UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Integración de Godot y SPADE para la construcción de entornos 3D para la navegación de robots

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Blasco Sevilla, Eduardo

Tutor/a: Rebollo Pedruelo, Miguel

Cotutor/a: Carrascosa Casamayor, Carlos

Director/a Experimental: Enguix Andrés, Francisco

CURSO ACADÉMICO: 2023/2024

Resumen

Uno de los aspectos más importantes de la programación de robots móviles es disponer de simuladores realistas que permitan probar algoritmos y estrategias de manera segura para los robots. El principal problema de las herramientas actuales es que solo funcionan para un conjunto limitado de robots y, en general, no escalan bien. La aplicación FIVE, creada en el ámbito de una línea de investigación del VRAIN, permite la generación de simulaciones complejas, usando las capacidades del motor de videojuegos Unity para solventar estos problemas. Cambios recientes en la política de licencias de Unity han hecho que pequeños estudios y desarrolladores independientes se cuestionen su uso, planteando Godot como alternativa para estos casos. El presente proyecto propone el diseño de una plataforma que permita integrar SPADE y Godot para construir simulaciones equivalentes a las de la aplicación FIVE.

Palabras clave: Entornos 3D, Simulación, Sistemas multi-agente, FIVE, Godot

Abstract

One of the main aspects of mobile robot programming is being able to use realistic simulations to evaluate algorithm and strategies in a safe way. The main issue with the current tools is that they only work for a limited set of robots and tend to scale in an inefficient way. The application FIVE, made in the context of an investigation line by VRAIN, allows the generation of complex simulations, using the capabilities of the videogame engine Unity to solve these problems. A recent change in the license model of Unity has led to many small studios and independent developers questioning its usage and proposing Godot as an alternative for this kind of cases. This project poses the design and implementation of a platform that allows to integrate SPADE and Godot to create complex simulations, equivalent to those generated by FIVE.

Keywords: 3D environment, Simulations, Multi-agent systems, FIVE, Godot.

A mi madre, por todo lo que me dio y no le pude devolver.

Índice general

Índice de imágenes

Índice de tablas

Capítulo 1 - Introducción

El presente trabajo consiste en el diseño e implementación de un sistema denominado *Flexible Intelligent Virtual Environment in Godot* (abreviado como FIVE-Godot), cuyo objetivo principal es permitir la simulación gráfica de sistemas multi-agente en un entorno virtual como paso previo a la experimentación con agentes en entornos reales. Este simulador pretende servir como alternativa a la plataforma *Flexible Intelligent Virtual Environment* (Javier Enguix, 2022), abreviada como FIVE. Esta aplicación, desarrollada en Unity, se utiliza actualmente en una línea de investigación activa del *Valencian Research Institute for Artificial Intelligence*, o VRAIN.

En este trabajo empezaremos analizando el simulador FIVE, su funcionamiento e identificando los requisitos que deberá cumplir FIVE-Godot. A continuación, definiremos y detallaremos la arquitectura de FIVE-Godot, y los distintos elementos que lo forman. Finalmente, haremos una comparación entre FIVE y FIVE-Godot, analizando como ambas herramientas ejecutan escenarios equivalentes, así como los requisitos de memoria y procesador.

1.1 Motivación

Como se ha mencionado en el anterior apartado, FIVE es una herramienta actualmente utilizada por una línea de investigación del VRAIN. Esta línea está relacionada con un proyecto a nivel nacional concedido por el Ministerio de Ciencia e Innovación, titulado Servicios Inteligentes Coordinados para Áreas Inteligentes Adaptativa[1] (*COSASS – Coordinated intelligent Services for Adaptative Smart Areas)*. Este proyecto estudia el uso de agentes inteligentes para ayudar al desarrollo de áreas rurales en España. Más concretamente, este proyecto investiga el uso de agentes inteligentes para automatizar el mantenimiento de áreas agrícolas, tales como plantaciones agrarias o vinícolas.

Dentro de este proyecto, FIVE se utiliza como herramienta para generar versiones virtuales de campos y plantaciones, en las que se simula el despliegue de maquinaria agrícola autónoma. Esta herramienta, creada con Unity por el estudiante de la UPV Francisco Enguix Andrés, ofrece a los investigadores un entorno en el que estudiar algoritmos de organización de agentes, permitiendo generar distintas simulaciones o modificar parámetros de la simulación de forma rápida y sencilla. Aunque esta aplicación se ha utilizado con éxito desde su creación en 2022, cambios introducidos en la política de precios de Unity en 2023 han llevado a los investigadores del VRAIN a buscar una aplicación alternativa a FIVE que no utilice Unity.

[1] https://cosass.usal.es

1.2 Objetivos

El objetivo principal de este proyecto es la creación de un marco de trabajo que permita crear simulaciones tridimensionales, poblarlas con agentes inteligentes y analizar distintos algoritmos de sistemas multi-agentes. Este marco de trabajo debe funcionar de la manera más similar posible a FIVE, siendo capaz de servir como sustituto en caso de que dicha aplicación deje de estar disponible. A partir de este objetivo principal, podemos identificar una serie de objetivos:

- Analizar la herramienta FIVE, entender su funcionamiento y extraer las características que debe cumplir FIVE-Godot para poder ser una alternativa viable.
- Estudiar distintas tecnologías para el desarrollo de la aplicación y decidir cuál es la más adecuada para la implementación de FIVE-Godot.
- Implementar un simulador que permita ejecutar simulaciones de manera análoga a FIVE.

Dado que FIVE es un proyecto que se ha ampliado desde su creación en 2022, podemos dividir este último objetivo en un objetivo principal y un objetivo secundario. El objetivo principal será crear un simulador que sea ofrezca las mismas funcionalidades que las que ofrecía FIVE cuando se creó. El objetivo secundario consistirá en, si es posible, añadir la funcionalidad que se ha ido a añadiendo a FIVE en sus distintas ampliaciones.

1.3 Estructura de la memoria

El primer capítulo de la memoria es una **introducción al proyecto**, dando una descripción general del trabajo y la herramienta a desarrollar. También se cubre la motivación que ha llevado a la creación de FIVE-Godot, así como los objetivos que pretendemos cumplir con esta aplicación.

El segundo capítulo consiste en la **definición** de los conceptos necesarios para entender el marco teórico de este proyecto. En este capítulo también analizaremos distintos simuladores de sistemas multi-agentes ya existentes, así como otras tecnologías relacionadas con el trabajo desarrollado. Finalmente, hablaremos de la motivación detrás de las tecnologías elegidas para desarrollar este proyecto.

El tercer capítulo engloba el **análisis de FIVE**. En primer lugar, analizaremos la herramienta FIVE, su funcionamiento y características. A continuación, veremos cómo se configuran las simulaciones. Finalmente, extraeremos los requisitos que deberá cumplir FIVE-Godot para poder considerarse una aplicación alternativa a FIVE.

El cuarto capítulo muestra el **diseño de la solución**, en el que se detallará la arquitectura de FIVE-Godot. Primero veremos, a nivel global, las distintas partes que componen la aplicación. Seguidamente, se hará una exposición de cada módulo que forma la solución de manera más profunda y técnica, mostrando las relaciones entre los elementos mediante diagramas de clases y figuras. Finalmente, se hará una explicación de cómo crear nuevos agentes y elementos de simulación.

El quinto capítulo cubrirá el **desarrollo de la solución**, en el hablaremos de las distintas decisiones que se tomaron durante el desarrollo de la aplicación y finalizaremos analizando los distintos problemas que han surgido durante el progreso del proyecto, así como su resolución.

El sexto capitulo analiza el **rendimiento y los casos de prueba**, donde se usará el simulador para construir un entorno poblado por agentes. Se analizarán el consumo de recursos de la simulación, así como los tiempos de carga. Finalmente, se ejecutará dos escenarios en FIVE y FIVE-Godot, analizando y comparando la ejecución en ambos sistemas.

En el capítulo de **conclusiones** discutiremos si los objetivos que habíamos planteado inicialmente se han alcanzado y tratado correctamente. Cerraremos el trabajo con la sección de trabajo futuro, donde detallaremos algunas ampliaciones y mejoras que se pueden desarrollar para mejorar FIVE-Godot.

Capítulo 2 - Estado del arte

2.1 Introducción

En este capítulo definiremos los conceptos teóricos que son utilizados en este trabajo. También analizaremos otros simuladores multi-agente ya existentes, así como otras tecnologías relacionadas con el proyecto. Seguidamente, veremos las tecnologías elegidas para el trabajo desarrollado y explicaremos por qué se han elegido.

2.2 Definiciones

Un **agente inteligente** (Stuart J. Russell, 1995) es un elemento *software* o *hardware* capaz de funcionar de manera autónoma en entornos cambiantes e impredecibles. Los agentes inteligentes utilizan sensores para percibir su entorno y, usando esta información, son capaces de decidir qué acciones tomar, siendo capaces de alterar dicho entorno mediante actuadores

Un **sistema multi-agente** (Yoav Shoham, 2009) consiste en un grupo de agentes inteligentes interactuando en un mismo entorno de manera descentralizada. Estos agentes son independientes entre sí, normalmente en entornos demasiado complejos como para que los agentes tengan una percepción global. Los agentes son capaces de comunicarse unos con otros, compartiendo información, organizándose, negociando y coordinando actividades. Esto permite a los sistemas multi-agente desarrollar comportamientos de organización complejos, incluso cuando los agentes individuales presentan comportamientos simples. Estos sistemas ofrecen ventajas tales como la eficiencia, ya que tienden a encontrar la solución óptima a un problema sin intervención externa; o la resistencia a fallos, al estar compuesto de agentes fácilmente replicables y reemplazables.

Un **motor de videojuegos** es un entorno de desarrollo que ofrece herramientas para la creación de videojuegos. Estos motores suelen incluir funcionalidad para renderizar gráficos en 2D y 3D, simular físicas y colisiones, así como animaciones, inteligencia artificial, gestión de memoria y muchos otros sistemas usados en el desarrollo de videojuegos. Aunque tradicionalmente se han usado para el desarrollo de videojuegos, en los últimos años estos motores han empezado a usarse para otras funciones, tales como la generación de gráficos CGI para películas y series, diseño arquitectónico o simulaciones de automóviles

2.3 Simuladores de sistemas multi-agente

En la actualidad, existen diversas herramientas capaces de simular sistemas multi-agente. En este apartado analizaremos 3 simuladores y veremos las características que ofrecen.

Netlogo[2] es un entorno de modelado de sistemas multi-agente desarrollado por Uri Wilensky en 1999 y desarrollado por la Universidad de Northwestern. Esta aplicación permite tanto la simulación de sistemas multi-agente como del entorno que dichos agentes habitan. Sin embargo, las capacidades de simulación de esta aplicación se limitan a entornos en dos dimensiones y las capacidades de *debugging* son muy limitadas, siendo imposible usar *breakpoints*. Finalmente, Netlogo no es capaz de paralelizar código o usar hilos de ejecución, por lo que simulaciones con muchos agentes o comportamientos complejos pueden afectar gravemente al rendimiento de la simulación.

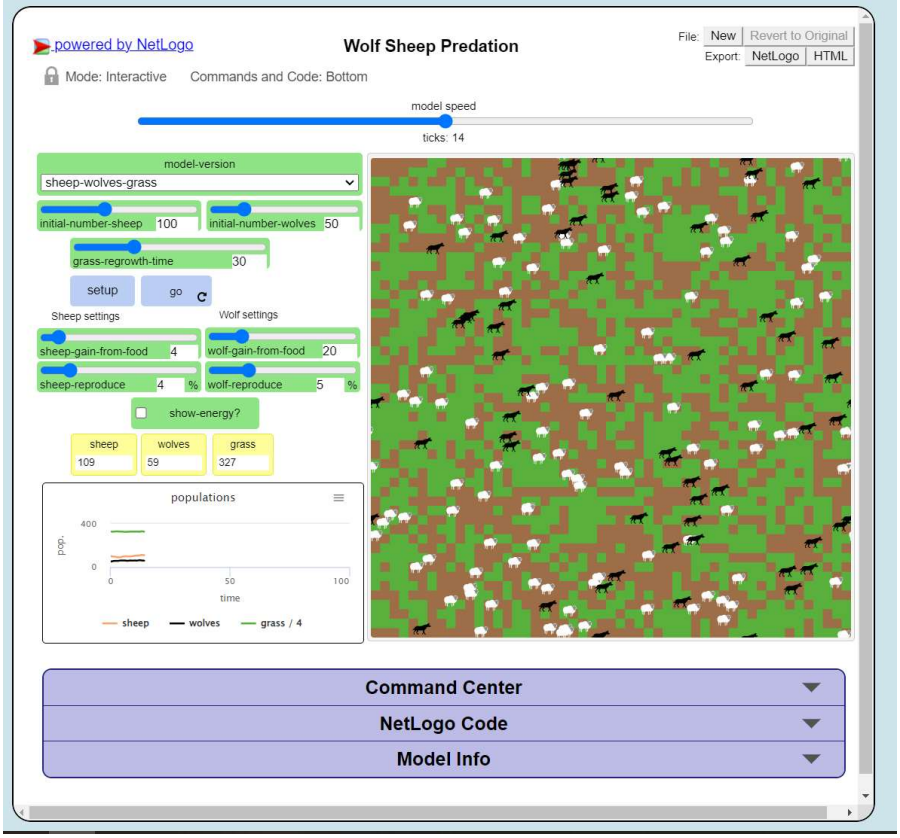

Imagen 1: Ejemplo de simulación en Netlogo

La imagen 1 muestra un modelo estabilidad depredador-presa basados en lobos, ovejas y hierba simulado en Netlogo.

GAMA Platform[3] es una plataforma de modelado y simulación de agentes capaz de simular agentes en un espacio tridimensional desarrollada por el Instituto de Investigación para el Desarrollo de Francia (Institut de Recherche pour le Development) en 2009. Aunque los agentes simulados existen en un espacio tridimensional, la capacidad de definir y mostrar este entorno es bastante limitada, siendo muy difícil crear

[2] https://ccl.northwestern.edu/netlogo

[3] https://gama-platform.org

un entorno tridimensional complejo y requiriendo mucho trabajo para aplicar cambios en dicho entorno.

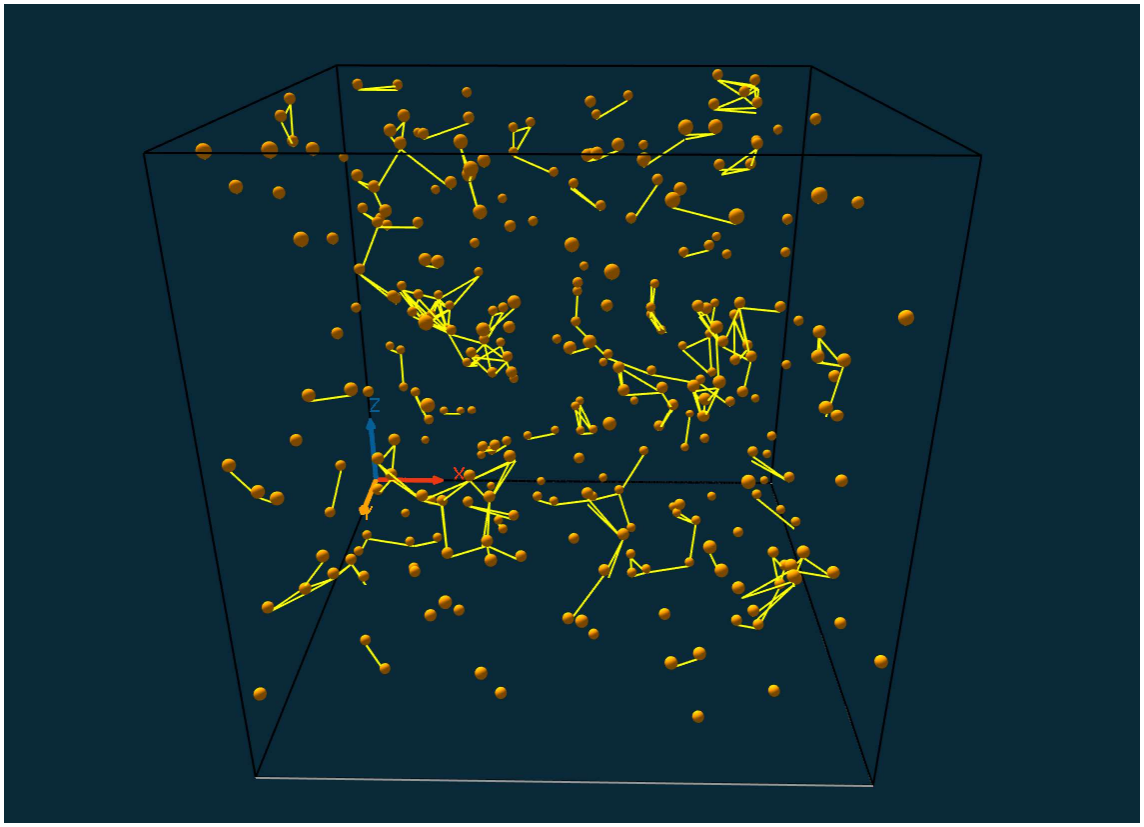

Imagen 2: Simulación de células en movimiento con GAMA Platform

En la imagen 2 podemos ver una simulación de células en GAMA Platform. En esta simulación, las células se mueven de manera aleatoria por el espacio, dibujando líneas que las unen con otras células que se encuentran a una distancia específica de ellas.

AnyLogic es un entorno de simulación de modelos desarrollado por The AnyLogic Company, inicialmente lanzada en 2000. Esta herramienta es posiblemente la herramienta más extensa actualmente disponible, siendo capaz no solo de simular sistemas multi-agente, si no también otro tipo de modelados, como dinámica de sistemas o sistemas de eventos discretos. AnyLogic ofrece simulación de entornos tridimensionales, y es utilizado por múltiples empresas como McDonalds, Siemens o Meta para investigar distintos tipos de simulaciones, tales como cadenas de suministros o gemelos digitales[4]. Lamentablemente, AnyLogic no ofrece versiones gratuitas para la investigación.

