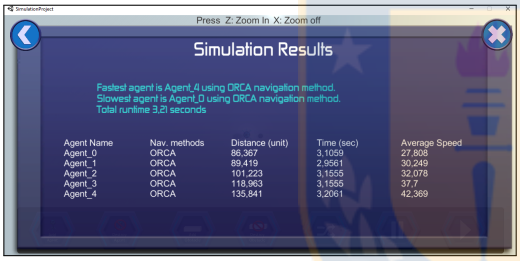
(b) Simulación de 2 agente.



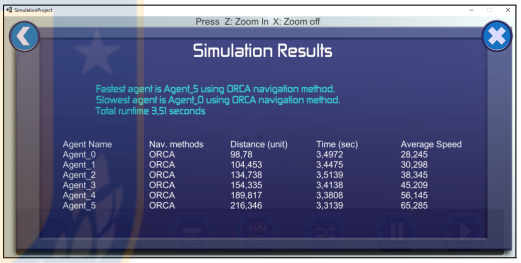
(c) Simulación de 3 agente.



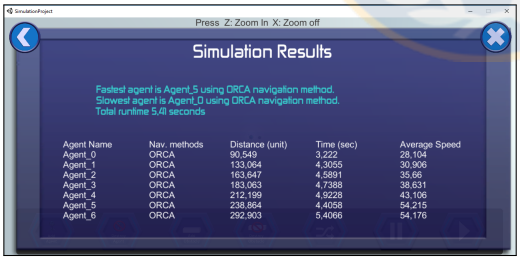
(d) Simulación de 4 agente.



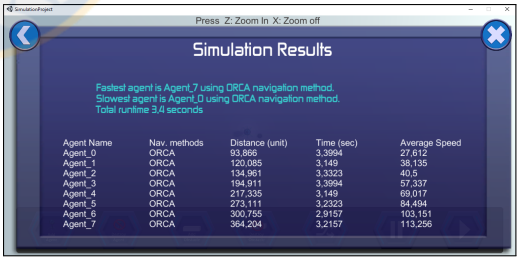
(e) Simulación de 5 agente.



(f) Simulación de 6 agente.

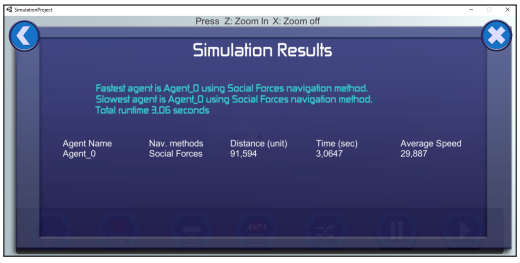


(g) Simulación de 7 agente.

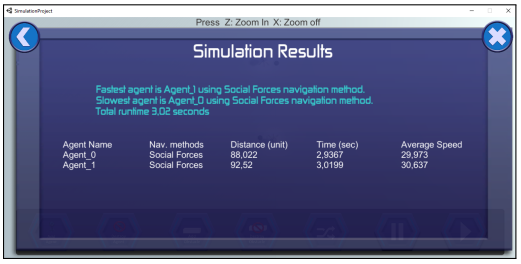


(h) Simulación de 8 agente.

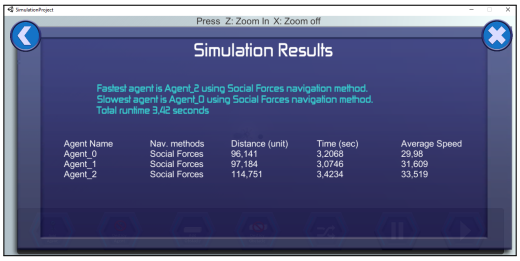
Figura 33: Escalabilidad de método ORCA.



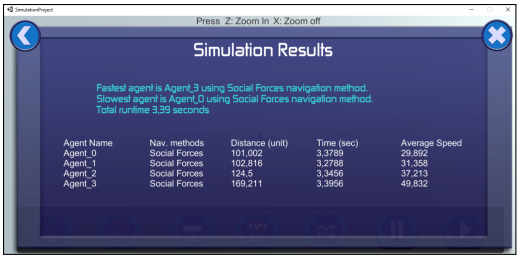
(a) Simulación de 1 agente.



