UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Desarrollo de un generador de simulaciones en Unity 3D para sistemas multi-agente basados en SPADE

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Enguix Andrés, Francisco

Tutor/a: Carrascosa Casamayor, Carlos

Director/a Experimental: RINCON ARANGO, JAIME ANDRES

CURSO ACADÉMICO: 2021/2022

Resum

Les simulacions gràfiques poden ser una manera senzilla de validar el comportament d'implementacions de sistemes multi-agents. Sobretot si hi ha un mètode senzill de generar i de modificar aquestes simulacions. Aquest és l'objectiu del treball presentat. El *framework* desenvolupat permet definir fàcilment un entorn virtual intel·ligent a Unity, habitat per un sistema multi-agent compost per agents SPADE. Després de detallar com definir una nova simulació, aquest treball presenta un exemple on una nova àrea intel·ligent adaptativa és simulada per validar la detecció de malalties en taronges per una xarxa neuronal profunda.

Paraules clau: sasamas, simulador, entorn virtual, ive, mas, agents

Resumen

Las simulaciones gráficas pueden ser una manera sencilla de validar el comportamiento de implementaciones de sistemas multi-agentes. Sobretodo si hay un método sencillo de generar y modificar estas simulaciones. Este es el objetivo del trabajo presentado. El *framework* desarrollado permite definir fácilmente un entorno virtual inteligente en Unity, habitado por un sistema multi-agente compuesto por agentes SPADE. Después de detallar cómo definir una nueva simulación, este trabajo presenta un ejemplo donde una nueva área inteligente adaptativa es simulada para validar la detección de enfermedades en naranjas por una red neuronal profunda.

Palabras clave: sasamas, simulador, entorno virtual, ive, mas, agentes

Abstract

Graphical simulations can be an easy way to validate the behaviour of multi-agent system (MAS) implementations. Moreover if there is an easy method of generating and modifying such simulations. This is the goal of the work presented. This framework allows to easily define an Intelligent Virtual Environment (IVE) in Unity, inhabited by a MAS composed of SPADE agents. After detailing how a new simulation is defined, this document presents an example where a new adaptive smart area is simulated to validate a deep neural network to detect diseases in oranges.

Key words: sasamas, simulator, virtual environment, ive, mas, agents

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

El siguiente trabajo está formado por un sistema diseñado por elaboración propia, al que hemos decidido nombrar SASAMAS (*Simulator of Adaptive Smart Areas based on SPADE Multi-Agent Systems*) y cuyo objetivo principal es permitir probar, de manera sencilla y rápida, sistemas multi-agente en un paso previo al despliegue de los experimentos en el entorno real. SASAMAS es un marco de trabajo que está compuesto por dos bloques. El primer bloque consiste en un simulador, fácilmente editable, que simula un entorno tridimensional que permite la interacción con agentes inteligentes que pueblan este entorno. El otro bloque consiste en un sistema que permite establecer el comportamiento que tendrán los distintos agentes inteligentes. En este trabajo empezaremos describiendo su arquitectura, así como los módulos que lo conforman. A continuación explicaremos cómo podemos programar el comportamiento de los agentes y cómo estos se configuran. Después mostraremos cómo podemos definir un entorno de simulación, y finalmente realizaremos un caso de ejemplo con un huerto de naranjas habitado por agentes inteligentes capaces de recorrer el huerto y hacer fotos a las naranjas, donde también pondremos a prueba una red neuronal profunda cuyo propósito es reconocer si las naranjas padecen alguna enfermedad.

1.1 Motivación

Las simulaciones gráficas siempre han sido un método de prueba y validación de aplicaciones. Comprobar de forma cualitativa si una simulación funciona correctamente puede ahorrar horas de trabajo comprobando tablas de números. El principal problema con estas simulaciones es que, por lo general, cuestan mucho tiempo de construir o incluso de ajustar para un algoritmo específico. Es por este motivo que hemos decidido crear un sistema que proporcione una solución eficaz y flexible a este problema.

SASAMAS también está vinculado con una de las **líneas de investigación activas** sobre sistemas multi-agente del VRAIN (*Valencian Research Institute for Artificial Intelligence*)[1], es decir, el instituto universitario valenciano de investigación en inteligencia artificial. En concreto, está relacionado con un **proyecto coordinado a nivel nacional**, que ha sido preconcedido por el Ministerio de Ciencia e Innovación, con el título de Servicios inteligentes Coordinados para Áreas Inteligentes Adaptativas (*COSASS-coordinated intelligent services for adaptative smart areas*). En este sentido, el uso de la herramienta desarrollada permitirá centrar el esfuerzo de los investigadores en el estudio de los algoritmos, en vez de invertir horas en la construcción y modificación de la simulación, así como la propia programación de los agentes.

[1] https://vrain.upv.es/

Otro motivo que ha incitado al desarrollo de SASAMAS es la **aportación al entorno científico**. A raíz de este trabajo realizamos un artículo científico que hemos enviado a la conferencia internacional PRIMA (*Principles and Practice of Multi-Agent Systems*)[2], donde actualmente se encuentra bajo revisión para ser publicado.

1.2 Objetivos

El objetivo principal consiste en desarrollar un marco de trabajo que permita crear una simulación en tres dimensiones, donde agentes inteligentes que pueblan el entorno tridimensional permitan probar diversos algoritmos de sistemas multi-agente, y también se puedan realizar cambios sobre la simulación de manera rápida y muy sencilla.

Los objetivos secundarios en los que deriva el objetivo principal son:

- Crear un sistema de **comunicaciones por red** para que los agentes se puedan ejecutar de forma desacoplada al simulador, es decir, en distintas máquinas y distintas redes locales. De esta manera también permitimos la **colaboración remota** desde distintos lugares geográficos del mundo.
- Crear una **API** (*Application Programming Interface*) que pueda utilizar el usuario para que la programación de agentes sea rápida y disponga de una manera sencilla de **interactuar con el simulador**.
- Crear una herramienta de **edición de mapas** basada en modificar archivos de texto para crear simulaciones de manera rápida y sencilla. De esta manera podremos **modificar la simulación** con un terminal o consola, prescindiendo de la necesidad de controlar la máquina donde se encuentre el simulador con un programa visor de escritorio remoto.
- Crear un sistema que permita **cargar imágenes como texturas** de elementos en la simulación en tiempo de ejecución, y que este sistema esté optimizado para que la carga del simulador sea apto con un alto número de elementos. Este punto es muy importante para aportar **agilidad y flexibilidad a los experimentos**, ya que evita tener que modificar el programa del simulador y compilarlo cada vez que queremos incluir nuevas texturas en los elementos actuales.

1.3 Estructura de la memoria

La estructura de la memoria ha sido realizada para partir de una base de conocimientos que nos permita entender el funcionamiento y propósito de todas las partes sobre las que está compuesto el marco de trabajo desarrollado, mostrando también la ventaja de velocidad y flexibilidad que proporciona SASAMAS frente a utilizar las piezas de las que se compone por separado.

El primer capítulo consiste en una **introducción al proyecto**, mostrando de forma global en qué consiste este trabajo y qué es la herramienta que se ha desarrollado. En este capítulo también podemos encontrar el motivo que incentiva la creación de SASAMAS, así como los objetivos que cubre la herramienta. Finalmente, como se ha utilizado un caso real propuesto por los tutores para probar el buen funcionamiento del marco de trabajo, también se ha incluido una sección adicional para mencionar los detalles de esta colaboración.

[2] https://prima2022.webs.upv.es/

El segundo capítulo contiene las **definiciones** necesarias para entender los conceptos teóricos que se tratan en este trabajo. También encontraremos **otros simuladores** de sistemas multi-agente ya existentes y se compararán las funcionalidades que proporcionan con SASAMAS. A continuación, expondremos la tecnología que subyace en el marco de trabajo desarrollado. Finalmente, veremos qué **oportunidades de innovación** existen y la solución que proponemos para explotarlas adecuadamente.

El tercer capítulo muestra la **arquitectura** que compone el marco de trabajo desarrollado. Primero veremos, de forma global, las partes que componen SASAMAS y cuáles son los archivos de configuración que nos permiten iniciar una simulación. A continuación, expondremos el diseño del marco de trabajo de manera mucho más extensa y técnica. Profundizaremos en cada módulo que conforma la arquitectura, con el apoyo visual de diagramas de clases y figuras. Finalmente, concluiremos el capítulo mostrando cómo funciona el protocolo de red que utilizan los agentes y el simulador para llevar a cabo la comunicación.

El cuarto capítulo trata el **desarrollo de la solución**, empezando por ver en qué consiste la propuesta inicial y terminando en la exposición de los problemas encontrados durante la implementación del marco de trabajo y las soluciones que han sido realizadas para abordar los problemas.

El quinto capítulo consiste en la **implantación** del sistema desarrollado, es decir, habla de los pasos que debemos realizar para desplegar cada una de sus partes en un entorno de producción. Este capítulo incluye las instrucciones para la puesta en marcha de los agentes, del simulador y también del servidor XMPP en los que los agentes basan su comunicación.

El sexto capítulo contiene el **caso de prueba**, donde se utilizará el marco de trabajo mostrar la construcción de un entorno habitado por un agente. Además, el agente está equipado con una red neuronal profunda con el objetivo de usar la simulación para probar su buen funcionamiento. De esta manera, usaremos la red para clasificar las imágenes del entorno que toma el agente con la cámara que lleva integrada. Finalmente, realizaremos una **modificación** sobre el entorno y el agente para ilustrar la sencillez y rapidez que SASAMAS brinda a sus usuarios.

El cierre del trabajo ocurre en la **conclusión**. En este último capítulo hablaremos sobre si hemos alcanzado los objetivos inicialmente planteados y de qué manera se han logrado abordar. Finalmente, también encontraremos una sección que trata el **trabajo futuro** que abre SASAMAS, donde mostraremos algunas líneas de trabajo futuro cuya realización resulta interesante explotar para mejorar este marco de trabajo.

1.4 Colaboraciones

El caso de prueba que se muestra en el capítulo seis es un caso real de una de las líneas de investigación de sistemas multi-agente del VRAIN. Este caso de prueba consiste en crear un agente inteligente que habita un huerto de naranjos y lo recorre, tomando imágenes sobre las naranjas para posteriormente identificar si la fruta padece una enfermedad o está sana. Fruto de esta línea se realizó la red neuronal profunda que se pone a prueba en el capítulo seis (y de la que es responsable el tutor, experto en inteligencia artificial, Dr. Jaime Andrés Rincón).

CAPÍTULO 2
Contexto tecnológico

2.1 Introducción

Este capítulo contiene las **definiciones** necesarias para entender los conceptos teóricos que se tratan en este trabajo. También encontraremos **otros simuladores** de sistemas multi-agente ya existentes y se compararán las funcionalidades que proporcionan con SASAMAS. A continuación, veremos la tecnología que subyace en el marco de trabajo desarrollado y se estudiará el porqué SASAMAS proporciona mayores beneficios al usuario que la suma de las partes sobre las que se compone. Finalmente, veremos qué **oportunidades de innovación** existen y la solución que proponemos para explotarlas adecuadamente.

2.2 Definiciones

Los **agentes inteligentes** [1] son piezas de *software* o *hardware* que, a diferencia de los agentes simples como puede ser un sistema de riego basado en un sensor de humedad, funcionan de manera autónoma y fiable en un entorno impredecible y cambiante. Los agentes inteligentes son capaces de percibir el entorno en el que se encuentran y en función de la percepción que reciben, realizan acciones de forma autónoma que pueden alterar el entorno para lograr sus objetivos.

Los **sistemas multi-agente** [2] están formados por agentes inteligentes que interactúan entre sí y modelan organizaciones computacionales complejas. Los agentes que lo conforman son capaces de negociar, delegar y coordinar actividades. Además, los sistemas multi-agente presentan una serie de ventajas como la eficiencia, gracias a la distribución de tareas entre los agentes; la fiabilidad, ya que al estar compuesto de un conjunto de unidades se pueden replicar agentes y reemplazarse en caso de fallo; la flexibilidad, ya que se puede modificar el número de agentes de forma dinámica y la modularidad, que se obtiene al trabajar con unidades más pequeñas.

Por otro lado, podemos encontrar lo que se denomina un **IVE** (*Intelligent Virtual Environment*) [3], es decir, un entorno virtual que simula un mundo real habitado por agentes inteligentes, donde pueden interactuar y su comportamiento puede ser fácilmente validado.

La **red neuronal profunda** que utilizaremos en el caso de prueba está basada en la red MobileNetV2 [4], sucesora de la red MobileNetV1 [5]. La arquitectura de MobileNetV2 se basa en el uso de *inverted residual blocks*, es decir, bloques residuales invertidos. El concepto de *inverted residual block* está propuesto por Sandler et al. [4] y consiste en usar capas delgadas en la entrada y en la salida del bloque residual, este enfoque opuesto al

tradicional permite mejorar la eficiencia en las redes neuronales usadas en el procesado de imágenes.

2.3 Otros simuladores de sistemas multi-agente

En esta sección mostraremos cuatro simuladores de sistemas multi-agente que representan las distintas opciones que se encuentran disponibles actualmente. Posteriormente, en la sección de oportunidades de innovación (2.5), detallaremos las novedades que incluye la herramienta desarrollada en este trabajo respecto a los demás simuladores existentes.

Tradicionalmente, los simuladores que incluyen conceptos de agentes simulan, no solo el entorno, sino también a los agentes que incluye. Este es el caso, por ejemplo, de **Netlogo** [6], donde los agentes habitan un entorno similar a una matriz formada por celdas. No obstante, este simulador está limitado a cuatro tipos de agentes diferentes, el entorno simulado es bidimensional y además no permite el desacoplamiento de sus partes, donde la configuración de este sistema monolítico se produce en un mismo archivo.

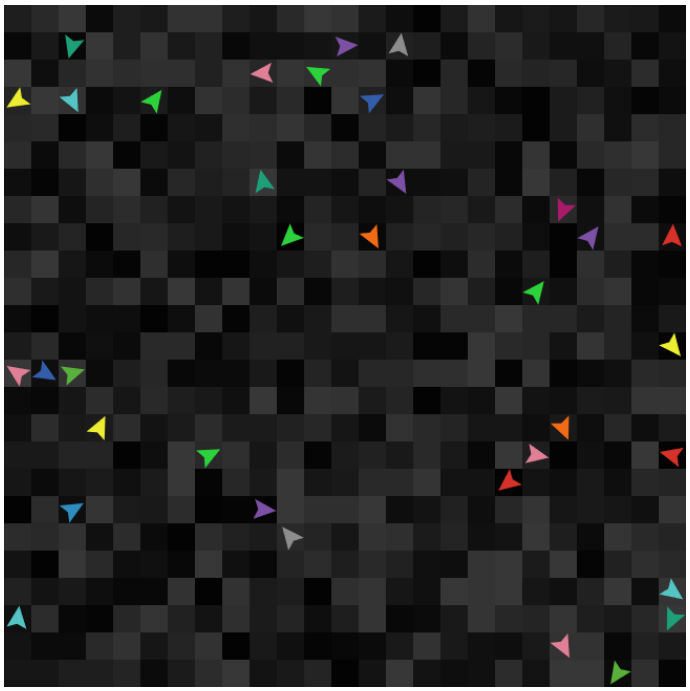


Figura 2.1: Ejemplo de visualización de Netlogo con agentes tortuga y celdas.

La figura 2.1 muestra una simulación realizada con Netlogo donde varios agentes de tipo tortuga se encuentran en un entorno compuesto por celdas cuadradas. Esta simulación está programada con la restricción de que dos agentes tortuga no pueden habitar una misma celda.

JaCalIVE (*Jason Cartago implemented Intelligent Virtual Environment*) [7] puede verse como un ejemplo de un marco para desarrollar un MAS (*Multi-Agent System*) que habita un IVE. Este marco se basa en el metamodelo MAM5 [8]. La idea detrás de esto es definir una simulación por medio de dicho metamodelo, que se compila en algunas plantillas de agentes de Jason [1] y artefactos de CaRTago [2] para que los complete el desarrollador de la

[1] https://github.com/jason-lang/jason
[2] http://cartago.sourceforge.net/

simulación. Entonces, este marco tiene un proceso de desarrollo muy formal, pero no es fácil crear una nueva simulación o incluso hacer cambios en una existente.

También podemos encontrar **MASON** (*Multi-Agent Simulator Of Neighborhoods*) [9], realizado puramente en el lenguaje de programación Java y con fecha de salida en 2003. Este simulador está principalmente orientado a realizar simulaciones de inteligencia en enjambre y de sistemas multi-agente. Además, permite elegir un espacio discreto o continuo en las simulaciones y visualizar el resultado en un espacio de dos o tres dimensiones (ver Figura 2.2). No obstante, conseguir una visualización tridimensional en este simulador no es sencillo ni rápido y requiere la instalación adicional de las librerías de Java3D y conocimientos de programación en Java.

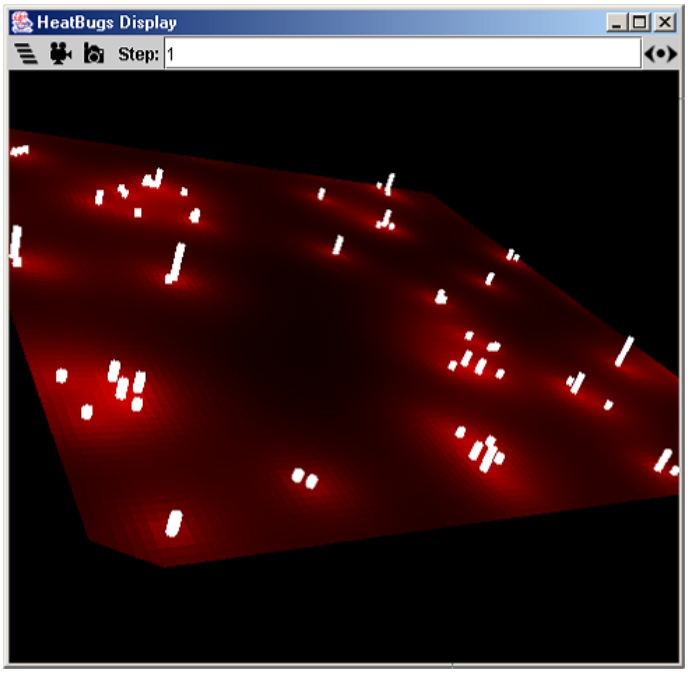


Figura 2.2: Ejemplo de visualización tridimensional de *heatbugs* en Mason, extraída del enlace: **https://cs.gmu.edu/ eclab/projects/mason/3DWindow.png**.

BimSim3D (*Building Information Model Simulator*) [10], es decir, un simulador de modelos de información en edificios. Está centrado en la simulación tridimensional de agentes en un edificio, haciendo uso de Unity como motor gráfico y sistemas multi-agente basados en el comportamiento humano. El propósito de este simulador no es generalista, sino que tiene como objetivo generar datos de un comportamiento humano y realista dentro de un edificio modelado. De esta manera, se puede usar la información sintetizada para realizar inferencia sobre el comportamiento real que tendrán los habitantes del edificio sin vulnerar la privacidad de las personas.

En esta muestra podemos ver que todos los simuladores tienen una o varias ventajas que han sido explotadas, pero a pesar de esto no se adaptan correctamente a la funcionalidad de probar los algoritmos en un paso previo al despliegue en el mundo real y que al mismo tiempo construir estas simulaciones sea un proceso sencillo, rápido y desacoplado.

Hemos visto simuladores que funcionan correctamente en un entorno bidimensional, pero después no permiten probar los algoritmos en un entorno de tres dimensiones, que es más apropiado en un paso previo al despliegue del experimento en el mundo real. Por otra parte, también podemos encontrar simuladores que sí permiten probar los algoritmos desarrollados en un entorno tridimensional, pero la construcción de estas simulacio-

nes requiere conocimientos avanzados de programación y son complejas y costosas de construir.

Finalmente, también hemos visto el caso de BimSim3D, que es un simulador que modela un entorno tridimensional para obtener datos que se usarán en el sistema real, pero en vez de tomar una aproximación generalista, está especializado en realizar simulaciones de agentes inteligentes que habitan edificios.

2.4 Tecnología del marco de trabajo SASAMAS

2.4.1. XMPP

XMPP (*Extensible Messaging and Presence Protocol*) [3] es un protocolo de mensajería instantánea basado en XML (*Extensible Markup Language*) que funciona a nivel de aplicación. Donde una de sus principales características es que en vez de tratarse de un protocolo propietario, es un protocolo abierto y extensible que permite aplicar capas de cifrado como SASL (*Simple Authentication and Security Layer*) y TLS (*Transport Layer Security*), y también permite la descentralización, ya que se pueden configurar servidores XMPP propios donde se comunicarán los agentes.

2.4.2. El motor gráfico: Unity

Unity [4] es un motor gráfico desarrollado en el año 2005, y está principalmente diseñado para crear videojuegos en dos y tres dimensiones, aunque también es usado en otros sectores como la creación de películas, la arquitectura, la industria aeroespacial y el diseño de vehículos [11] [12]. Es por este motivo que su utilización resulta idónea en la programación del simulador, ya que permite construir escenarios con precisión y brinda al usuario del simulador la posibilidad de crear sistemas complejos y detallados. Además, cuenta con un sistema de física integrada en el propio motor, que permite realizar ajustes de manera sencilla, como modificar el valor de la gravedad.

El lenguaje de programación que emplea el usuario de Unity es C#. Este lenguaje ha sido desarrollado por Microsoft, aunque no es cerrado a su sistema operativo Windows, ya que también **puede ser usado en otros sistemas operativos** como los basados en Linux, Android y macOS. C# es multi-paradigma y de propósito general, que son características que favorecen su versatilidad.

También es uno de los motores gráficos que más se emplean de la industria de los videojuegos [13], lo que favorece a que exista abundante documentación en internet para poder utilizarlo.

2.4.3. SPADE

SPADE (*Smart Python Agent Development Environment*) [14] es un marco para desarrollar agentes inteligentes en Python, cuya característica principal es el uso de XMPP como protocolo de mensajería instantánea. En los sistemas multi-agente la comunicación entre agentes es primordial y los agentes SPADE cuentan con un despachador de mensajes integrado que permite la comunicación entre estos.

El modelo de agente SPADE **se basa en comportamientos**, donde los comportamientos son tareas que se repiten en base a un patrón de tiempo determinado, teniendo no

[3] https://xmpp.org/
[4] https://unity.com

solo del tipo one-shot, periódicos o FSM (*Finite State Machine*), sino también un comportamiento BDI (*Belief Desaire Intention*) [15], que permite tener comportamientos reactivos y deliberativos en el agente.

Por otra parte, SPADE es un sistema que se encuentra en auge, recibiendo trescientas sesenta y nueve descargas en enero de dos mil veinte [14]. En su tercera versión (que es la usada en este proyecto) se hizo una reestructuración del proyecto [14] y actualmente es un sistema robusto y formal que permite la programación de agentes inteligentes.

2.5 Oportunidades de innovación

Las oportunidades de innovación que aparecen en este trabajo están basadas en explotar las carencias de simuladores de entornos virtuales inteligentes, como los que hemos podido ver en la sección 2.3.

Actualmente no existe un sistema en tres dimensiones, flexible y distribuido que permita simular un entorno virtual inteligente habitado por agentes inteligentes SPADE, que al mismo tiempo sea muy sencillo de crear y modificar, y que permita el desacoplamiento de sus partes para fomentar la colaboración desde distintos lugares geográficos del mundo.

La investigación en algoritmos basados en sistemas multi-agente se beneficiará del uso de una herramienta que cumple con estas características y en concreto será usado por una línea de investigación activa de sistemas multi-agente del VRAIN, como hemos señalado en la sección 1.1.