[4] https://www.anylogic.com/resources/case-studies

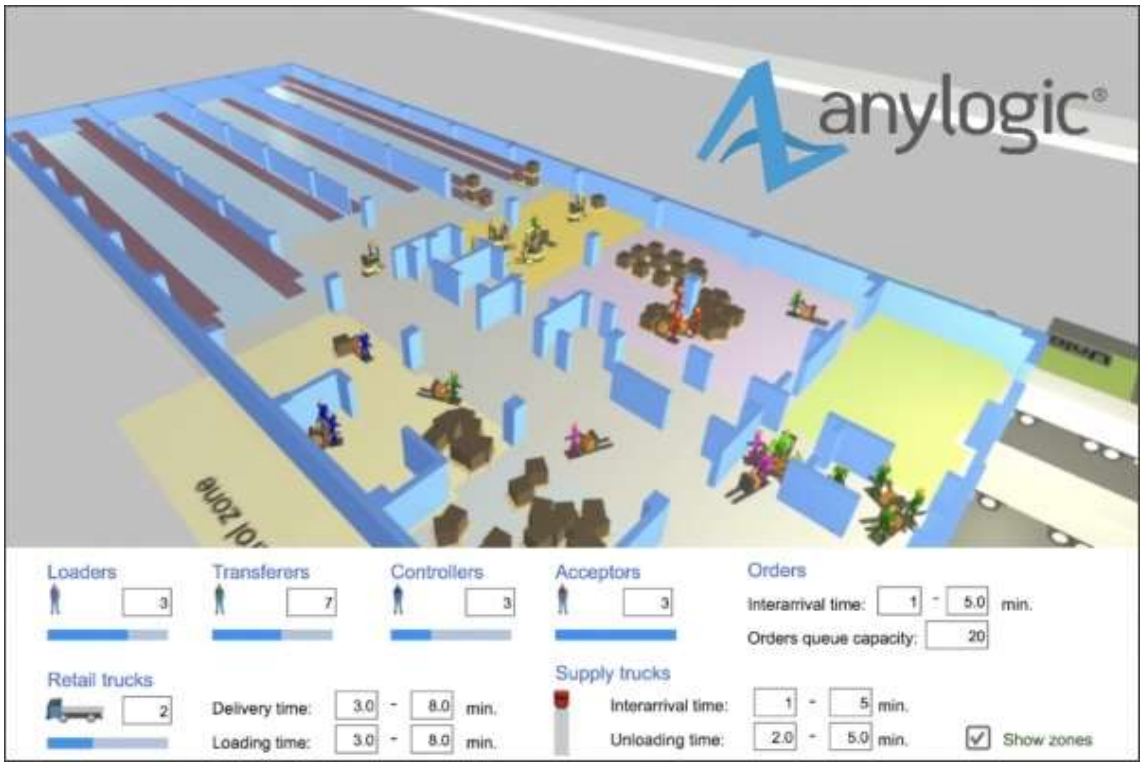

Imagen 3: Simulación de almacén en AnyLogic

En la imagen 3 podemos ver un ejemplo de AnyLogic en el que se usa un sistema multi-agente para simular el funcionamiento de un almacén.

Como podemos ver, en la mayoría de los casos los simuladores solo son capaces de generar entornos bidimensionales, lo cual limita mucho su utilidad para probar algoritmos que después serán desplegados en el mundo real. Por otro lado, los simuladores que si permiten generar entornos tridimensionales tienen capacidades muy limitadas, o en el caso de AnyLogic, son aplicaciones de pago, lo que dificulta su uso para tareas de investigación. Muchos de estos simuladores utilizan además lenguajes propietarios, lo que añade una capa extra de dificultad a la hora de utilizarlos.

2.4 Motores de videojuegos

Actualmente, existe una gran cantidad de motores de videojuegos. En este apartado analizaremos los tres más utilizados en la actualidad.

Unreal Engine[5] es un motor de videojuegos desarrollado por Epic Games. Fue presentado al público por primera vez en 1998, con el lanzamiento del juego *Unreal*. Unreal Engine se distribuye como código abierto, y ofrece licencias gratuitas, siempre y cuando las aplicaciones creadas con dichas licencias generen ingresos anuales inferiores a un millón de dólares, momento a partir del se empiezan a cobrar regalías.

[5] https://www.unrealengine.com

Ejemplos de juegos hechos con Unreal Engine incluyen *Fortnite* (Epic Games, 2017) o *Final Fantasy 7 Remake* (Square Enix, 2020).

Con la presentación de la última versión de Unreal Engine, Unreal Engine 5, en abril de 2022, se distribuyó *The Matrix Awakens*, una demostración técnica en la que cientos de coches y viandantes se movían por una ciudad de aproximadamente 16 kilómetros cuadrados, tal y como muestra la imagen 4.

Imagen 4: Fotograma de The Matrix Awakens

Aunque es un motor con grandes capacidades, es muy complejo de usar, ya que tiene una capa de abstracción sobre C++ que dificulta su aprendizaje y su documentación oficial es muy poco explicativa, siendo en la mayoría de los casos la firma de la función y el fichero de cabecera en el que reside.

Unity[6] es un motor de videojuegos creado por Unity Technologies y estrenado en junio de 2005. Es un software cerrado con distintas licencias, y permite al usuario trabajar con C#. Algunos ejemplos de juegos desarrollados con Unity son Hearthstone (Blizzard Entertainment, 2014) o Genshin Impact (miHoyo, 2020).

En 2023, Unity Technologies anunció un cambio de la política de precios. Este cambio iba a permitir a Unity cobrar una cuota por cada instalación de programas hechos con Unity, aplicándose retroactivamente a juegos hechos en Unity ya comercializados. Esto provocó que cientos de desarrolladores anunciaran que iba a dejar de usar Unity y moverse a otro motor de videojuegos, lo que acabó llevando a la rectificación por parte de Unity Technologies, que decidieron cambiar a un sistema de regalías similar al que usa Unreal Engine. Aun así, este movimiento provocó una pérdida de confianza en Unity, y es uno de los motivos que llevan al desarrollo de este trabajo.

[6] https://www.unity.com

Godot[7] es un motor gratuito, creado por Juan Linietsky y Ariel Manzur y lanzado al público en enero de 2014 bajo la licencia MIT. Es un motor gratuito de código abierto desarrollado en C++ que permite la creación de aplicaciones usando el lenguaje C# o GDScript, un lenguaje propietario con sintaxis similar a Python. *Sonic Colors: Ultimate* (Blind Squirrel Entertainment, 2023) o *Cassette Beasts* (Bytten Studios, 2023) son ejemplos de juegos desarrollados usando Godot.

Godot destaca por su facilidad de uso y portabilidad. El motor en si puede descargarse y empezar a usarse en cuestión de segundos, no requiriendo ningún tipo de registro o trabajo extra. La documentación oficial es muy extensa, con muchos tutoriales y explicaciones de cómo usar las distintas funcionalidades del motor, y es capaz de crear aplicaciones para Windows, Linux o Mac sin ningún tipo de configuración extra.

2.5 Otras tecnologías relevantes

2.5.1 XMPP

XMPP[8] (*Extensible Messaging and Presence Protocol),* anteriormente Jabber, es un protocolo de mensajería en tiempo real basado en XML *(Extensibe Markup Language)* abierto y extensible que funciona a nivel de aplicación. Ofrece la posibilidad de aplicar capas de cifrado tales como TLS (*Transport Layer Security)* o SASL (*Simple Authentication and Security Layer)*. Existe una gran cantidad de servidores gratuitos disponibles, así como instrucciones para crear servidores XMPP privados.

2.5.2 SPADE

SPADE (Miguel Gregori, 2006) *(Smart Python multi-Agent Development Environment*) es una plataforma de desarrollo de sistemas multi-agente que utiliza XMPP como método de comunicación, teniendo cada agente SPADE un gestor de mensajes integrado. SPADE utiliza un modelo de agentes basado en comportamientos, en el que los comportamientos son tareas que pueden ejecutarse una vez; de manera periódica, o utilizar una máquina de estados finitos para crear comportamientos complejos. SPADE también permite el uso del modelo BDI (*Belief-Desire-Intention)*, el cual permite crear agentes con comportamientos reactivos.

2.5.3 ROS

ROS[9] (*Robot Operating System)* es un framework para el desarrollo de software para robots. Esta plataforma incluye funcionalidad como abstracción de hardware, múltiples algoritmos de localización y pathfinding, simulación de robots, comunicación

[7] https://godotengine.org

[8] https://xmpp.org

[9] https://www.ros.org

entre procesos y otras muchas capacidades. ROS se utiliza de forma extensa para la investigación y desarrollo de robots industriales y vehículos autónomos.

2.6 FIVE (Flexible Intelligent Virtual Environment)

Como hemos visto en el apartado 2.3, la mayoría de los simuladores de sistemas multi-agente tienen dificultades para simular entornos tridimensionales complejos de manera efectiva. **FIVE** utiliza las herramientas de Unity para crear entornos de simulación de sistemas multi-agente tridimensionales, ofreciendo la posibilidad de generar entornos de manera rápida y eficiente. Está construido de forma que el usuario apenas requiere conocimientos básicos de Unity para crear o alterar entornos, ya que la configuración de la simulación se realiza mediante ficheros de texto y ficheros JSON, ajenos a los tipos de fichero propios de Unity.

Aunque no permite la simulación directa de agentes inteligentes, FIVE permite la comunicación con agentes externos mediante el protocolo de mensajería instantánea XMPP. Esto permite poblar el entorno con agentes inteligentes capaces de comunicarse mediante este protocolo, como es el caso de SPADE. Esta capacidad de comunicación también permite trabajar con FIVE de manera descentralizada, siendo posible ejecutar FIVE y los distintos agentes inteligentes en diferentes máquinas, disminuyendo los requisitos de *hardware* y permitiendo la colaboración desde distintos lugares del mundo.

Desde su creación, FIVE ha recibido una serie de actualizaciones, aumentando sus funcionalidades. Actualmente, FIVE utiliza las herramientas de integración de Unity y ROS[10] para permitir la simulación de movimiento de robots mediante ROS, así como el uso de *artifacts*, elementos de la simulación que no son capaces de realizar acciones autónomas, pero ofrecen otras funcionalidades que los agentes pueden utilizar durante la ejecución de la simulación.

2.7 Solución propuesta: FIVE-Godot

La propuesta de este trabajo consiste en la elaboración de una herramienta gemela a FIVE, llamada **FIVE-Godot**. Esta herramienta permitirá la simulación de entornos tridimensionales para sistemas multi-agente ofreciendo un funcionamiento lo más parecido posible a FIVE. El usuario deberá ser capaz de usar los mismos archivos de configuración que usaría en FIVE para obtener un entorno de simulación equivalente al que se generaría en FIVE. También deberá poder poblar el entorno con agentes SPADE, y hacer que ejecuten su comportamiento e interactúen con el entorno de la misma forma que lo hacen en FIVE.

Esta nueva herramienta deberá también ser escalable y expandible, permitiendo añadir nuevas características y funcionalidades que se añadan con el tiempo a FIVE,

[10] https://github.com/Unity-Technologies/Unity-Robotics-Hub

tales como el uso de otros tipos de comunicaciones, nuevos tipos de agentes con características distintas o diferentes tipos de entornos.

Siguiendo la filosofía de desarrollo de FIVE, esta nueva aplicación se desarrollará usando un motor de videojuegos, para poder aprovechar las capacidades de *rendering,* simulación de físicas y *pathfinding* que estos ofrecen. Tras analizar los motores de videojuego disponibles y sus características, se usará el motor Godot, ya que ofrece una experiencia de uso mucho más sencilla, con documentación muy detallada y extensa. Además, el hecho de que este distribuido con licencia MIT nos asegura que no puede darse una situación similar a la de Unity mencionada en el apartado 2.4, en la que se ponga en riesgo la viabilidad de usar la aplicación en el futuro.

Finalmente, Godot permite el desarrollo de aplicaciones usando el lenguaje de programación C#, lenguaje que también utiliza Unity y en el que está programado FIVE. Que ambos simuladores utilicen el mismo lenguaje simplificará trasladar funcionalidades entre ambos y reducirá el conocimiento necesario para poder trabajar con las dos aplicaciones.

Capítulo 3 – Análisis de FIVE

3.1 Introducción

En este apartado, haremos un análisis de la herramienta FIVE. Veremos que componentes la forman, cómo funcionan y cómo interactúan entre sí. Una vez entendamos el funcionamiento y sus características, extraeremos los requisitos que FIVE-Godot deberá cumplir para poder funcionar como alternativa viable a FIVE.

3.2 Elementos de FIVE

FIVE se compone de tres elementos diferentes, representados en la imagen 5. El simulador, en el que se genera el entorno y se simulan las acciones de los agentes inteligentes; los agentes SPADE, que pueblan la simulación a través de avatares digitales; y el servidor XMPP necesario para la comunicación de los agentes, tanto entre sí como con el simulador. Estos elementos pueden convivir en la misma red o estar distribuidos en distintas redes alrededor del mundo, con los agentes y el simulador usando el servidor XMPP para mandarse información entre sí.

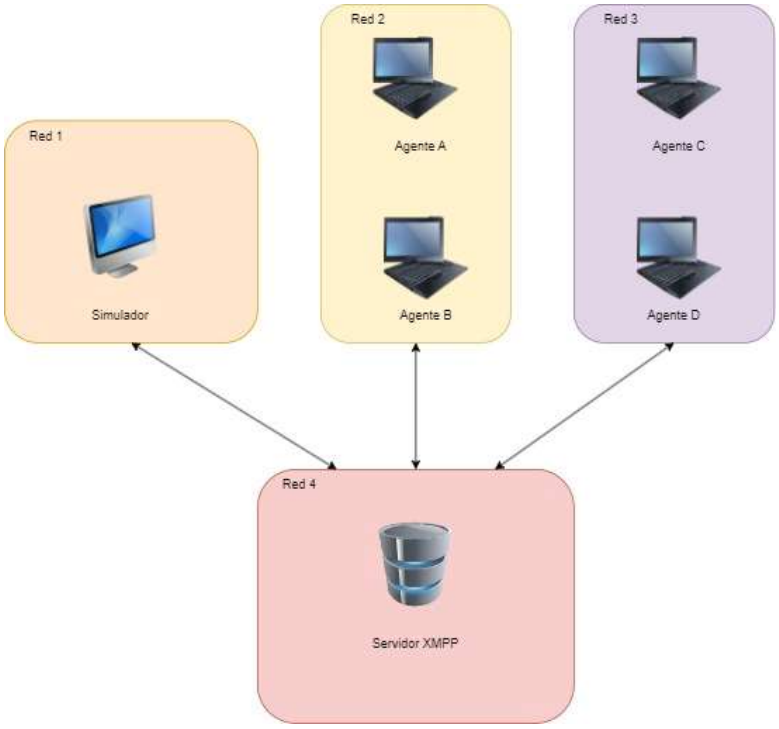

Imagen 5: Ejemplo de arquitectura FIVE desplegada

3.2.1 Agentes

Como hemos explicado en el apartado 2.6, los agentes que usan FIVE son agentes SPADE programados en Python, utilizando el protocolo XMPP para comunicarse entre sí, así como para enviar comandos al simulador, lo que les permite manipular sus avatares en la simulación. Estos agentes están diseñados con vistas a escalabilidad, pudiendo ejecutarse en hilos, proceso o incluso redes diferentes.

Los agentes interactúan con el simulador utilizando avatares de FIVE que disponen de cámaras, lo que les permite recibir imágenes con las que reconocer su entorno y decidir cómo actuar. Para ello, ejecutan una máquina de estados finitos con 4 posibles estados, tal y como muestra la imagen 6:

1. Estado de inicialización: En este estado se ejecutan los métodos de preparación oportunos para el agente. Esto incluye inicializar los parámetros que usará el agente, así como solicitar a FIVE que genere un avatar.
2. Estado de percepción: El agente procesa las imágenes que ha recibido del simulador
3. Estado de cognición: En este estado, el agente analiza la información que ha obtenido a partir de las imágenes analizadas en el estado de percepción y decide cómo actuar
4. Estado de acción: El agente, a través de XMPP, envía comandos a su avatar en el simulador. Estos comandos pueden incluir acciones tales como cambiar la configuración de la cámara del avatar, mover el avatar a una nueva posición o cambiar la frecuencia con la que el avatar manda imágenes al agente.

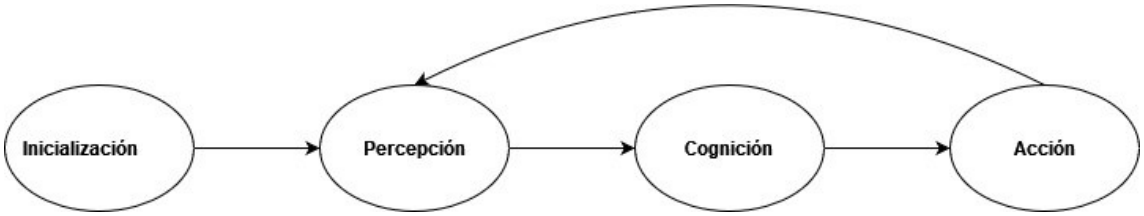

Imagen 6: Maquina de estados finitos del agente

3.2.2 Simulador

El simulador está desarrollado en Unity, y permite simular los entornos virtuales que después poblarán los agentes SPADE, como se muestra en la imagen 7. El simulador utiliza una serie de ficheros de texto para obtener los parámetros de la simulación, tales como el entorno a generar, los elementos que formarán el entorno inicial y sus posiciones o los distintos avatares que podrán usar los agentes una vez se inicie la simulación. Estos ficheros se analizarán más adelante.

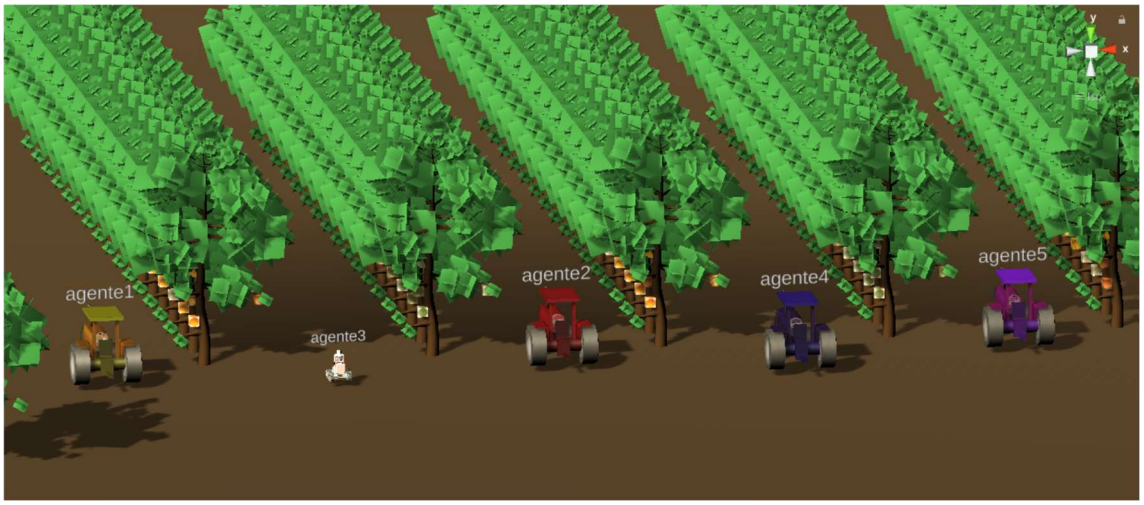

Imagen 7: Ejemplo de simulación en FIVE

Durante la simulación, el simulador recibirá comandos de los agentes con instrucciones tales como crear avatares nuevos para los agentes, mover dichos avatares o cambiar la configuración de sus cámaras. Estos comandos se envían por XMPP codificados en formato JSON[11] y contienen dos elementos:

- ***commandName:*** Contiene una cadena de texto con el nombre del comando que se va a ejecutar.
- ***data:*** Contiene un *array* con la información que el simulador necesita para poder ejecutar el comando. Esta información puede tener un número de elementos arbitrario, y cada dato puede tener un tipo distinto, siendo responsabilidad del simulador convertir cada elemento a su tipo correspondiente.

```
{
  "commandName": "move_agent",
  "data": ["agent1", "{10, 0, 10}"]
}
```

Imagen 8: Ejemplo de comando enviado por un agente

La imagen 8 muestra un ejemplo de comando. En este caso, el mensaje solicita que se ejecute el comando move_agent, que mueve un avatar dentro de la simulación. El campo de datos del comando contiene el nombre del agente a mover (*agent1)* y la posición a la que tiene que moverse (10,0,10).