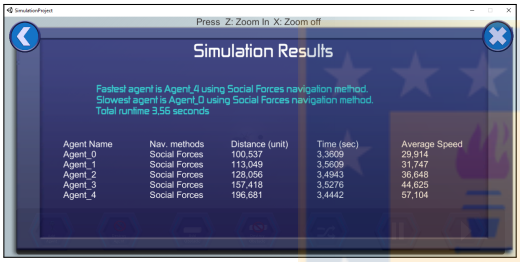
(b) Simulación de 2 agente.



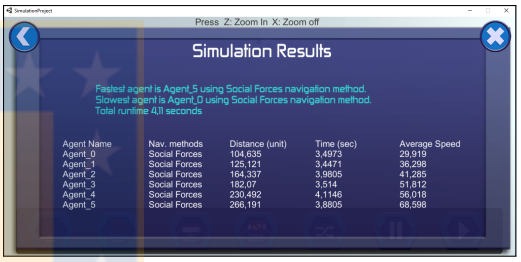
(c) Simulación de 3 agente.



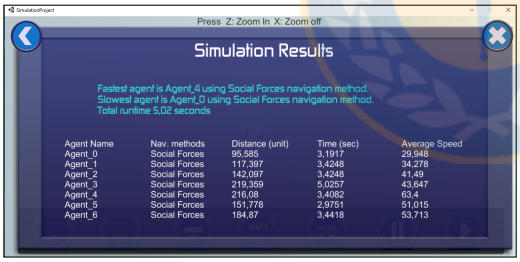
(d) Simulación de 4 agente.



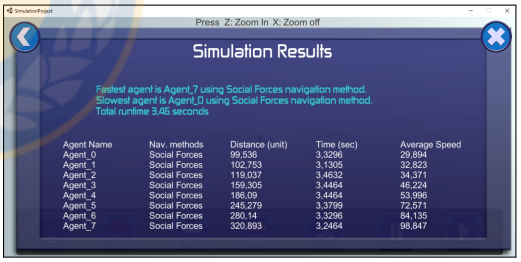
(e) Simulación de 5 agente.



(f) Simulación de 6 agente.

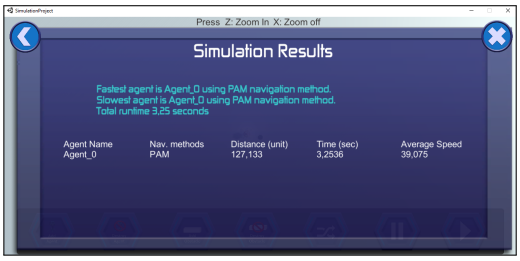


(g) Simulación de 7 agente.

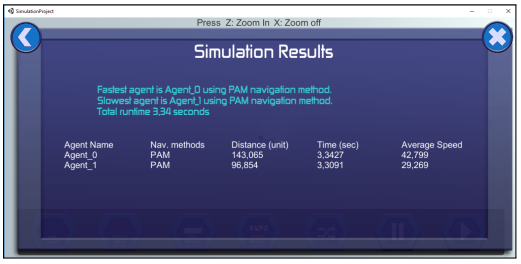


(h) Simulación de 8 agente.

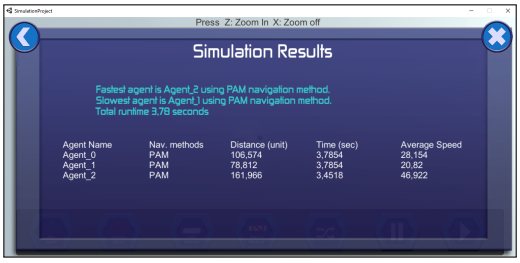
Figura 34: Escalabilidad de método Social Forces.