2.6 Solución propuesta

La propuesta de este trabajo busca explotar las oportunidades de innovación mencionadas en la sección 2.5, desarrollando una herramienta que proporcione una manera fácil y rápida de definir un entorno virtual inteligente, en tres dimensiones y habitado por agentes inteligentes. Este IVE se formará en Unity, pero el usuario del marco de trabajo no tendrá que programar los detalles de implementación, ya que el propio sistema se los proporcionará listos para su uso. Por otra parte, los agentes serán agentes SPADE, donde el usuario de SASAMAS únicamente tendrá que programar el comportamiento que desea que tengan los agentes durante la simulación. De esta manera, el usuario no requiere de conocimientos avanzados de programación, y el esfuerzo y la inversión de tiempo se destina principalmente a la investigación de algoritmos de inteligencia artificial.

Por otra parte, el desacoplamiento de los agentes permite repartir la carga computacional entre distintas máquinas, lo que produce una disminución en la demanda de los componentes de la máquina que ejecuta el simulador y al mismo tiempo fomenta la colaboración desde distintos lugares geográficos del mundo.

En definitiva, el marco de simulación SASAMAS permite, no solo definir de manera sencilla el entorno con herramientas específicamente construidas para agilizar la modificación del terreno y de las condiciones ambientales, sino también facilitar la programación de los agentes para elaborar los algoritmos que serán validados usando los agentes SPADE que habitan dicho entorno.

CAPÍTULO 3
Diseño de la solución

3.1 Introducción

Este capítulo muestra la **arquitectura** que compone el marco de trabajo desarrollado. Primero veremos, de forma global, las partes que componen SASAMAS y cuáles son los archivos de configuración que nos permiten iniciar una simulación. A continuación, expondremos el diseño del marco de trabajo de manera mucho más extensa y técnica. Profundizaremos en cada módulo que conforma la arquitectura, con el apoyo visual de diagramas de clases y figuras. Finalmente, concluiremos el capítulo mostrando cómo funciona el protocolo de red que utilizan los agentes y el simulador para llevar a cabo la comunicación.

3.2 Arquitectura

Esta sección introduce la arquitectura del marco de trabajo SASAMAS desarrollado, que se compone de los siguientes elementos: servidor XMPP para la comunicación entre agentes, los agentes SPADE que poblarán la simulación y el servidor que ejecutará el simulador.

La figura 3.1 muestra un ejemplo de una simulación, con el objetivo de poder hacernos una idea global de la arquitectura de SASAMAS. Esta simulación está implementada en cuatro redes diferentes marcadas por rectángulos de colores y las flechas representan las conexiones de red. *Agente 1* y *Agente 2* están en la misma red local (*Red 2*) y se ejecutan en diferentes máquinas, aunque podrían estar ejecutándose en la misma máquina, e incluso el mismo proceso. *Agente 3* se está ejecutando en otra red (*Red 3*). *Agente 1*, *Agente 2* y *Agente 3* están conectados tanto al simulador como al servidor XMPP mediante *sockets*, es decir, enlaces de red. Como se comentó en la sección 2.4.3, los agentes SPADE basan sus comunicaciones en el protocolo XMPP, por lo que deben conectarse a un servidor XMPP. Es por este motivo que el servidor XMPP también se encuentra presente en la figura, ubicado en la red local *Red 4*.

3.2.1. Agentes

Los agentes (basados en SPADE) están programados en el lenguaje de programación Python [1], y se comunican con el simulador a través de conexiones de red, para interactuar con su avatar virtual que se encuentra ubicado en el entorno virtual inteligente

[1] https://www.python.org/

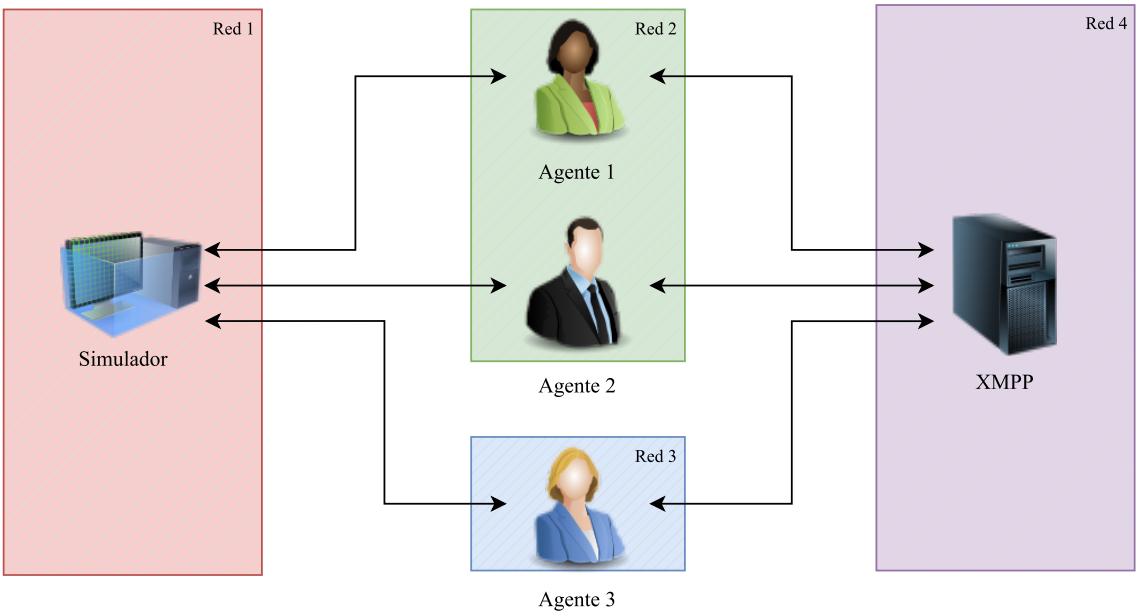


Figura 3.1: Ejemplo de la arquitectura desplegada. Hay cuatro redes locales distintas marcadas con cuadrados coloreados y las flechas representan los enlaces de red.

administrado por el simulador. Los agentes también pueden compartir mensajes entre ellos usando el protocolo XMPP. El diseño de los agentes está orientado hacia la escalabilidad del sistema y hacia permitir un sistema distribuido que se adapte a las necesidades de los usuarios que lo usen. Es por este motivo que los agentes se pueden ejecutar en diferentes hilos, procesos, máquinas e incluso redes. De esta manera, varios ordenadores pueden ejecutar diferentes agentes que coexisten en el mismo entorno.

3.2.2. Simulador

El simulador es una nueva herramienta hecha con Unity que permite definir entornos virtuales inteligentes para ser habitados por agentes SPADE. El sistema desarrollado otorga la capacidad de crear entornos tridimensionales, utilizando un editor de mapas de elaboración propia, integrado en el simulador y basado en texto. Además, permite la creación fácil y rápida de avatares personalizados de agentes. Por ejemplo, estos avatares pueden ir equipados con cámaras o sin ellas.

Definir una simulación

La creación de una simulación es un proceso fácil y rápido que involucra configurar los siguientes archivos (ver Figura 3.2):

1. El archivo `map.txt` permite editar un entorno de forma muy sencilla y rápida, ya que se trata de una herramienta basada en texto que podemos utilizar para crear un mapa en segundos. Simplemente escribimos con caracteres el aspecto que tendrá nuestro mapa, y el simulador lo cargará en la escena.

2. En el archivo `map_config.json` describiremos todos los elementos que pueden ser usados en el archivo `map.txt`. De esta manera el simulador puede relacionar el carácter con el objeto que representa. También incluye otras características generales,

que son: el punto de origen donde se empezarán a instanciar los elementos descritos en `map.txt` y la distancia de separación entre estos elementos en el eje de abscisas y en el de ordenadas.

3. Con los dos archivos anteriores ya podemos crear un mapa básico y funcional, pero carecería de flexibilidad. Es por esto que existe este tercer archivo llamado `map.json`, que permite mucho más control sobre el escenario. En este archivo podemos agregar y modificar luces en la escena, así como instanciar elementos en una posición concreta y asignarles una rotación determinada. También podemos definir unos objetos especiales llamados *Spawner*, que carecen de cuerpo visible, pero pueden ser referenciados en la configuración de los agentes, y así los agentes aparecerían en la posición en la que se encuentre el *Spawner* que han referenciado.

4. La definición de los agentes inteligentes que habitan la simulación se encuentra en el archivo `configuration.json`. Esta parte contiene toda la información que necesitan los agentes, incluida la dirección de red del simulador SASAMAS, el tipo de avatar de los agentes y la posición donde serán generados.

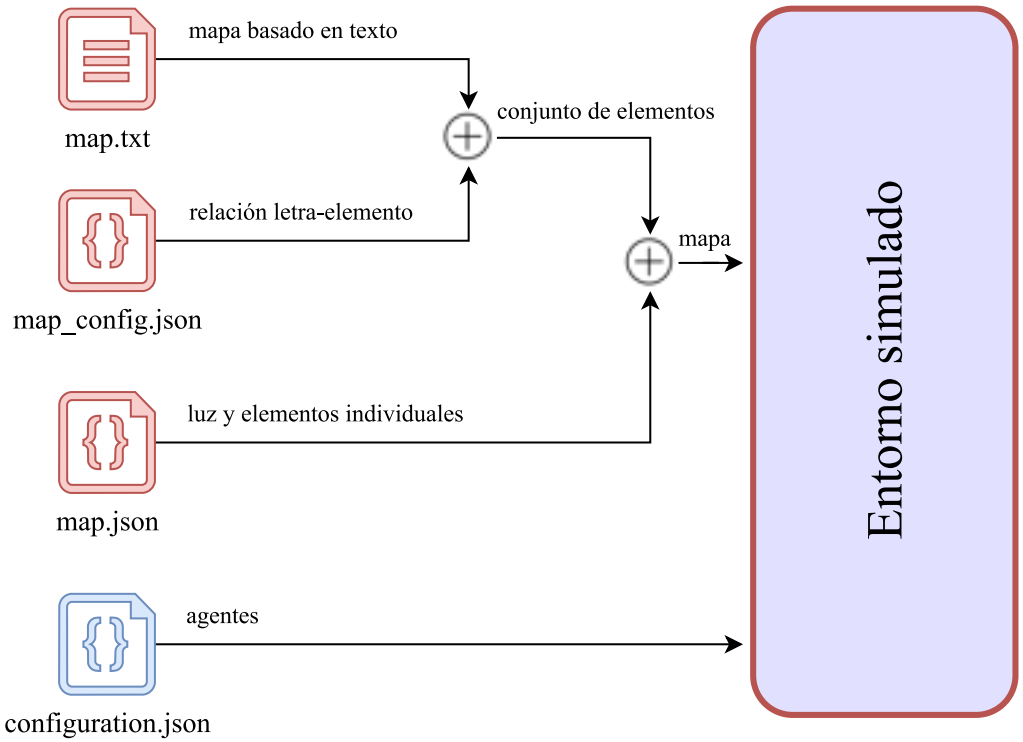


Figura 3.2: Generación del entorno de la simulación y de los agentes desde los ficheros de entrada. Los primeros tres archivos rojos generan el entorno virtual inteligente (compuesto por objetos de luz, puntos de generación de agentes y otros elementos), y el último archivo azul se usa para generar los agentes.

3.3 Diseño Detallado

Esta sección presenta de forma extensa los subsistemas introducidos en la sección 3.2. También contiene figuras con extractos representativos de ejemplos de archivos con el fin de ilustrar su funcionamiento.

3.3.1. Agentes

Los agentes SASAMAS se pueden ejecutar en diferentes hilos, procesos, máquinas e incluso redes. Para comunicarse con el simulador y el servidor XMPP debemos configurar un archivo JSON llamado `configuration.json`.

Propiedades del archivo configuration.json

Donde los nombres en inglés *Object*, *String*, *Integer* y *Float* corresponden con los tipos: objeto, cadena, entero y decimal (respectivamente) de la estructura de datos JSON (*JavaScript Object Notation*).

- simulator: Object. Datos de la red del simulador
  - address: String. Dirección IP (*Internet Protocol*) del simulador SASAMAS.
  - commandPort: Integer. Puerto de envío de comandos del simulador SASAMAS.
  - imagePort: Integer. Puerto de envío de imágenes del simulador SASAMAS.
- agent: Lista de Objects de agente
  - name: String. Nombre del agente.
  - at: String. Dirección del servidor XMPP.
  - password: String: Contraseña del agente.
  - imageBufferSize: Integer. Número máximo de imágenes por agente.
  - imageFolderName: String. Nombre de la carpeta donde se guardan las imágenes.
  - enableAgentCollision: Booleano. Si este valor se establece en *true*, este agente colisionará con otros agentes. De lo contrario, no lo hará.
  - prefabName: String. Nombre de referencia del avatar para el agente.
  - position: String u Object. Posición del punto de aparición.

    String: Nombre de referencia del punto de generación del agente.

    Object: Coordenadas del punto de generación.

    x: Float. Coordenada del punto de generación del eje X.

    y: Float. Coordenada del punto de generación del eje Y.

    z: Float. Coordenada del punto de generación del eje Z.

La figura 3.3 ilustra las propiedades descritas del archivo de configuración sobre un ejemplo de dos agentes: *agente1* y *agente2*. Ambos agentes son tractores porque la propiedad `prefabName` así lo indica y aparecerán en la escena en las posiciones donde se encuentren los objetos *Spawner 1* y *Spawner 2*, respectivamente. Estos objetos estarán definidos en el simulador que se encuentra en la dirección IP `127.0.0.1`, y en la sección 3.3.2 veremos la forma de definirlos.

Comportamiento de los agentes

El comportamiento se basa en una FSM (*Finite State Machine*), es decir, una máquina de estados finitos. El ciclo de ejecución (ver Figura 3.4) del agente se compone de los siguientes estados:

```
{
    "simulator": {
        "address": "127.0.0.1",
        "commandPort": 6066,
        "imagePort": 6067
    },
    "agents": [
        {
            "name": "agente1",
            "at": "localhost",
            "password": "xmppserver",
            "imageBufferSize": 70,
            "imageFolderName": "capturas1",
            "enableAgentCollision": true,
            "prefabName": "Tractor",
            "position": "Spawner 1"
        },
        {
            "name": "agente2",
            "at": "localhost",
            "password": "xmppserver",
            "imageBufferSize": 70,
            "imageFolderName": "capturas2",
            "enableAgentCollision": true,
            "prefabName": "Tractor",
            "position": "Spawner 2"
        }
    ]
}
```

Figura 3.3: Extracto del archivo `configuration.json` modificado.

1. ESTADO INICIAL: El agente realiza los procedimientos de preparación que considere oportunos, previos al ciclo de estados. Por ejemplo, podemos usar este estado para inicializar parámetros del agente, como la altura a la que tiene la cámara, el campo de visión y detalles de la programación del mismo. Internamente también inicia una instancia de la clase `ImageManager` en un hilo que se ejecuta de fondo. La clase `ImageManager` maneja el flujo entrante de capturas tomadas por el agente en el simulador SASAMAS y agrega los datos de la imagen a una cola compartida segura para hilos, para un procesamiento posterior.

2. ESTADO DE PERCEPCIÓN: Las capturas de la cola de imágenes se procesan en este estado. Por cada visita que recibe, retira la primera imagen de la cola y la almacena en el sistema de archivos con una marca temporal.

3. ESTADO DE COGNICIÓN: En este estado es donde ocurre el proceso de cognición. El agente decide realizar una acción en base a la información de la que dispone en ese momento.

4. ESTADO DE ACCIÓN: El agente envía comandos a su avatar, que se encuentra en el simulador. Un ejemplo podría ser un comando de rotación de cámara o un comando de movimiento. Para realizar este proceso de forma sencilla, disponemos de una API que reúne todos los comandos con los que el agente puede comunicar al servidor.

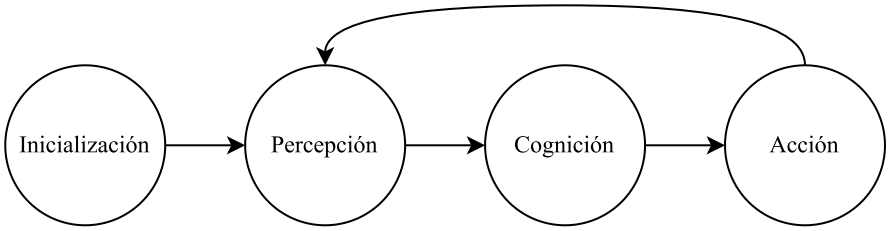


Figura 3.4: Ciclo de la FSM (máquina de estados finitos) del agente.

Programación de los agentes

La programación la realizamos en un archivo llamado `entity_shell.py`. Este archivo es una abstracción del comportamiento del agente explicado anteriormente y contiene cuatro métodos que se pueden completar: `init`, `perception`, `cognition` y `action`. Cada uno de estos métodos controla la ejecución del agente en el estado FSM del mismo nombre.

El agente tiene acceso a la clase `Commander`, que crea una capa de abstracción con el protocolo de red (ver sección 3.3.3) desarrollado para el uso en SASAMAS y contiene métodos para permitir una fácil comunicación con el simulador. Los comandos actuales cubiertos por `Commander` son los siguientes:

- `create_agent`: Envía una solicitud de instanciación al simulador, y el simulador devuelve al agente las coordenadas de la posición inicial. Este comando siempre se envía de manera automática durante el estado inicial para crear el avatar del agente.
- `move_agent`: Envía un comando al simulador para mover el avatar del agente a la posición deseada definida como `(x, y, z)`. La posición deseada es una lista de *floats*, donde el primer elemento es el valor del eje X, el segundo es el valor del eje Y y el tercero es el valor del eje Z. Finalmente, el simulador devuelve al agente la posición de destino solo si el avatar del agente pudo llegar a esta posición, en caso contrario, el simulador devuelve la posición en la que el agente se quedó atascado.
- `fov_camera`: Envía un comando para cambiar el valor del campo de visión de la cámara.
- `move_camera`: Envía un comando para mover la posición de la cámara. El argumento de este comando toma tres posiciones, de igual manera que en el comando `move_agent`.
- `rotate_camera`: Envía un comando para girar la cámara del agente a los grados deseados. Está diseñado de forma que si queremos girar la cámara para que apunte al norte, debemos indicar en el argumento el valor cero grados; si la queremos girar al este, debemos indicar noventa grados; si la queremos girar al sur, debemos indicar ciento ochenta grados y si la queremos girar al oeste, debemos indicar doscientos setenta grados. También es posible usar valores intermedios entre las cuatro orientaciones nombradas.
- `take_image`: Dependiendo del valor del argumento `image_mode`, este comando se usa para:
  - Si image_mode $= 0$: Toma una sola imagen instantáneamente.
  - Si image_mode > 0: Toma una imagen cada `image_mode` segundos deseados (o fracciones de segundos).

- Si image_mode < 0: Termina el ciclo de tomar imágenes.

Si se envían dos comandos `take_image` con el argumento `image_mode` mayor que cero, el último comando sobrescribe el ciclo del primero.

- `change_color`: Envía un comando para cambiar el color del agente. El color tiene formato `(r, g, b, a)`, donde cada elemento es de tipo *float* y sus valores deben estar comprendidos entre cero y uno.
  - *r*: Indica la cantidad de rojo del color.
  - *g*: Indica la cantidad de verde del color.
  - *b*: Indica la cantidad de azul del color.
  - *a*: Indica el grado de transparencia del color.

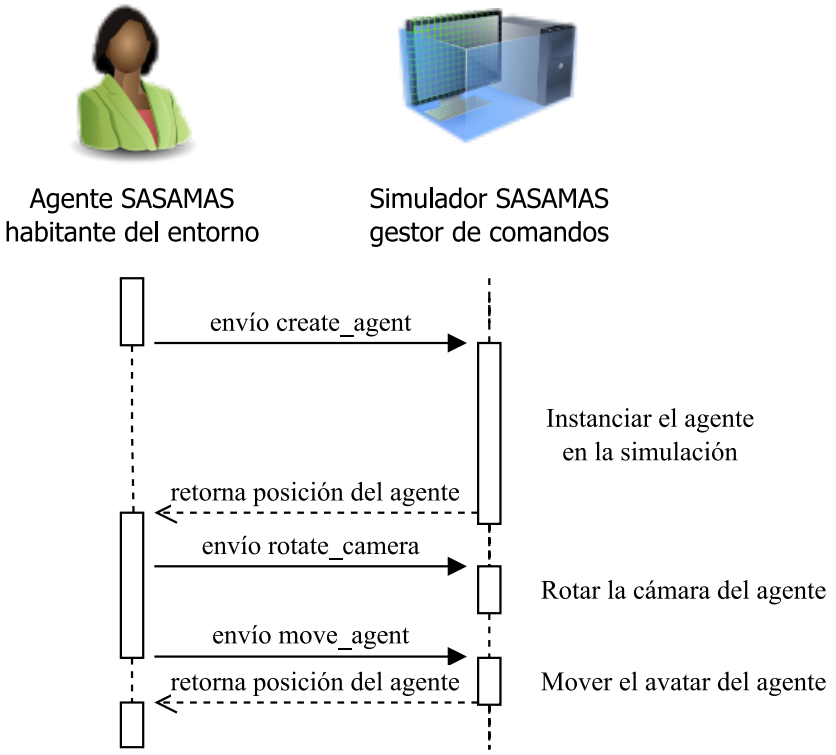


Figura 3.5: Ejemplo de comunicación de un agente con el simulador en el *socket* de comandos.

La figura 3.5 muestra una posible interacción de ejecución entre un agente y el simulador, donde el agente envía diferentes comandos —vistos en la sección 3.3.1— al simulador: `create_agent`, que crea el agente en el IVE; `rotate_camera`, que orienta la cámara del agente a los grados deseados y el comando `move_agent`, para mover al agente a una nueva posición.

Código de los agentes

En esta sección veremos las clases y paquetes del marco de trabajo desarrollado, que permiten simplificar la programación de los agentes a los usuarios de SASAMAS. Empezaremos con una vista general de los paquetes que intervienen en la ejecución del marco de trabajo y profundizaremos en el funcionamiento de las clases que contienen con el apoyo visual de varios diagramas UML.

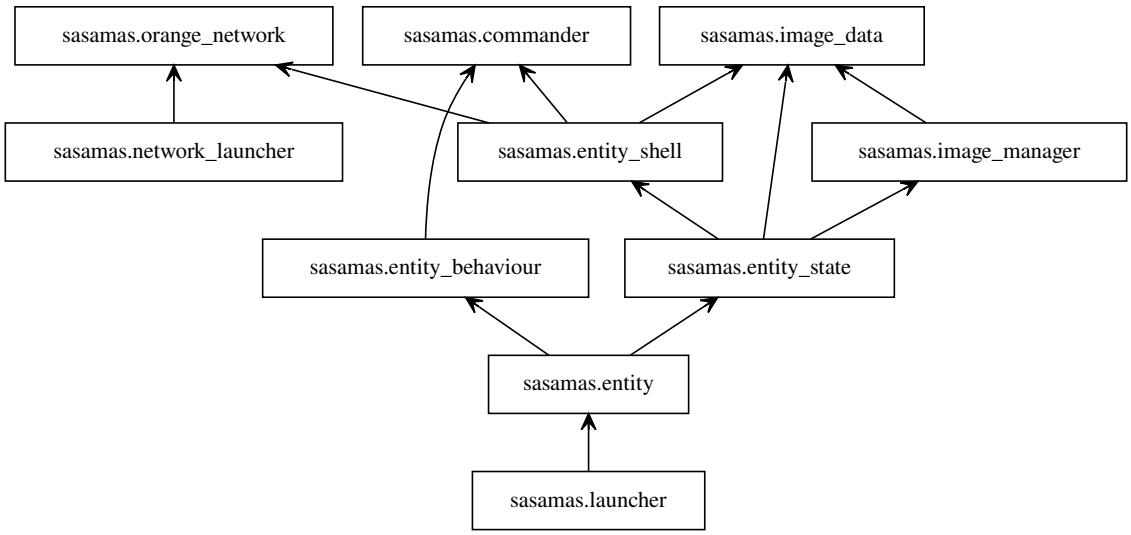


Figura 3.6: Esquema de dependencias entre los paquetes del marco de trabajo SASAMAS.