Los comandos que pueden utilizar los agentes son los siguientes:

[11] https://www.json.org/json-en.html

- *create_agent:* Este comando solicita al simulador que instancie un nuevo avatar. Contiene el nombre que tendrá dicho avatar, el tipo de avatar y su localización inicial, así como si debe iniciarse con las colisiones activadas. Una vez creado, el simulador devolverá al agente un mensaje indicando la posición inicial del avatar.
- *move_agent:* Este comando sirve para mover el avatar del agente a una posición concreta, enviada como una lista de 3 *floats*, formando un vector (x, y, z). El simulador moverá al agente a esa posición, mandando al agente un mensaje con el vector de la posición final, ya sea la posición de destino o la posición en la que el avatar se haya quedado atascado.
- *fov_camera:* Este comando permite alterar el campo de visión de la cámara del agente. Dado que los agentes soportan múltiples cámaras, el comando incluye una lista de *floats*, con el primer float indicando el índice de la cámara que va a cambiar, y el segundo indicando el nuevo campo de visión.
- *move_camera:* Este comando, similar a *move_agent*, permite mover la cámara de un avatar dentro de su espacio local. Para ello, manda una lista con 4 *floats*, en el que el primero es el índice de la cámara y los tres siguientes forman un vector (x, y, z). Por ejemplo, un mensaje con el vector (0,1,0) moverá la posición de la cámara una unidad en el eje "y" local del agente.
- *rotate_camera:* Este comando gira la cámara del agente de forma global. Esto quiere decir que, sin importar la orientación de un avatar, la rotación final de la cámara será siempre la misma para un mensaje específico. El comando debe incluir una lista con tres *floats*, el primero indicando el índice de la cámara que se va a rotar; el segundo indicando el eje en el que la cámara rotará, siendo el 0 el eje "x", 1 el eje "y", y 2 el eje "z"; y el tercero indicando la nueva rotación en este eje, siendo 0 grados orientación norte, 90 grados este, 180 grados sur y 270 grados oeste.
- *take_image:* Este comando se utiliza para alterar el comportamiento de una cámara. El comando contiene dos *floats*, el primero indicando el índice de la cámara que va a cambiar su comportamiento y el segundo, al que llamaremos *behavior*, es un número indicando el nuevo comportamiento:

    - Si *behavior* es mayor a 0, la cámara empezará a tomar y mandar fotos al agente de forma periódica, enviando una nueva imagen cada *behavior* segundos. Si la cámara ya estaba aplicando este comportamiento, el tiempo entre imágenes se actualizará al nuevo valor.
    - Si *behavior* es igual a 0, la cámara enviará una imagen de manera única e instantánea.
    - Si *behavior* es menor que 0, la cámara dejará de enviar imágenes.
- *change_colour:* Este comando permite cambiar el color de un avatar. El comando incluye el nombre del agente a cambiar y 4 *floats* entre 0 y 1, que forman un color con el formato (r, g, b, a), representando respectivamente la cantidad de rojo, verde, azul y transparencia del nuevo color.

La lógica para activar la ejecución de los comandos está contenida en clases, teniendo cada comando su propia clase. Cuando el simulador recibe un comando, crea una instancia de la clase relevante, añade la información del comando y la introduce en una cola. Cuando el comando se ejecuta, la instancia recibe un diccionario con todos

los avatares disponibles, busca el avatar en el que tiene que actuar y le ordena que ejecute el comando, pasándole la información necesaria. Los avatares guardan la lógica para ejecutar todos los comandos en su clase base.

3.3 Archivos de configuración de FIVE

En este apartado, veremos en detalle los distintos ficheros de configuración que FIVE necesita para su correcto funcionamiento.

3.3.1 map.txt

Como hemos mencionado, el archivo map.txt contiene una representación en caracteres de los elementos que forman la simulación. Estos caracteres se separan entre si mediante espacios, y a la hora de generar la simulación, se leen de izquierda a derecha y de arriba a abajo. El fichero puede contener espacios extra entre elementos, aumentando la distancia entre un elemento y el siguiente en su fila; así como líneas vacías, que aumentarán la distancia entre las distintas filas de elementos. La imagen 9 incluye un ejemplo de este fichero.

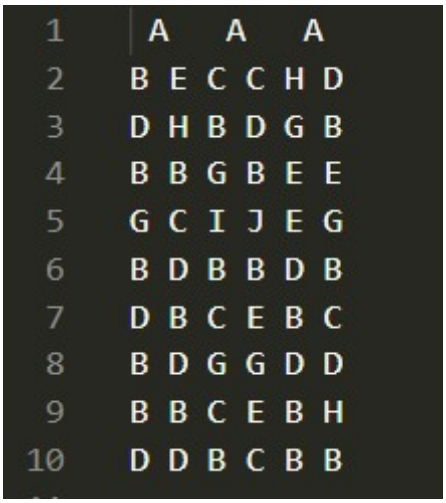

Imagen 9: Ejemplo de representación de mapa en el fichero map.txt

Este fichero también permite definir los lugares de instanciación de los avatares, llamados *spawners.* Los *spawners* se generan con nombres correlativos (*Spawner 1, Spawner 2…)* conforme se van instanciando. Cuando un agente solicita la instanciación de un avatar mediante el comando *create_agent*, puede elegir entre mandar un vector de posición o el nombre de un *spawner* como localización inicial del avatar.

3.3.2 map_config.json

Este archivo JSON contiene la relación entre los caracteres de map.txt y los objetos que representa, así como el origen de coordenadas de la generación del mapa y la distancia entre elementos. En la imagen 10, podemos ver un ejemplo de este fichero, formado por los siguientes elementos:

- *Origin:* Este vector tridimensional de *floats* (x,y,z) representa el punto inicial donde se empezarán a colocar elementos en la simulación.
- *Distance:* Vector bidimensional de *floats* (x, y). Indica la distancia de separación en cada eje que tiene que haber entre dos elementos contiguos en la simulación.
- *symbolToPrefabMap:* Lista de elementos donde cada elemento contiene los siguientes parámetros:
    - *symbol:* Carácter que representa un elemento en map.txt.
    - *prefabName:* Nombre interno del elemento asociado a dicho carácter.
    - *dataFolder* (opcional): ruta de carpeta. Algunos elementos de FIVE pueden aplicar las imágenes contenidas en la carpeta como texturas a distintos componentes del elemento de forma aleatoria.


```
{
    "origin": {
        "x": 0.0,
        "y": 0.0,
        "z": 15.0
    },
    "distance": {
        "x": 3,
        "y": 1.5
    },
    "symbolToPrefabMap": [
        {
            "symbol": "T",
            "prefabName": "Tree"
        },
        {
            "symbol": "B",
            "prefabName": "Tree Fruit Variant",
            "dataFolder": "../../release/windows-server/oranges/black spot"
        },
        {
            "symbol": "C",
            "prefabName": "Tree Fruit Variant",
            "dataFolder": "../../release/windows-server/oranges/canker"
        },
        {
            "symbol": "A",
            "prefabName": "Spawner"
        }
    ]
}
```


Imagen 10: Ejemplo de fichero map_config.json

3.3.3 map.json

Este fichero JSON contiene la lista de luces que contendrá el simulador, así como una lista de objetos que se deben instanciar en lugares específicos. Esto permite

añadir más flexibilidad a la simulación, pudiendo cambiar las condiciones ambientales y hacer cambios puntuales en los objetos de la simulación sin necesidad de generar nuevos ficheros de mapa o configuración. Los elementos que forman el objeto JSON son los siguientes:

- *Lights:* Lista de luces que existirán en la simulación. Cada una tiene los siguientes parámetros:
  - *active:* Booleano que indica si el objeto debe ser creado.
  - *objectName:* Nombre interno del objeto de luz que se instanciará.
  - *position:* Vector tridimensional de *floats* (x,y,z) que indica donde se posicionará la luz.
  - rotation: Vector tridimensional de *floats* con la rotación, en grados, que tendrá la luz.
  - *color:* Objeto que contiene la información del color que tendrá la luz. Contiene cuatro *floats r,g,b,a* con valores entre 0 y 1, que representan la cantidad de rojo, verde, azul y transparencia del color.
- *objects:* Lista de elementos a instanciar. Al igual que un elemento *light,* tienen las características de active, objectName, position y rotation, pero no la característica de color, al ser específica de los objetos de luz.

Debido a la longitud del fichero, la imagen con el ejemplo de este fichero se puede encontrar en el anexo 1 de este documento.

3.4 Requisitos de FIVE-Godot

Una vez vistos el funcionamiento de las distintas partes que conforman FIVE, es sencillo ver los requisitos que deberá de cumplir FIVE-Godot. Para poder ofrecer una alternativa a FIVE, FIVE-Godot debe de ser capaz de leer los distintos ficheros que definen una simulación en FIVE y utilizarlos para generar un entorno de simulación equivalente. Así mismo, deberá permitir que los agentes SPADE envíen, mediante protocolo XMPP, mensajes con comandos, y ejecutarlos en la simulación de manera correcta. En definitiva, queremos que un usuario pueda usar los mismos ficheros en FIVE y FIVE-Godot y obtener una simulación lo más similar posible.

Así mismo, dado que FIVE es una herramienta actualmente en uso y expansión, debemos asegurarnos de que FIVE-Godot se desarrolla de forma escalable, y que permita añadir nuevas funcionalidades y características a la par que FIVE.

Capítulo 4 – Diseño de la Solución

4.1 Introducción

En este capítulo se expondrá la arquitectura del simulador FIVE-Godot. En primer lugar, veremos los archivos que necesita FIVE-Godot para poder ejecutar una simulación. A continuación, veremos los distintos elementos que forman lo forman y sus responsabilidades. Tras esto, hablaremos de cómo se han diseñado los avatares que los agentes usarán para interactuar en el entorno. Finalmente, expondremos de manera más profunda la arquitectura del simulador, analizando las distintas clases que lo forman y cómo se relacionan, y explicaremos el proceso para nuevos avatares y elementos al simulador.

4.2 Archivos de configuración de FIVE-Godot

Ya hemos visto en el apartado 3.3 los archivos de configuración que utiliza FIVE para definir los parámetros de la simulación. FIVE-Godot utilizará estos mismos ficheros para generar la simulación. Además, FIVE-Godot necesita dos ficheros de configuración extra para su correcto funcionamiento.

4.2.1 folders_config.json

Como hemos visto en el apartado 3.2.2, *map_config.json* contiene una lista que permite asociar los caracteres de map.txt con la ruta de archivo en la que se encuentra el objeto a instanciar. Dado que esta ruta es exclusiva de Unity, tenemos que convertirla a una ruta propia de Godot. La información necesaria para ello está guardada en el fichero *folders_config.json*, mostrado en la imagen 11, que está formado por dos elementos:

- *UnityDataFolder: String* con la base de la ruta de los archivos en Unity.
- *GodotDataFolder: String* con la base de la ruta de archivos de Godot.

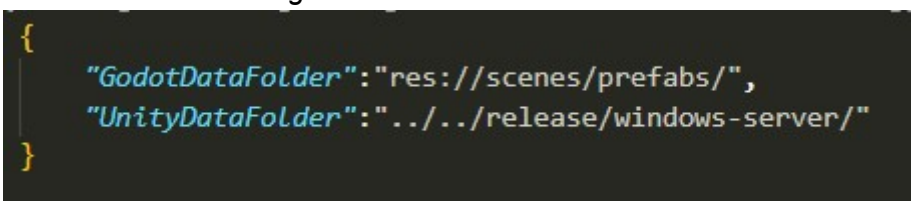

Imagen 11: Ejemplo de fichero folders_config.json

4.2.2 server_config.json

El fichero *server_config.json* contiene la información necesaria para que el simulador se pueda conectar al servidor XMPP. A diferencia de *folder_configs.json*, este fichero es opcional, y en caso de su ausencia, se usarán los valores por defecto para la conexión, especificados en el editor de Godot. Como se puede apreciar en la imagen 12, los elementos que lo forman son:

- *ServerName:* Nombre del servidor al que se conectará el simulador.
- *UserName:* Nombre de usuario con el que el simulador se conectará.
- *Password:* Contraseña con la que se intentará la conexión al servidor.

Imagen 12: Ejemplo de fichero server_config.json

4.3 Arquitectura del simulador

La simulación de FIVE-Godot está compuesto por 4 mánagers, que se coordinan entre sí para crear el entorno que después poblarán los agentes a través de sus avatares. En este apartado haremos una descripción general de las responsabilidades de cada uno y sus relaciones.

Los resultados de la ejecución de los distintos mánagers pueden estar en una de las siguientes categorías:

- **Resultado correcto**: El mánager ha ejecutado su funcionalidad de forma correcta, sin encontrar ningún tipo de problema.
- **Resultado con alerta**: El mánager ha encontrado problemas durante su ejecución, pero estos no son lo suficientemente graves como para justificar detener la ejecución de la simulación (Por ejemplo, el mánager de entidades no ha podido encontrar el objeto asociado a un símbolo en *map.txt*). En estos casos, la simulación continuará, pero puede haber problemas, tales como objetos ausentes, que provoquen que los resultados de la simulación difieran de lo esperado. En estos casos, el propio mánager registrará los problemas que ha encontrado.
- **Resultado con error**: El mánager ha encontrado un problema que va a impedir la ejecución de la simulación. La ausencia de ficheros de configuración o la imposibilidad de conectarse al servidor XMPP son ejemplos de problemas que causarían este tipo de resultado. En estos casos, la simulación se detiene por completo, siendo necesario reiniciar el simulador.

4.3.1 Mánager de simulación

El mánager de simulación (*Simulation Manager)* es el encargado de gestionar las distintas fases de la generación de la simulación. Cuando la simulación se inicia, este mánager lee los ficheros de configuración *folders_config.json* y *map_config.json* y extrae la información relevante. Tras esto, el mánager va llamando a la funcionalidad de los otros mánagers de manera ordenada, esperando a recibir el resultado de un mánager antes de llamar al siguiente.

En el caso del mánager de simulación, la simulación se detendrá si el mánager no puede encontrar alguno de los ficheros de configuración que necesita, o si alguna de las instancias de los otros mánagers es inválida.

4.3.2 Mánager de mapa

El mánager de mapa (*Map Manager)* es el primer mánager activado por el mánager de simulación, y se encarga de leer el fichero *map.txt* y guardar la información obtenida en una matriz de caracteres de forma que otros mánagers puedan acceder a ella cuando lo necesiten. También contiene funcionalidad que permite cambiar el tamaño del suelo, usando la información de *map.txt y map_config.json* para calcular el espació que ocupará el mapa. Finalmente, este mánager también se encarga de añadir las distintas fuentes de luz declaradas en *map_config.json.*

Si el mánager de mapa no es capaz de leer el fichero *map.txt*, o dicho fichero está vacío, lanzará un resultado con error, deteniendo la simulación. En caso contrario, devolverá un resultado correcto.

4.3.3 Mánager de entidades

El Manager de entidades (*Entity Manager)* se activa una vez *el* mánager de mapa ha cumplido sus tareas. Este mánager es el encargado de gestionar la instanciación de objetos en el mapa, tanto al inicio de la simulación, añadiendo los objetos especificados en *map.txt* y *map_config.json*, como durante el transcurso de esta, instanciando e inicializando los avatares requeridos por los agentes SPADE.

Cuando el mánager de simulación se inicia, empieza a recorrer la matriz de caracteres que ha generado el mánager de mapa. Para cada símbolo, intenta buscar el objeto asociado a dicho símbolo en el mapa de prefabs, creando una nueva instancia y posicionándolo en la posición correcta. En caso de que el símbolo sea un espacio, el hueco que ocuparía ese símbolo queda vacío. Si el símbolo no tiene un objeto asociado, el mánager registrará una alerta, pero continuará su ejecución. Finalmente, una vez ha instanciado todos los objetos, el mánager reconstruye la *navmesh*, para permitir que los avatares puedan moverse correctamente por el entorno.

Este mánager tiene además un componente *TextureBankComponent*. Tal y como se menciona en el apartado 3.3.2, los objetos de la simulación pueden tener ciertos elementos cuyo aspecto se genera de forma aleatoria usando una serie de imágenes. Este componente se encarga de cargar las imágenes necesarias y convertirlas en texturas para que el objeto instanciado pueda aplicarlas a estos elementos.

Si el mánager no puede acceder a la información que el mánager de simulación o el mánager de mapa han extraído, devolverá un resultado con error. En caso contrario, devolverá un resultado correcto.

Una vez este mánager ha terminado correctamente su inicialización, el proceso de generación de la simulación ha terminado. El último paso es inicializar el mánager de comunicaciones para permitir que los agentes SPADE se puedan comunicar con el simulador.

4.3.4 Mánager de comunicaciones

El mánager de comunicaciones (*Communication Manager*) es el encargado de recibir los mensajes de los agentes vía XMPP y reenviarlos internamente a las entidades relevantes. Para ello, los avatares deben registrarse como receptores de mensajes. Este mánager también se encarga de enviar a los agentes los mensajes resultantes de la ejecución de los comandos, como el mensaje con la posición inicial del avatar, así como las imágenes tomadas por las cámaras.

Cuando este mánager es activado por el mánager de simulación, intenta acceder al fichero *server_config.json*. En caso de que este fichero no exista, el mánager intentará usar los valores por defecto. Si no consigue conectarse a un servidor XMPP, lanzará un resultado de error.

Una vez el mánager de comunicaciones se conecta al servidor XMPP, la simulación está preparada. A partir de este momento, los agentes pueden empezar a mandar mensajes al simulador para interactuar con la simulación.

En la imagen 13 podemos ver un diagrama de secuencias, ilustrando la activación de los distintos mánagers por parte del manager de simulación.

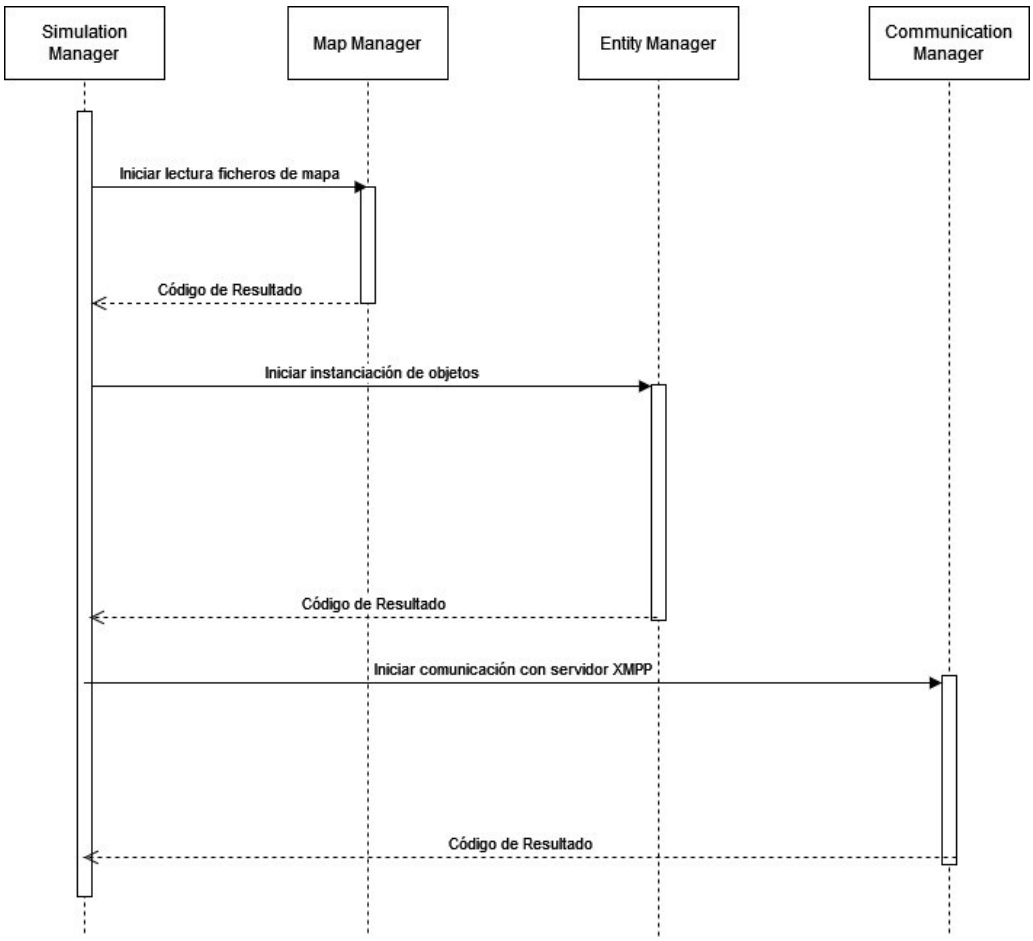

Imagen 13: Diagrama de secuencias de la generación de una simulación

4.4 Arquitectura de los avatares y ejecución de comandos

Como hemos visto en el apartado 3.2.2, la lógica ejecución de los comandos en FIVE está implementada de forma monolítica en los agentes. Aunque esto simplifica los avatares y su creación, también implica una disminución de su flexibilidad. En caso de querer hacer avatares que ejecuten comandos de forma distinta a la habitual, es necesario crear una clase derivada y sobrescribir la lógica que se llama al ejecutar un comando. Conforme aumenta el número de agentes que requieren este tipo de cambios, aumenta el número de clases que hay que mantener, dificultándose la reutilización de código y haciendo que aparezca código duplicado.