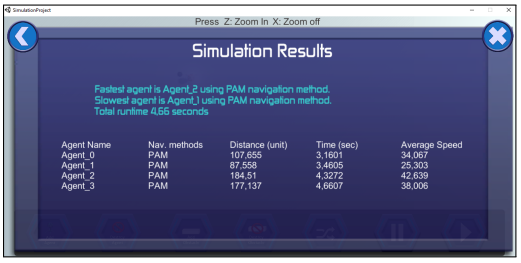
(a) Simulación de 1 agente.



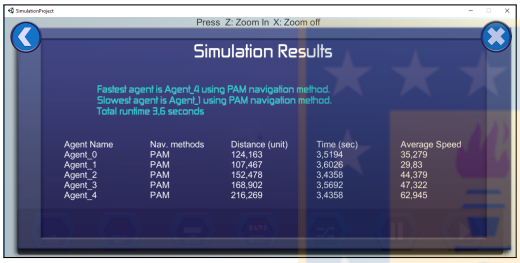
(b) Simulación de 2 agente.



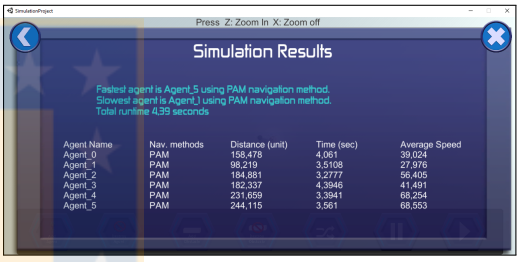
(c) Simulación de 3 agente.



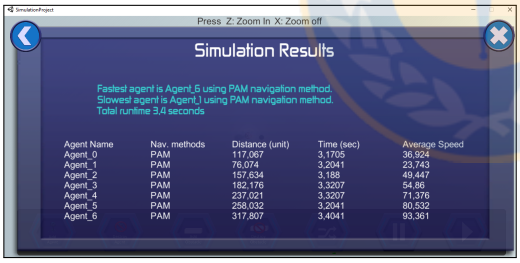
(d) Simulación de 4 agente.



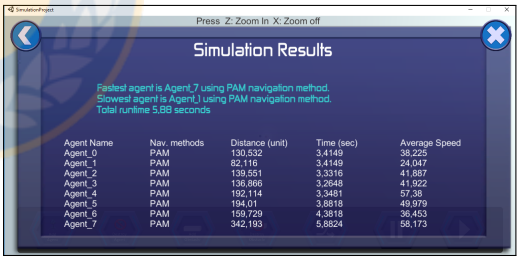
(e) Simulación de 5 agente.



(f) Simulación de 6 agente.



(g) Simulación de 7 agente.



(h) Simulación de 8 agente.

Figura 35: Escalabilidad de método PAM.

7.2. Script desarrollados en C# para ejecución en Unity

En esta sección se muestra el código correspondiente a los script utilizados en Unity.