El archivo que ejecutaremos para cargar la configuración contenida en el documento `configuration.json` e iniciar a los agentes, es el paquete `launcher`. Al ejecutar el método principal del paquete, el código de este intentará cargar la configuración de los agentes y si no se encuentra el archivo `configuration.json` que la contiene, generará uno por defecto. Una vez ha cargado la información de configuración, recorre la lista de agentes y crea un hilo que se encarga de transferir la información del agente al constructor de la clase `EntityAgent`, para finalmente iniciar su instancia. El programa `launcher` también se encarga de atender las peticiones de interrupción, es decir, si deseamos finalizar de forma inmediata la ejecución del módulo de agentes del sistema SASAMAS, únicamente tenemos que dirigirnos a la consola de ejecución y pulsar las teclas `Ctrl+C` para mandar una señal de interrupción y el código de `launcher` se detendrá finalizando correctamente a los agentes que ya se encontraban en ejecución.

El siguiente paquete se denomina `entity` y contiene a la clase `EntityAgent`. Esta clase hereda de la clase `Agent` de SPADE y es utilizada para definir el comportamiento que tendrá el agente. En la clase `EntityAgent` se crea el comportamiento y se agregan los estados y las transiciones entre estos.

El paquete `entity_behaviour` contiene la definición de la clase `AgentBehaviour` usada en la clase `EntityAgent`. La clase `AgentBehaviour` hereda de la clase `FSMBehaviour`, de esta manera los agentes del marco de trabajo tienen un comportamiento basado en una máquina de estados finitos (ver Figura 3.4). `AgentBehaviour` se encarga de crear una instancia de la API `Commander`, que posteriormente será pasada por referencia a los métodos expuestos de la clase `EntityShell` para ser utilizada por el usuario en las comunicaciones con el simulador. Finalmente, esta clase también establece las conexiones de red con el simulador SASAMAS al inicio del comportamiento y los cierra al finalizar el comportamiento.

El paquete `commander` contiene tres clases. Las dos primeras han sido nombradas `Axis` e `ImageMode` y son clases que heredan de la clase `IntEnum`. Están creadas para facilitar la programación al usuario dentro del marco de trabajo SASAMAS. La clase `Axis` relaciona los ejes `X`, `Y` y `Z` con los números cero, uno y dos respectivamente. De este modo el usuario puede indicar que quiere rotar la cámara sobre el eje Y, en vez de introducir como argumento del eje el número uno, que no sería significativo para la tarea descrita. La clase `ImageMode` es análoga a la clase `Axis`, pero para las tareas que tienen que ver con el manejo de la cámara. Relaciona los modos de la cámara del avatar del agente `Disable`

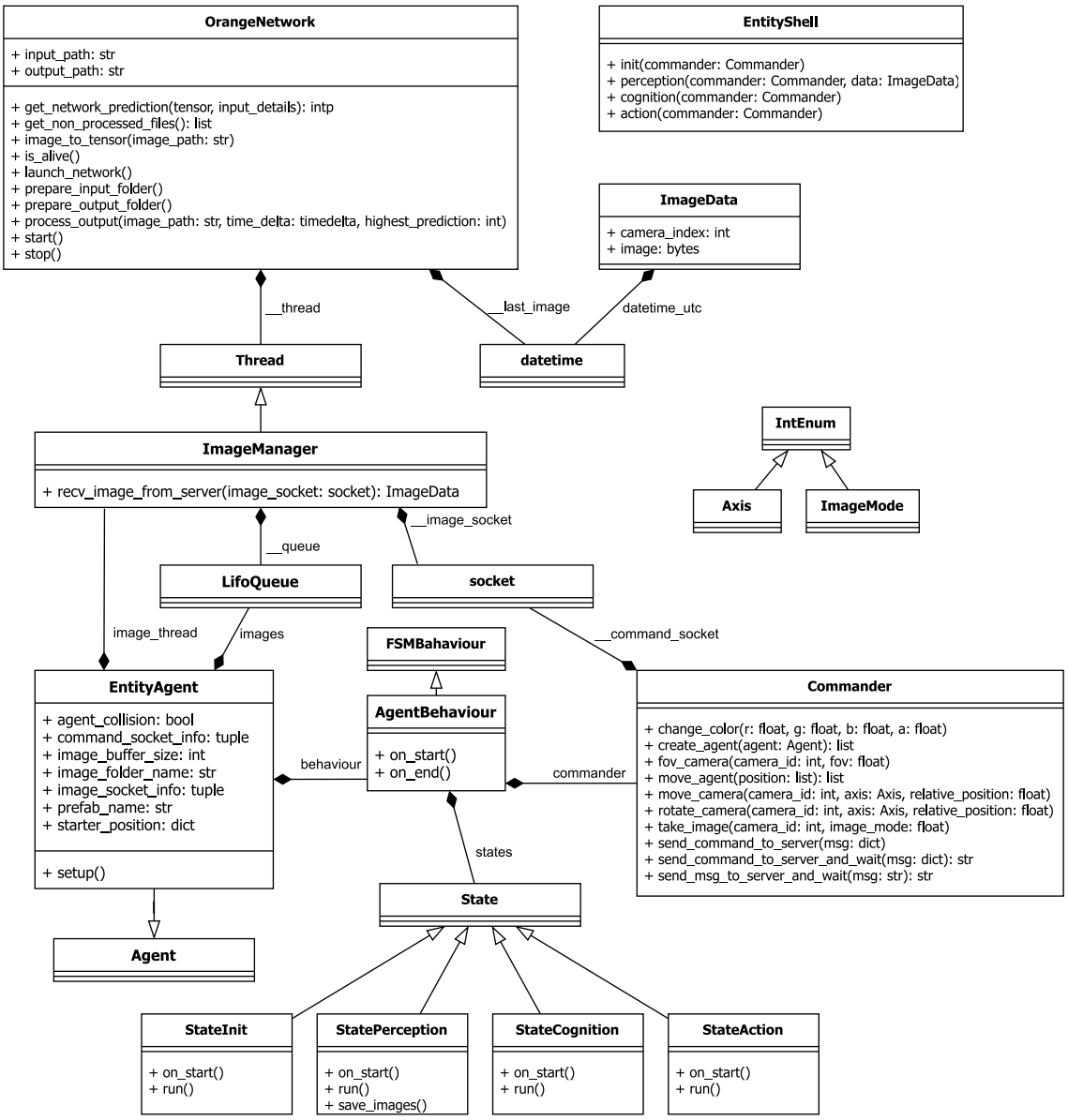
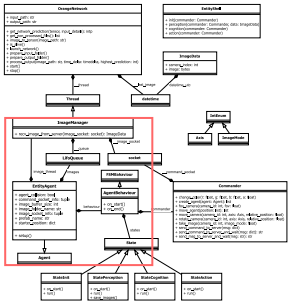


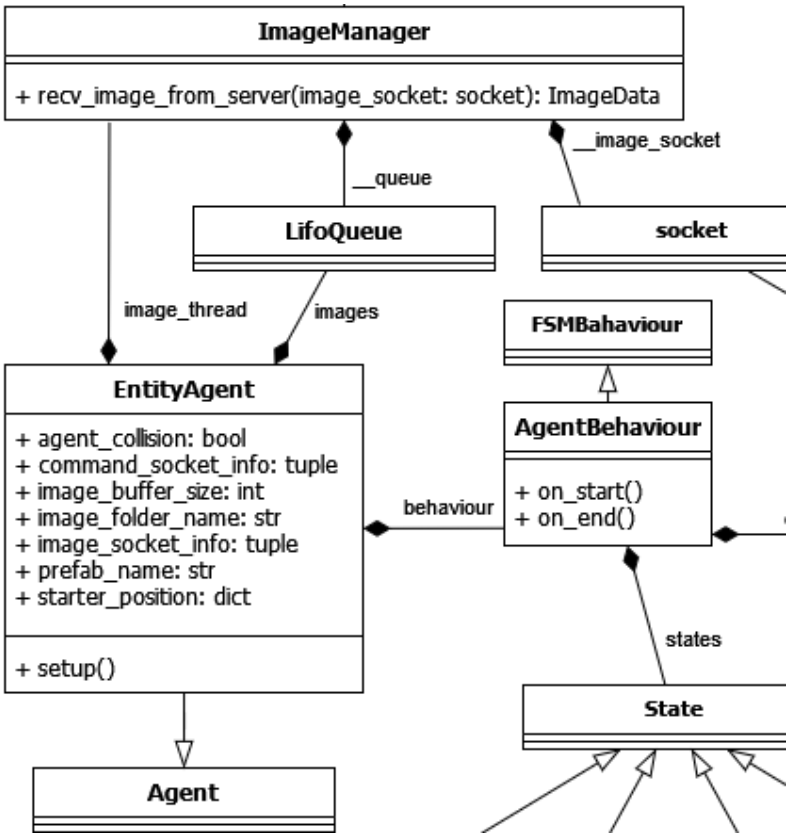
Figura 3.7: Diagrama UML con todas las clases del marco de trabajo SASAMAS y la clase `OrangeNetwork` del caso de prueba.

que termina la toma de imágenes periódica establecida anteriormente, e `Instant` que permite tomar una captura instantánea sin establecer una frecuencia periódica de toma de imágenes. Por otra parte, la clase `Commander` es una API que permite realizar las comunicaciones de comandos con el simulador sin tener que lidiar con la programación de *sockets* de red, es decir, sin que el usuario de SASAMAS tenga que programar los canales de red, ni utilizar el formato del protocolo que subyace en las comunicaciones de este marco de trabajo.

El paquete `entity_state` contiene cuatro clases que representan los cuatro estados de la máquina de estados finitos del agente. Las cuatro clases heredan de la clase `State` de SPADE y tienen un método que se ejecuta una única vez al inicializarse, y otro método que se ejecuta cada vez que el agente pasa por ese estado. De esta manera, se pueden inicializar las variables en el primero, y el segundo puede ser usado para implementar las funciones que queramos que el agente realice tras el paso de cada estado.



(a) Vista general con resaltado.

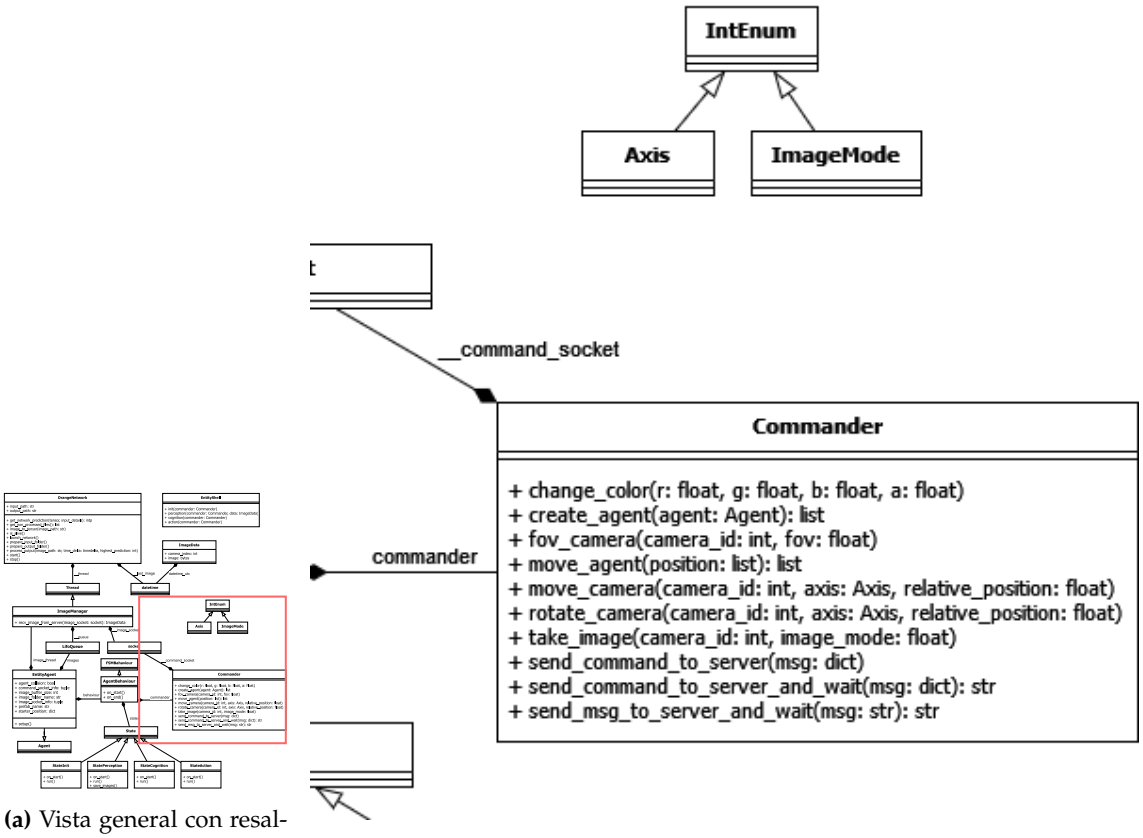


(b) Vista ampliada de la clase y sus relaciones cercanas.

Figura 3.8: Vista general y vista ampliada de la clase `EntityAgent` en el diagrama UML, con sus relaciones cercanas a otras clases.

- La clase `StateInit` se encarga de iniciar la petición de creación del avatar del agente al servidor. También crea una pila (segura para hilos) que comparte con una instancia de la clase `ImageManager`, que posteriormente ejecuta. Finalmente ejecuta una única vez el código del método `init` de la clase `EntiyShell`, que es el recipiente donde el usuario del marco SASAMAS deposita su código.
- La clase `StatePerception` se encarga de ejecutar el método `perception` de la clase `EntityShell` y es donde el agente capta la información de su entorno. Si el avatar del agente posee una cámara para obtener imágenes de su entorno y han sido enviadas por el simulador y recogidas por la pila compartida con la clase `ImageManager`, son desencoladas y procesadas. La información de la imagen, junto con la propia imagen, es pasada como argumento al método `perception`, y si el archivo `configuration.json` tiene la propiedad `image_buffer_size` con un valor mayor a cero, la captura también es guardada en el directorio de capturas establecido en la configuración del agente.
- La clase `StateCognition` se encarga de ejecutar el método `cognition` de la clase `EntityShell` y es donde el agente toma una decisión con la información que posee en ese momento.
- La clase `StateAction` se encarga de ejecutar el método `action` perteneciente a la clase `EntityShell` y es donde se llevan a cabo el conjunto de acciones que el agente ha deliberado en el estado anterior.

El paquete `image_manager` contiene la clase `ImageManager` que hereda de la clase `Thread`, es decir, posee un método `run` que se ejecuta en un hilo. Esta clase se encarga



(a) Vista general con resaltado.

(b) Vista ampliada de la clase y sus relaciones cercanas.

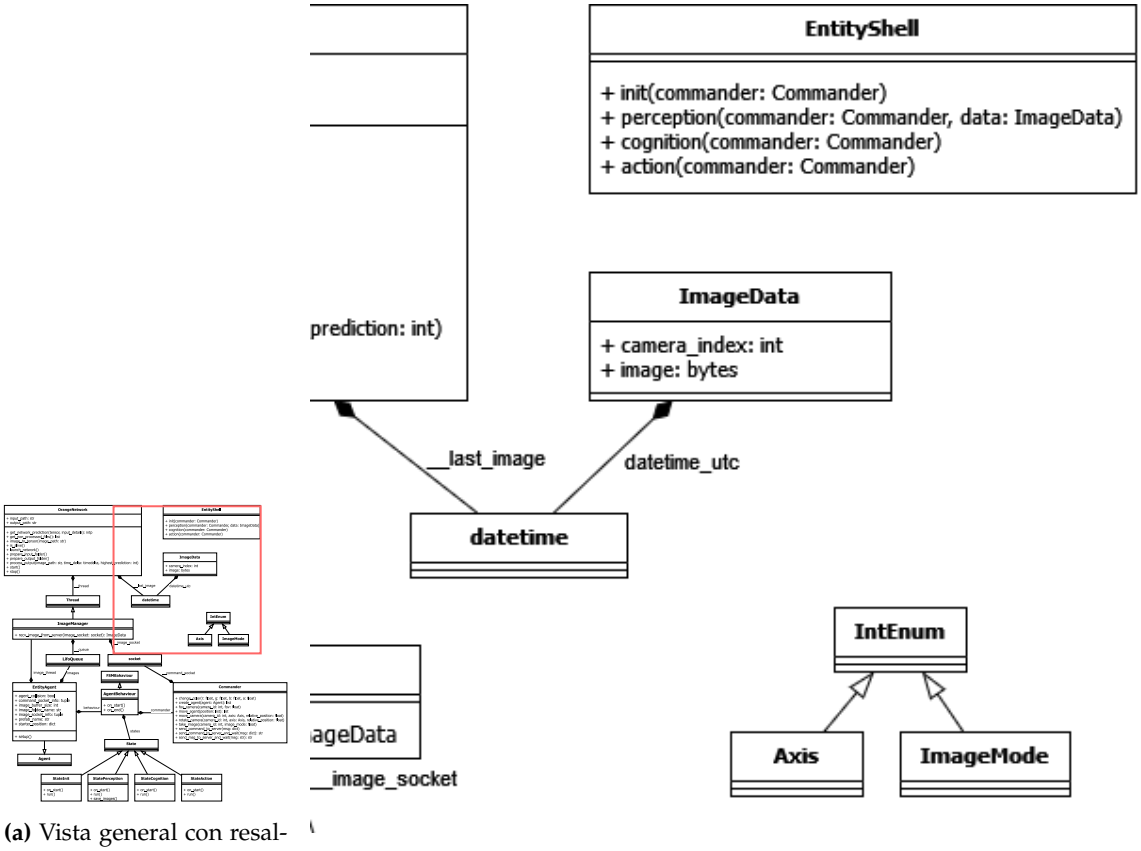
Figura 3.9: Vista general y vista ampliada de la clase `Commander` en el diagrama UML, con sus relaciones cercanas a otras clases.

de gestionar la recepción de imágenes que el avatar del agente envía desde el simulador. Cuando una imagen es recibida, esta clase la añade a una pila (segura para hilos) y compartida para su posterior procesado.

El paquete `image_data` contiene la definición de la clase `ImageData`, cuyo propósito es encapsular las propiedades de la imagen que recibe por red la clase `ImageManager` desde el simulador. Esta clase tiene tres atributos: la propia información de la imagen representada en *bytes*, el índice de la cámara del avatar con la que se tomó la imagen y la fecha y hora en la que se tomó en formato UTC.

El paquete `entity_shell` contiene la definición de la clase `EntityShell`, que es utilizada como una plantilla para que el usuario del marco de trabajo SASAMAS pueda escribir su código en ella. Contiene cuatro métodos: `init`, `perception`, `cognition` y `action`. Estos métodos son ejecutados en los estados que tienen el mismo nombre. La clase `EntityShell` usa de manera adicional la clase `OrangeNetwork`, que es usada en el caso de prueba de la sección 6.3, y es por este motivo que resulta conveniente incluirla en los diagramas y explicarla a continuación.

El paquete `orange_network` contiene la clase `OrangeNetwork`, que es la clase que se encarga de gestionar y abstraer la red neuronal profunda utilizada en el caso de prueba. La clase recibe como parámetros del constructor la ruta al modelo de la propia red, el directorio de entrada donde se encuentran las imágenes que se clasificarán y el directorio de salida donde obtendremos el resultado. La clase posee un método `start` que establece una conexión con el sistema de eventos para poder transmitir una señal de parada con el método `stop` en caso de desear abortar el proceso. El método `start` también prepara un hilo de ejecución donde se ejecuta el método `launch_network`, que se encarga de leer



(a) Vista general con resaltado.

(b) Vista ampliada de la clase y sus relaciones cercanas.

Figura 3.10: Vista general y vista ampliada de la clase `ImageData` en el diagrama UML, con sus relaciones cercanas a otras clases.

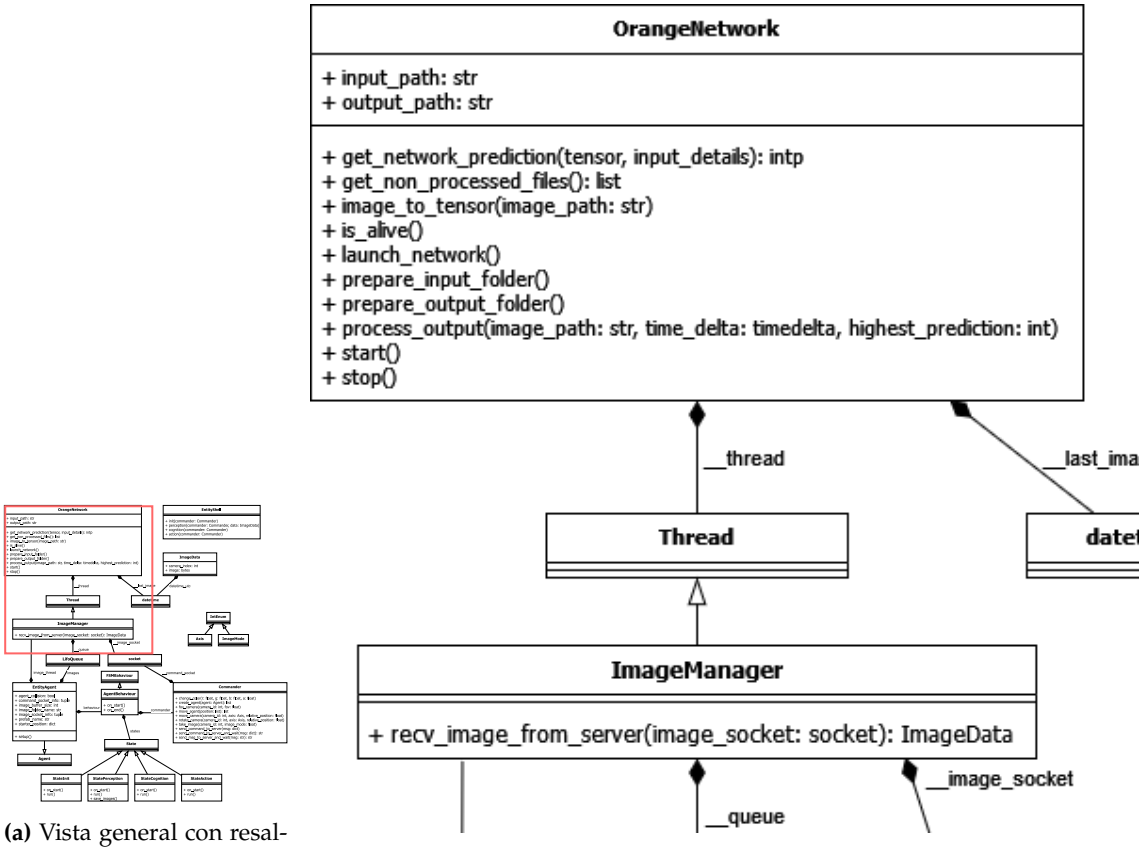
las imágenes que se encuentran en el directorio de entrada y realizar una predicción para clasificarlas. El método `launch_network` crea un archivo `log.txt` en el directorio de salida donde escribe los resultados de la clasificación y también copia la imagen que ha procesado, de este modo la validación resulta muy sencilla y si las imágenes son borradas o sobrescritas en el directorio de entrada los resultados permanecen seguros. Si el método `start` se ejecuta por segunda vez sobre el directorio con imágenes ya procesadas, la red no las vuelve a procesar. Esto es debido a un mecanismo de control que consiste en procesar las imágenes en orden ascendente a la fecha de modificación de la imagen, de esta manera guarda la fecha de modificación de la última imagen procesada en una variable interna llamada `__last_image` y solo procesa las imágenes que tienen una fecha de modificación superior a la almacenada en esta variable.

Por último el paquete `network_launcher` contiene código ejecutable por si deseamos iniciar la red para clasificar las imágenes de un directorio sin que los agentes estén en funcionamiento.