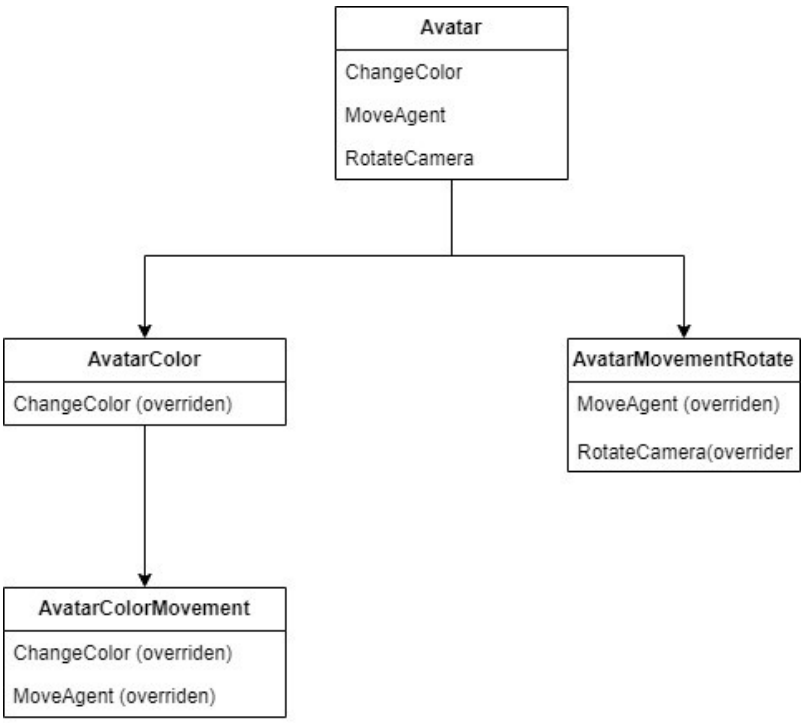

Imagen 14: Ejemplo de reutilización de código en FIVE

La imagen 14 ilustra este problema. En este ejemplo, la necesidad de tener un avatar que ejecute de forma distinta el comando de cambio de color requiere la creación de una nueva clase, *AvatarColor,* en la que la lógica de cambio de color es sobrescrita. Tras esto, se crea una nueva clase *AvatarMovementRotate*, en la que la lógica de movimiento y de rotación de cámara se sobrescriben. Finalmente, se requiere un agente que ejecute la lógica de cambio de color como *AvatarColor* y la de movimiento como *AvatarMovement,* lo que lleva a la creación de una tercera clase, hija de *AvatarColor*, llamada *AvatarColorMovement*, en la que se sobrescribe la lógica de movimiento para que sea igual a *AvatarMovement*. Como podemos ver, la lógica de movimiento de *AvatarMovementRotate* no se puede reutilizar, por lo que debe ser duplicada en *AvatarColorMovement*

Para evitar este problema, se ha optado por utilizar un enfoque basado en la composición para la implementación de los avatares en FIVE-Godot. La lógica para ejecutar los distintos comandos (a excepción del comando *create_agent*, que es

ejecutado por el mánager de entidades) está agrupada en distintos componentes. Un avatar puede tener los componentes que necesite según los comandos a los que quiera reaccionar. Por ejemplo, el comando *change_color* se ejecuta en el componente de aspecto, por lo que un avatar que no vaya a cambiar de color puede optar por no tener este componente. Los componentes que existen actualmente son:

- **Componente de cámara:** Encargado de ejecutar los comandos relacionados con la toma de imágenes. Contiene la lógica de los comandos *move_camera, fov_camera, rotate_camera* y *take_image.*
- **Componente de movimiento:** Responsable de los comandos de movimiento del agente. Ejecuta el comando *move_agent.*
- **Componente de aspecto:** Este componente ejecuta los comandos que cambian el aspecto del avatar, más concretamente el comando *change_color*.

Siguiendo este enfoque, el problema presentado anteriormente se podría resolver sin duplicidad de código. Los cambios requeridos en la lógica de movimiento, rotación de cámara y cambio de color llevarían a la creación de nuevos componentes de cámara, aspecto y movimiento. Una vez creados, los nuevos avatares solo tendrían que incluir el componente con la funcionalidad que requieren. Así pues, *AvatarColor* usaría el nuevo componente de aspecto; *AvatarMovementRotate* los nuevos componentes de movimiento y cámara; y *AvatarColorMovement* los nuevos componentes de aspecto y movimiento. Esto eliminaría la duplicidad de código entre *AvatarColorMovement* y *AvatarMovementRotate*, ya que ambos usan el mismo código, existente en el nuevo componente de movimiento.

4.5 Diseño detallado

En este apartado analizaremos el código que forma el simulador, así como el proceso para expandir el simulador añadiendo nuevos elementos de simulación y avatares

4.5.1 Código del simulador

El simulador está construido con el lenguaje de programación C#. En este apartado veremos en detalle las distintas clases que intervienen en el simulador, ayudándonos con diagramas UML. Debido a su tamaño, esta sección incluye partes recortadas del diagrama, que se puede consultar en el siguiente enlace: https://rb.gy/p3wmfz.

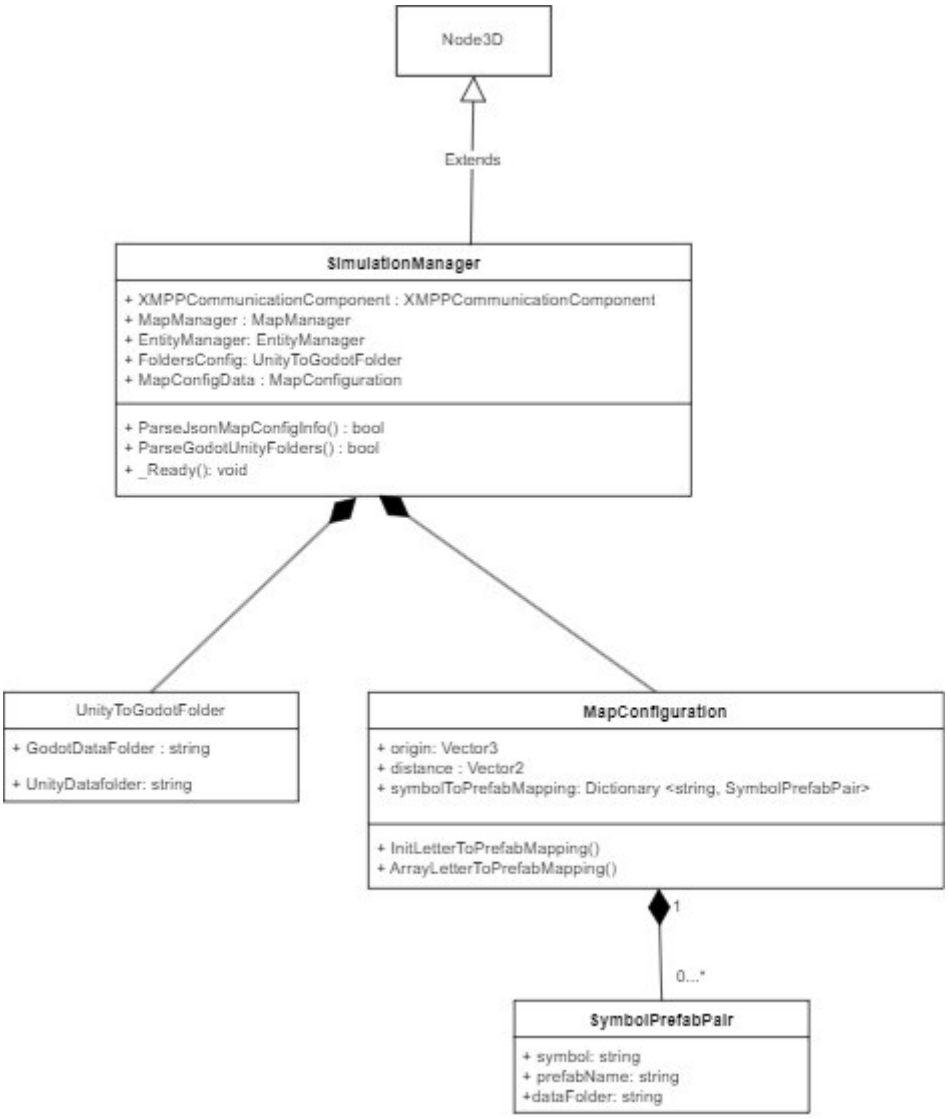

Imagen 15: Diagrama UML centrado en la clase SimulationManager

En la imagen 15 podemos ver el diagrama UML de la primera clase que se ejecuta en el simulador, *SimulationManager*. Esta clase, como ya hemos explicado en el apartado 4.3.1, se encarga de leer archivos de configuración específicos e ir activando los distintos mánagers que se encargan de los distintos elementos de la simulación. El primer archivo que lee es *folder_configs.json*, guardando la información de las rutas de ficheros en la clase *UnityToGodotFolder*.

Tras esto, el mánager lee la información del fichero *map_config.json*. Esta información esta encapsulada en la clase *MapConfiguration*, que contiene el vector con el punto de origen del mapa, un vector representando la distancia entre elementos y un diccionario de *strings* a elementos de la clase *SymbolPrefabPair*. Esta clase guarda la información que relaciona un símbolo con el tipo de objeto que representa, así como, opcionalmente, la ruta al directorio de imágenes que puede usar como texturas.

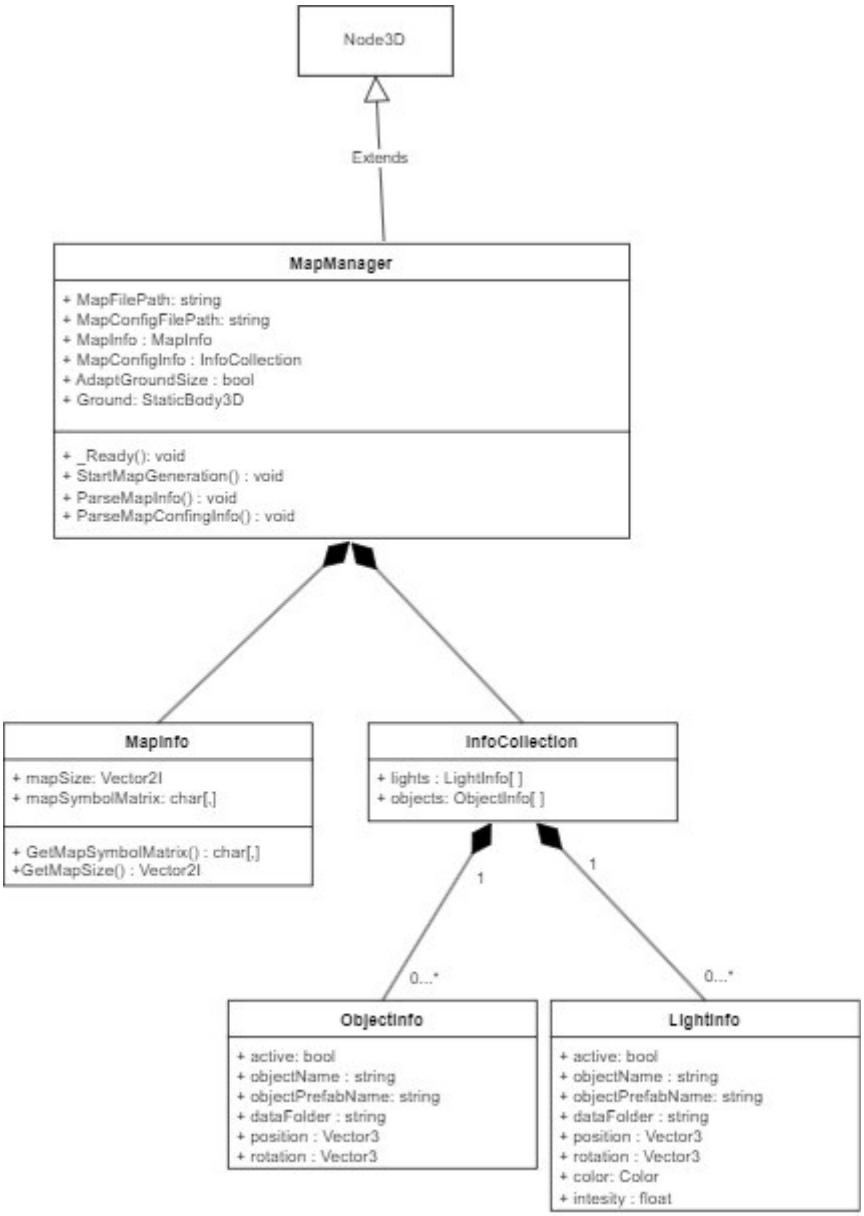

Imagen 16: Diagrama UML centrado en la clase MapManager

Una vez el mánager ha terminado de leer los ficheros, activa el mánager de mapa, encapsulado en la clase *MapManager*, cuyo diagrama se muestra en la imagen 16. Esta clase se encarga de leer los archivos de mapa, más concretamente *map.txt* y *map.json*. Tras leer el primero, guarda los distintos caracteres que encuentra en una estructura *MapInfo*. Esta estructura contiene un vector bidimensional con el número de elementos en los ejes “x” y “z”, así como una matriz bidimensional que contiene el símbolo que se ha leído en cada posición del mapa.

Si el mánager ha sido capaz de leer el fichero *map.txt* y guardar la información de forma correcta, procede a hacer lo mismo con el fichero *map.json*. Para guardar la información de este fichero, se utiliza la clase *InfoCollection*, que está compuesta por dos *arrays* de tipo *LightInfo* y *ObjectInfo.* El primero contiene la información de los

objetos de luz que hay en el mapa, mientras que el segundo contiene la información de los otros objetos que se quieren instanciar de forma específica en la simulación.

Una vez se han leído correctamente ambos ficheros, si la opción *Adapt Ground Size* está seleccionada, el mánager procederá a cambiar el tamaño del suelo de la simulación. Para ello utiliza la información del número de elementos en el mapa, guardada en *MapInfo*, junto a la información de distancia entre elementos, guardada en *MapConfiguration*, para calcular la extensión que tendrá el mapa.

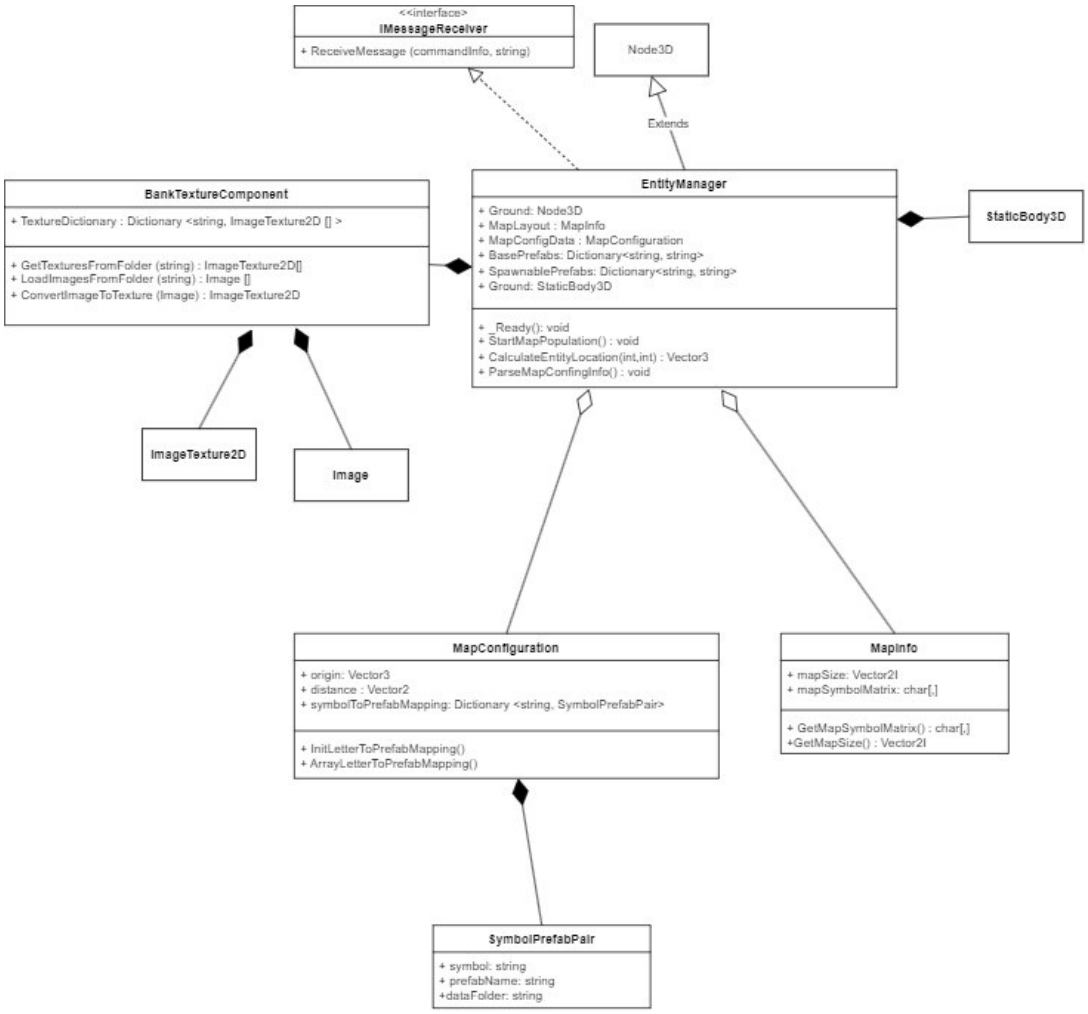

Imagen 17: Diagrama UML centrado en la clase EntityManager

Finalizada estas tareas, se activa el mánager de entidades, encapsulado en la clase *EntityManager* tal y como se puede ver en la imagen 17. Este mánager implementa la interfaz *IMessageReceiver*, haciendo que pueda recibir mensajes reenviados por el mánager de comunicaciones, y se encarga de instanciar las distintas entidades que poblarán la simulación. Para ello, cuenta con un diccionario *BasePrefabs* en el que asocia los nombres de los distintos objetos de la simulación que se pueden instanciar durante la generación de mapas. El mánager obtiene de *MapManager* la información de *MapInfo* y *MapConfiguration*, tras lo cual comienza a recorrer la matriz de símbolos de

MapInfo, usando *MapConfiguration* para obtener el nombre del símbolo que ocupa cada posición del mapa e instanciando el objeto asociado a dicho nombre en el diccionario.