7.2.1. GeneralAgentMove.cs

```
using UnityEngine;
using UnityEngine.UI;
using System.Runtime.InteropServices;
using System;

/*      Estructura que permite la compatibilidad con la estructura
        de puntos de C++*/
[StructLayout(LayoutKind.Sequential, Pack = 1)]
struct Point
{
    public float x;
    public float y;

    public Point(float ejex, float ejey)
    {
        x = ejex;
        y = ejey;
    }
}

public class GeneralAgentMove : MonoBehaviour
{
    [DllImport("MetodosMT.dll")]
    static extern IntPtr CreateRVOSimulator();
    [DllImport("MetodosMT.dll")]
    static extern void setupScenario(IntPtr sim, Point[]
        agent, Point [] obstacle, Point[] target);
    [DllImport("MetodosMT.dll")]
    static extern Point move(IntPtr sim, int agent, int
        method);
    [DllImport("MetodosMT.dll")]
    static extern bool reachedGoalUnit(IntPtr sim, int
        agent, Point target);
    [DllImport("MetodosMT.dll")]
    static extern float GetTimeStep(IntPtr sim);


    private Point[] Agent;
```

```
private Point[] Target;
private Point[] VerticeObstacle;

public int count_agent;
private int count_target;
private int count_obs;
private IntPtr sim;

private Point[] LastPosition;
private Point[] InitialPosition;

private Point TargetPosition;
private Boolean onPlay = false;
private int AgentInTarget = 0;

private Boolean[] InTarget;
private float[] DeltaTime;
private float[] DistanceToTarget;
private float[] AverageSpeed;
private string[] AgentName;
private string[] MethodsAgent;

public Text stdName;
public Text stdMethod;
public Text stdDistance;
public Text stdTime;
public Text stdVelocity;
public Text stdGeneral;
public GameObject panel;

private int EndSimulationinFrame;
private float TotalRuntime;

//Función aplicada al iniciar la escena.
private void Start()
{
    panel.SetActive(false);
}

/* Función aplicada al hacer click en el boton PLAY
   Da inicio a la simulación y crea el escenario en
   base a los elementos establecidos en la escena.*/
public void Play()
{
    count_agent = GameObject.FindGameObjectsWithTag("Agent").
        Length;
```

```
        count_target = GameObject.FindGameObjectsWithTag("Target").
            Length;
        count_obs = GameObject.FindGameObjectsWithTag("Obstacle").
            Length;

        Agent = new Point[count_agent];
        Target = new Point[count_target];
        VerticeObstacle = new Point[count_obs * 4];

        InitialPosition = new Point[count_agent];
        LastPosition = new Point[count_agent];

        DeltaTime = new float[count_agent];
        DistanceToTarget = new float[count_agent];
        AgentName = new string[count_agent];
        InTarget = new Boolean[count_agent];
        MethodsAgent = new string[count_agent];
        AverageSpeed = new float[count_agent];

        sim = CreateRVOSimulator();

        Target = GetTargetInScene();
        VerticeObstacle = GetObstacleInScene();
        Agent = GetAgentInScene();

        EndSimulationinFrame = 0;
        TotalRuntime = Time.time;

        setupScenario(sim, Agent, VerticeObstacle, Target);
        onPlay = true;
    }

    /* Da la opción de pausar la simulación
        eliminando los elementos de los arreglos de agentes
        obstáculos y objetivos*/
    public void Stop()
    {
        if(onPlay==true){
            panel.SetActive(false);
            onPlay = false;
            Array.Clear(Agent,0,count_agent-1);
            Array.Clear(Target, 0, count_target - 1);
            Array.Clear(VerticeObstacle, 0, count_obs*4 - 1);
        }
    }
```

```
/* Calcula la estadística grupal, obtiene el agente más
rápido y el más lento y el tiempo total de la ejecución*/
private void GroupStatistics()
{
    float MaxSpeed = 0;
    int FastestAgent = 0;
    float MinSpeed = 9999;
    int SlowestAgent = 0;


    for (int i=0; i<count_agent; i++)
    {
        if (AverageSpeed[i] > MaxSpeed)
        {
            MaxSpeed = AverageSpeed[i];
            FastestAgent = i;
        }
        if (AverageSpeed[i] < MinSpeed)
        {
            MinSpeed = AverageSpeed[i];
            SlowestAgent = i;
        }
    }

    if (EndSimulationinFrame == 1)
    {
        TotalRuntime = Time.time - TotalRuntime;
    }

    stdGeneral.text = "Fastest agent is " +
            AgentName[FastestAgent].Remove(AgentName[FastestAgent].
            Length - 7) + " using " + MethodsAgent[FastestAgent] +
            " navigation method.\n";

    stdGeneral.text += "Slowest agent is " + AgentName[SlowestAgent
        + " using " + MethodsAgent[SlowestAgent] + " navigation me
    stdGeneral.text += "Total runtime " + System.Math.Round(TotalRu
}

/* Obtiene los datos de cada agente tabulandolo en un panel*/
public void Statistics()
{
    panel.SetActive(true);
    stdName.text = "Agent Name\n";
    stdMethod.text = "Nav. methods\n";
    stdDistance.text = "Distance (unit)\n";
```

```
        stdTime.text = "Time (sec)\n";
        stdVelocity.text = "Average Speed\n";

        for (int i=0; i<count_agent; i++)
        {
            string name = AgentName[i];
            name = name.Remove(name.Length - 7);
            stdName.text += name + "\n";
            stdMethod.text += MethodsAgent[i] + "\n";
            stdDistance.text += System.Math.Round(DistanceToTarget[i]
                , 3).ToString() + "\n";
            stdTime.text += System.Math.Round(DeltaTime[i], 4).
                ToString() + "\n";
            AverageSpeed[i] = DistanceToTarget[i] / DeltaTime[i];