3.3.2. Simulador

Escenas del simulador

El simulador está formado por dos escenas. La primera es el menú principal (ver Figura 3.12), compuesto de tres botones: *Start*, *Options* y *Quit*. El botón *Start* inicia la escena de simulación. El botón *Options* muestra un nuevo panel de configuración, donde se pueden ajustar los siguientes parámetros:



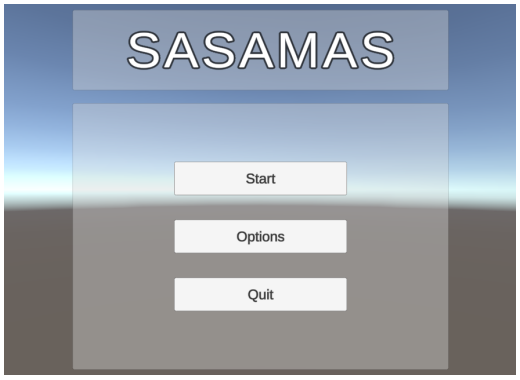
(a) Vista general con resaltado.

(b) Vista ampliada de la clase y sus relaciones cercanas.

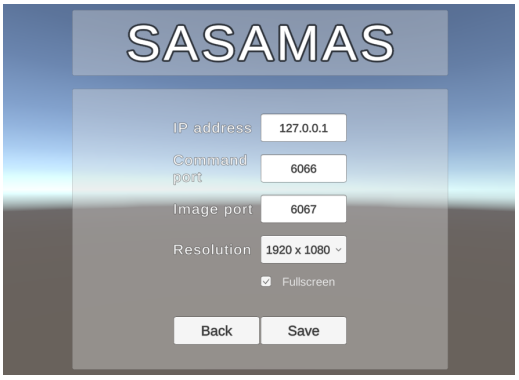
Figura 3.11: Vista general y vista ampliada de la clase `OrangeNetwork` en el diagrama UML, con sus relaciones cercanas a otras clases.

- IP address: Dirección IP donde el servidor de red del simulador va a atender las peticiones de los agentes.
- Command port: Puerto de red donde el simulador va a atender a los comandos de los agentes.
- Image port: Puerto de red que el simulador usará para el envío de imágenes a los agentes.
- Resolution: Resolución de pantalla a la que el simulador se ejecutará.
- Fullscreen: Alterna el modo de pantalla completa del simulador.
- Back: Botón para retroceder al menú principal.
- Save: Botón para guardar los cambios.
    - En el sistema operativo Windows son guardados en el registro, en la clave: `HKCU\Software\fraenan\sasamas`.
    - En sistemas operativos basados en Linux son guardados en la ruta: `~/.config/unity3d/fraenan/sasamas`.

La segunda escena (ver Figura 3.13) es la simulación. En esta escena se muestran los elementos que conforman el mapa de nuestra simulación y los agentes conectados. Podemos terminar la simulación pulsando la tecla *escape* y la cámara puede ser manejada por el usuario, usando los controles:



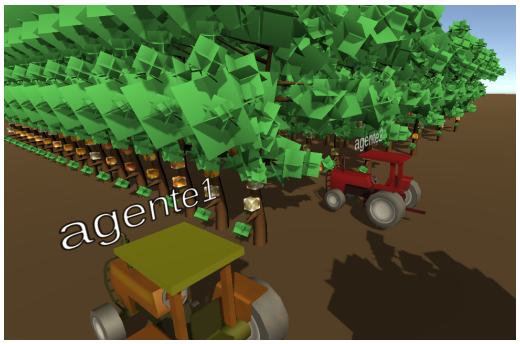
(a) Menú principal del simulador.



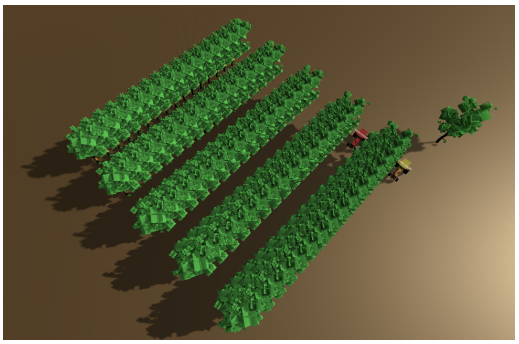
(b) Menú de opciones del simulador.

Figura 3.12: La figura 3.12a muestra el aspecto del menú principal, mientras que la figura 3.12b muestra el menú de opciones que aparece al pulsar el botón *Options* del menú principal.

- `W`: Avanza hacia la posición a la que apunta la cámara.
- `S`: Retrocede de la posición a la que apunta la cámara.
- `A`: Se desplaza horizontalmente en sentido izquierdo respecto a la posición a la que mira la cámara.
- `D`: Se desplaza horizontalmente en sentido derecho respecto a la posición a la que mira la cámara.
- `E`: Se desplaza verticalmente en sentido ascendente respecto a la posición a la que mira la cámara.
- `Q`: Se desplaza verticalmente en sentido descendente respecto a la posición a la que mira la cámara.
- `Shift izquierdo`: Al mantener esta tecla, la velocidad de la cámara se dobla.



(a) Vista cercana de los agentes, en un mapa de un huerto de naranjos.



(b) Vista alejada de los agentes, en un mapa de un huerto de naranjos.

Figura 3.13: Diferentes vistas del mapa del simulador, proporcionadas por la cámara libre que puede manejar el usuario, con dos agentes inteligentes: agente1 y agente2.

Archivos configurables del simulador

El entorno lo podemos modificar de forma rápida y efectiva simplemente editando el contenido de los siguientes tres primeros archivos vistos en la sección 3.2.2:

- `map.txt`: Primero creamos un mapa básico escribiendo caracteres que serán reemplazados por los elementos que representan, espaciándolos con el carácter *espacio*. Por ejemplo, si lo que deseamos son naranjos como los que aparecen en la figura 3.13, podemos realizar varias columnas con el carácter *O* y usar un espacio entre las columnas. También podemos definir lugares de instanciación de los agentes, denominados *Spawners*. Los *Spawners* tienen un nombre asociado, y son usados en el archivo de configuración (`configuration.json`) de los agentes. Los *Spawners* que son definidos en el archivo `map.txt` se generan con nombres correlativos: *Spawner 1*, *Spawner 2*, *Spawner 3*, etc. La secuencia de nombres sigue el orden de aparición del carácter que representa al *Spawner*, de izquierda a derecha, y de arriba a abajo. Para añadir estos elementos simplemente usaremos el carácter de nuestra elección, de igual manera que en el ejemplo del naranjo, y a continuación lo relacionaremos con el *Spawner* en el archivo `map_config.json`.

- `map_config.json`: El archivo JSON nos permite definir una relación entre caracteres del archivo `map.txt` y los elementos que representan, además de ajustar el origen de la generación del mapa y la distancia de separación entre elementos. Está formado por:

  - `origin`: Representa el punto de origen (como una coordenada tridimensional) donde se van a colocar los elementos en la simulación.
    - x: Float. Coordenada del punto de generación del eje X.
    - y: Float. Coordenada del punto de generación del eje Y.
    - z: Float. Coordenada del punto de generación del eje Z.
  - `distance`: Representa la distancia de separación entre elementos en los diferentes ejes.
    - x: Float. Distancia en el eje X entre elementos y espacios.
    - y: Float. Distancia en el eje Y entre elementos y espacios.
  - `symbolToPrefabMap`: Es una lista de los siguientes parámetros:
    - `symbol`: Carácter que representa el elemento en el archivo `map.txt`.
    - `prefabName`: Nombre del prefab del elemento que reemplazará al carácter.
    - `dataFolder`: Propiedad opcional que contiene una ruta de carpeta, y se puede usar con elementos que tienen un componente `ChangeTexture` para cargar las imágenes de la carpeta especificada y aplicarlas como texturas aleatorias al elemento (ver Figura 3.14).

- `map.json`: Los elementos también pueden ser creados de manera individual para otorgar mayor grado de precisión en la creación del mapa, así como crear y ajustar tantas fuentes de luz como se requieran para proporcionar las condiciones ambientales deseadas. En este archivo también podemos crear elementos de tipo *Spawner* con nombres personalizados.

  - `lights`: Lista de objetos de tipo luz.
    - `active`: Boolean. Bandera para crear el objeto (si el valor es verdadero) o para ignorarlo (si el valor es falso).
    - `objectName`: Nombre interno del objeto, que en el caso de ser un *Spawner* puede ser usado por los agentes para aparecer en su posición.
    - `objectPrefabName`: Nombre de referencia del elemento de los objetos de luz disponibles.
    - `position`: Coordenada tridimensional del objeto.
      - x: Float. Coordenada del punto de generación del eje X.

        - ◇ y: Float. Coordenada del punto de generación del eje Y.
        - ◇ z: Float. Coordenada del punto de generación del eje Z.
    - ○ `rotation`: Rotación (en grados) de los tres ejes.
        - ◇ x: Float. Grados en el eje X.
        - ◇ y: Float. Grados en el eje Y.
        - ◇ z: Float. Grados en el eje Z.
    - ○ `color`: Objeto que contiene la información del color. Todos los valores están comprendidos entre cero y uno, inclusives.
        - ◇ *r*: Indica la cantidad de rojo del color.
        - ◇ *g*: Indica la cantidad de verde del color.
        - ◇ *b*: Indica la cantidad de azul del color.
        - ◇ *a*: Indica el grado de transparencia del color.
    - ○ `intensity`: Intensidad del rayo de luz.
- `objects`: Lista de elementos. Sus propiedades son exactamente las mismas que las de los objetos de luz de arriba, pero estos elementos no tienen color ni intensidad.

Figura 3.14: Varios naranjos con texturas aleatorias aplicadas en tiempo de ejecución a las naranjas, cargando las imágenes de la ruta especificada en el archivo `map_config.json`.

Para ilustrar lo fácil que es modificar una simulación compuesta por agentes y elementos del entorno usando solo los archivos explicados en la sección 3.2.2, mostramos la figura 3.15, donde solo los archivos `map_config.json` y `configuration.json` han sido modificados. En la figura 3.15b, hemos disminuido el valor de la propiedad vertical del campo `distancia` de nueve unidades a tres unidades en el archivo `map_config.json`, y en el archivo `configuration.json` agregamos cuatro agentes y solo necesitamos cambiar el nombre y la ubicación de generación de los nuevos agentes.

Agregar nuevos agentes y elementos a la simulación

Si deseamos agregar nuevos tipos de agentes con modelados y características distintas a los agentes predeterminados debemos realizar el siguiente proceso:

(a) Simulación de un agente tractor en un campo de naranjos.

(b) Simulación de cuatro agentes tractores y un agente robot en un campo de naranjos.

Figura 3.15: Ejemplo de simulación compuesta por un campo de naranjos y agentes. En la figura 3.15a solo hay un agente, y en la figura 3.15b hay cinco agentes y el espacio entre los árboles es tres veces menor que en la figura 3.15a.

1. Abrimos el proyecto del simulador en el editor de Unity.

2. Importamos el modelo del nuevo agente en el editor de Unity.

3. Arrastramos y soltamos el modelo en la escena, para que Unity cree una instancia de *GameObject*.

4. Agregamos un componente *NavMeshAgent* a la instancia y configuramos sus parámetros para obtener las capacidades deseadas del agente.

5. Creamos una nueva instancia de *TextMeshPro* como hija de la instancia del modelo, para que se pueda mostrar el nombre del agente.

6. Agregamos un componente *Entity* (Script) a la instancia del modelo y configuramos el parámetro *Text* para que referencie a la instancia *TextMeshPro* creada en el punto anterior. También configuramos el parámetro llamado *Colored Part* que apunta a la parte de la instancia del modelo que cambiará de color cuando se procese un comando de cambio de color.

Opcional: Agregamos un componente *CameraManager* (Script) al agente para que este pueda tomar fotografías. El parámetro *Capturar dimensiones* es una lista de resoluciones de cámara, donde el elemento en el índice *i* corresponde a la cámara en el índice *i* en la lista de *Cameras*.

7. Finalmente guardamos la instancia del agente como un prefab de Unity, y referenciamos el prefab resultante en una ranura vacía del parámetro *Agent Prefabs* del objeto *Agent Manager*.

Si deseamos agregar nuevos tipos de elementos a la simulación debemos realizar el siguiente proceso:

1. Abrimos el proyecto del simulador en el editor de Unity.
2. Importamos el modelo del nuevo elemento en el editor de Unity y creamos una instancia de *GameObject*.
3. Agregamos un componente *NavMeshObstacle* a la instancia y establecemos sus parámetros para ajustar el modelo a la malla de obstáculos. La casilla de verificación *Carve* debe estar marcada para evitar que los agentes colisionen con el nuevo elemento.

Opcional: Agregamos un componente *ChangeTexture* (Script) al nuevo elemento para poder ajustar las texturas que se cargarán en tiempo de ejecución desde la ruta del directorio definido en el archivo `map_config.json`. El componente *ChangeTexture* requiere la configuración de dos atributos:

- `instantiableObject`: Referencia al prefab del elemento que se instanciará como hijo del elemento que estamos añadiendo. Las texturas de las instancias generadas a partir de este prefab son las que serán alteradas. En el ejemplo de la figura 3.15, el valor de este atributo sería el prefab de la naranja.
- `instantiableObjectPositions`: Lista de posiciones donde se instanciarán los elementos del tipo establecido en `instantiableObject`.

4. Guardamos la instancia del elemento como un prefab de Unity y colocamos el prefab resultante en la lista del atributo *Instantiable Prefabs* del objeto *MapLoader* que se encuentra en la escena del simulador.

Código del simulador

El lenguaje de programación usado es C#, dado que el simulador está construido usando el motor gráfico de Unity. En esta sección veremos en detalle las clases que intervienen en el funcionamiento del simulador, acompañando las descripciones con diferentes vistas del diagrama UML. El diagrama UML completo en una sola imagen no se incluye en esta sección, debido a que su dimensión excede el tamaño de página de este documento, y por lo tanto se encuentra disponible para su consulta en la ruta del proyecto github **other/diagrams/simulator uml.png**, accesible desde el enlace: **https://github.com/FranEnguix/sasamas**

En la primera escena (el menú) interviene el archivo `MenuController.cs` que contiene la clase `MenuController`. Esta clase es la responsable de:

- Cargar las resoluciones de pantalla disponibles y poblar con esta información el control donde el usuario puede seleccionar una.

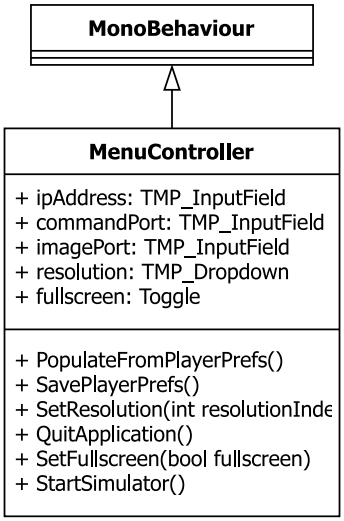


Figura 3.16: Extracto del diagrama UML centrado en la clase `MenuController`

- Gestionar las preferencias del usuario. Tanto leer la información del archivo que las almacena para poblar los controles, como el método ejecutado por el botón `Save` que las guarda en el disco.
- Validar la información que el usuario introduce en los controles y mostrar cuando es incorrecta cambiando el color por defecto de los controles a rojo e inhabilitando el botón de guardado.
- Modificar la resolución de la pantalla y alternar el modo de pantalla completa.
- Cerrar la aplicación cuando el usuario ejecuta el procedimiento del botón `Quit`.
- Cargar la escena de la simulación cuando el usuario ejecuta el método del botón `Start`.

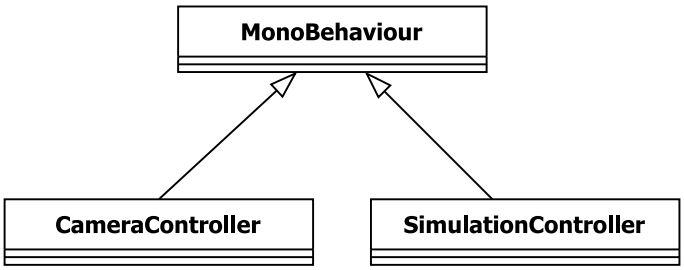


Figura 3.17: Extracto del diagrama UML con la vista de las clases `CameraController` y `SimulationController`

En la segunda escena (la simulación) intervienen más clases que se mostrarán a continuación, empezando por la clase `CameraController`. El movimiento y rotación de la cámara libre está controlado por esta clase, que permite ajustar la velocidad y sensibilidad de la cámara. La cámara responde a los controles descritos en la sección 3.3.2.

La versión actual del simulador contempla un único control para el simulador, que es iniciado al pulsar la tecla `escape` y se gestiona en la clase `SimulationController`. Pulsar esta tecla durante la simulación hace que termine la ejecución del programa de manera inmediata.

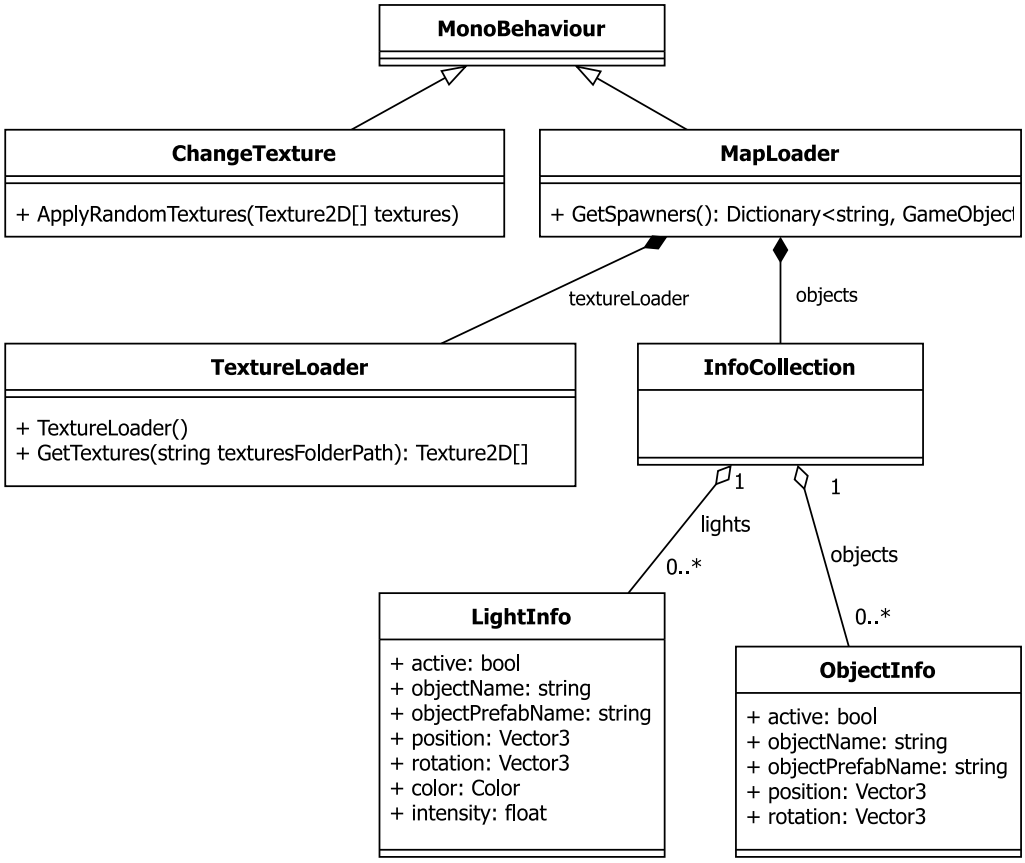


Figura 3.18: Extracto del diagrama UML con la vista de varias clases y sus relaciones, centrado en la clase `MapLoader`

La clase `MapLoader` es la primera en ser ejecutada en la escena del simulador. Es la encargada de generar el mapa a partir de la configuración que hemos establecido en los archivos descritos en la sección 3.3.2.

- Si los archivos no existen, genera unos archivos básicos y funcionales a modo de plantilla, para que podamos rellenarlos de manera sencilla.
- Cuando el proyecto está abierto en el editor de Unity, podemos agregar nuevos elementos en el atributo `instantiablePrefabs` para ser utilizados en los archivos de generación de mapas.
- Guarda una lista de los *spawners* (objetos especiales usados para generar agentes) para ser utilizada posteriormente por la clase `TcpServer` en el proceso de instanciado de agentes.

La clase `InfoCollection` encapsula las listas de los objetos de luz y los otros elementos que conforman el mapa de una simulación. Es usada por la clase `MapLoader` para procesar el archivo `map.json` descrito en la sección 3.3.2.

Los objetos de luz son representados por la clase `LightInfo`, que posteriormente serán utilizados para formar estos objetos y colocarlos en la escena de la simulación, y así crear las condiciones ambientales.

La información de los otros elementos no lumínicos de la escena es encapsulada por la clase `ObjectInfo`, que posteriormente será utilizada para formar estos objetos y colocarlos en la simulación.

La clase `TextureLoader` tiene un método que es invocado por la clase `MapLoader` y se encarga de gestionar la carga de imágenes de la ruta que se encuentra descrita en el archivo `map_config.json`, para ser convertidas a texturas. Este método recibe como argumento la ruta al directorio donde se encuentran las imágenes, e internamente se encarga de gestionar el proceso de conversión a texturas. Si la ruta obtenida del argumento es la primera vez que la procesa, entonces convierte las imágenes a un arreglo de texturas y almacena la información en una tabla de dispersión, usando la ruta como clave y el arreglo de texturas como valor. El método finalmente devuelve la referencia del arreglo de texturas al código que lo invocó. De esta manera, si la misma ruta es consultada de nuevo, ya no tiene que volver a cargar las imágenes del disco y convertirlas de nuevo en texturas, sino que simplemente devuelve la referencia al arreglo de texturas almacenado.

La clase `ChangeTexture` permite instanciar varios objetos de un mismo tipo (establecido en la variable `intantiableObject`) con texturas aleatorias. Para ello, tiene un método que por cada objeto a instanciar, selecciona una textura aleatoria de un arreglo de texturas que obtiene como argumento. Instancia tantos objetos como posiciones hayamos definido en la otra variable del modelo, denominada `intantiableObjectPositions`.

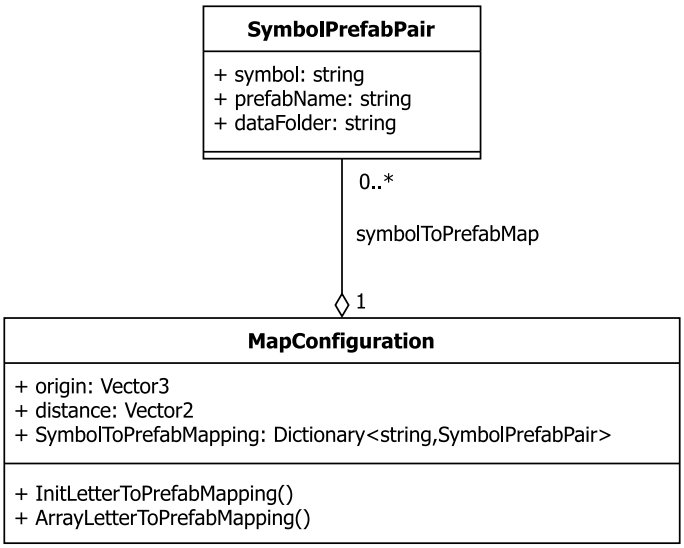


Figura 3.19: Extracto del diagrama UML, centrado en `MapConfiguration`

La clase `SymbolPrefabPair` encapsula la información sobre la relación entre un símbolo y el tipo de objeto que representa, y puede ir acompañado de la ruta a un directorio de imágenes para que la clase `TextureLoader` cargue las imágenes como texturas. Es usado durante el proceso de construcción de la escena, a partir de los archivos `map.txt` y `map_config.json` explicados en detalle en la sección 3.3.2.