En el caso de que el objeto tenga asociado un fichero con imágenes para aplicar de forma aleatoria, el mánager solicita las texturas requeridas al banco de componentes, encapsulado en la clase *BankTextureComponent*. Este componente contiene un diccionario que asocia las rutas de directorios con las texturas creadas a partir de las imágenes del fichero. La primera vez que un objeto quiere usar las imágenes de una carpeta específica, el componente carga las imágenes y las convierte a ficheros de tipo *ImageTexture2D*, tras lo cual devuelve un *array* con las texturas creadas para que el objeto pueda aplicarlas. Las texturas se guardan en el diccionario, usando la ruta de carpeta como clave. De esta forma, si otro elemento quiere utilizar las imágenes de esta carpeta, las texturas creadas se reutilizan, evitando tener que repetir el proceso de carga de imágenes y creación de texturas.

Imagen 18: Diagrama UML centrado en la clase TextureChangerComponent

El proceso de aplicar las texturas se lleva a cabo en el *TextureChangerComponent*, mostrado en la imagen 18. Este componente recibe del *EntityManager* un *array* con las texturas posibles a usar y recorre un *array* con los elementos a los que hay que aplicar dichas texturas, eligiendo una de las candidatas de forma aleatoria y aplicándola.

Una vez instanciados todos los elementos del mapa, el mánager procede a obtener *InfoCollectio*n del mánager de mapa, y, usando el mismo método, instancia los objetos especificados en *ObjectInfo* y *LightInfo.*

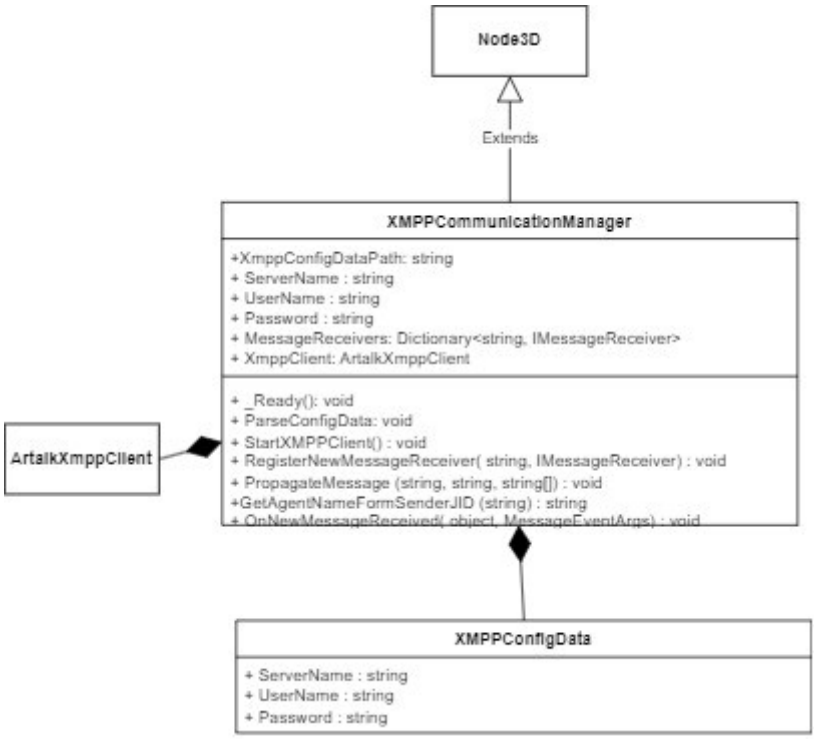

Imagen 19: Diagrama UML centrado en la clase XMPPCommunicationManager

Tras esto, se activa *XMPPCommunicationManager*, la clase que se puede ver en la imagen 19, encargada de gestionar las comunicaciones XMPP. Esta clase intenta conectarse al servidor XMPP usando la información del fichero *server_config.json*, que se lee en un objeto de la clase *XMPPConfigData*. En caso de que el fichero no exista, el mánager intenta conectarse usando los valores por defecto, especificados en el editor. Una vez el mánager se ha conectado de forma satisfactoria, la simulación da comienzo, y los agentes pueden empezar a interactuar.

Cuando un agente envía un comando al simulador, el comando se analiza en *XMPPCommunicationManager*, guardando la información en un objeto de la clase *CommandInfo*. Para poder reenviar el comando al objeto que debe ejecutarlo, el mánager tiene un diccionario de elementos de la simulación que son capaces de gestionar comandos. Solo clases que implementen la interfaz *IMessageReceiver* pueden recibir comandos de esta forma.

En el caso de que el comando sea *create_agent*, el comando es reenviado al *EntityManager*, que se encarga de crear el nuevo agente. Para ello, el *EntityManager* tiene un diccionario con los distintos tipos de avatar que se pueden instanciar.

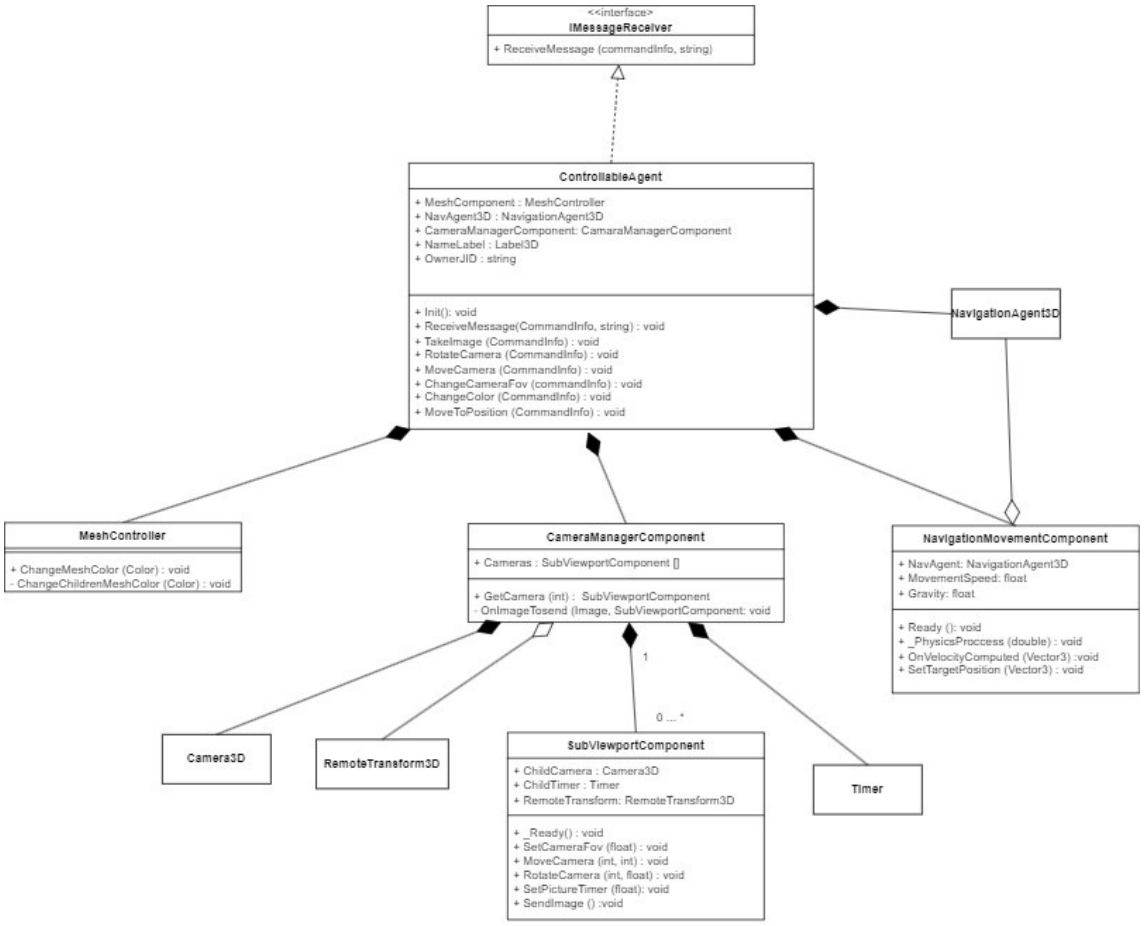

Imagen 20: Diagrama UML centrado en la clase ControllableAgent

Los avatares, cuyo diagrama de clase se muestra en la imagen 20, pertenecen a la clase *ControllableAgent*, e implementan la interfaz *IMessageReceiver*. Heredan de la clase de Godot *CharacterBody3D*, una clase específica para personajes que van a ser controlados y se moverán durante la ejecución del programa. Cuando un avatar se instancia, lo primero que hace es usar la clase de utilidades para registrarse como receptor de comandos. Tras esto espera a recibir nuevos comandos, analizando la información del comando y reenviándola al componente encargado de ejecutarlo. Como se mencionó en el apartado 4.4, los avatares tienen tres componentes, que se explican a continuación:

El componente de aspecto *MeshController* contiene la lógica más simple. Cuando recibe un comando change_color, recorre todos los nodos que forman el avatar, buscando los nodos de tipo *MeshInstance3D*, y cambiando su color.

El componente de movimiento *MovementComponent* se encarga de mover el avatar, usando las funciones internas de *pathfinding* y movimiento de Godot.

Finalmente, el componente de cámara *CameraManagerComponent* se encarga de gestionar las cámaras del agente. Este componente contiene un *array* de objetos *SubViewportComponent* y su única función es mandar los comandos a las cámaras

relevantes, así como recibir las imágenes de la cámara y mandarlas al agente responsable del avatar.

El *SubViewportComponent*, objeto de tipo *SubViewport,* es la clase que contiene la funcionalidad de la cámara del agente. Contiene un componente de tipo *Camera3D*, uno de tipo *Timer* y, finalmente una referencia a un *RemoteTransform3D*. La *Camera3D* proyecta las imágenes que captura en el *SubViewport*, desde el cual la podemos transformar en ficheros JPG y devolver al *CameraManagercomponent* para su envío al agente. Para hacer este envío, la información de la imagen y el tiempo en el que se capturó se guardan en un objeto de la clase *ImageData*, que contiene un número para identificar la cámara que hizo la foto, un *DateTime* con el momento en que se capturó, y una *string* con la imagen codificada en formato Base64.

El *RemoteTransform3D* se utiliza para resolver un problema inherente a la arquitectura del agente. Debido a como está diseñado Godot, la cámara va a ocupar la posición que herede de su nodo padre, pero el *SubViewport* es un tipo de nodo que no tiene una posición. Esto hace que la cámara se sitúe en la posición global (0,0,0) al inicializarse. Por otro lado, la cámara tiene que ser hija del *SubViewport* ya que esto nos permite obtener las imágenes de la cámara para mandarlas a los agentes. El *RemoteTransform3D* es un tipo de nodo que "impone" su posición y rotación a otro nodo. De esta forma, podemos poner este nodo en la posición en la que queremos que esté la cámara y hacer que copie su posición a la cámara, resolviendo así este problema.

Además de las clases mencionadas anteriormente, existe un namespace llamado *Utilities*. Este namespace contiene las siguientes clases estáticas con funciones de utilidad:

- *ConfigData:* Esta clase contiene funciones para acceder con facilidad a los objetos de tipo *MapConfiguration, MapInfo, InfoCollection y UnityToGodotFolder,* que contienen la información necesaria para ejecutar la simulación.
- *Files:* Esta clase contiene funciones para leer ficheros y convertir ficheros JSON a las clases adecuadas.
- *Messages:* Contiene funciones para convertir la información incluida en los mensajes recibidos, así como para enviar mensajes y registrar avatares y otros elementos como receptores de los mensajes.
- *Math:* Esta clase contiene una función que permite orientar vectores de Unity a Godot y viceversa.

Finalmente, las clases *UnityVector3* y *UnityColor* actúan como clases intermedias para traducir los tipos de color y vector tridimensional entre Godot y Unity, y su funcionamiento se detallará más adelante.

4.5.2 Agregar nuevos avatares

El proceso para añadir nuevos tipos de avatares a la simulación es el siguiente:

1. Abrir el proyecto del simulador en el editor de Godot.
2. Importar el modelo del avatar en el editor de Godot.
3. Crear una nueva *scene* de tipo *CharacterBody3D* y añadir el script *ControllableAgent* y el modelo que se ha importado.

4. Añadir un nodo hijo de tipo *CollisionShape3D* y configurar sus valores. Esto permitirá que el avatar pueda detectar colisiones con otros elementos de la simulación.
5. Añadir un nodo hijo de tipo *NavigationAgent3D*. Esto permitirá que el agente pueda moverse usando el sistema de *pathfinding*.
6. Agregar un nodo hijo del tipo *Node3D*, y añadir el *script MeshController*.
7. Agregar un nodo hijo de tipo *Node3D* y añadir el script *MovementController*.
8. Añadir un nodo de tipo *Label3D* y cambiar su valor de *Layers* a 3.
9. Agregar un nodo de tipo *Node3D* y añadir el script *CameraManager*. Para cada cámara que se quiere añadir hay que seguir el siguiente proceso:
    a. Crear un nodo hijo al nodo que contiene el mánager de cámara, de tipo *SubViewport*, con el script *SubviewportComponent* añadido.
    b. Crear un nodo de tipo *Timer*, hijo del nodo *Subviewport*.
    c. Crear un nodo de tipo *Camera3D*, hijo del nodo *Subviewport*, y configurar con los valores de cámara deseados. En la configuración de este nodo, deseleccionar el valor 3 en la *Cull Mask*. Esto hará que los nombres de los agentes no aparezcan en las capturas que tome esta cámara.
    d. Crear un nodo de tipo *RemoteTransform3D* en la escena (no tiene por qué ser hijo de estos nodos que estamos creando) y seleccionar la cámara que hemos creado en el paso anterior como *Remote Path*.
    e. En el nodo *Subviewport*, asignar los nodos que hemos creado a las variables *Child Camera*, *Timer Child* y *Remote Transform.*
    f. Finalmente, en el nodo *CameraManager,* añadir el nodo *Subviewport* al *array* de cámaras.
10. Finalmente, en el nodo que contiene el script *ControllableAgent,* asignar las variables *Mesh Component, Nav Agent 3d, Movement Component, Camera Manager Component* y *Name Label* a los nodos respectivos.

Una vez construido el avatar, hay que añadirlo a la lista de avatares del mánager de entidades, para que pueda instanciarlo cuando un agente lo solicite:

1. Seleccionar el nodo *EntityManager.*
2. En el inspector, seleccionar la variable *Spawnable Prefabs.*
3. Añadir un nuevo elemento al diccionario:
    a. La clave será el nombre que usarán los agentes para referirse al agente (por ejemplo, “Tractor” para el avatar que ya existe).
    b. El valor será la ruta a la *scene* que hemos creado. Una forma fácil de obtenerla es en la vista de *FileSystem*, seleccionar la *scene*, hacer click con el botón derecho y seleccionar “Copy Path”.

Una vez hecho esto, nuevo avatar está listo para su uso en la simulación.

4.5.3 Agregar nuevos elementos a la simulación

El proceso para agregar nuevos elementos a la simulación es similar al proceso para agregar nuevos avatares:

1. Abrir el proyecto del simulador en el editor de Godot.
2. Importar el modelo del elemento en el editor de Godot.
3. Crear una nueva *scene* de tipo *Node3D* y añadir el modelo que se ha importado.

4. Añadir un nodo *CollisionShape3D.*
5. Guardar la *scene*.

Una vez guardada la *scene*, los pasos para que se pueda usar en la simulación son idénticos a los pasos para usar un avatar, con la diferencia de que en el nodo *EntityManager* hay que seleccionar la variable *Base Prefabs*, y que la clave de elemento debe ser una única letra.

Si queremos que el elemento que hemos creado tenga partes cuyo aspecto se decide aleatoriamente hay que hacer los siguientes pasos:

1. Añadir a la *scene* un componente *StaticMesh3D* por cada elemento con aspecto aleatorio que se quiera añadir.
2. Añadir a la *scene* base del elemento (el nodo Node3D creado en el apartado 3) el script *TextureChangerComponent.*
3. Añadir al *array TextureChangeableElements* los componentes creados en el primer apartado.

Finalmente, añadir una carpeta nueva al proyecto con las imágenes que queremos utilizar.

Capítulo 5 – Desarrollo de la solución

5.1 Introducción

Este apartado las decisiones que se tomaron a la hora de implementar la simulación, así como los problemas que aparecieron durante el desarrollo y como se solucionaron.

5.2 Lenguaje de programación y bibliotecas externas

5.2.1 Lenguajes de programación

Uno de los primeros problemas que hubo que abordar al inicio de la implementación fue decidir en qué lenguaje se iba a implementar el simulador. Como se menciona en el apartado 2.4, Godot permite el desarrollo de aplicaciones con dos lenguajes: GDScript, un lenguaje propio; y C#, un lenguaje multiparadigma creado y mantenido por Microsoft.

Godot ofrece muchas prestaciones que hacen que trabajar en GDScript sea más atractivo. Por un lado, Godot incorpora un editor de código, mostrado en la imagen 21, que permite trabajar en GDScript sin requerir programas externos, ofreciendo características como destacado de palabras reservadas, autocompletar, abrir ventanas partidas o conexión con sistemas de control de versiones. A la hora de probar el código, permite añadir *breakpoints*, así como medir el rendimiento de las distintas funciones con un analizador de rendimiento visual. Además, los cambios en el código no requieren recompilar el proyecto, lo que permite trabajar más rápido.

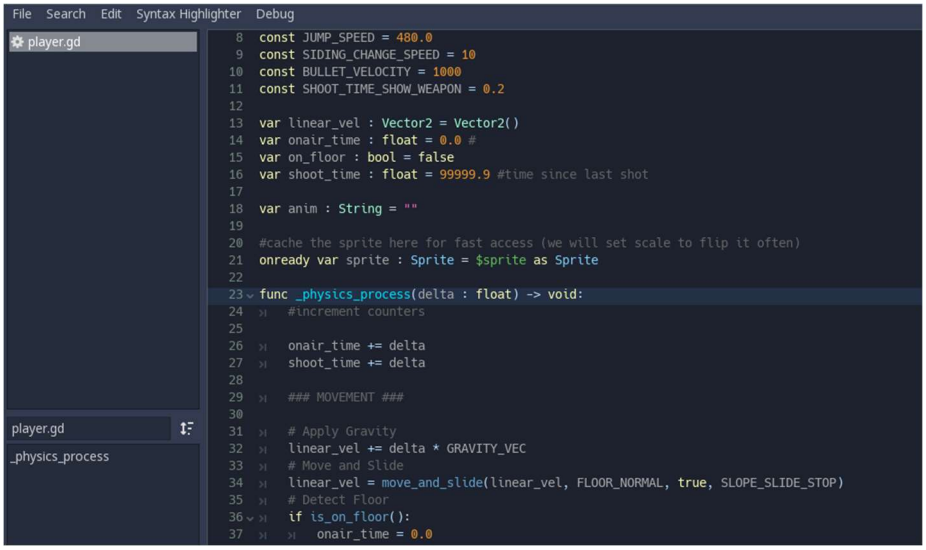

Imagen 21: Ejemplo de código GDScript en el editor interno de Godot

Por otro lado, el desarrollo en GDScript tiene algunos inconvenientes. Uno de los principales es que, al tratarse de un lenguaje dinámicamente tipado, el rendimiento es inferior al de un lenguaje estáticamente tipado. Esto también hace que el análisis gramatical sea menos estricto, detectando menos errores en tiempo de compilación. A la hora de escribir código, la naturaleza dinámica del lenguaje dificulta el auto completado de código (el tipo de algunas variables solo puede saberse durante la ejecución). Esta naturaleza también hace que sea más fácil que se provoquen bugs al asumir el tipo de una variable en el código. Otro problema que tiene GDScript es la dificultad para encontrar bibliotecas con funcionalidad nueva, especialmente funcionalidad que no está directamente relacionada con el desarrollo de videojuegos.