            stdVelocity.text += System.Math.Round(AverageSpeed[i], 3)
                .ToString() + "\n";
        }
        GroupStatistics();
    }

    /* Si se ha apretado play, calcula la nueva posición
        de los agentes, la función se aplica en cada cuadro,
        por lo que en cada cuadro se calcula la nueva posción
        del agente.*/
    void Update()
    {
        if (onPlay == true)
        {
            int i = 0;
            foreach (GameObject go in GameObject.
                FindGameObjectsWithTag("Agent") as GameObject[])
            {
                float distance = Vector3.Distance(new
                        Vector3(LastPosition[i].x, 0,
                                LastPosition[i].y), new Vector3(
                                TargetPosition.x, 0, TargetPosition.y));

                TargetPosition = Target[i];
                int meth_go = go.GetComponent<DragObject>().method;
                Point point = new Point(go.transform.position.x,
                        go.transform.position.z);

                if (!reachedGoalUnit(sim, i, TargetPosition))
                {
                    Point NewPoisition = new Point();
                    NewPoisition = move(sim, i,meth_go);
```

```
                    DistanceToTarget[i] += Vector3.Distance(
                        new Vector3(NewPoisition.x, 0,
                                NewPoisition.y),
                        new Vector3(LastPosition[i].x, 0,
                                LastPosition[i].y));
                    LastPosition[i] = NewPoisition;
                    Vector3 new_position = new Vector3(
                        NewPoisition.x, 0, NewPoisition.y);
                    Quaternion new_rotation = new Quaternion(0, 0, 0,
                     0);
                    go.transform.SetPositionAndRotation(new_position,
                     new_rotation);
                    DeltaTime[i] += Time.deltaTime;
                    InTarget[i] = false;
                }
                else
                {
                    Vector3 new_position = new Vector3(
                        LastPosition[i].x, 0, LastPosition[i].y);
                    Quaternion new_rotation = new Quaternion(0, 0, 0,
                     0);
                    go.transform.SetPositionAndRotation(new_position,
                     new_rotation);
                    if (InTarget[i] == false)
                    {
                        AgentInTarget++;
                        InTarget[i] = true;
                    }
                }
                i++;
                if (AgentInTarget == count_agent)
                {
                    EndSimulationinFrame++;
                    Statistics();
                }
            }
        }
    }

    /* Obtiene todos los agentes ubicados en la escena, para llevar
        un registro de ellos.*/
    Point[] GetAgentInScene()
    {
        Point[] aux = new Point[count_agent];
        int i = 0;
        foreach (GameObject go in GameObject.FindGameObjectsWithTag(
```

```
            "Agent") as GameObject[])
        {
            Point point = new Point(go.transform.position.x, go.
                transform.position.z);
            InitialPosition[i] = point;
            AgentName[i] = go.name;
            InTarget[i] = false;
            aux[i] = point;

            int meth_go = go.GetComponent<DragObject>().method;
            if (meth_go == 0)
            {
                MethodsAgent[i] = "ORCA";
            }
            else if (meth_go == 1)
            {
                MethodsAgent[i] = "Social_Forces";
            }
            else if (meth_go == 2)
            {
                MethodsAgent[i] = "PAM";
            }
            else
            {
                MethodsAgent[i] = "ORCA";
            }
            i++;
        }
        return aux;
    }

    /* Obtiene todos los objetivos ubicados en la escena, para
        llevar un registro de ellos.*/
    Point[] GetTargetInScene()
    {
        Point[] Tar_aux = new Point[count_target];
        int j = 0;
        foreach (GameObject tar in GameObject.FindGameObjectsWithTag(
            "Target") as GameObject[])
        {
            Point point = new Point(tar.transform.position.x, tar.
                transform.position.z);
            Tar_aux[j] = point;
            j++;
        }
        return Tar_aux;
```

```
    }

    /* Obtiene todos los obstáculos ubicados en la escena, para
       llevar un registro de ellos.*/
    Point[] GetObstacleInScene()
    {
        Point[] VertObs_aux = new Point[count_obs*4];
        int k = 0;
        foreach (GameObject obs in GameObject.FindGameObjectsWithTag(
            "Obstacle") as GameObject[])
        {
            Vector3 position = obs.transform.position;
            Vector3 scale = obs.transform.localScale;

            Point aux1 = new Point(position.x - (scale.x / 2),
                position.z - (scale.z / 2));
            Point aux2 = new Point(position.x + (scale.x / 2),
                position.z - (scale.z / 2));
            Point aux3 = new Point(position.x + (scale.x / 2),
                position.z + (scale.z / 2));
            Point aux4 = new Point(position.x - (scale.x / 2),
                position.z + (scale.z / 2));

            VertObs_aux[k] = aux1;
            VertObs_aux[k + 1] = aux2;
            VertObs_aux[k + 2] = aux3;
            VertObs_aux[k + 3] = aux4;
            k = k + 4;
        }
        return VertObs_aux;
    }

}
```