La clase `MapConfiguration` existe porque la librería proporcionada por Unity para utilizar el formato JSON no dispone de estructuras de datos del tipo diccionario, por este motivo, esta clase permite convertir listas de `SymbolPrefabPair` a tablas de dispersión, y viceversa. La tabla de dispersión usa el símbolo como clave y el objeto que representa dicho símbolo corresponde al valor, de esta manera puede ser consultada durante el proceso de construcción de la escena y se reduce el coste computacional, de tener una cota superior lineal (siendo la talla la cantidad de símbolos distintos que se han definido) en la búsqueda del símbolo por la utilización de arreglos, a tener una cota superior constante que es el coste de la búsqueda en la tabla de dispersión que utiliza.

La clase `TcpServer` es la encargada de gestionar la comunicación por red con los agentes y de la ejecución de los comandos que los agentes mandan por red. Sus características principales son:

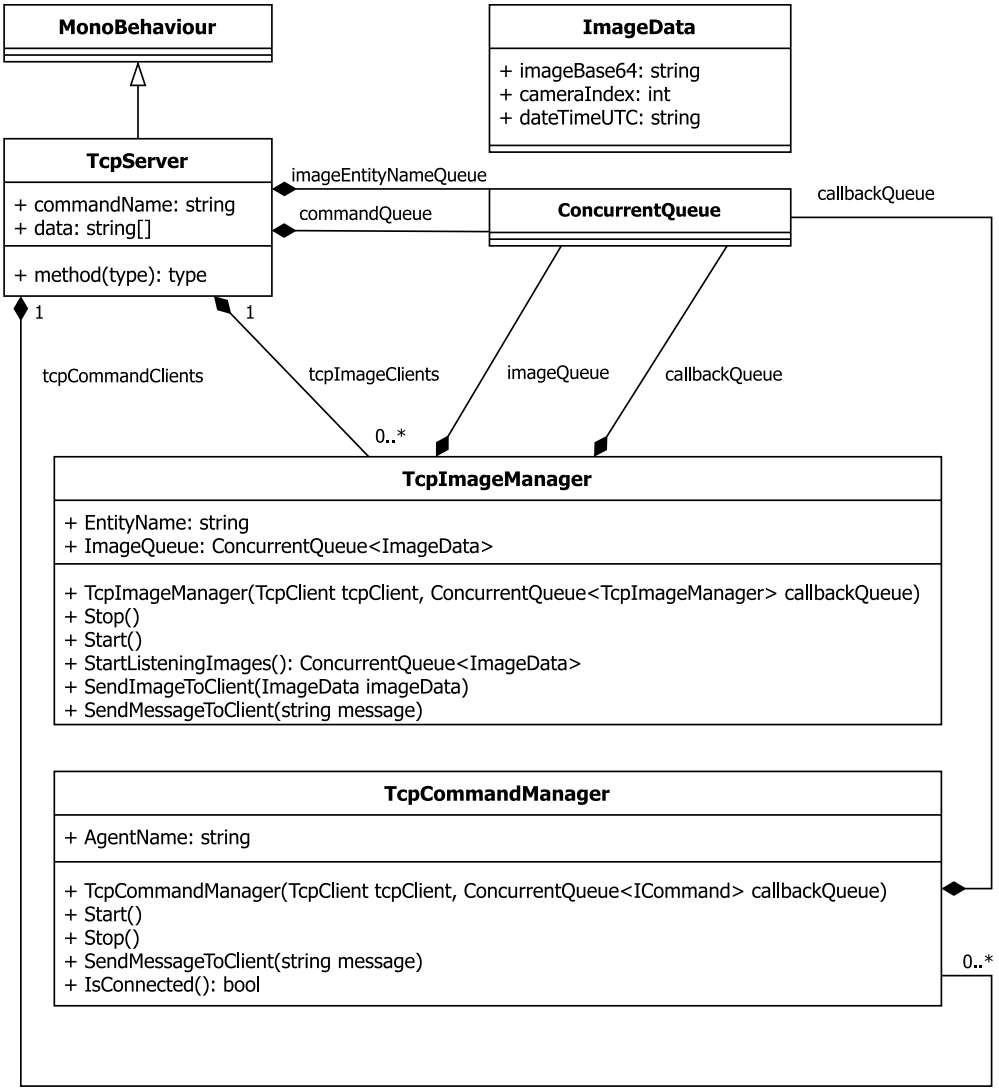


Figura 3.20: Extracto del diagrama UML con la vista de varias clases y sus relaciones, centrado en la clase `TcpServer`

- Antes de que se ejecute la simulación, esta clase lee el contenido del archivo donde están almacenadas las preferencias que hemos establecido en el menú de opciones (`Options`) del simulador y carga su contenido en variables que serán usadas en los siguientes puntos.

- Al inicio de la simulación ejecuta un hilo en segundo plano que escucha las comunicaciones de red en el puerto `commandPort` configurado. Cuando el puerto es contactado por un cliente, incluye el *socket* de este cliente y la referencia de la cola segura para hilos (`commandQueue`) en una nueva instancia de la clase `TcpCommandManager`. Finalmente, ejecuta la nueva instancia de la clase `TcpCommandManager`, que se encargará de traducir y atender las peticiones del cliente en un nuevo hilo.

- Al inicio de la simulación también ejecuta otro hilo en segundo plano que escucha las comunicaciones de red en el puerto `imagePort` configurado. Que funciona de manera análoga al hilo explicado en el punto anterior, con la diferencia de que la clase instanciada es `TcpImageManager` y la cola segura para hilos que se le pasa como referencia es la de `imageEntityNameQueue`.

- En cada fotograma de la simulación se comprueba si alguna de las dos colas existentes (`commandQueue` y `imageEntityNameQueue`) tienen contenido, y en caso de que sea así se procesan:
  - En `commandQueue` se desencola el siguiente comando y se ejecuta. Gracias al uso de la interfaz `ICommand` utilizada por todos los comandos, todos los comandos tienen la función `Execute` que puede ser ejecutada directamente. Existen dos casos especiales con los comandos para crear y reconectar agentes, porque la clase `TcpServer` mantiene una lista de los agentes activos que estos dos comandos actualizan y los agentes necesitan que se le devuelva la posición en la que se encuentran inicialmente sus avatares, por este motivo se ejecutan en las funciones `CreateEntity` y `ReconnectEntity` de la clase `TcpServer`.
  - En la cola (segura para hilos) `imageEntityNameQueue` se desencola una instancia de la clase `TcpImageManager`, que permite relacionar al avatar del agente con el `TcpImageManager` desencolado a través del nombre del agente.
- Existen otros métodos auxiliares que son usados por las funciones descritas. Por ejemplo, para mandar por red la información de la posición actual del avatar del agente al propio agente, existe el método `SendCurrentPosition`. También existe la función `GetAgentPrefab`, que recibe como parámetro el nombre del *prefab* del avatar del agente y devuelve el propio avatar en forma de objeto.

La clase `TcpCommandManager` encapsula y gestiona las comunicaciones de red relacionadas con los comandos que el agente envía a su avatar. Si detecta que el agente envía un mensaje a través del *socket* de comandos, utiliza la clase `CommandParser` para obtener el comando y lo encola en la cola (segura para hilos) que tiene compartida con la clase `TcpServer` para su posterior procesamiento.

La clase `TcpImageManager` encapsula y gestiona las funciones de red relacionadas con las imágenes que captura el avatar del agente. Comparte una cola (segura para hilos) con la clase `CameraManager` y si detecta que la cola tiene contenido, desencola su contenido y lo envía por red al agente en formato JSON.

La clase `Entity` se encarga de gestionar los parámetros que intervienen en el control del avatar del agente.

- Si el agente requiere que su avatar cambie de color, existe una función en esta clase llamada `ChangeColor` que cambia el color del avatar del agente.
- La función `CameraFov` que permite cambiar el campo de visión de la cámara vinculada al avatar del agente.
- La función `CameraMove` mueve la cámara del avatar a la posición deseada por el agente.
- La función `CameraRotate` rota la cámara en el eje deseado para orientarla hacia los grados en los que indica el agente. Por ejemplo, podemos rotar la cámara sobre el eje vertical, donde los grados de rotación corresponden con: cero grados, enfocar al norte; noventa grados, el este; ciento ochenta, el sur y doscientos setenta, el oeste.
- `LaunchTakePicture` donde se puede establecer la frecuencia a la que el agente toma capturas del entorno.
- La función `SetTargetPosition` que se usa para indicar al avatar del agente la posición a la que deseamos que se desplace.

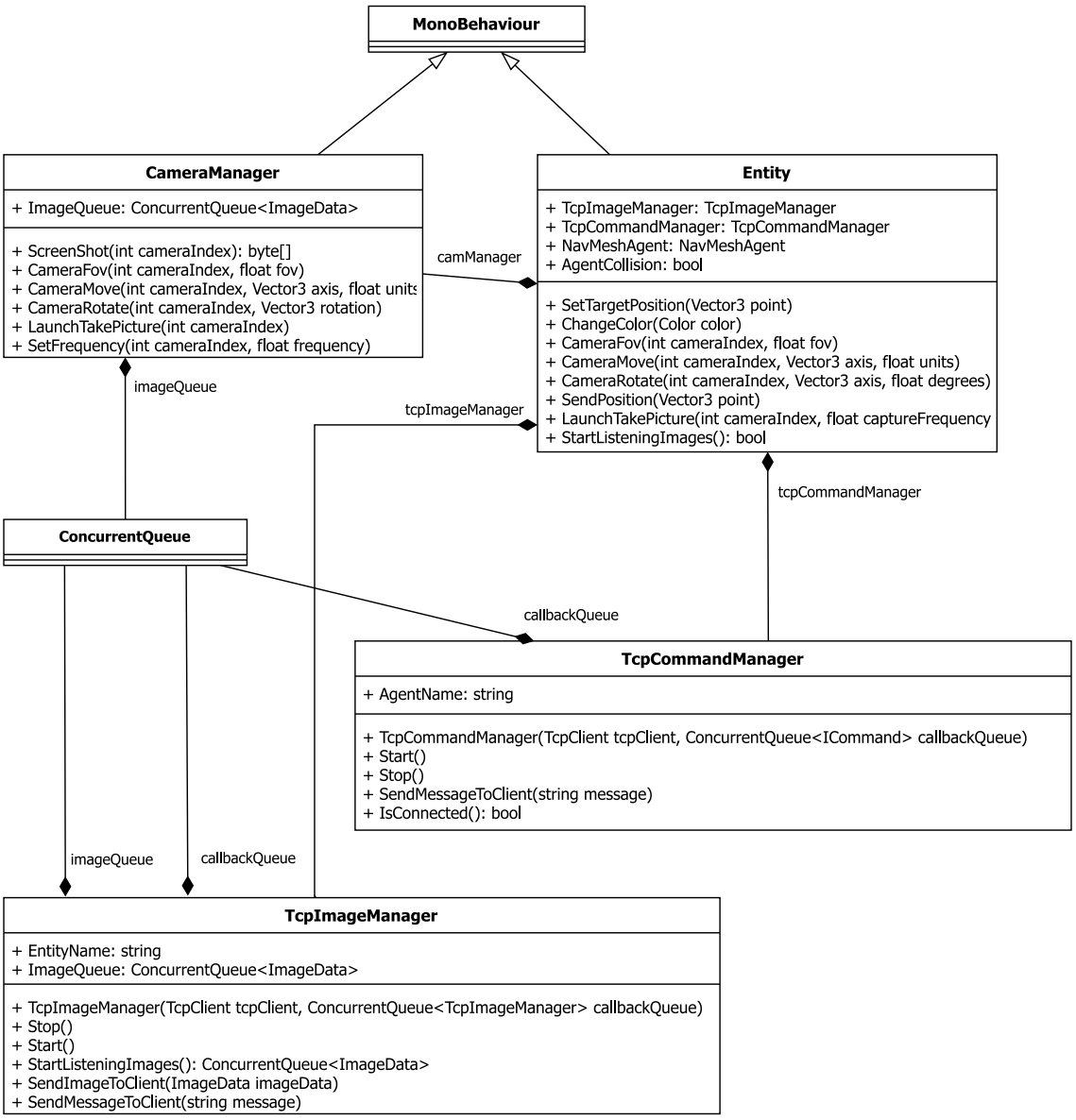


Figura 3.21: Extracto del diagrama UML con la vista de varias clases y sus relaciones, centrado en la clase `Entity`.

La clase `CameraManager` es utilizada por la clase `Entity` para gestionar las funciones internas de la cámara:

- Cuando el avatar del agente se instancia en la escena de la simulación, esta clase obtiene información de las cámaras que tiene el avatar vinculadas para utilizarla en los métodos descritos a continuación.

- `TakePicture` es utilizado para obtener una captura con la cámara indicada por el agente. Este método codifica los *bytes* que forman la imagen a *base64* y los encapsula, junto con la fecha y hora actual, para encolarlos en la cola compartida (segura para hilos) de imágenes. Finalmente la imagen tomada será enviada por red al agente.

- `SetFrequency` es una función que establece la frecuencia a la que toma imágenes una cámara determinada.

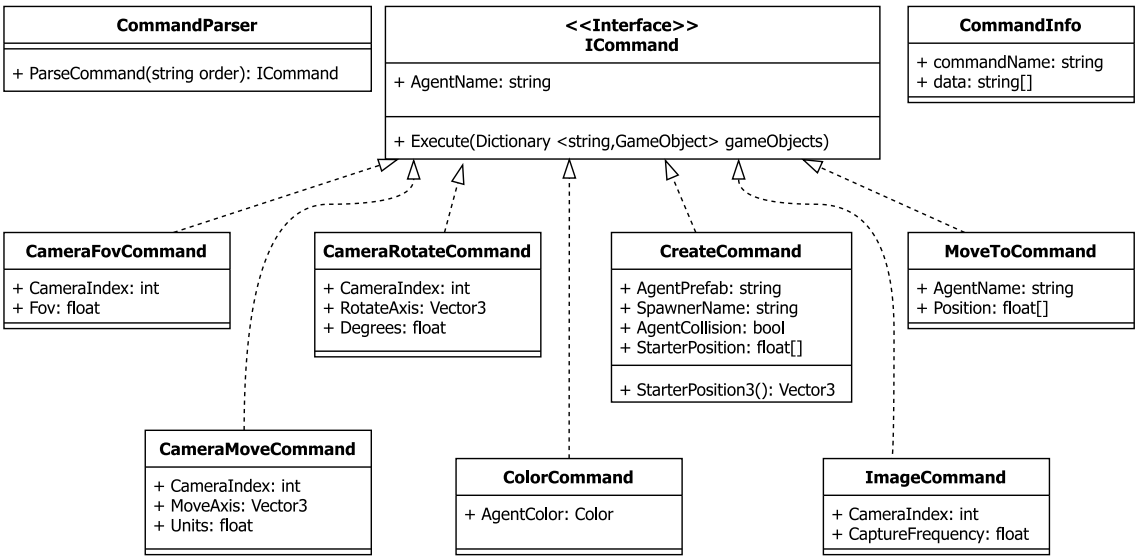


Figura 3.22: Extracto del diagrama UML con la vista de varias clases y sus relaciones, centrado en la interfaz `ICommand`.

La clase `CommandParser` es la encargada de traducir los mensajes de red del agente que están en texto plano y con formato JSON, a objetos de tipo comando (implementan la interfaz `ICommand`).

Los comandos que utiliza el agente para generar una respuesta en su avatar implementan la interfaz `ICommand`, que tiene la cabecera de los métodos: `Execute`, usado para ejecutar el comando y `AgentName` usado para poder relacionar el comando con el agente al que corresponde. Los comandos disponibles que implementan dicha interfaz, son:

- `CameraFovCommand`: Utilizado para cambiar el campo de visión de una cámara del avatar del agente.
- `CameraMoveCommand`: Utilizado para mover la cámara deseada del avatar del agente.
- `CameraRotateCommand`: Utilizado para rotar la cámara deseada del avatar del agente, en los ejes requeridos de manera simultánea.
- `ColorCommand`: Utilizado para que el agente cambie de color la pieza designada para este uso por el modelo del avatar.
- `ImageCommand`: Utilizado para que el agente pueda dar instrucciones de toma de imágenes a una cámara.
- `MoveToCommand`: Utilizado para que el agente pueda proporcionar un destino a su avatar, y este se desplace a él.

Existen otras clases se utilizan para convertir los datos a formato JSON, o para convertir los datos en formato JSON a instancias de clases. Estas clases son:

- `ImageData`: Encapsula la información de las capturas que toman las cámaras del avatar del agente: la propia captura en formato *base64*, el índice de la cámara con la que fue tomada y la fecha y hora en formato UTC.
- `CommandInfo`: La clase encapsula la información de los comandos enviados por el agente a través de la red. Esta clase la expondremos en detalle en la sección 3.3.3.

3.3.3. Protocolo de red

La comunicación entre los avatares de los agentes, que se encuentran en el simulador, y los propios agentes que pueden ser ejecutados en cualquier parte del mundo, se efectúa mediante un protocolo de comunicaciones de elaboración propia. Existen dos canales entre los agentes y el simulador, que son: el canal de comandos, donde el agente expresa las acciones que su avatar debe realizar y el canal de imágenes, donde el simulador envía las imágenes tomadas por el avatar del agente. En esta sección veremos en detalle el protocolo que subyace en estas comunicaciones.

Protocolo de comandos

En esta sección vamos a ilustrar el recorrido que realiza un comando, desde lo que ocurre en la ejecución del propio comando en el agente, hasta que llega al servidor del simulador.

El simulador es el primero que debemos ejecutar, ya que estará a la escucha para atender las peticiones de los agentes. A continuación, cada agente solicita la instanciación de su avatar en la simulación enviando un comando de creación. La forma de enviar este comando desde el código del agente es haciendo uso de la API `Commander` desarrollada, y es suficiente con utilizar el método `create_agent` de esta, que se encargará de crear el avatar y de devolver la posición al agente. Los usuarios del marco de trabajo SASAMAS no necesitamos hacer esta llamada, porque se realiza de manera automática en la clase interna `StateInit`, como se muestra en la figura 3.23, pero sí es interesante conocer qué ocurre después de que se realice la llamada.

```
class StateInit(State):
  async def on_start(self):
    behaviour = self.agent.behaviours[0]
    self.image_socket = behaviour.image_socket
    self.commander = behaviour.commander

    print(f"{self.agent.name}: Create command sent.")
    self.agent.position = await self.commander.
        create_agent(self.agent)
```

Figura 3.23: Extracto de código interno de la clase `StateInit`.

La construcción de todos los comandos, en el código de la API `Commander`, siempre sigue la estructura de crear un diccionario con las claves:

- `commandName`: El valor es el nombre interno del comando. Es usado para que la clase `CommandParser` del simulador (ver sección 3.3.2) sepa qué comando le estamos enviando y así lo decodifique correctamente.

- `data`: El valor es la lista que contiene los parámetros o la información adicional que requiere el comando.

La figura 3.24 muestra el código del comando `create_agent` y podemos ver los valores que se utilizan en el diccionario. La clave `commandName` tiene el valor `create`, la clave `gameObject` tiene el valor del nombre del agente y la clave `data` contiene el nombre del tipo del avatar del agente, la posición inicial donde aparecerá y la bandera que indica si tiene las colisiones activadas.

```
async def create_agent(self, agent: Agent) -> list:
  command = { 'commandName': 'create', 'data': [agent.
     name, agent.prefab_name] }
  position = agent.starter_position
  if isinstance(position, str):
    command['data'].append(position)
  else:
    command['data'].append(f"({position['x']} {
       position['y']} {position['z']})")
  command['data'].append(agent.agent_collision)
  agent_position = (await self.
     send_command_to_server_and_wait(command)).decode(
     'utf-8')
  return [float(x) for x in (agent_position.split())
     [1:]]
```

Figura 3.24: Extracto del código interno del método `create_agent` de la clase `Commander`.

El diccionario es convertido en una cadena de texto con formato JSON (ver Figura 3.25) y es enviado por red directamente al puerto de comandos del servidor del simulador. La comunicación usa el protocolo TCP (*Transmission Control Protocol*) para asegurar la correcta transmisión de los paquetes de red y garantizar la integridad de los mensajes que se transmiten, independientemente de su tamaño.

```
{
    "commandName": "create",
    "data": ["agent1", "Tractor", "Spawner 1",
       ↪ true]
}
```

Figura 3.25: Ejemplo de comando de creación en formato JSON.

La información llega al puerto de comandos del servidor del simulador (ver Figura 3.26), y a continuación el mensaje es recogido por el hilo de la instancia de la clase `TcpCommandManager` al que está vinculado el agente que envió el comando. La clase `TcpCommandManager` convierte el flujo de datos en una cadena de texto con formato JSON y después es procesada por la clase `CommandParser`, que convierte la cadena JSON en una instancia de la clase `CommandInfo` que contiene la información necesaria para generar el comando enviado por el agente. Finalmente, la clase `CommandParser` genera el comando `CreateCommand` con la información retornada por el objeto `CommandInfo` y se lo devuelve a la clase `TcpCommandManager` para que lo encole en la cola (segura para hilos) compartida con la clase `TcpServer` para su posterior ejecución.

Protocolo de imágenes

En esta sección se ilustrarán las dos fases del protocolo que interviene en el envío de imágenes, mostrando la ruta que sigue una captura del entorno desde que es tomada por el avatar del agente, hasta que llega al estado de cognición del agente.

La primera fase del protocolo de imágenes consiste en la identificación del agente en el simulador. La clase de los agentes que gestiona las comunicaciones de red por el puerto de imágenes del simulador (que de igual forma que el puerto de comandos, usa TCP) se llama `ImageManager`. La clase `ImageManager` se ejecuta en un hilo, y cuando el

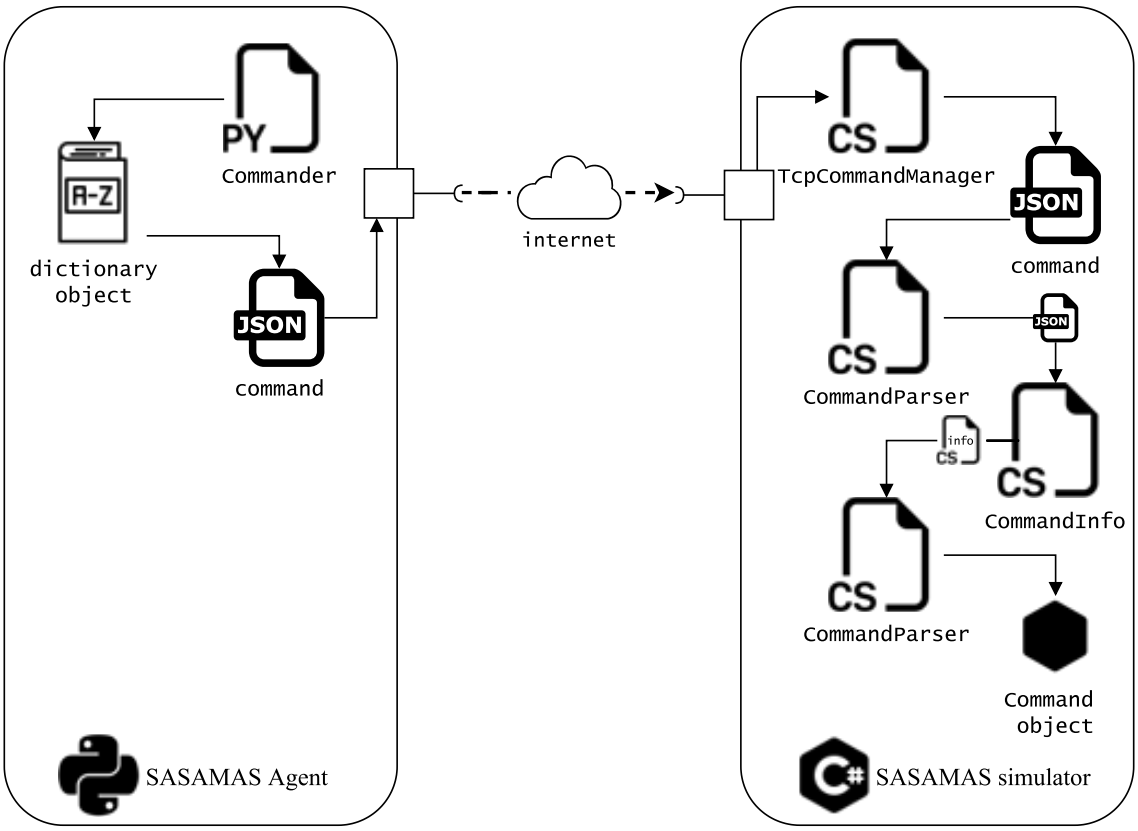


Figura 3.26: Esquema de creación y envío de un comando por el canal de comunicaciones TCP entre el agente y el simulador SASAMAS.

agente ejecuta su instancia en el estado inicial, lo primero que hace esta clase es mandar el nombre del agente en texto plano para que el simulador pueda relacionar el avatar del agente, con el *socket* que mantendrá la comunicación.