Este último problema, unido al conocimiento y la experiencia previa con C#, fue el mayor aliciente a la hora de decantarse por usar C#. En caso de haber querido utilizar GDScript, no solo habría hecho falta dedicar una parte importante de tiempo en aprender el lenguaje, si no que habría hecho falta implementar la funcionalidad para conectarse e interactuar con servidores XMPP desde cero, lo que seguramente habría sido más complejo y llevado más tiempo que la totalidad de este proyecto.

5.2.2 Bibliotecas externas

Aunque la mayoría de la funcionalidad del simulador se ha creado desde cero, hay dos elementos que se han implementado utilizando librerías externas, ya que se consideró que la implementación de estos elementos excedía el ámbito de este trabajo. Así pues, el proyecto utiliza las siguientes librerías:

- **Artalk.XMPP**[12]**:** Librería .Net que permite la comunicación con un servidor XMPP
- **Newtonsoft.Json**[13]**:** Librería para la serialización y deserialización de ficheros JSon.

5.3 Problemas encontrados durante la implementación

5.3.1 JSON en Godot

Uno de los primeros problemas que se encontró al empezar el desarrollo fue que Godot ofrece un soporte muy limitado para trabajar con ficheros JSON. A diferencia de otros lenguajes, Godot solo permite importar ficheros JSON como un diccionario que utiliza *strings* como clave y valor de sus elementos. Esto dificulta su uso, especialmente en casos como *map_config*.json, en el que el fichero contiene objetos complejos con variables anidadas. Godot también limita los tipos de objeto que puede exportar a JSON, requiriendo que hereden de la clase *Variant*.

Esta limitación se debe a que Godot favorece usar un tipo propio de fichero, el *Resource*, para guardar y transferir información. Aunque este tipo de dato ofrece bastantes funcionalidades que permite usarlo con facilidad en Godot, tales como auto

[12] https://github.com/araditc/Artalk.Xmpp

[13] https://www.newtonsoft.com/json

serialización o la capacidad de editar este tipo de datos en el editor, el hecho de que la aplicación debe ser compatible con FIVE, así como mandar y recibir información codificada en JSON a los distintos agentes hace que sea necesario trabajar con JSON. Para solventar este problema, se utilizó la librería Newtonsoft.Json, que permite la serialización y deserialización de todo tipo de ficheros JSON de manera rápida y eficiente.

5.3.2 Diccionarios en Godot

Otro problema que se encontró durante el desarrollo fue la implementación de los diccionarios en Godot. Los diccionarios en Godot no están estáticamente tipados, lo que permite que un diccionario contenga claves o valores de distintos tipos. Esto puede llevar a errores en los que se trabaja con un tipo de diccionario asumiendo el tipo de datos que va a contener, solo para provocar un error al intentar acceder a un valor del diccionario que en realidad es de un tipo distinto.

Una opción sopesada para resolver este problema fue utilizar un tipo de dato distinto para guardar la información. Ejemplos de opciones barajadas fue crear clases específicas para guardar la información o usar dos *arrays* ordenados, uno de los cuales contendría las claves del diccionario y otro los valores. En este caso, para obtener el dato asociado a una clave, se buscaría dicha clave en el primer *array*, y una vez encontrado, se usaría la posición de la clave en el *array* para extraer su valor asociado en el *array de valores*.

Esta opción finalmente se descartó ya que, aunque solucionaría el problema de los tipos, se consideró que podía causar más problemas, al depender de que el orden de ambos *arrays* fuera correcto. Además, utilizar diccionarios en Godot ofrece una gran ventaja, y es que pueden modificarse desde el editor de forma fácil y rápida.

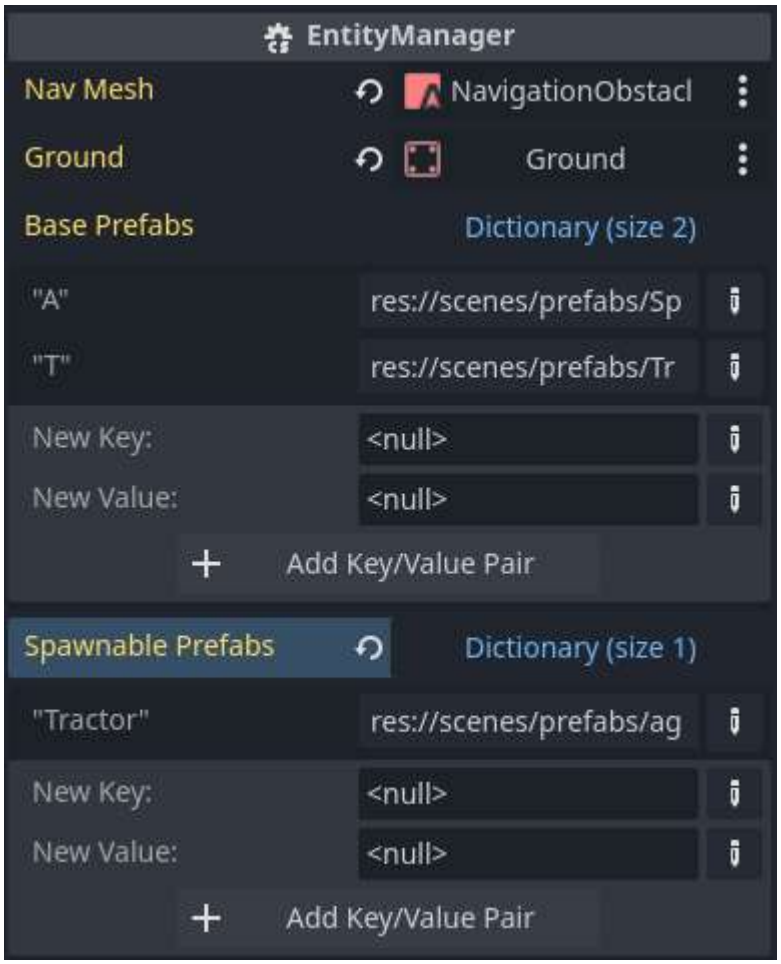

Imagen 22: Ejemplo de diccionarios en Godot

En la imagen 22 se ve el ejemplo de dos diccionarios pertenecientes al mánager de entidades. Añadir nuevos elementos a estos diccionarios es tan fácil como seleccionar el tipo adecuado de dato para la clave y el valor (*string* en este caso) pulsando el lápiz a la derecha de los datos a añadir, introducir los valores que se quieren agregar y apretar el botón 'Add Key/Value Pair'.

Eventualmente, se descubrió que hay una *pull request* en el repositorio de Godot que contiene una implementación de diccionarios tipados[14]. Esta implementación ya ha sido aprobada por Juan Linietsky, y está en los pasos finales previos a ser integrada en el código de Godot. Por esta razón, se decidió que lo mejor era usar los diccionarios que Godot ofrece, y actualizarlos a diccionarios tipados una estos cambios vez se hayan introducido en Godot.

5.3.3 Traducción de información entre Godot y Unity

Uno de los primeros problemas que se encontraron durante el desarrollo fue la dificultad a la hora de traducir la información que recibía el simulador de los agentes,

[14] https://github.com/godotengine/godot/pull/78656

que están preparados para trabajar con FIVE. Aunque la mayoría de la información se podía utilizar sin necesitar ningún tipo de modificación, surgieron dos problemas que impedían a Godot usar los comandos recibidos por los agentes: La diferencia de ejes y la de nomenclatura.

La **diferencia de ejes** hace referencia a como comprenden las coordenadas Unity y Godot. Ambos utilizan un sistema de coordenadas *Y-up*, en el que el eje Y es el eje del plano vertical, mientras que los ejes X y Z forman el plano horizontal. El problema que se encontró es que Unity usa coordenadas diestras, mientras que Godot usa coordenadas zurdas. Eso significa que el eje que Unity considera como Z positivo, Godot lo considera Z negativo. La imagen 23 muestra las diferencias entre los ejes de Unity (izquierda) y Godot(derecha).

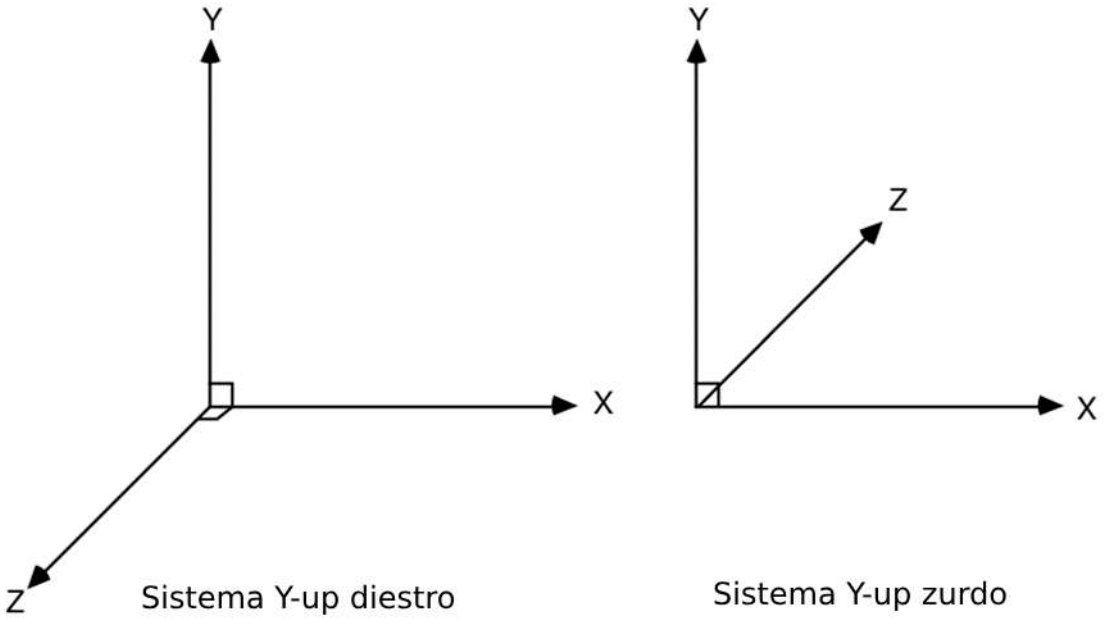

Imagen 23: Sistemas de coordenadas en Unity y Godot

La **diferencia de nomenclatura** hace referencia a la diferencia entre como Godot y Unity nombran las variables de sus objetos. Mientras que, en Unity, los campos de las clases comienzan su nombre con una letra minúscula, como *deltaTime* o *border*, los campos de las clases de Godot empiezan su nombre con letra mayúscula, como *DeltaTime* o *Border*. Esto provocó problemas en el intercambio de información con los agentes mediante comandos. Por ejemplo, intentar deserializar la posición de un comando *move_agent* o el color de un comando *change_*color provocaba un error, ya que la librería no era capaz de convertir el *string* serializado a un objeto del tipo adecuado.

Esto a su vez también causaba problemas en los agentes, ya que esperaban recibir del simulador objetos serializados con la nomenclatura de Unity. Por ejemplo, al recibir una posición de FIVE-Godot, los agentes extraen la información de la posición a

un diccionario e intentan obtener los valores usando las claves “X”, “Y” y “Z”, pero las claves que tenía el diccionario creado con el mensaje enviado por FIVE-Godot eran “x”, “y” y “z”.

Para resolver este problema, se crearon clases intermediarias para los tipos Vector y Color, llamadas *UnityVector* y *UnityColor* respectivamente. Estas clases contienen los campos con la nomenclatura que utiliza Unity, así como una función que utiliza estos datos para crear una variable del tipo de dato nativo de Godot y, en el caso de *UnityVector*, una función para crear una variable de este tipo intermedio usando un objeto del tipo vector nativo de Godot, así como lógica para cambiar el signo de la coordenada Z del vector, permitiendo cambiar entre el eje de coordenadas diestro de Unity y el zurdo de Godot. Un ejemplo de cómo el simulador usa estos tipos es el siguiente:

1. Uno de los agentes envía el comando *move_agent* con un vector diestro de la forma (X, Y, Z) en formato JSON.
2. Cuando el simulador lo recibe, extrae el vector a un objeto de tipo *UnityVector*
3. El simulador llama al método *GetGodotVector*. Este método crea un objeto del tipo vector nativo de Godot, cambiando el signo de la coordenada z y haciendo así el vector zurdo.
4. Eventualmente, cuando el avatar termina su movimiento, el simulador debe mandar al agente la nueva localización del avatar.
5. Para ello llama al constructor de *UnityVector*, pasando al constructor la posición del avatar.
6. El constructor crea un nuevo objeto *UnityVector*, cambiando el signo de la coordenada z para hacer el vector diestro.
7. El simulador puede ahora serializar el objeto *UnityVector*, creando así un texto en formato JSON equivalente al que habría mandado FIVE.

5.3.4 Múltiples instancias del mismo mánager

Durante el desarrollo de la solución, se dio el caso de que un día, sin motivo aparente, el mánager de comunicaciones comenzó a fallar. El mánager se conectaba correctamente al servidor XMPP, pero a la hora de mandar mensajes, daba un error de que no estaba conectado. Tras investigarlo, se descubrió que el problema era que, por error, se había añadido un segundo mánager de comunicaciones a la simulación, que era el que se estaba usando para intentar mandar mensajes.

Aunque este es un error que se puede detectar y resolver fácilmente, se decidió añadir una salvaguarda para evitar que pueda volver a suceder. Para ello, se implementaron los mánagers usando el patrón *Singleton*. Este patrón asegura que solo exista una única instancia de una clase a la vez durante la ejecución del programa. En caso de que se inicie el programa con varios mánagers del mismo tipo, el único que permanecerá será el primero que se haya instanciado. El resto detectarán que ya hay un mánager, registrarán un aviso, y se destruirán.

Capítulo 6 – Rendimiento y casos de prueba

En este capítulo, estudiaremos el rendimiento del simulador y su consumo de recursos. En primer lugar, analizaremos el rendimiento de FIVE-Godot, incluyendo su consumo en entornos de gran tamaño. Tras esto, lanzaremos simulaciones en FIVE y FIVE-Godot, y compararemos el rendimiento en ambas versiones.

Estas pruebas se han realizado en un ordenador con un procesador AMD Ryzen Threadripper 3960X, 128 gigabytes de RAM y una tarjeta gráfica NVIDIA GeForce RTX3080.

6.1 Rendimiento de FIVE-Godot

En este apartado se analizará el rendimiento de FIVE-Godot, así como el tiempo que tarda en generar una simulación, incluyendo gráficas para ver de forma visual como el número de objetos en la simulación afecta al consumo de recursos.

6.1.1 Métricas

Godot ofrece múltiples monitores que permiten medir diferentes aspectos del rendimiento de una aplicación, tales como el número de polígonos en la navegación o el número de primitivas dibujadas en un fotograma. Para medir el rendimiento del simulador, utilizaremos las siguientes métricas:

La primera métrica que se utilizará es el **tiempo de carga**. Esta métrica medirá el tiempo que pasa desde que el simulador inicia la generación de la simulación hasta el momento en que la simulación esta lista para ser utilizada por los agentes. Más concretamente, se mide el tiempo entre que el manager de simulación inicia el proceso de carga de la simulación hasta que recibe el aviso de que el mánager de comunicaciones se ha conectado y los agentes pueden empezar a interactuar con la simulación. Esta métrica se medirá en milisegundos.

La segunda métrica relevante es el **número de objetos instanciados**. Este número cuenta los distintos subobjetos (nodos en Godot) que componen los objetos de la simulación, por lo que aumentará de forma lineal con el tamaño de la simulación.

La tercera métrica utilizada será el **consumo de memoria**. Esta medirá la cantidad de memoria RAM que la simulación utiliza una vez generada. La medida de esta métrica será el megabyte.

La cuarta métrica que se usará es el **consumo de GPU**. Esta métrica mide la cantidad de memoria RAM de la tarjeta gráfica (*GPU, Graphics Processing Unit* en inglés*)* utiliza la simulación. Al igual que el consumo de memoria, esta métrica se medirá en megabytes.

La quinta métrica que se tendrá en cuenta son los **fotogramas por segundo**. Esta métrica calcula los fotogramas que la simulación es capaz de generar por segundo. Esta es una métrica muy variable ya que Godot, como la mayoría de los motores de videojuegos modernos, utiliza la técnica de tronco piramidal, o view *frustrum*, que evita

que se dibujen elementos que no van a aparecer en la pantalla. Para medir esta métrica, se coloca la cámara en una posición en la que se vean la mayor cantidad posibles de elementos de la simulación, lo que nos permite obtener una medida de los fotogramas por segundo en el peor caso posible.

La prueba de rendimiento consistirá en generar simulaciones con un cierto número de elementos en ella, como muestran las imágenes 24 y 25, y medir las métricas explicadas. Cada simulación se generará tres veces, tomando las métricas y calculando los valores medios de cada una de ellas. Tras esto, se aumentará el número de elementos en la simulación para poder ver como entornos más grandes afectan al rendimiento del simulador.