7.2.2. moveCamera.cs

```
using UnityEngine;

public class moveCamera : MonoBehaviour
{
    float deltaMove = 30f;
    /* Se llama una vez por fotograma, permite el movimiento de la
       cámara se aplica un límite de movimiento de la cámara para
       que no salga de las fronteras del plano.*/
    void Update()
    {
```

```
        if (Camera.main.transform.position.x<240 && Input.GetKey(
         KeyCode.UpArrow))
         {
             transform.Translate(Vector3.up * deltaMove * Time.
                 deltaTime);
         }
        else if (Camera.main.transform.position.x > -240 && Input.
         GetKey(KeyCode.DownArrow))
         {
             transform.Translate(Vector3.down * deltaMove * Time.
                 deltaTime);
         }
         else if (Camera.main.transform.position.z < 240 &&
             Input.GetKey(KeyCode.LeftArrow))
         {
             transform.Translate(Vector3.left * deltaMove * Time.
                 deltaTime);
         }
         else if (Camera.main.transform.position.z > -240 && Input.
             GetKey(KeyCode.RightArrow))
         {
             transform.Translate(Vector3.right * deltaMove * Time.
                 deltaTime);
         }
         else if (Camera.main.transform.position.y> 50 && Input.GetKey
             (KeyCode.Z))
         {
             transform.Translate(Vector3.forward * deltaMove * Time.
                 deltaTime);
         }
         else if (Camera.main.transform.position.y < 200 && Input.
             GetKey(KeyCode.X))
         {
             transform.Translate(Vector3.back * deltaMove * Time.
                 deltaTime);
         }
    }
}
```