La segunda fase del protocolo (ver Figura 3.27) comienza una vez formada la asociación entre el agente y su avatar. Permite que las imágenes tomadas por el avatar sean enviadas por red cuando el agente envía el comando que activa la cámara del avatar. La imagen no se envía directamente, sino que después de ser tomada por la clase `CameraManager` se convierte primero a formato `base64` y se incluye en una instancia de la clase `ImageData`, que también contendrá el índice de la cámara con la que fue tomada y la fecha y hora en formato UTC. La instancia de la clase `ImageData` se encola en una cola (segura para hilos) compartida con la clase `TcpImageManager` para su posterior proceso. Cuando la clase `TcpImageManager` desencola la instancia de `ImageData`, la instancia es convertida a una cadena de formato JSON y es enviada al agente mediante el *socket* de imágenes.

El agente recompone la cadena de texto en formato JSON en la clase `ImageManager` y la usa en el constructor de la clase `ImageData` para obtener una instancia con todos los datos que han sido recibidos, donde en el código del propio constructor la imagen es decodificada de `base64` a un conjunto de *bytes*.

Finalmente, en el estado interno de cognición del agente, la instancia de `ImageData` se extrae de la pila (segura para hilos) de imágenes compartida, se guarda la imagen en un archivo si el parámetro `image_buffer_size` del archivo `configuration.json` tiene un valor mayor a cero, y se pasa la referencia de la instancia de la clase `ImageData` al estado de cognición expuesto en la clase `EntityShell` para que podamos utilizarla de manera cómoda y sencilla.

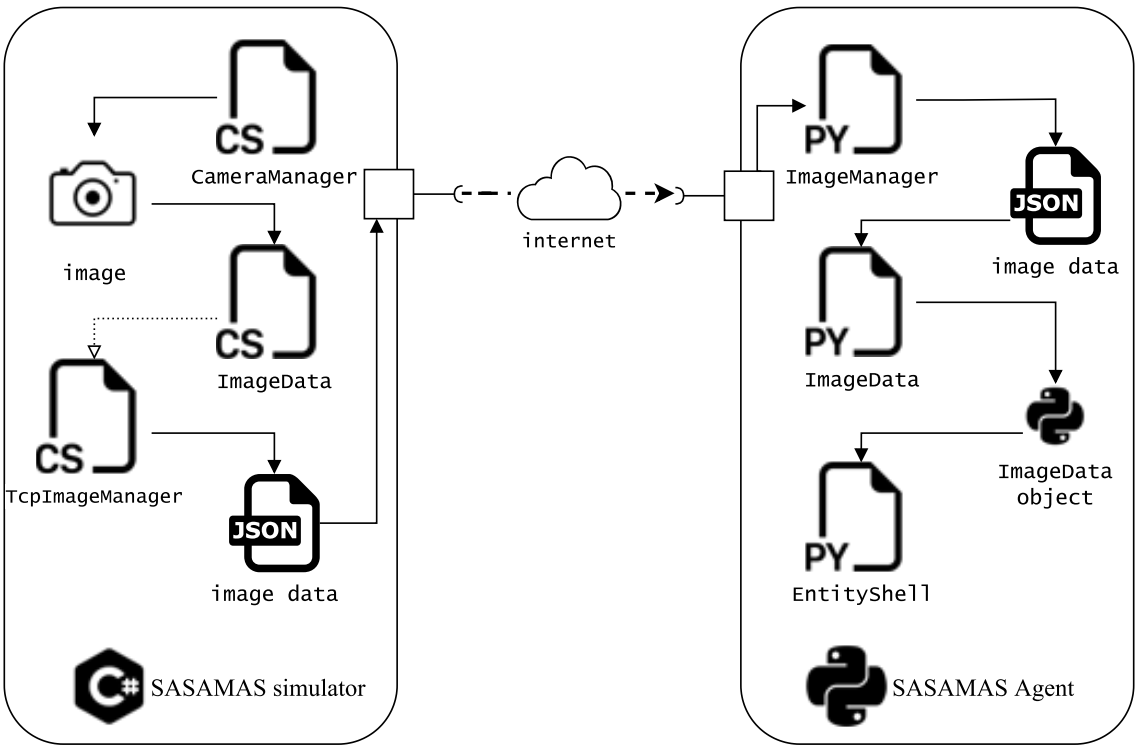


Figura 3.27: Esquema de creación y envío de una imagen por el canal de imágenes TCP entre el agente y el simulador SASAMAS.

CAPÍTULO 4
Desarrollo de la solución

4.1 Introducción

En este capítulo veremos en qué consistió la propuesta inicial, así como los problemas encontrados durante la implementación del marco de trabajo y las soluciones que han sido realizadas para abordar los problemas surgidos.

4.2 Propuesta inicial

La propuesta inicial fue evolucionando con el tiempo. Si bien desde el primer momento el proyecto consiste en construir un sistema completo formado por un simulador y agentes inteligentes que interactúan con el entorno simulado, se contemplaron diversos escenarios y arquitecturas.

Se estudió simular la física en un servidor programado con Python, al que los agentes inteligentes se conectan. Así como programar un visor tridimensional, que también iría conectado al simulador (ver Figura 4.1), para observar el comportamiento de los agentes.

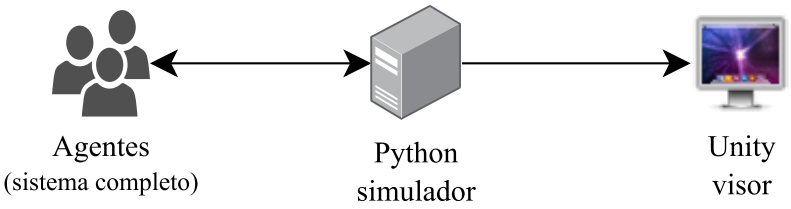


Figura 4.1: Esquema de la propuesta inicial donde la visualización y el simulador estaban desacoplados.

Esta idea se descartó, porque si bien es posible su realización, existe una mejor solución. La propuesta final consiste en implementar la simulación y la visualización en una misma pieza. De esta manera, la máquina ahorra el cómputo de procesar un mecanismo de comunicación entre el simulador basado en Python y el visor Unity, y se mejora la fluidez de la escena. Esto es así porque la herramienta desarrollada ha sido pensada para validar los algoritmos gráficamente, por lo que siempre que se ejecute el simulador para probar un experimento el usuario comprobará su correcto funcionamiento viendo el comportamiento de los agentes en la escena.

Por otra parte, se barajaron distintas opciones para probar el sistema completo y finalmente se decidió usar un problema real de la línea de investigación de sistemas multiagente del VRAIN. El problema consiste en un conjunto de agentes inteligentes habitando

un IVE, equipados con una red neuronal profunda desarrollada bajo la guía de uno de los investigadores expertos en IA (Inteligencia Artificial). Los agentes recorren un huerto de naranjos e identifican enfermedades en las frutas con la red neuronal profunda. La prueba consistirá en validar el funcionamiento de la red neuronal profunda en un entorno simulado, no obstante, cabe recalcar que la construcción de la red neuronal no es parte de este trabajo.

4.3 Problemas en la implementación y soluciones realizadas

4.3.1. Modificar los avatares desde hilos

El principal problema que surgió al usar Unity como simulador es que para poder modificar las propiedades de un objeto —como son los avatares de los agentes— de la escena, únicamente se puede hacer desde el hilo principal de Unity. Esto plantea una dificultad adicional, porque los comandos son gestionados por los hilos generados por el hilo principal y los comandos requieren modificar las propiedades del agente al que están asociados.

Este problema se solucionó usando una estructura de datos de tipo cola, segura para hilos, compartida entre el servidor y los hilos de los agentes. De esta manera, cada vez que un agente envía un comando a su avatar, el comando es recibido por el hilo del agente y encolado para que el hilo principal sea el que ejecuta la orden.

4.3.2. Retardos en la comunicación por red

Existen otras dificultades que aparecieron en el diseño de las comunicaciones entre los agentes y el simulador. Como la resolución de la cámara de los agentes es modificable y la frecuencia de realización de las imágenes es variable por el código de los agentes, era posible que con valores tiempo entre capturas muy bajos y una resolución de imagen alta, el flujo de imágenes enviado por el canal de red entre el simulador y los agentes ocasionara retardos para el flujo de comandos.

La solución realizada consiste en crear un nuevo canal de comunicaciones entre los agentes y el simulador, únicamente dedicado a la transmisión de imágenes, de manera que los agentes poseen un hilo que gestiona el flujo de imágenes y las transfiere al hilo principal. Compartir recursos entre hilos es un proceso delicado, porque si no se usan mecanismos de protección pueden ocurrir situaciones indeseadas como la condición de carrera. Por lo tanto, para la solución ha sido empleada una cola, segura para hilos, y de esta manera se ha conseguido liberar el canal de envío de comandos.

4.3.3. Vincular a los agentes con su canal de imágenes

La creación de un nuevo canal dedicado al envío de imágenes (ver la sección 4.3.2) conlleva un nuevo problema a resolver. El problema consiste en vincular (desde el servidor) el nuevo canal de imágenes con el agente al que pertenece.

El simulador es el encargado de generar a los avatares de los agentes cuando estos mandan un comando de creación en la escena y los agentes tienen nombres únicos. Estas dos características nos permiten crear una estructura de datos llamada tabla de dispersión, donde la clave es el nombre del agente y el valor es la referencia al avatar que crea el simulador.

Por lo tanto, la solución a la cuestión inicial consiste en poblar la tabla en el momento de creación del agente. De esta manera, cuando el canal de imágenes se crea y el agente envía su nombre (por primera y única vez) por el nuevo canal al principio de la comunicación, el simulador puede buscar en la tabla de dispersión (usando el nombre del agente como clave) la referencia del avatar del agente al que pertenece. Este proceso se realiza en un tiempo de cómputo constante respecto a la cantidad de agentes en la simulación, debido a las propiedades intrínsecas de la tabla de dispersión.

4.3.4. Tiempo de carga

El tiempo de carga de la simulación no es solo una propiedad relacionada con la comodidad del usuario, ya que si bien un tiempo de carga bajo resulta satisfactorio, también ahorra tiempo a los usuarios —que es uno de los objetivos planteados en la sección 1.2—. Por otra parte, un tiempo de carga excesivamente alto es un problema importante, ya que puede convertir a la herramienta en un programa inutilizable.

Además, SASAMAS permite la carga de texturas en tiempo de ejecución (ver Figura 4.2) y su aplicación aleatoria sobre elementos —como se detalló en la sección 3.3.2—, para así proporcionar al usuario flexibilidad en las simulaciones sin tener que editar el mapa de nuevo o directamente tener que realizar una compilación completa. Esta característica requiere realizar un proceso de optimización especialmente cuidadoso para garantizar un tiempo de carga asumible.

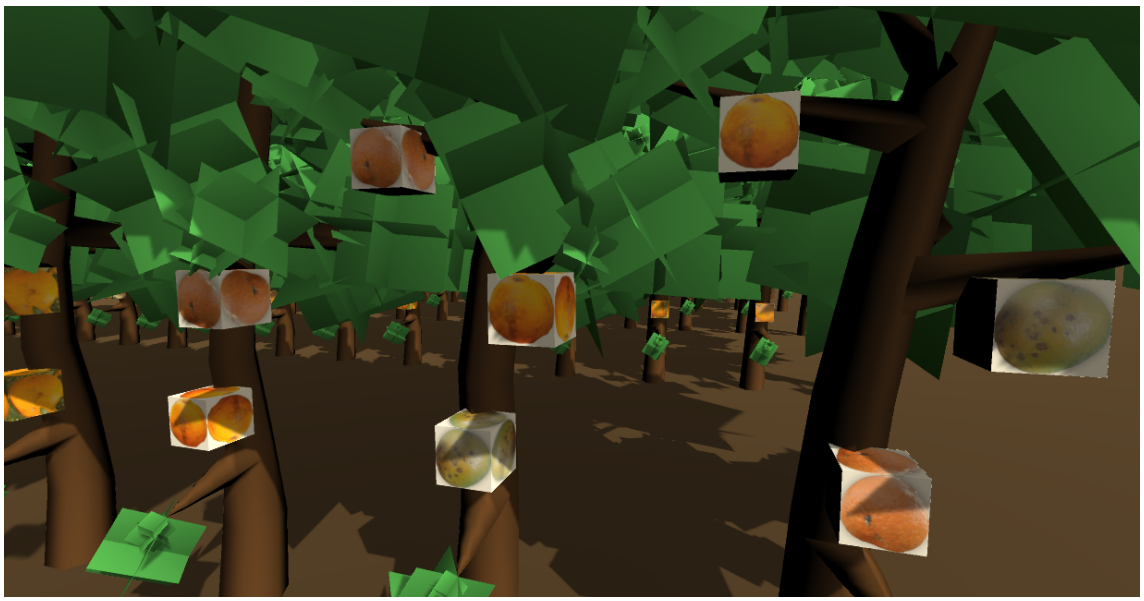


Figura 4.2: Muestra de texturas cargadas en tiempo de ejecución y aplicadas de forma aleatoria.

En este punto cabe destacar que si se desea que un elemento tenga siempre la misma textura en vez de escoger una aleatoria tras cada ejecución, simplemente se puede depositar esa textura de manera aislada en un directorio. De esta forma, cuando cargue las texturas de los elementos no existirá aleatoriedad, porque solo hay una disponible en el conjunto de texturas asociadas a ese elemento.

El comportamiento de la clase `TextureLoader` es clave para la optimización del proceso de carga, porque si realizamos la carga de imágenes sin tener en consideración el objetivo de minimizar el tiempo de carga de la escena, en simulaciones como la del caso de prueba podría resultar inviable. El proceso de lectura de un archivo de imagen en disco es un proceso costoso y a este coste se le suma el coste del siguiente proceso que es la transformación y aplicación de la textura.

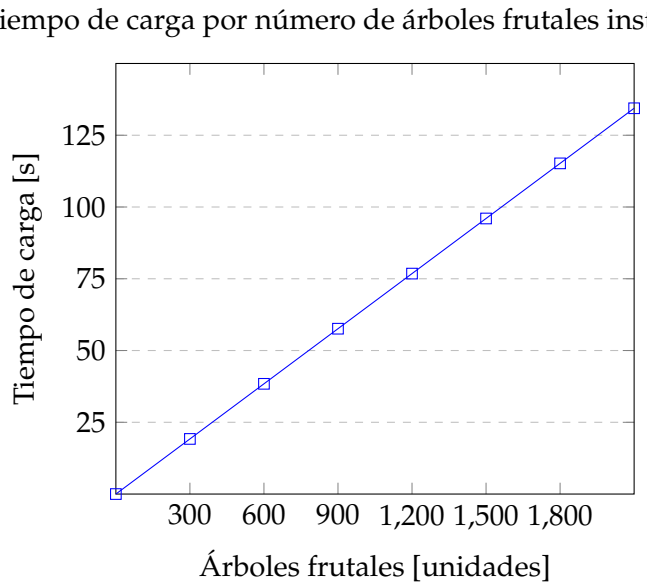


Figura 4.3: Gráfica que muestra el tiempo que costaría generar los árboles frutales, con cuatro frutas cada uno y cargando sus texturas en tiempo de ejecución sin un proceso de optimización

A continuación, realizaremos un experimento para ilustrar la importancia de este problema. El experimento consiste en medir el tiempo de carga de las texturas sin realizar la optimización sobre la clase `TextureLoader` y extrapolar los datos con un número variable de elementos. Los árboles frutales son elementos idóneos para probar el tiempo de carga, ya que cada árbol frutal tiene cuatro objetos del tipo fruta que cargan su textura a partir de la conversión de imágenes en tiempo de ejecución. La dimensión de las imágenes utilizadas para formar las texturas es de doscientos veinticuatro por doscientos veinticuatro píxeles y se escogen de manera aleatoria sobre cinco clases ubicadas en carpetas con cuatro imágenes cada una, sumando un total de veinte imágenes diferentes. Por último, la máquina sobre la que se ha tomado la medición es un ordenador portátil, sin tarjeta gráfica externa y formado por los siguientes componentes: una gráfica integrada modelo *Intel Iris Plus Graphics*, un procesador *Intel Core i7 1035G7*, dieciséis *gigabytes* de RAM (Random Access Memory) y una placa blase *ASUSTeK X421JAY*.

El tiempo medio de carga de las texturas en un huerto de ochenta árboles y trescientas veinte naranjas es de dieciséis milisegundos, por lo que podemos extrapolar estos datos para calcular el tiempo de carga de texturas para un número variable de árboles (ver Figura 4.3).

La optimización consiste en evitar el proceso de carga de imágenes y conversión de imagen a textura por cada elemento que requiera la asignación de una textura en tiempo de ejecución. Para ello, se ha utilizado una estructura de datos denominada tabla de dispersión, que permite relacionar una clave con un valor y el tiempo computacional de búsqueda en dicha tabla es constante en relación a la cantidad de entradas de la tabla. Como se detalló en la sección 3.3.2, la tabla relaciona la ruta al directorio que contiene las imágenes con una lista de las texturas que han sido cargadas. De modo que si es la primera vez que un elemento demanda las imágenes de una carpeta, la clase `TextureLoader` las procesa y almacena la lista de texturas en la tabla de dispersión, para que si posteriormente la mismas texturas son demandadas, únicamente es necesario devolver la referencia a la lista de texturas en vez de volver a realizar el proceso de lectura en disco y conversión a textura por cada imagen.

La validación del correcto funcionamiento del sistema se encuentra en la figura 4.4, donde podemos ver una gráfica que muestra el tiempo (en segundos) que ha tardado

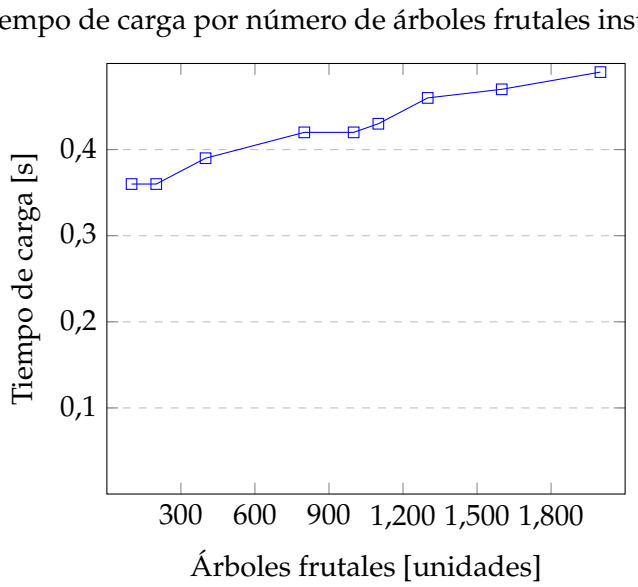


Figura 4.4: Gráfica que muestra el tiempo que ha costado cargar la escena con árboles frutales, con cuatro frutas cada uno y cargando sus texturas en tiempo de ejecución

en cargar la escena completa al poblarla con un objeto de luz, el terreno y un número variable de árboles frutales —indicado en el eje de abscisas—. Cabe destacar que en este experimento empírico se ha medido el tiempo de carga de la escena completa, no solo el proceso de carga de texturas.

La comparación entre ambas gráficas concluye en que gracias a la optimización, no solo hemos pasado de cargar una simulación con mil árboles frutales —cuya carga de texturas duraba más de un minuto— a menos de medio segundo, sino que con el nuevo sistema también es posible la simulación de entornos complejos con una cantidad de elementos que sin la optimización no eran factibles.

4.3.5. Herramientas para crear simulaciones

La elección de las características que iban a tener las herramientas de creación de mapas también supuso un sumidero de horas de trabajo. Las herramientas de edición tienen que proporcionar flexibilidad y detalle al usuario, pero al mismo tiempo tienen que ser fáciles de usar y rápidas para lograr los objetivos planteados en la sección 1.2 y aportar un marco de trabajo innovador al entorno científico.

La primera aproximación consistió en crear una herramienta que permita diseñar el mapa pintando un lienzo con colores, donde los colores se traducirían posteriormente a elementos del entorno. Si bien esta característica es rápida y fácil, ya que el usuario no requiere conocimientos de programación, no es la más deseable por las desventajas que presenta. Existen tres desventajas que propiciaron buscar una alternativa mejor:

- La primera desventaja consiste en la limitación del lienzo. Para poder editar y visualizar el lienzo se requiere de un programa de edición de imágenes, que si bien es fácil encontrar varios sistemas operativos que ya vienen con un programa instalado de estas características, no sería posible usarlo en conexiones a una terminal de texto con protocolos como SSH (*Secure SHell*) o Telnet (*Telecomunication Network*). Es por este motivo que pierde versatilidad.

- La segunda desventaja consiste en la imposibilidad de establecer propiedades sobre elementos individuales. La rotación sobre elementos, la intensidad y color de la luz ambiental son características que no pueden ser definidas intuitivamente en un mapa de colores. El lienzo permite definir su ubicación, pero no editar propiedades individuales de los elementos, por lo que pierde flexibilidad.
- La tercera desventaja consiste en establecer el valor de la tercera componente del eje tridimensional. Las herramientas de edición de imágenes son programas que trabajan sobre dos dimensiones, por este motivo uno de los ejes del espacio tridimensional no podría ser editado con el uso del lienzo.

Finalmente, decidimos que una mejor alternativa era la construcción de una herramienta de edición de mapas basada en texto y desacoplada. En la sección 3.3.2 se detalló la herramienta basada en texto, y podemos observar como la herramienta actual cubre todas las necesidades que surgen con el uso de la herramienta del lienzo.

4.3.6. Capa de abstracción con SPADE

SPADE es un marco de trabajo robusto y formal para desarrollar agentes inteligentes usando el lenguaje de programación Python, como vimos en la sección 2.4.3. Pero el uso directo de SPADE incrementa considerablemente la cantidad de esfuerzo y tiempo que tiene que invertir el usuario para construir una simulación —por simple que esta sea—.

Es por este motivo que se tomó la decisión de crear un sistema que abstraiga al usuario de la dificultad y el coste temporal de definir las bases de los agentes SPADE. Este sistema está compuesto por un conjunto de clases —detalladas en la sección 3.3.1— que facilitan y resumen toda la programación que debe hacer el usuario en una única clase. Esta clase, llamada `EntityShell`, es donde el usuario de SASAMAS programa el comportamiento de los agentes.

En la tercera modificación del caso de prueba (ubicado en la sección 6.8) se compara la cantidad de esfuerzo y tiempo, medido en líneas de código escritas, que necesita el usuario para dotar de un comportamiento simple al agente. Este comportamiento está basado en cambiar el color del avatar del agente cada vez que este alcanza un estado de su FSM (detallada en la sección 3.3.1).

CAPÍTULO 5
Implantación

5.1 Introducción

Este capítulo muestra los pasos que debemos realizar para desplegar cada una de sus partes en un entorno de producción. Este capítulo incluye las instrucciones para la puesta en marcha de los agentes, del simulador y también del servidor XMPP en los que los agentes basan su comunicación.