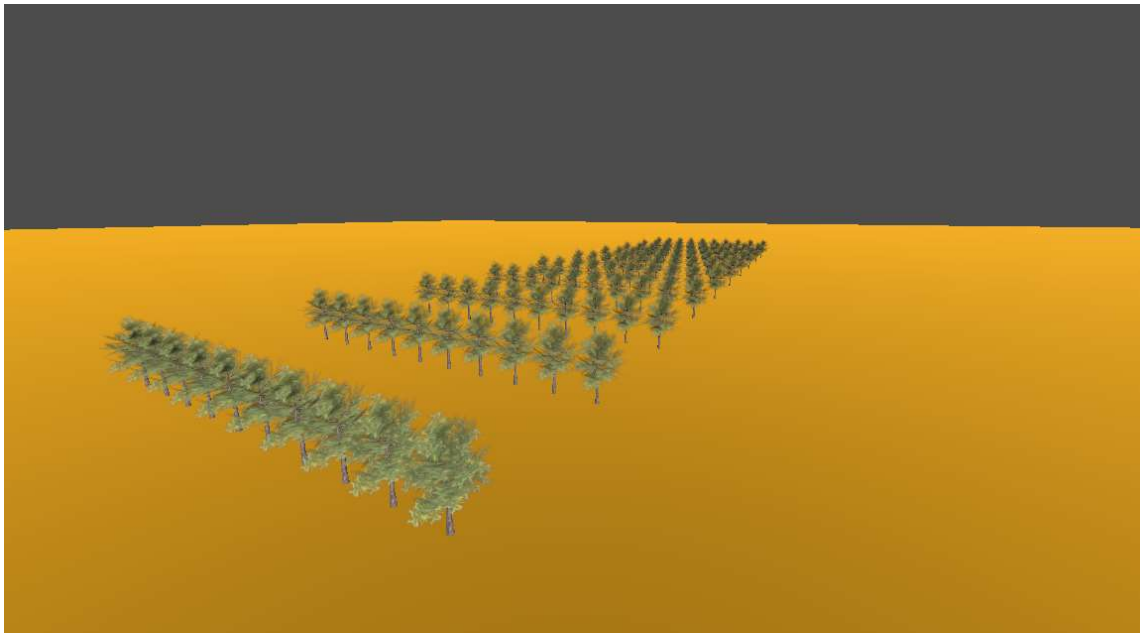

Imagen 24: Ejemplo de simulación con 100 elementos

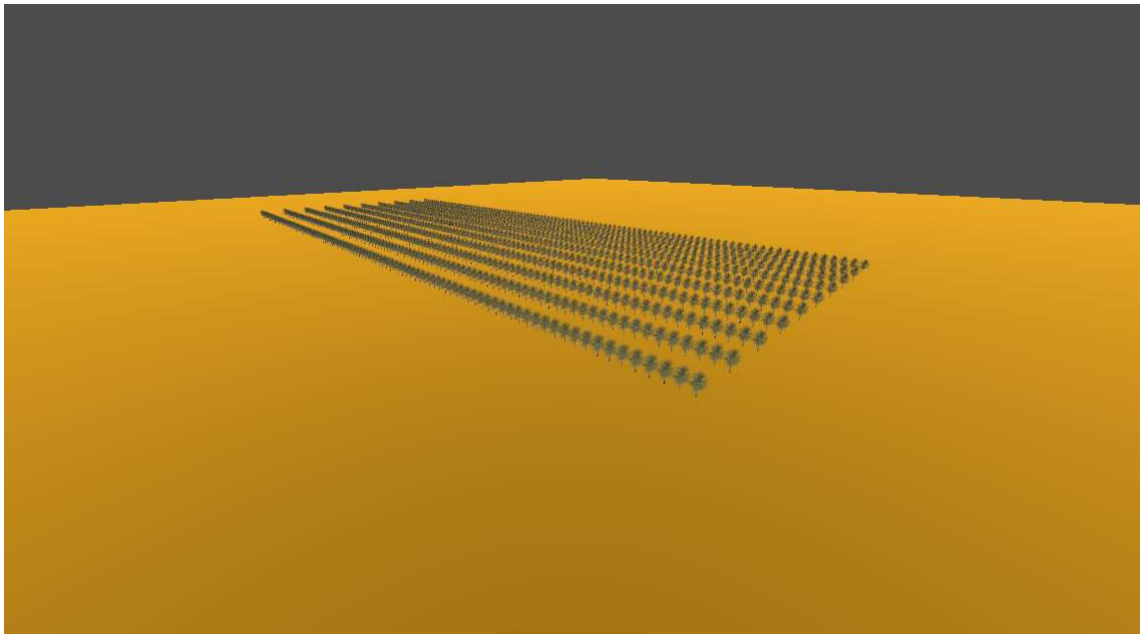

Imagen 25: Ejemplo de simulación con 1000 elementos

6.1.2 Resultado de las pruebas

Tras realizar las pruebas, los resultados son los siguientes:

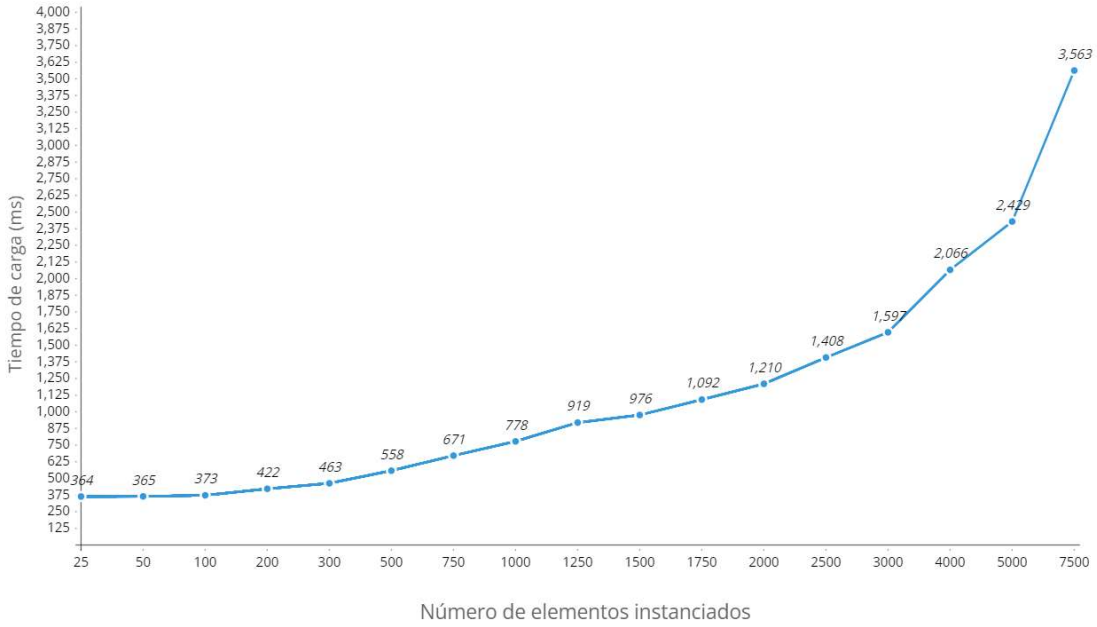

Imagen 26: Medida de tiempo de carga

En la imagen 26 podemos ver los resultados de medir el tiempo de carga. El eje vertical indica el número de elementos que se cargan en la simulación, mientras que el eje horizontal muestra el tiempo de carga de la simulación. Como se puede apreciar, el simulador es capaz de cargar una gran cantidad de elementos en un tiempo muy pequeño, pudiendo cargar miles de elementos en cuestión de segundos.

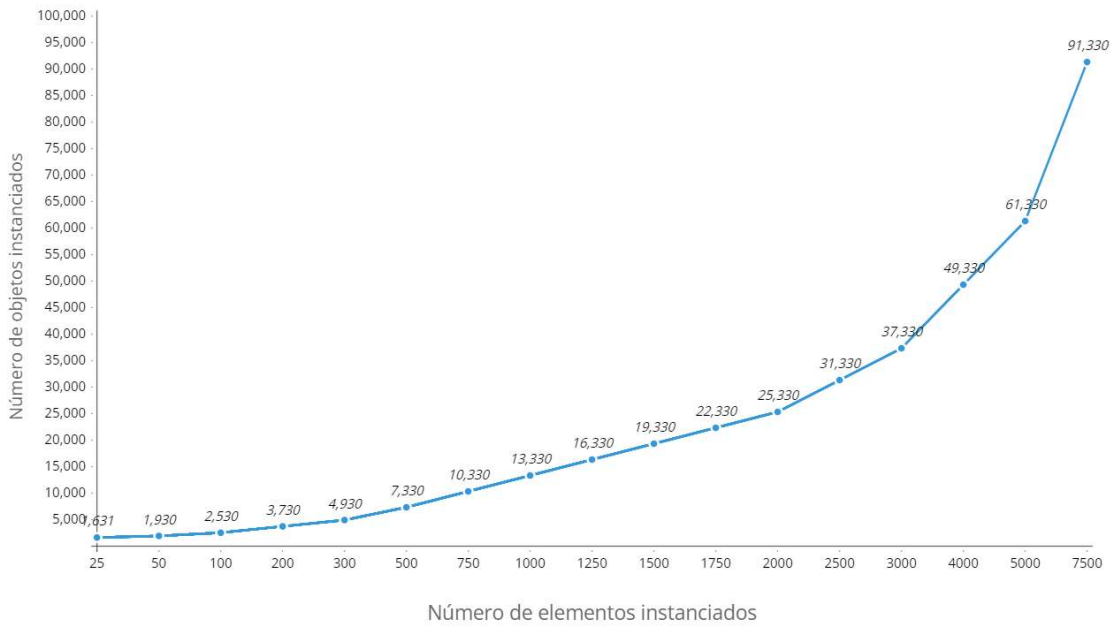

Imagen 27: Medida de objetos instanciados

En la gráfica de la imagen 27, podemos ver que el número de objetos instanciados crece con el número de elementos instanciados. Analizando el aumento del número de objetos respecto al número de elementos instanciados, podemos ver que cada elemento de la simulación contiene internamente 12 objetos.

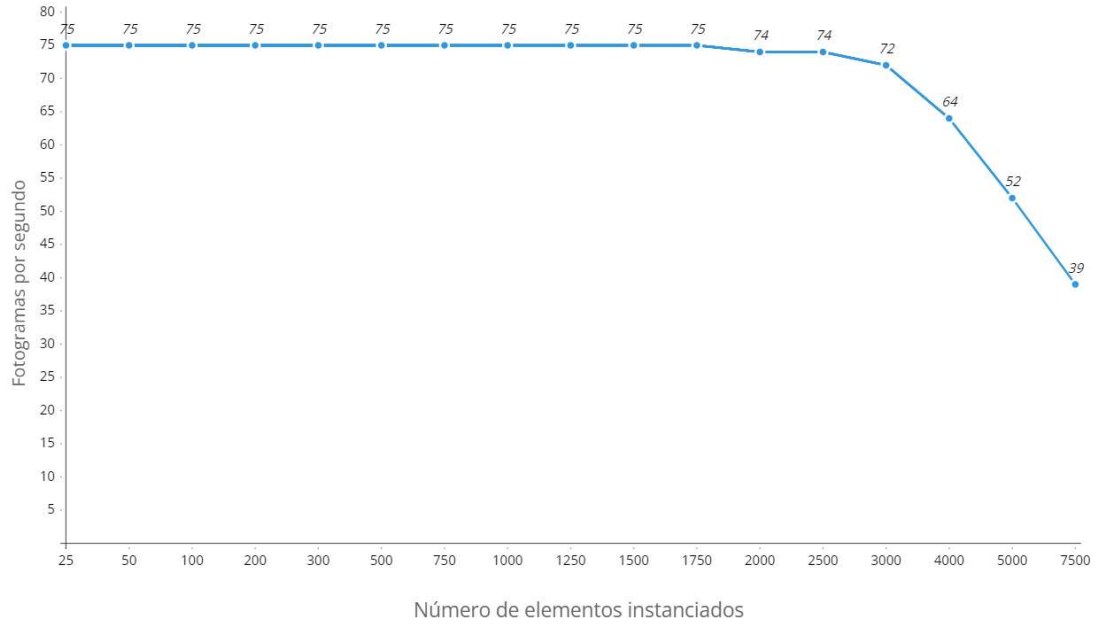

Imagen 28: Medida de fotogramas por segundo

Como podemos ver en la imagen 28, el número de fotogramas por segundo se mantiene estable hasta que el número de elementos instanciados se acerca a 2000, tras lo que empieza a reducirse. Aun así, podemos ver que, para una simulación con 7500 elementos, el número de fotogramas por segundos se sitúa en unos 39, lo cual permite observar la simulación de forma cómoda y sin problemas.

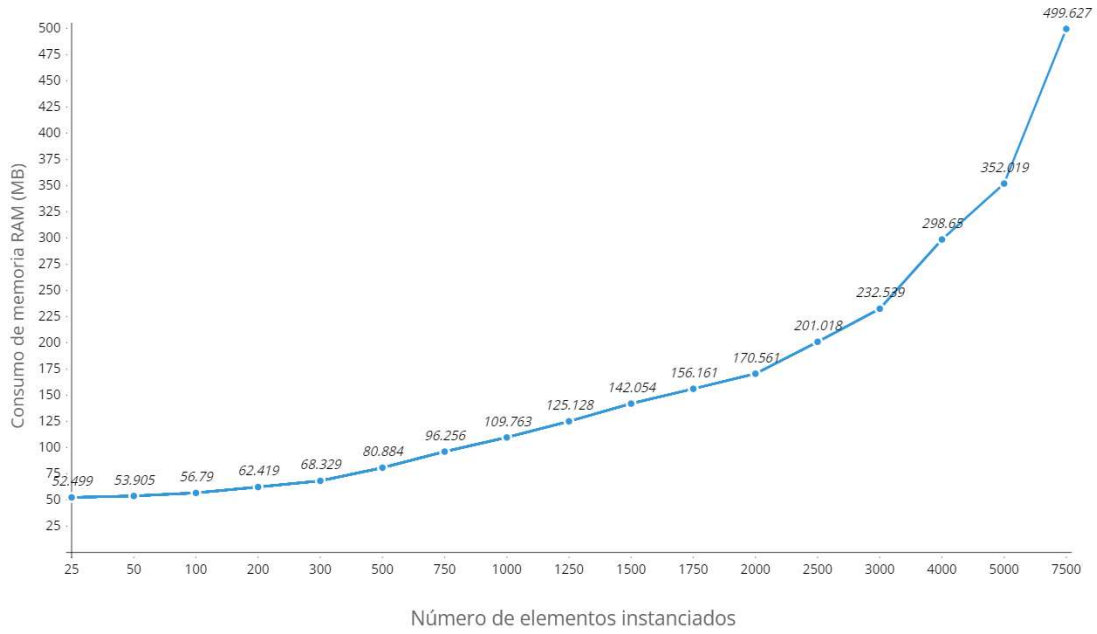

Imagen 29: Medida de consumo de RAM

Como podemos apreciar en la imagen 29, el consumo de memoria RAM por el simulador no es muy excesivo. Empieza en 52.499 MB con 25 elementos, ocupando la mayoría de la memoria los archivos de Godot, y como podemos apreciar, crece de manera bastante lenta. Si calculamos la diferencia de consumo de memoria entre dos valores contiguos de la gráfica y lo dividimos entre la diferencia de elementos, podemos ver que los elementos de la simulación ocupan muy poco espacio en memoria, alrededor de unos 55 KB.

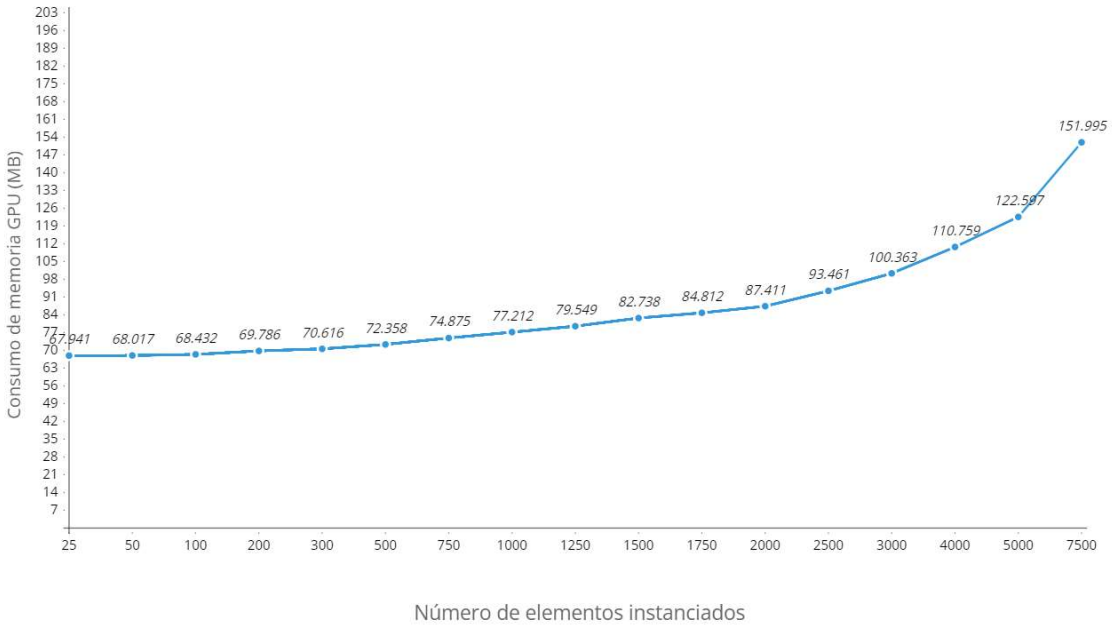

Imagen 30: Medida de consumo de GPU

El consumo de GPU, ilustrado en la imagen 30, crece de una forma muy lenta, comparado con el consumo de memoria RAM. Esto se debe a que los elementos utilizados en la simulación son árboles con un aspecto bastante simple, lo que reduce el impacto de los elementos en la GPU.

6.2 Comparativa FIVE y FIVE-Godot

En este apartado compararemos el rendimiento de FIVE-Godot con el de FIVE. Para ello, ejecutaremos dos simulaciones en ambos simuladores, y analizaremos su rendimiento. Las métricas en FIVE se obtendrán usando la herramienta de monitoreo integrada en Unity. Para probar la naturaleza distribuida del simulador, los agentes se ejecutarán en una máquina separada, utilizando el servidor XMPP público jabbers.one[15] para establecer la comunicación entre los agentes y el simulador.

Las métricas que se analizarán serán las mismas, a diferencia de la métrica de tiempo de carga. En su lugar, se medirá el tiempo de instanciación de los elementos de la simulación, ya que FIVE incluye funcionalidad para medir este tiempo.

6.2.1 Primer caso de prueba

El primer caso de prueba consistirá en una simulación de pequeño tamaño. En esta simulación, se generará un entorno con 60 elementos, divididos en 6 filas de 10 elementos, y se situará un agente entre cada fila, recorriendo dicha fila y tomando fotos

[15] https://jabbers.one/

de los elementos a su derecha. Las métricas de consumo de recursos se tomarán una vez se haya generado la simulación y los agentes hayan creado sus avatares.