7.2.3. SelectScene.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class SelectScene : MonoBehaviour
{
```

```
    /*  Carga la escena para modelar escenarios*/
    public void CreateScene()
    {
        SceneManager.LoadScene("CreateScene");
    }

    /*  Carga la escena predefinida*/
    public void StaticScene()
    {
        SceneManager.LoadScene("StaticScene");
    }
    /*  Carga la escena de inicio*/
    public void HomeScene()
    {
        SceneManager.LoadScene("Home");
    }

    /*  Cierra la plataforma*/
    public void Exit()
    {
        Application.Quit();
    }
}
```

7.2.4. Button.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class Button : MonoBehaviour
{
        /*Inclusión de los modelos de agentes para identificar
        cada método de navegación con un color.*/
    public GameObject agentORCA;
    public GameObject agentSocialForce;
    public GameObject agentPAM;

    /*Inclusión de los modelos de objetivo y obstáculo
    para la ubicarlos en la simulación*/
    public GameObject target;
    public GameObject obstacle;
    int count_agent;

    /*Agrega agente indicando método de navegación entre
    los incluidos en la plataforma.*/
    public void AddObject(int method)
```

```
{
    //Method = 0 -> ORCA
    //Method = 1 -> Social Forces
    //Method = 2 -> PAM
    count_agent = GameObject.FindGameObjectsWithTag("Agent").
    Length;
    Vector3 pos = new Vector3(Camera.main.transform.position.x,
            0, Camera.main.transform.position.z);

    if (method == 0)
    {
        agentORCA.name = "Agent_" + count_agent;
        Instantiate(agentORCA, pos, Quaternion.identity);
    }
    else if(method == 1)
    {
        agentSocialForce.name = "Agent_" + count_agent;
        Instantiate(agentSocialForce, pos, Quaternion.identity);
    }
    else if(method == 2)
    {
        agentPAM.name = "Agent_" + count_agent;
        Instantiate(agentPAM, pos, Quaternion.identity);
    }
    else
    {
        agentORCA.name = "Agent_" + count_agent;
        Instantiate(agentORCA, pos, Quaternion.identity);
    }
    target.name = "Target" + count_agent;
    Instantiate(target, pos, Quaternion.identity);
}

/*Agrega el obstáculo en el centro de la escena
donde se encuentra la cámara.*/
public void AddObstacle()
{
    Instantiate(obstacle, new Vector3(Camera.main.transform.
            position.x, 0, Camera.main.transform.position.z),
            Quaternion.identity);
}

/*Cambia la posición de los agentes en escena
a posiciones aleatorias.*/
public void RandomScene()
{
```

```
    foreach (GameObject go in GameObject.FindGameObjectsWithTag(
     "Agent") as GameObject[])
     {
          Point point = new Point(go.transform.position.x, go.
              transform.position.z);
          float x_agent = Random.Range(-200, 200);
          float z_agent = Random.Range(-200, 200);
          Vector3 pos_agent = new Vector3(x_agent, 0, z_agent);
          Quaternion new_rotation = new Quaternion(0, 0, 0, 0);
          go.transform.SetPositionAndRotation(pos_agent,
              new_rotation);
     }
}


//Elimina todos los agentes y objetivos de la escena.
public void DestroyAgents()
{
     foreach (GameObject go in GameObject.FindGameObjectsWithTag(
             "Agent") as GameObject[])
     {
          Destroy(go);
     }
     foreach (GameObject tar in GameObject.FindGameObjectsWithTag(
             "Target") as GameObject[])
     {
          Destroy(tar);
     }
}

//Elimina todos los obstáculos de la escena.
public void DestroyObstacle()
{
     foreach (GameObject obs in GameObject.FindGameObjectsWithTag(
             "Obstacle") as GameObject[])
     {
          Destroy(obs);
     }
}

//Elimina todos los elementos de la escena.
public void DestroyAll()
{
     DestroyAgents();
     DestroyObstacle();
}
```

```
}
```

7.2.5. DragObject.cs

```
  using    System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class DragObject : MonoBehaviour
{
    public int method;
    private Vector3 mOffset;
    private float mZCoord;

    /*Funciones que dan la opciones de cliquear los objetos y
    eliminar en caso de hacer click y presionar suprimir al mismo
     tiempo.*/
    private void OnMouseDown()
    {
        mZCoord = Camera.main.WorldToScreenPoint(
            gameObject.transform.position).z;
        mOffset = gameObject.transform.position - GetMouseWorldPos();
        if (Input.GetKey(KeyCode.Delete))
        {
            Destroy(this.gameObject);
        }
    }

    /*  Obtiene la posición donde se hizo click con respecto a
        la escena.*/
    private Vector3 GetMouseWorldPos()
    {
        Vector3 mousePoint = Input.mousePosition;
        mousePoint.z = mZCoord;
        return Camera.main.ScreenToWorldPoint(mousePoint);
    }

    /*  Permite arrastrar y mover los elementos cliqueados
        (aplicado a agentes, obstáculos y objetivos)*/
    private void OnMouseDrag()
    {
        transform.position = GetMouseWorldPos() + mOffset;
    }


}
```