5.2 Servidor XMPP

La instalación del servidor XMPP es opcional, ya que existen servidores gratuitos y abiertos que podemos usar para que nuestros agentes SPADE se comuniquen. Cualquier servidor XMPP que tenga activo el registro *in-band*[1] es apto para ser utilizado, porque esta propiedad permite a los agentes entrar al servidor sin la necesidad de haber sido registrados previamente con usuario y contraseña. Si los agentes están registrados con usuario y contraseña en un servidor XMPP que no tenga activo el registro *in-band*, hay que verificar que el nombre del agente y la contraseña que hemos introducido en el archivo `configuration.json` (ver sección 3.3.1) coincide con la del usuario que está dado de alta en el servidor XMPP.

Project Name	**Platforms**
AstraChat	Linux / macOS / Solaris / Windows
ejabberd	Linux / macOS / Windows
Isode M-Link	Linux / Windows
MongooseIM	Linux / macOS
Openfire	Linux / macOS / Solaris / Windows
ProsodyIM	BSD / Linux / macOS
Tigase XMPP Server	Linux / macOS / Solaris / Windows

Tabla 5.1: Tabla de servidores XMPP aptos para ser usados por agentes SPADE extraída del enlace **https://xmpp.org/software/servers/**

[1] https://xmpp.org/extensions/xep-0077.xml

El servidor XMPP puede ser escogido entre las opciones que proporciona SPADE (ver tabla 5.1), aunque el recomendado por SPADE y el que usaremos en este despliegue es ProsodyIM[2]. El servicio puede ser instalado en la misma máquina donde se ejecutan los agentes, o en una máquina distinta. Lo recomendable es instalar el servicio XMPP en una máquina distinta si queremos dedicar la mayor cantidad de recursos a los agentes. Si nos interesa el rendimiento, también podemos instalar un sistema operativo basado en Linux, sin interfaz gráfica, para dedicar la mayor cantidad de recursos al servicio XMPP. Una buena alternativa es el sistema operativo Debian 11 [3] ya que es un sistema que cumple los requisitos nombrados y es estable y seguro.

Para instalar ProsodyIM debemos seguir las instrucciones que se muestran en su página web, pero cabe destacar que si queremos habilitar el registro *in-band* deberemos modificar el archivo `prosody.cfg.lua` y establecer la propiedad `allow_registration` a `false`.

El servidor ProsodyIM no está disponible para sistemas operativos Windows[4]. En caso de querer realizar una instalación en ese sistema operativo, podemos usar Openfire[5]. Para instalar Openfire debemos seguir las instrucciones que se muestran en su página web y no es necesario realizar una configuración adicional, ya que la opción del registro *in-band* viene habilitada por defecto.

5.3 Agentes

La implantación de los agentes la podemos hacer de distintas maneras, dependiendo de la configuración que deseemos. La podemos desplegar en el mismo equipo que el simulador y el servidor XMPP, en diferentes equipos dentro de la misma red local que el simulador y el servidor XMPP, en una red distinta al simulador y al servidor XMPP o incluso combinar las opciones anteriores según nuestra preferencia o necesidad. En cualquier caso, la puesta en marcha de los agentes es idéntica y solo variará la configuración del archivo `configuration.json` detallado en la sección 3.3.1.

Una vez hayamos escogido una arquitectura de despliegue, debemos instalar el programa de los agentes accediendo a la dirección de github [6] donde se encuentra y descargarlo en el directorio de nuestra preferencia. Por ejemplo, podemos descargar los archivos en una carpeta llamada *Agentes* contenida en la carpeta de documentos del usuario.

A continuación, definiremos el comportamiento que deseamos que tenga el agente en el archivo `entity_shell.py`. Para ello, usaremos la API explicada en la sección 3.3.1, ya que permite agilizar la programación y reduce la cantidad de líneas de código que requiere nuestro programa, como se mostrará en un caso práctico en la sección 6.8.2 donde se concluye que un código de cuatro líneas de SASAMAS equivale a cien líneas de código sin su uso.

El siguiente paso consiste en editar las propiedades del archivo `configuration.json` según nuestra necesidad. Los detalles del archivo de configuración se encuentran explicados en la sección 3.3.1, pero podemos destacar que en este punto debemos conocer la dirección IP donde se encuentra el servidor XMPP y el simulador, para que los agentes puedan comunicarse con su avatar haciendo uso del protocolo desarrollado en la sección

[2] https://prosody.im
[3] https://www.debian.org
[4] https://www.microsoft.com/es-es/windows
[5] https://www.igniterealtime.org/projects/openfire/
[6] https://github.com/FranEnguix/sasamas/tree/main/release

3.3.3. También escogeremos el lugar de aparición del agente que hemos programado, así como el tipo de avatar que manejará en el simulador.

Si en lugar de un agente queremos que aparezcan varios agentes con el mismo comportamiento, simplemente tenemos que añadir nuevas entradas a la lista de agentes del archivo `configuration.json`, como se muestra en el extracto de ejemplo de configuración en la figura 5.1. De este modo, todos los agentes compartirán el código del archivo `entity_shell.py`.

```
{
...
  "agents": [
    {
      "name": "agente 1",
      ...
    },
    {
      "name": "agente 2",
      ...
    },
    ...
  ]
}
```

Figura 5.1: Extracto de archivo `configuration.json` con varios agentes.

Si por el contrario deseamos tener comportamientos diferenciados para distintos tipos de agentes, como podría ser el caso de un IVE que simule un hormiguero donde existen hormigas (agentes) obreras y guerreras, entonces debemos tener una carpeta con el sistema de los agentes SASAMAS por cada comportamiento distinto. En el ejemplo de las hormigas, tendríamos una carpeta con el comportamiento de los agentes hormiga obrera en el archivo `entity_shell.py` y su configuración en el archivo `configuration.json`, y en otra carpeta distinta tendríamos otro sistema SASAMAS descargado, con el comportamiento `entity_shell.py` de los agentes hormiga guerrera y su archivo de configuración. De este modo, podemos ejecutar de manera independiente los dos comportamientos en el mismo simulador, incluso si estos dos comportamientos están en distintas máquinas y en distintas redes locales.

Para poder ejecutar el sistema hay que poner en funcionamiento a los agentes y para ello, debemos tener instalado en nuestro sistema operativo la versión 3.9 de Python y el paquete SPADE en la versión 3.2.2. La instalación del paquete SPADE la podemos realizar con el gestor de paquetes de Python, llamado *pip*. Simplemente escribimos la orden *pip install spade==3.2.2* y es suficiente para que se descargue e instale SPADE correctamente.

Si de manera adicional quisiéramos ejecutar el código del caso de prueba del capítulo seis, y usar la clase desarrollada `OrangeNetwork` que facilita el acceso y funcionamiento de la red neuronal profunda, debemos instalar los siguientes paquetes adicionales de forma análoga al paquete SPADE: *numpy* en la versión 1.23.0, *pillow* en la versión 9.1.1 y *tensorflow* en la versión 2.9.1. No obstante, en el caso de funcionamiento normal del sistema de agentes de SASAMAS, estos paquetes no son necesarios.

Finalmente, para iniciar el sistema y que los agentes comiencen a conectarse con el simulador, ejecutaremos el archivo `launcher.py` que se encargará de iniciar a los agentes según los parámetros de configuración que hemos establecido.

5.4 Simulador

La puesta en marcha del simulador es más simple que la de los agentes, porque no requiere programación. Primero descargamos el simulador (que se encuentra disponible para los sistemas operativos Linux, Windows y macOS) usando el enlace donde se encuentra la versión compilada del mismo [7] y lo ubicamos en la carpeta de nuestra preferencia, por ejemplo, una carpeta contenida en los documentos del usuario del sistema operativo.

El archivo compilado viene con una plantilla básica de los archivos de configuración (`map.txt`, `map_config.json` y `map.json`) que detallamos en la sección 3.3.2, por lo que los editaremos para generar un entorno personalizado. Si deseamos agregar nuevos tipos de elementos al entorno para usarlos en los archivos de configuración, debemos descargar el proyecto fuente del simulador[8]. Una vez lo tenemos descargado en el directorio de nuestra preferencia, instalaremos el editor de Unity en la versión 2020.3.20f1 y abriremos el proyecto SASAMAS descargado. Con el editor de Unity abierto y el proyecto SASAMAS cargado, seguiremos los pasos descritos en la sección 3.3.2 para añadir nuevos tipos de elementos. Finalmente, abriremos la ventana *Build Settings* que se encuentra en el botón *File* del menú superior del editor y compilaremos el proyecto indicando la plataforma a la que queremos exportarlo, como se muestra en la figura 5.2.

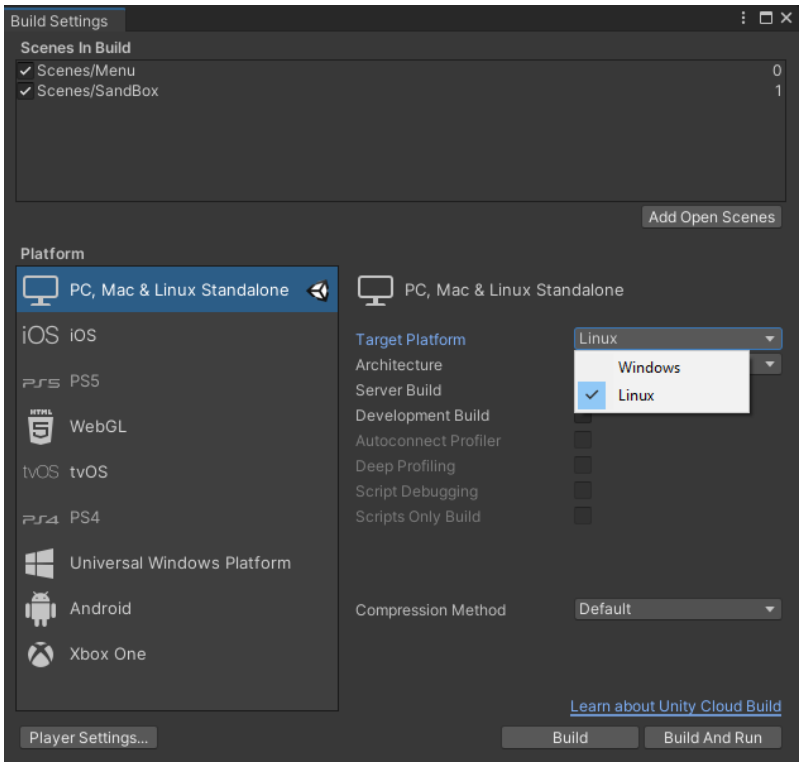


Figura 5.2: Ventana de compilación del proyecto SASAMAS en el editor Unity.

Una vez tenemos el archivo del simulador compilado y los archivos de configuración editados para obtener las condiciones ambientales y la disposición de elementos en el entorno deseada, simplemente tenemos que ejecutar el archivo compilado. A continuación, escena principal del simulador se mostrará en pantalla (ver Figura 3.12a), donde podemos ver que se nos presentan tres botones. El último botón contiene el texto *Quit* y sirve para cerrar la aplicación. El botón del medio contiene el texto *Options* y sirve para

[7] https://github.com/FranEnguix/sasamas/tree/main/release
[8] https://github.com/FranEnguix/sasamas/tree/main/src

abrir un menú de opciones y poder configurar los puertos de red del simulador, tanto el de comandos como el de imágenes. En el menú de opciones también podemos encontrar ajustes gráficos que nos permiten habilitar o deshabilitar el modo de pantalla completa, así como cambiar la resolución a la que se muestra el simulador, escogiendo la deseada de un control desplegable que automáticamente reconoce la lista de resoluciones que acepta el monitor donde está siendo ejecutado. Finalmente se encuentra el primer botón, que contiene el texto *Start*. Este botón inicia la construcción del entorno usando los archivos de configuración que hemos editado, así como la escucha a través de los puertos de red que hemos configurado en el menú de opciones del simulador, dando inicio a la simulación del entorno virtual inteligente.

5.5 Orden de ejecución

El orden de ejecución es relevante en la puesta en marcha del sistema, ya que si los agentes inteligentes SPADE carecen de un servidor XMPP activo en la dirección que hemos establecido en su archivo de configuración, fallarán y no podrán conectarse al simulador SASAMAS.

La figura 5.3 muestra el grafo de dependencias entre los módulos que intervienen en la ejecución del sistema SASAMAS. Las flechas que inciden sobre un nodo indican las dependencias vinculadas a dicho nodo, por lo que podemos observar que la ejecución del módulo compuesto por los agentes SASAMAS depende directamente de que el servidor XMPP esté operativo y que el servidor que ejecuta el simulador SASAMAS también esté iniciado y a la escucha de nuevas peticiones de red.

Figura 5.3: Grafo de dependencias de los módulos que intervienen en la ejecución del sistema SASAMAS.

Por otra parte, el simulador SASAMAS no utiliza el servicio proporcionado por el servidor XMPP. Debido a esto, el simulador SASAMAS no depende del servidor XMPP y puede ser ejecutado antes de ejecutar el servicio de mensajería. Del mismo modo, el servidor XMPP no utiliza los servicios del simulador SASAMAS, ya que no están relacionados. Por este motivo, el servidor XMPP también puede ejecutarse antes que el simulador SASAMAS y no existe una dependencia entre ellos.

CAPÍTULO 6
Caso de prueba

6.1 Introducción

En este capítulo vamos a utilizar el marco de trabajo desarrollado para probar un experimento real del área de investigación de sistemas multi-agente del VRAIN. El experimento consiste en implementar un agente inteligente que conduce un tractor y recorre un huerto de naranjos. El agente está equipado con una red neuronal profunda y una cámara, con el objetivo de tomar capturas de las naranjas y clasificarlas en cinco clases (dependiendo de si la naranja está sana o de si padece alguna enfermedad) haciendo uso de la red, de este modo podemos probar si la red neuronal profunda tiene una tasa de acierto apta para llevar a cabo el experimento en la vida real. Finalmente, realizaremos una **modificación** sobre el entorno y el agente para ilustrar la sencillez y rapidez que SASAMAS brinda a sus usuarios.

6.2 Construcción de la simulación

El entorno lo construiremos usando los archivos explicados en la sección 3.3.2. Empezaremos creando la **estructura del huerto de naranjos**, añadiendo columnas de árboles frutales y un lugar de aparición para nuestro agente. Para hacer la validación de la prueba más visual, vamos a hacer cinco columnas (una por cada clase) y cada columna tendrá dieciséis árboles. Para añadir variabilidad, todos los árboles de la misma columna tendrán frutos de la clase a la que pertenezcan, escogidos de manera aleatoria tras cada ejecución.

Modificamos el archivo `map.txt` y escogemos las letras que mejor representen al elemento que instanciarán en la escena. Por ejemplo, podemos usar: la letra `A` para el lugar de aparición del agente, la `B` para la clase *Black Spot*, la `C` para la clase *Canker*, la `G` para la clase *Greening*, la `H` para la clase *Healthy* y la `S` para la clase *Scab*.

A continuación, crearemos el **vínculo entre la letra definida** en el archivo `map.txt` (ver Figura 6.1) y el elemento que representa. A tal efecto, el siguiente archivo que modificaremos será `map_config.json`, agregando un nuevo elemento en `symbolToPrefabMap` por cada letra que hayamos usado en el archivo `map.txt`. En este caso, además de especificar que una letra (`symbol`) se relaciona con un elemento (`prefabName`), necesitamos especificar de qué directorio serán cargadas las imágenes para poder **generar las texturas** de las naranjas. Por lo tanto, como queremos que cada columna de naranjos escoja una imagen aleatoria para cada naranja, debemos crear cinco directorios, cada uno con las imágenes que deseemos que se conviertan en texturas y se apliquen a las naranjas, y finalmente establecer la propiedad `dataFolder` con la ruta a los directorios (ver Figura 6.2).

```
1  A
2    B  C  G  H  S
3    B  C  G  H  S
4    B  C  G  H  S
5    B  C  G  H  S
6    B  C  G  H  S
7    B  C  G  H  S
8    B  C  G  H  S
9    B  C  G  H  S
10   B  C  G  H  S
11   B  C  G  H  S
12   B  C  G  H  S
13   B  C  G  H  S
14   B  C  G  H  S
15   B  C  G  H  S
16   B  C  G  H  S
17   B  C  G  H  S
```

Figura 6.1: Contenido del archivo `map.txt`.

También podemos establecer el espaciado horizontal y vertical entre elementos modificando los valores de la propiedad `distance`, en este caso podemos usar seis unidades de espaciado horizontal para que el agente pueda pasar entre las columnas, y tres unidades de espaciado vertical.

Finalmente podemos añadir las **condiciones ambientales** que deseemos para nuestra simulación editando el archivo `map.json`. En este caso podemos usar las condiciones ambientales por defecto —diurna—, pero podríamos cambiar las condiciones ambientales modificando la posición de la luz, la rotación, el color o la intensidad. También podemos definir dos condiciones ambientales, como una diurna y otra nocturna (o usar las que vienen dadas por defecto) y utilizar la bandera `active` para habilitar únicamente la que necesitemos en la simulación actual (ver Figura 6.3), de este modo evitamos tener que borrar o volver a editar las propiedades de la luz para cambiar las condiciones ambientales.

6.3 Red neuronal profunda

La clasificación de naranjas consiste en detectar si una naranja está sana o si por el contrario, la naranja padece alguna enfermedad. Para ello, se entrenó a la red neuronal profunda bajo la guía de un investigador experto en IA, usando la base de datos de Citrus [16]. Esta base de datos tiene cinco clases de frutas y hojas: *Black Spot*, *Canker*, *Greening*, *Healthy* y *Scab*. El número total de imágenes disponibles en el conjunto de datos es de setecientas cincuenta y nueve, resultado de sumar las imágenes de los frutos y de las hojas repartidas en estas cinco clases.

La red utilizada para los experimentos es una MobilenetV2[4]. Se decidió utilizar un máximo de cien *epochs*, ya que un mayor número de *epochs* podría dar lugar a un sobreajuste. La tasa de aprendizaje de todos los modelos es la misma (lr=0,001). Se activó el parámetro *data augmentation* y el *fine training* para mejorar el entrenamiento.

La figura 6.4 muestra la matriz de confusión obtenida en el proceso de entrenamiento. Como se puede apreciar en la matriz, el grueso de la probabilidad ocurre en la diagonal. Por lo tanto, la red está clasificando los datos de entrenamiento con una precisión

```
{
  "origin": {
      "x": 0.0,
      "y": 0.0,
      "z": 15.0
  },
  "distance": {
      "x": 6,
      "y": 3
  },
  "symbolToPrefabMap": [
      {
        "symbol": "B",
        "prefabName": "Tree Fruit Variant",
        "dataFolder": "C:/Users/Fran/Pictures/oranges/black spot"
      },
      {
        "symbol": "C",
        "prefabName": "Tree Fruit Variant",
        "dataFolder": "C:/Users/Fran/Pictures/oranges/canker"
      },
      {
        "symbol": "G",
        "prefabName": "Tree Fruit Variant",
        "dataFolder": "C:/Users/Fran/Pictures/oranges/greening"
      },
      {
        "symbol": "H",
        "prefabName": "Tree Fruit Variant",
        "dataFolder": "C:/Users/Fran/Pictures/oranges/healthy"
      },
      {
        "symbol": "S",
        "prefabName": "Tree Fruit Variant",
        "dataFolder": "C:/Users/Fran/Pictures/oranges/scab"
      },
      {
        "symbol": "A",
        "prefabName": "Spawner"
      }
  ]
}
```

Figura 6.2: Contenido del archivo `map_config.json`.

aceptable, ya que para cualquier entrada de la clase *i* del conjunto de entrenamiento, la probabilidad de que la red prediga que pertenece a esa clase *i* es de mínimo el cincuenta y nueve por ciento.

6.4 Preparación del agente

6.4.1. Configuración

La configuración del agente consiste en editar el archivo `configuration.json` visto en detalle en la sección 3.3.1. Los parámetros que modificaremos serán la dirección IP y

```json
{
  "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
},
```

(a) Configuración de luz diurna activa.

```json
{
  "active": false,
  "objectName": "Moon Light",
  "objectPrefabName": "Light",
  "position": {
    "x": 0.0,
    "y": 3.0,
    "z": 0.0
  },
  "rotation": {
    "x": 0.0,
    "y": -30.0,
    "z": 0.0
  },
  "color": {
    "r": 0.51,
    "g": 0.70,
    "b": 1.0,
    "a": 1.0
  },
  "intensity": 0.2
}
```

(b) Configuración de luz nocturna deshabilitada.

Figura 6.3: Extracto del archivo `map.json`.

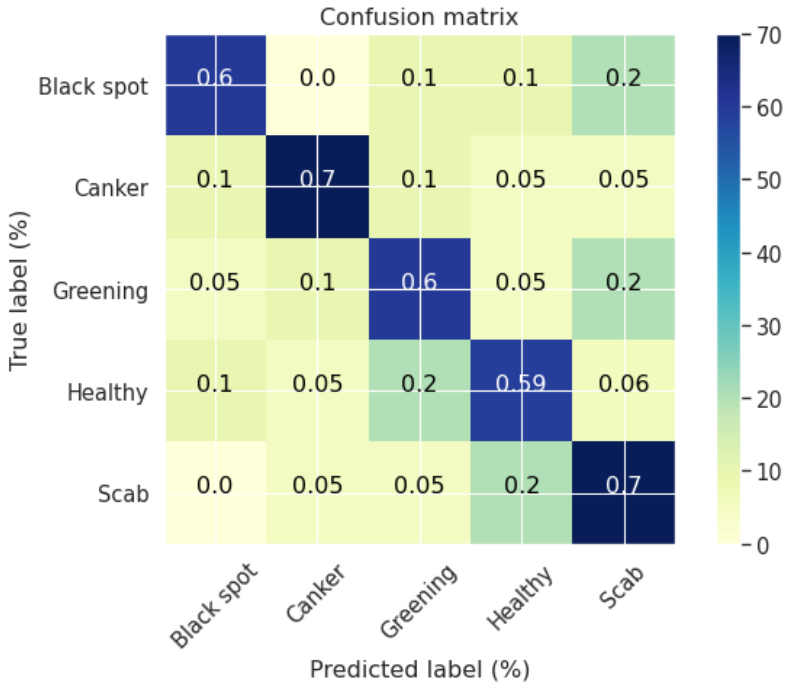


Figura 6.4: Matriz de confusión de la red obtenida durante el entrenamiento.

los puertos del simulador, y la configuración del agente. En este caso de prueba, el agente recibirá el nombre de *agente1* y dispondrá de un `buffer_size` de ochenta —dieciséis árboles, en cinco columnas— unidades, para poder almacenar todas las capturas de ma-

nera automática sin que se sobrescriban entre ellas. Finalmente, ajustaremos el lugar de aparición del agente al *Spawner 1* que se creará por la letra `A` del archivo `map.txt` que aparece en la figura 6.1.

6.4.2. Programación

La cámara del avatar del agente la ajustamos en el método `init`. Para conseguir un enfoque preciso en la fruta podemos modificar tanto el campo de visión, como la posición de la cámara, que gracias al uso de la clase `Commander` se realiza con dos sencillas instrucciones. De esta manera, conseguimos un enfoque correcto de las naranjas mientras el avatar del agente se mueve por el huerto.

Las imágenes llegan al método `perception`, pero como hemos establecido la propiedad `buffer_size` del archivo `configuration.json` en ochenta (dieciséis árboles, en cinco columnas) unidades, el marco de trabajo SASAMAS se encarga de guardar las imágenes automáticamente en el directorio de capturas que hemos establecido en la configuración. Por lo que no necesitamos programar este método.

Después de entrenar la red se creó una nueva clase para ser utilizada en los agentes, llamada `OrangeNetwork`. La clase `OrangeNetwork` tiene la red neuronal integrada y posee un método que se encarga de procesar todas las imágenes de un directorio destino y las clasifica en una de las cinco clases. También se puede ejecutar mientras el agente está capturando las imágenes, porque la nueva clase tiene un mecanismo de control para no procesar dos veces la misma imagen. El mecanismo consiste en guardar la fecha de modificación de la última imagen procesada, así si se reanuda el proceso sobre el mismo directorio de capturas, únicamente procesará las imágenes restantes.