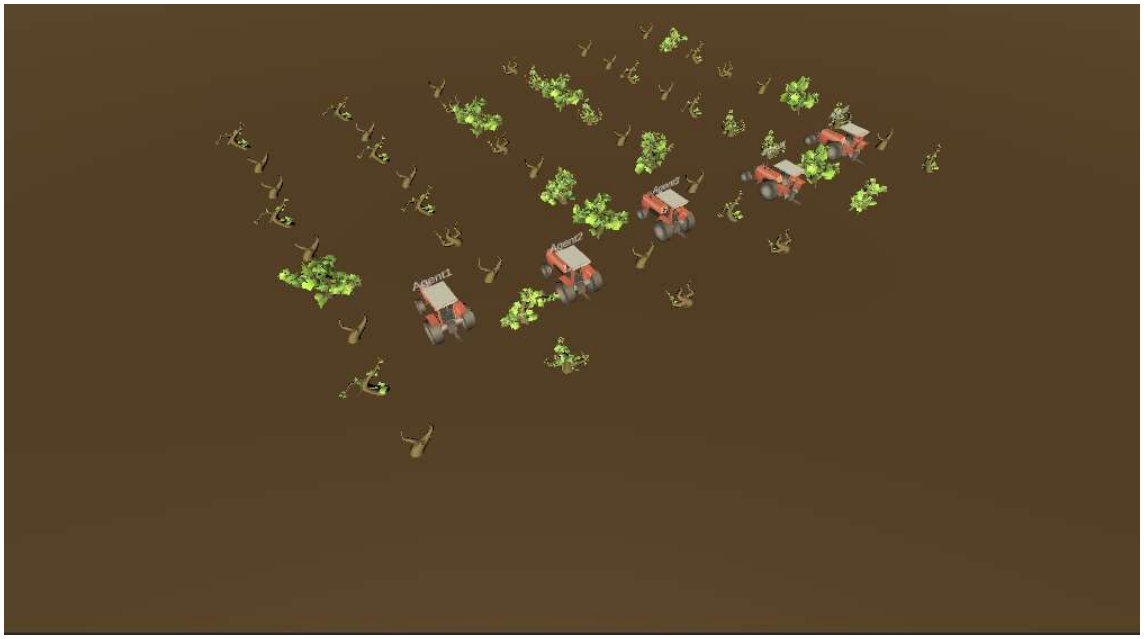

Imagen 31: Ejecución del primer caso de prueba en FIVE

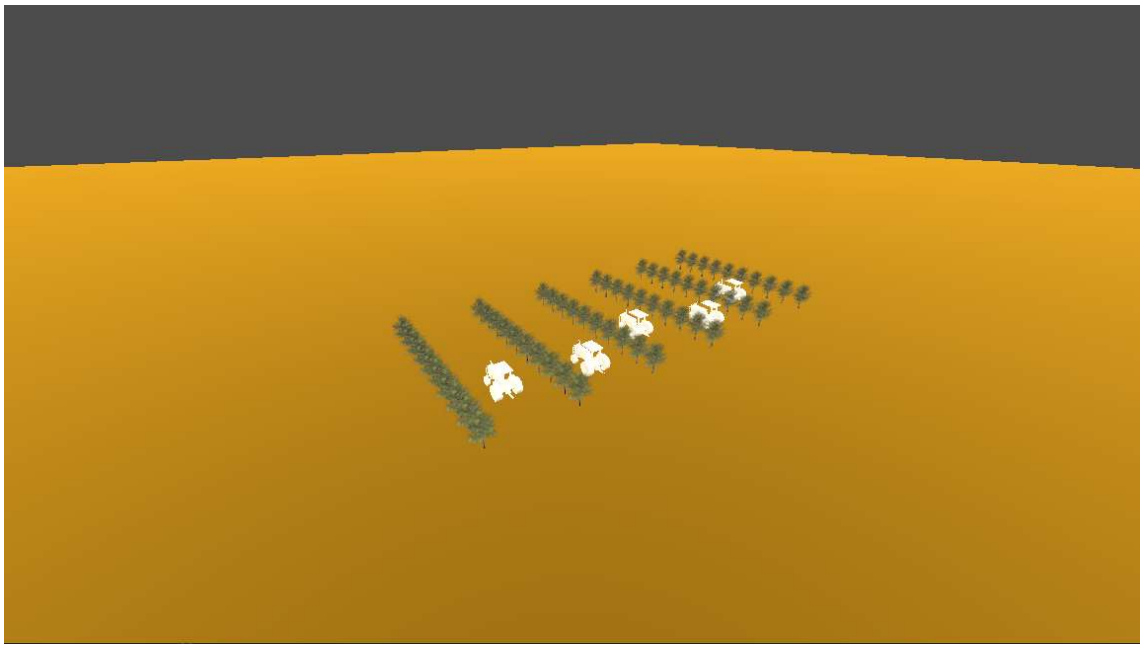

Imagen 32: Ejecución del primer caso de prueba en FIVE-Godot

Tras ejecutar la simulación tres veces, tal y como muestran las imágenes 31 y 32, se calcularon los valores medios de las métricas, obteniendo los resultados de la tabla 1.

Métrica	FIVE	FIVE-Godot
Tiempo de instanciación	260 ms	82 ms
Número de objetos	2178	2168
Consumo de memoria RAM	811.6 MB	55.4 MB
Consumo de memoria GPU	98.5 MB	147.2 MB
Fotogramas por segundo	60	75

Tabla 1: Comparativa de métricas en el primer caso de prueba

Como podemos ver, los resultados son bastante similares entre los dos simuladores, siendo el consumo de memoria RAM la métrica que más diverge entre ambos. Unity es un motor bastante más pesado que Godot y tiene requerimientos de *hardware* más elevados, lo que explicaría la diferencia entre el consumo de RAM.

6.2.2 Segundo caso de prueba

El segundo caso de prueba, mostrado en las imágenes 33 y 34, conllevará generar una simulación de grandes dimensiones, con un número mayor de elementos y agentes en ella. Para ello, se creará una simulación con 1000 elementos, divididos en 10 filas de 100 elementos, y se introducirán 9 agentes ejecutando el mismo comportamiento que en el caso de prueba anterior.

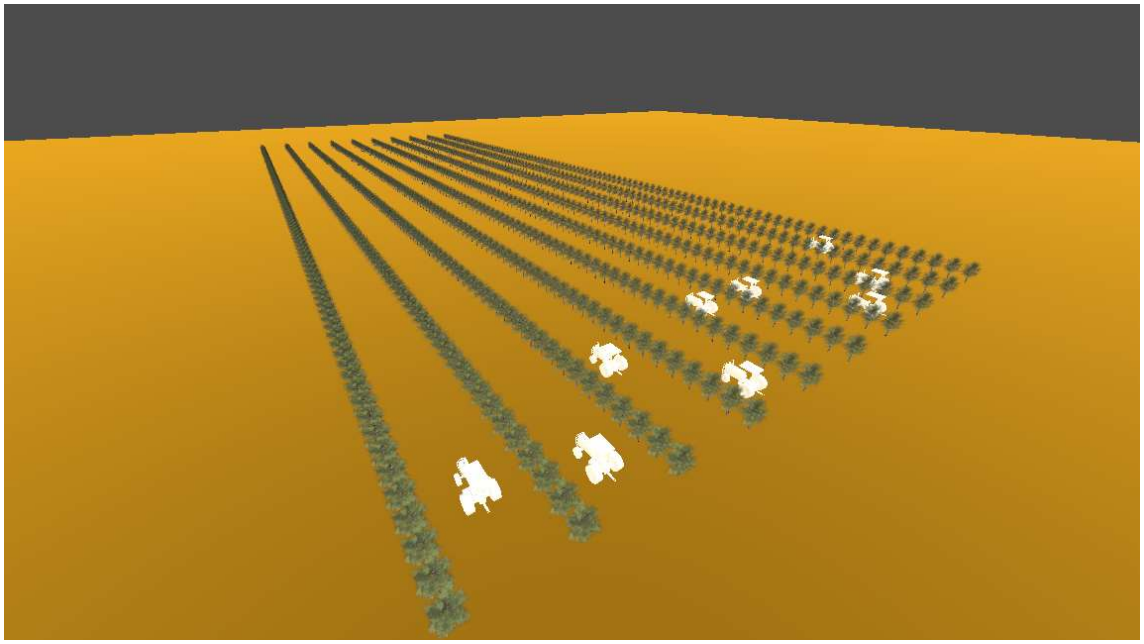

Imagen 33: Ejecución del segundo caso de prueba en FIVE-Godot

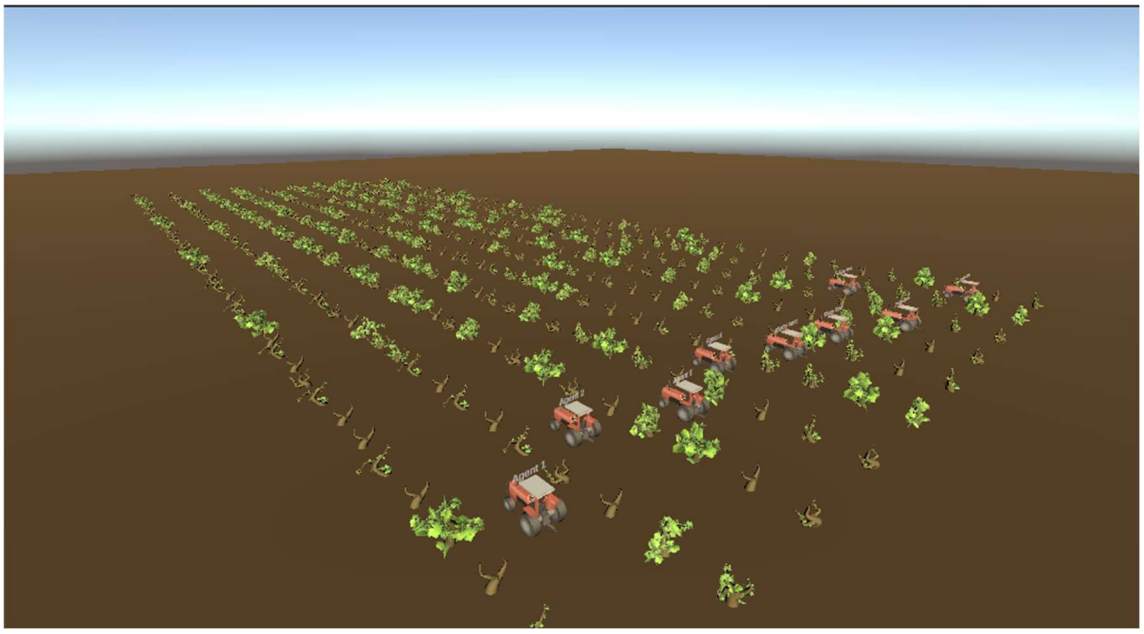

Imagen 34: Ejecución del segundo caso de prueba en FIVE

Tras ejecutar la simulación tres veces y calcular los valores medios, las métricas resultantes son las mostradas en la tabla 2:

Métrica	**FIVE**	**FIVE-Godot**
Tiempo de instanciación	406 ms	481 ms
Número de objetos	6655	13578
Consumo de memoria RAM	825.3 MB	108.8 MB
Consumo de memoria GPU	101.3 MB	251.4 MB
Fotogramas por segundo	60	75

Tabla 2: Comparativa de métricas en el segundo caso de prueba

Como podemos apreciar, los tiempos de instanciación son bastante similares entre ambos simuladores. De la misma forma, podemos comprobar que, aunque el consumo de memoria de FIVE es mayor que el de FIVE-Godot, la diferencia del consumo de memoria RAM entre ambos casos de prueba en FIVE es muy pequeña, lo cual nos lleva a pensar que el consumo aumentado de memoria de Unity se debe mayoritariamente a los componentes internos de Unity, no a la complejidad de la simulación.

En conclusión, aunque existen diferencias entre los tiempos de carga y el consumo de recursos entre FIVE y FIVE-Godot, estas son relativamente pequeñas, y no representan una mejora sustancial que lleve a desear usar uno de los simuladores por encima del otro.

Capítulo 7 - Conclusiones

Este capítulo resume las distintas tareas que se han llevado a cabo en este proyecto, así como los resultados obtenidos.

En el primer capítulo hemos introducido FIVE, su aplicación en la investigación de sistemas multi-agente y los cambios en las políticas de Unity que han llevado la creación de este proyecto, así como los objetivos que buscábamos cumplir.

En el segundo capítulo hemos analizado el estado y las debilidades de distintos simuladores de sistemas multi-agente actualmente disponibles, y hemos visto como FIVE utiliza la funcionalidad que ofrece el motor de videojuegos Unity para superar estas debilidades. También hemos visto otras tecnologías relevantes, incluyendo varios motores de videojuegos.

El tercer capítulo ha servido para analizar más profundamente FIVE, su funcionamiento y los elementos que lo componen. Hemos visto también los archivos que permiten configurar una simulación y el funcionamiento básico de los agentes que interactúan con el simulador. Finalmente, hemos visto los requisitos que FIVE-Godot debe cumplir para poder servir como alternativa a FIVE.

En el cuarto capítulo hemos analizado el diseño e implementación de FIVE-Godot. Hemos visto los distintos mánagers que componen el simulador y las responsabilidades de cada uno, así como el proceso de generación de una simulación y los archivos de configuración que requiere. Tras esto, hemos profundizado en la implementación de FIVE-Godot, viendo las distintas clases que lo forman y como se relacionan entre ello. Hemos cerrado este capítulo explicando el proceso para añadir nuevos elementos y agentes al simulador.

El quinto capítulo ha servido para discutir el lenguaje elegido para el desarrollo del simulador, así como las bibliotecas externas que se han utilizado. También se ha hecho una presentación de distintos problemas que surgieron durante el desarrollo, y cómo se solventaron.

Finalmente, en el sexto capítulo se ha mostrado el rendimiento del simulador, comprobando el consumo de recursos en distintas situaciones, así como el tiempo de generación de la simulación. Tras esto, se han hecho medidas de consumo en FIVE y FIVE-Godot, comparando los recursos utilizados por ambos simuladores.

Reflexionando sobre los objetivos que nos habíamos propuesto al principio de este proyecto, podemos concluir que hemos alcanzado los objetivos principales. FIVE-Godot ofrece las mismas capacidades básicas de FIVE, permitiendo usar los mismos ficheros de configuración de FIVE para crear y poblar simulaciones análogas a las que generaría FIVE. Los agentes SPADE que actualmente se utilizan en las simulaciones de FIVE pueden interactuar con las simulaciones de FIVE-Godot sin necesidad de aplicar ningún cambio a su programación, obteniendo los mismos resultados que obtendrían al interactuar con FIVE.

Lamentablemente, los objetivos secundarios no se han podido cumplir. La integración con ROS no ha sido posible ya que, a diferencia de Unity, Godot no posee

bibliotecas especializadas que le permitan trabajar con ROS. Debido a la complejidad de ROS, crear desde cero una integración de este tipo entre ambas plataformas es una tarea de gran tamaño, que queda fuera del ámbito de este proyecto. De forma parecida, la falta de tiempo ha evitado que se pudiera investigar la implementación de *artifacts* en FIVE-Godot.

7.1 Relación con los estudios cursados

El desarrollo de este trabajo ha requerido el uso de muchas de las capacidades y conocimientos adquiridos durante el estudio del Grado en Ingeniería Informática.

Por un lado, ha sido necesario aprender el uso de Godot, una herramienta de creación de videojuegos con la que no se tenía experiencia previa. Además, se ha requerido estudiar la aplicación FIVE y entender su funcionamiento, en muchas ocasiones analizando el código fuente para comprender las partes del simulador que no se pueden intuir simplemente viendo la simulación en ejecución, tales como el funcionamiento de los comandos. Esto demuestra capacidad para adaptarse a proyectos ya existentes, que utilizan herramientas para las que no se tiene experiencia previa.

Por otro lado, el diseño y desarrollo de FIVE-Godot prueba la capacidad para diseñar y llevar a término aplicaciones complejas. FIVE-Godot es una aplicación con múltiples partes, que trabajan juntas de forma coordinada para crear una simulación tridimensional. Además, la aplicación está implementada con una serie de clases que mantienen el principio de responsabilidad única, lo que facilita la expansión y ampliación del simulador.

Finalmente, el problema encontrado en la arquitectura de agentes de FIVE, explicado en el apartado 4.4, demuestra que el autor no solo es capaz de entender el funcionamiento de la aplicación, si no también prever problemas que pueden surgir en el futuro y buscar formas de solventar, o en este caso evitar, dichos problemas.

7.2 Trabajo futuro

Las líneas de trabajo futuro de este proyecto se orientan principalmente a mejorar el simulador, especialmente añadiendo la funcionalidad de FIVE que no se ha podido implementar en FIVE-Godot, así como formas de facilitar el traslado a FIVE-Godot de elementos nuevos de FIVE. Estas líneas incluyen:

- Implementación de *artifacts* que puedan ser usados por los agentes en la simulación.
- Analizar la viabilidad de integrar Godot y ROS. Actualmente, existe una librería mantenida por la empresa Siemens, Ros#[16], que permite la comunicación entre

[16] https://github.com/siemens/ros-sharp

ROS y aplicaciones desarrolladas en C# y que se perfila como un buen punto de partida para esta línea de trabajo.

- Investigar la creación de un sistema que permita transformar archivos de Unity en archivos de Godot. Aunque ya existen algunos proyectos que permiten esta funcionalidad, tales como *UnityToGodot*[17] o *unity_to_godot_converter*[18], aún se encuentran en un estado muy experimental, y en muchos casos los resultados son erróneos o inutilizables.
- Actualizar el simulador para que utilice los diccionarios tipados, tal y como se menciona en el apartado 5.3.2, una vez se introduzcan en Godot.

[17] https://github.com/Anthogonyst/UnityToGodot

[18] https://github.com/Zylann/unity_to_godot_converter

Bibliografía y referencias

Javier Enguix, F. (2022). *Desarrollo de un generador de simulaciones en Unity 3D.* Valencia: Universidad Politécnica de Valencia.

Miguel Gregori, J. P. (2006). A jabber-based multi-agent system platform. *Proceedings of the International Conference on Autonomous Agents.*, 1282-1284.

Stuart J. Russell, P. N. (1995). *Artificial Intelligence: A Modern Approach.* Upper Saddle River, N.J.: Prentice Hall.

Yoav Shoham, K. L.-B. (2009). *Multiagent Systems. Algorithmic, Game-Theoretic, and Logical Foundations.* Cambridge: Cambridge University Press.

Modelo de árbol utilizado en la simulación: tree low-poly, creado por Ricardo Sánchez, distribuido con licencia Creative Commons Attribution

Apéndices

Apéndice 1: Ejemplo de fichero map.json

```
{
    "lights": [
        {
            "active": true,
            "objectName": "Sun Light",
            "objectPrefabName": "Light",
            "position": {
                "x": 0.0,
                "y": 3.0,
                "z": 0.0
            },
            "rotation": {
                "x": 35.0,
                "y": 40.0,
                "z": 0.0
            },
            "color": {
                "r": 1.0,
                "g": 0.95,
                "b": 0.83,
                "a": 1.0
            },
            "intensity": 2.0
        }
    ],
    "objects": [
        {
            "active": true,
            "objectName": "Spawner 0",
            "objectPrefabName": "Spawner",
            "position": {
                "x": 3.5,
                "y": 0.0,
                "z": 0.0
            },
            "rotation": {
                "x": 0.0,
                "y": 0.0,
                "z": 0.0
            }
        },
        {
            "active": false,
            "objectName": "Tree 1",
            "objectPrefabName": "Tree",
            "position": {
                "x": -2.6,
                "y": 0.0,
                "z": 0.0
            },
            "rotation": {
                "x": 0.0,
                "y": 0.0,
                "z": 0.0
            }
        }
    ]
}
```

Imagen 35: Ejemplo de fichero de configuración map.json

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**			X	
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**			X	
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.				X
ODS 12. **Producción y consumo responsables.**			X	
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**	X			

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Los ODS más relevantes con el proyecto desarrollado son el ODS 17 (Alianzas para lograr objetivos) y el ODS 9 (Industria, innovación e infraestructuras). En menor grado también está relacionados el ODS 8 (Trabajo decente y crecimiento económico), así como el ODS 4 (Educación de calidad) y el ODS 12 (Producción y consumo responsables).

El ODS 17 se relaciona con el trabajo debido a la capacidad del FIVE-Godot de funcionar de manera descentralizada, habitando en un entorno separado a los agentes que poblarán la simulación, lo cual permite la colaboración de equipos de investigación en distintas zonas del mundo.

El ODS 9 se relaciona con este trabajo ya que tiene una aplicación directa sobre la investigación de sistemas multi-agente, debido a que el objetivo de este proyecto es que FIVE-Godot pueda ser utilizado por la línea de investigación activa del VRAIN (Valencian Research Institut for Artificial Intelligence) como sustituto al simulador FIVE, en caso de que sea necesario. De la misma forma, dado que esta línea de investigación está estudiando las posibilidades de usar sistemas multi-agentes para crear plantaciones agrarias automatizadas, se podría ver una pequeña relación con el ODS 8.

EL ODS 4 está relacionado con el proyecto debido a que la sencillez de uso del simulador y la facilidad para añadir nuevos agentes y elementos de simulación permite que el simulador pueda ser usado por docentes para enseñar las materias relacionadas con agentes inteligentes. La facilidad para instalar el simulador y los bajos requisitos de *hardware* permiten que pueda usarse en laboratorios y ordenadores de bajas prestaciones

Finalmente, el ODS 12 está relacionado en un pequeño grado con el proyecto, ya que el uso de la aplicación para estudiar formas de organización de sistemas multi-agentes permite hacer experimentos en un entorno virtual con costes muy reducidos, permitiendo evitar los costes de preparación de los experimentos en entornos reales hasta que haya un alto nivel de confianza en que resultarán exitoso