A continuación, integramos el uso de la nueva clase `OrangeNetwork` en el método `cognition` de los agentes. Para integrarla es suficiente con usar el método `is_alive` para comprobar si ha finalizado de procesar el lote actual, y si es así y quedan más lotes por procesar, se vuelve a ejecutar su método `start`. El método `start` reanuda el proceso de clasificación seleccionando únicamente las imágenes que no han sido procesadas ya. El resultado del proceso de imágenes lo encontraremos en el directorio de salida de la red, genera un archivo `log.txt` que muestra el tiempo de proceso de cada imagen junto con la predicción de la clase.

Finalmente, la programación del método `action` incluye el movimiento del agente por el huerto y la toma de imágenes. Es importante especificar que el avatar ajuste y rote la cámara, porque cuando termina el primer conjunto de dieciséis árboles y se dirige a la nueva columna, la posición del agente está invertida y si no se realizan estos ajustes, obtendremos capturas de la parte trasera de los árboles que ya habíamos capturado frontalmente.

Ejecutar los agentes nos permite validar, de forma sencilla, el comportamiento de la red neuronal profunda en un entorno simulado y previo al real.

6.5 Resultado del caso de prueba

Las ochenta imágenes capturadas por el agente han sido procesadas por la red neuronal y la figura 6.5 muestra la matriz de confusión obtenida. Como se puede apreciar, la matriz de confusión obtenida en la fase de entrenamiento (ver Figura 6.4) con la partición de imágenes de prueba, es mejor que la obtenida en la etapa de validación.

Estos resultados se deben a que las imágenes de prueba fueron tomadas en condiciones controladas. Mientras que las imágenes de simulación fueron capturadas en am-

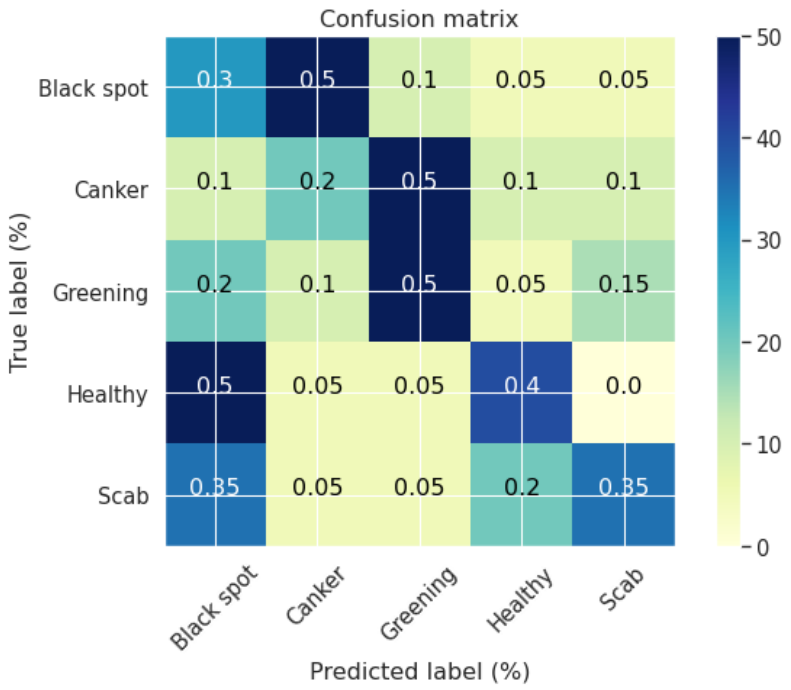


Figura 6.5: Matriz de confusión de la red obtenida durante la validación.

bientes no controlados, pudiéndoles afectar las condiciones de iluminación, la posición y orientación de las naranjas. Por lo tanto, dado que los resultados de clasificación no son los deseables, es recomendable volver a entrenar la red.

Aunque la clasificación de las imágenes no llega a ser totalmente precisa, se puede considerar como un buen resultado para el simulador. Si bien el clasificador puede fallar, el resultado obtenido evita un despliegue real del experimento con tractores, robots u otra maquinaria especializada, ahorrando recursos humanos y materiales.

6.6 Modificación uno del caso de prueba

6.6.1. Introducción

En esta sección vamos a realizar modificaciones sobre el caso de prueba con el fin de ilustrar lo sencillo que es cambiar una simulación en el marco de trabajo desarrollado. La modificación consiste en obtener más velocidad a la hora de capturar las imágenes del huerto de naranjos, por lo que vamos a incluir a cuatro agentes más, para sumar un total de cinco agentes que se encargarán de tomar imágenes. Los nuevos agentes aparecerán cada uno en una columna distinta, de esta manera solo será necesario que todos recorran su columna una vez para obtener capturas de todos los árboles del huerto. También vamos a modificar las condiciones ambientales para que las capturas sean tomadas de noche.

6.6.2. Cambios en el simulador

El cambio de las condiciones ambientales para transformar un escenario diurno, a uno nocturno se realiza modificando los objetos de luz descritos en el archivo `map.json`. Podemos modificar los parámetros `color` e `intensity` del objeto activo `Sun Light` y así convertirlo en una luz nocturna, pero el simulador SASAMAS proporciona la descripción

de un objeto `Moon Light`, desactivado por defecto, que ya tiene las condiciones ambientales que necesitamos. Para desactivar el objeto `Sun Light` y activar el objeto `Moon Light` simplemente debemos modificar la propiedad `active` que se muestra en la figura 6.6.

```
{
  "active": false,
  "objectName": "Sun Light",
```

(a) Configuración de luz diurna deshabilitada.

```
{
  "active": true,
  "objectName": "Moon Light",
```

(b) Configuración de luz nocturna activa.

Figura 6.6: Extracto del archivo `map.json` modificado.

El último cambio que vamos a realizar en el simulador no es necesario, ya que consiste en agregar nuevos *spawners*, es decir, puntos de generación de los nuevos avatares. Los agentes no necesitan un *spawner* para generar su avatar en el escenario, pueden aparecer en la posición que deseen definiendo el punto (en coordenadas tridimensionales) de aparición en su archivo de configuración. Igualmente, es una buena idea definir puntos de aparición y usar su referencia en la configuración de los agentes. De esta manera, si queremos hacer alguna modificación del escenario en el futuro y volver a ubicar a los agentes en él, únicamente tendremos que modificar el archivo `map.txt`. Es por este motivo que agregaremos *spawners* para los nuevos agentes, y la forma de hacerlo consiste en agregar cuatro nuevas letras `A` en la fila donde se encontraba la primera (ver Figura 6.7), de modo que, de izquierda a derecha recibirán los nombres internos *Spawner 1*, *Spawner 2*, *Spawner 3*, *Spawner 4* y *Spawner 5* que usaremos en la configuración de los agentes.

```
A  A  A  A  A
 B  C  G  H  S
```

Figura 6.7: Extracto del archivo `map.txt` modificado.

6.6.3. Cambios en los agentes

La modificación de los agentes la podemos agrupar en dos cambios. Un cambio que consiste en modificar el archivo de configuración de los agentes para incluir a los nuevos agentes y el otro cambio consiste en modificar el comportamiento de los agentes, ya que ahora no queremos que todos los agentes recorran el huerto, sino que cada agente recorrerá la columna en la que se generó su avatar.

El archivo de configuración `configuration.json` debe incluir a los cinco agentes, para ello simplemente copiamos el objeto JSON que describe a nuestro primer agente (ver Figura 6.8) y lo duplicamos cuatro veces. A continuación, simplemente hay que modificar cuatro parámetros: `name`, `password` en caso de que estemos usando un servidor XMPP con el registro *in-band* desactivado (ver sección 5.2), `imageFolderName` y `position`. La propiedad `imageFolderName` que especifica el directorio donde se guardarán las capturas podríamos no cambiarla, de esta manera todas las imágenes de todos los agentes estarían agrupadas en un único directorio, pero es mucho más interesante hacer que cada agente deposite sus capturas en un directorio distinto. Al cambiar el directorio donde cada agente guarda sus capturas, podemos hacer que la red neuronal profunda de cada agente se encargue de clasificar las imágenes que ha tomado ese mismo agente. Esto es muy interesante, ya que nos permite desacoplar el sistema de forma muy sencilla y cada agente podría estar gobernado por una máquina distinta, de manera que podríamos distribuir el cómputo que conlleva la clasificación de todas las capturas del huerto de naranjos.

```
{
  "name": "agente1",
  "at": "localhost",
  "password": "xmppserver",
  "imageBufferSize": 80,
  "imageFolderName": "captures1",
  "enableAgentCollision": true,
  "prefabName": "Tractor",
  "position": "Spawner 1"
},
```

Figura 6.8: Extracto del archivo `configuration.json` modificado.

Finalmente, realizaremos el cambio en la programación. En este nuevo escenario los agentes van a recorrer únicamente su columna de árboles, por lo que debemos acortar el recorrido de sus avatares. La modificación sobre este nuevo escenario la haremos en el método `action`.

6.6.4. Resultado de la modificación

El escenario se forma con las nuevas condiciones establecidas y el trabajo que realizaba el primer agente es repartido entre los cinco agentes creados. En este nuevo escenario las capturas de los agentes muestran que las naranjas tienen menos luz incidiendo sobre ellas, esto es debido a que hemos modificado el objeto de luz del entorno para simular un escenario donde solo disponemos de luz de luna.

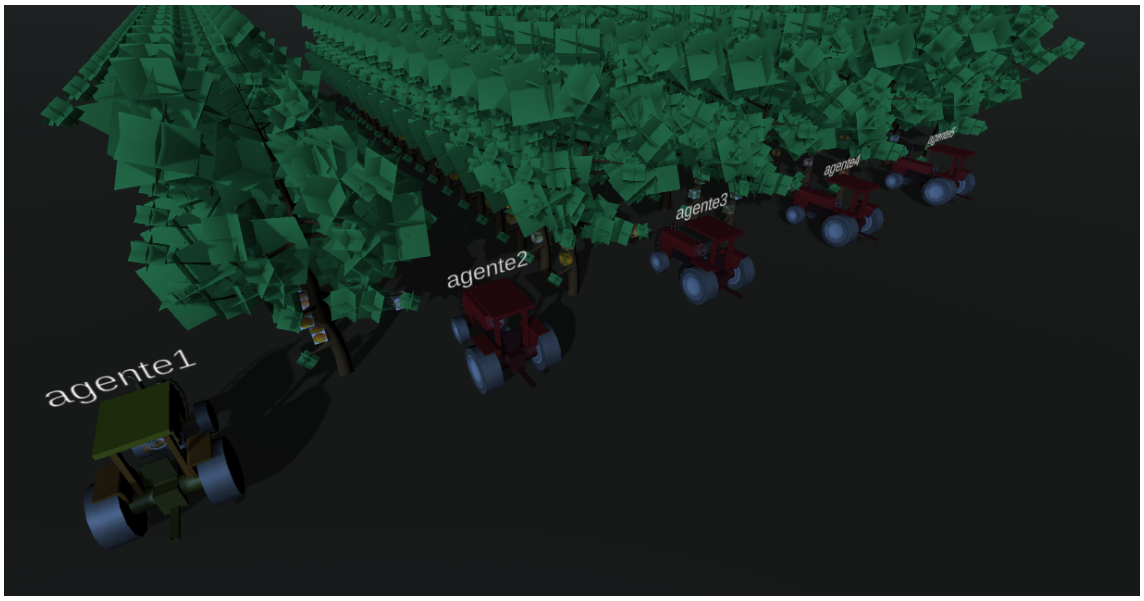


Figura 6.9: Cinco agentes en el escenario del caso de prueba con condiciones ambientales de nocturnidad.

6.7 Modificación dos del caso de prueba

6.7.1. Introducción

La segunda modificación del caso de prueba consiste en aprovechar el sistema desarrollado de carga de texturas en tiempo real y optimizado, que se detalló en la sección 4.3.4, y utilizarlo para extender el área que ocupan los naranjos del huerto. En el caso original habían un total de ochenta naranjos, cada naranjo puede ser de una de las cinco clases totales y posee cuatro naranjas. Las texturas de las frutas son extraídas del directorio al que referencia la clase del árbol en el archivo `map_config.json`, y aplicadas de manera aleatoria para obtener variedad tras cada simulación. La nueva extensión que se abordará en este caso de prueba consiste en escalar los ochenta naranjos iniciales, a un total de dos mil quinientos árboles de naranjas.

6.7.2. Cambios en el simulador

El cambio requerido para realizar el escalado de árboles consiste en editar únicamente el archivo `map.txt`. En este caso, simplemente podemos definir una fila de cincuenta naranjos alternando entre las cinco clases posibles y a continuación replicarla cincuenta veces, como se muestra en la figura 6.10.

```
1 A
2  B C G H S B C G H S B C G H S B C G H S B C G H S B C G H...
```

Figura 6.10: Extracto del archivo `map.txt` modificado.

6.7.3. Resultado de la modificación

El resultado de la modificación es inmediato gracias al sistema desarrollado de carga de texturas en tiempo de ejecución, comentado en la sección 4.3.4. Además, la carga de los dos mil quinientos árboles supone un tiempo de cómputo inferior al segundo (ver Figura 4.4) usando un ordenador portátil sin tarjeta gráfica externa. En la figura 6.11 podemos ver el nuevo aspecto que tiene el entorno simulado cargando los dos mil quinientos naranjos.

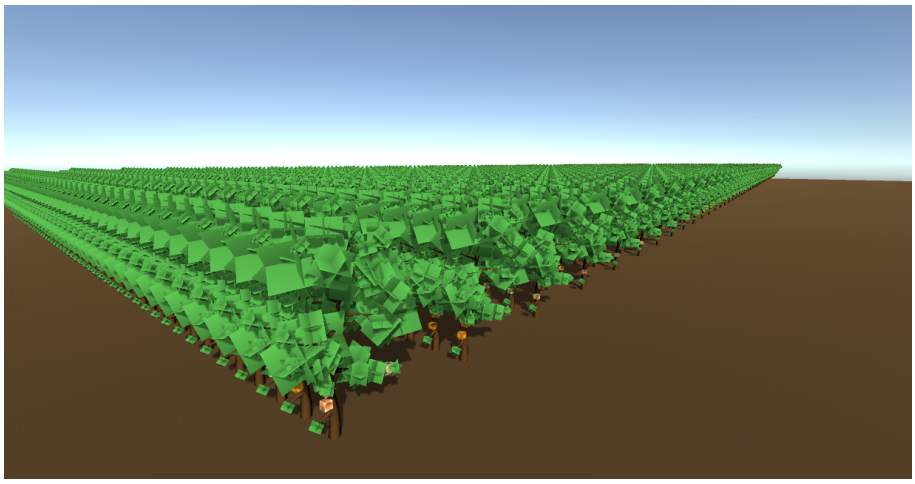


Figura 6.11: Caso de prueba modificado con dos mil quinientos naranjos.

6.8 Modificación tres del caso de prueba

6.8.1. Introducción

La última modificación del caso de prueba consiste en cambiar completamente el comportamiento del agente. En vez de recorrer el huerto, tomar imágenes de las naranjas y clasificarlas haciendo uso de la red neuronal profunda que hemos probado, el agente cambiará de color (ver Figura 6.12) cada vez que alcanza un estado de su FSM (la máquina de estados finitos del agente está detallada en la sección 3.3.1). El avatar del agente empezará por el color gris cuando se encuentre en el estado inicial, después cambiará a rojo en el estado de percepción, para pasar a verde en cognición y finalmente asumirá el azul en acción.

(a) Agente en estado inicial.

(b) Agente en estado de percepción.

(c) Agente en estado de cognición.

(d) Agente en estado de acción.

Figura 6.12: Un agente recorriendo los cuatro estados de la FSM asociada a su comportamiento y cambiando de color tras alcanzar cada uno.

Esta modificación la vamos a realizar con el marco de trabajo SASAMAS que hemos desarrollado, y la realizaremos otra vez usando únicamente SPADE. Esto es así porque concluiremos esta sección comparando el número de líneas de código requerido en ambos casos, con el fin de ilustrar la cantidad de trabajo que se ahorran los usuarios de SASAMAS al usar el marco desarrollado.

6.8.2. Cambios en los agentes

La cantidad de código necesario para realizar este caso de prueba usando el marco de trabajo SASAMAS consiste en **cuatro líneas**. Una línea de código para realizar una llamada al método `change_color` en cada estado, y si deseamos que el agente realice una pausa tras cambiar de color, podemos agregar una línea más tras cada cambio de color (ver Figura 6.13) para detener la ejecución durante el tiempo deseado.

```
1 await commander.change_color(0, 1, 0, 0.8)
2 time.sleep(4)
```

Figura 6.13: Código que se ejecuta durante el estado de cognición del agente de la figura 6.12. La primera línea de código tinta de verde al agente y la última línea de código realiza una espera de cuatro segundos.

Finalmente, realizar este mismo caso de prueba en SPADE conlleva programar del orden de **cien líneas** de código. Dada la gran cantidad de código que implica, no se encuentran incluidas en este trabajo, pero están disponibles para su consulta en el siguiente enlace: **https://github.com/FranEnguix/sasamas**

6.8.3. Resultado de la modificación

Los resultados muestran que para esta modificación del caso de prueba, el marco de trabajo SASAMAS permite ahorrar una gran cantidad de tiempo y esfuerzo a los investigadores. Esto es así gracias al diseño que presenta la herramienta desarrollada, facilitando a sus usuarios, no solo una plantilla de programación como es la clase `EntityShell`, sino también una API `Commander` (detallada en la sección 3.3.1) que proporciona una capa de abstracción que incluye la comunicación con el servidor y los comandos que permiten hacer uso de las funcionalidades del avatar del agente.

CAPÍTULO 7
Conclusiones

La reciente expansión en el campo de los sistemas multi-agente demanda nuevas herramientas para probar algoritmos MAS (Multi-Agent System) en un entorno virtual inteligente, en lo que puede ser un paso previo a la ejecución en el entorno real. Por ello es muy importante, no solo poder **definir rápida y fácilmente entornos virtuales inteligentes**, sino también modificarlos. El nuevo marco de trabajo SASAMAS es una **herramienta versátil**, que proporciona una creación rápida y sencilla de diferentes tipos de entornos virtuales inteligentes y ayuda a los investigadores a probar algoritmos en los que intervienen sistemas multi-agente. Hemos mostrado cómo se puede usar SASAMAS para crear un nuevo IVE con agentes SPADE que lo habitan, y hemos presentado un ejemplo de un huerto de naranjos con agentes robot que prueban una red neuronal profunda para identificar enfermedades en las frutas. También hemos visto cómo de sencillo resulta la modificación del entorno, cambiando no solo la disposición de elementos y el número de agentes de la escena, sino también las condiciones ambientales de la simulación.

El objetivo principal, por lo tanto, hemos logrado cumplirlo exitosamente. Del mismo modo, también han sido cumplidos los **objetivos secundarios** planteados en el primer capítulo. Hemos visto el sistema que permite a los agentes realizar una comunicación por red con el simulador, y entre ellos. También hemos visto cómo el uso de la API (Application Programming Interface) `Commander` desarrollada permite la rápida y fácil programación de los agentes. Otro objetivo inicialmente planteado consistía en crear una herramienta de edición de mapas, que actualmente el simulador cuenta con ella y permite una sencilla creación y modificación de los mismos. Finalmente, también hemos visto cómo de eficaz y fácil es cargar imágenes como texturas sobre elementos del entorno en tiempo de ejecución, por lo que no necesitamos compilar el simulador de nuevo para cambiar estos elementos, ahorrando de este modo tiempo y esfuerzo.

El trabajo presentado está relacionado con un **proyecto coordinado a nivel nacional**, que ha sido preconcedido por el Ministerio de Ciencia e Innovación, con el título de *Servicios inteligentes Coordinados para Áreas Inteligentes Adaptativas* (COSASS-coordinated intelligent services for adaptative smart areas).

Fruto de este trabajo se ha enviado un **artículo científico** a la conferencia internacional PRIMA (*Principles and Practice of Multi-Agent Systems*) donde actualmente se encuentra bajo revisión para ser publicado. Esto es así porque realmente pensamos que esta herramienta supone una gran diferencia respecto a las herramientas que se encuentran disponibles actualmente para la creación de entornos virtuales inteligentes, y también la fácil y rápida modificación de los mismos. SASAMAS no solo ahorra recursos materiales y humanos al permitir probar el experimento en un entorno simulado, en un paso previo al que sería el despliegue en el mundo real, sino que también **permite la colaboración**

desde distintos lugares geográficos del mundo gracias a la capacidad de desacoplamiento de sus piezas y a la conectividad por red que proporciona.

7.1 Resumen de ventajas

En este punto recogeremos todas las ventajas que hemos ido exponiendo a lo largo del trabajo. Realizaremos un repaso sobre ellas indicando la sección donde se encuentran detalladas.

- La API `Commander` facilita la programación de los agentes y la comunicación por red con el servidor. También permite dar instrucciones al avatar del agente que se encuentra en el simulador, como se muestra en la sección 3.3.1.
- La clase `EntityShell` de los agentes actúa como una plantilla donde el usuario de SASAMAS puede programar el comportamiento de los agentes. Esta clase crea al mismo tiempo una capa de abstracción sobre las clases SPADE, que requieren un conocimiento avanzado de programación. Todo esto se encuentra detallado en la sección 3.3.1 y ahorra tiempo y esfuerzo al usuario de SASAMAS, como se muestra en la sección 6.8.2.
- El menú inicial del simulador permite adaptar la resolución del simulador según las resoluciones que permita el monitor de la máquina en el que se ejecuta. El menú inicial también cuenta con una opción para configurar los parámetros de red del simulador. En la sección 3.3.2 se encuentran los detalles.
- Definir una simulación es un proceso muy sencillo y rápido que involucra editar dos archivos de texto, si no necesitamos rotar elementos ni cambiar las condiciones ambientales del entorno, o tres archivos de texto en caso contrario. Esto se encuentra detallado en la sección 3.3.2.
- El marco de trabajo desarrollado permite agregar nuevos modelos de elementos y agentes sin necesidad de programar nada, como se muestra en la sección 3.3.2.
- Diseño propio de un protocolo de comunicaciones basado en JSON para el envío de comandos y de imágenes, detallado en la sección 3.3.3.
- El simulador permite cargar las texturas de los elementos a partir de una ruta a un directorio con imágenes en tiempo de ejecución, así como asignar estas texturas de manear aleatoria para añadir variabilidad a las simulaciones. Este proceso está optimizado para minimizar el tiempo de carga de la escena, como se muestra en la sección 4.3.4.
- El sistema desarrollado permite validar los algoritmos de sistemas multi-agente en un paso previo a lo que sería el despliegue en el mundo real, pudiendo detectar anomalías o fallos que habían pasado desapercibidas. Esta detección temprana puede ahorrar el coste de recursos humanos y materiales ocasionados por la puesta en marcha del experimento en el mundo real. Esto lo hemos podido ver en la sección 6.5.

7.2 Trabajo futuro

Las líneas de desarrollo futuro que surgen a partir de este proyecto están principalmente orientadas a la mejora del simulador, pero también existen mejoras sobre los agentes. Estas líneas son:

- Desarrollar **un programa dedicado a la edición de mapas**, que trabaje por encima de los archivos de configuración. Esta herramienta adicional debe permitir editar los mapas con una interfaz gráfica, que posteriormente traduzca el mapa editado a los archivos de configuración de mapas que hemos visto en la sección 3.3.2.
- Crear un **sistema de guardado** de simulaciones, para después poder volver a visionarlas.
- Crear un menú integrado con distintas configuraciones preestablecidas de la cámara libre. Por ejemplo, puede haber una combinación de teclas o un botón que permita mover la vista libre de la escena al seguimiento de un agente.
- Integrar un menú con **ayudas gráficas**, como poder mostrar un rastro en los agentes con un color configurable para ver su camino y un mapa de calor de las zonas más transitadas.
- Agregar un comando que permita **cambiar la resolución de las cámaras** de los agentes. De este modo, se evita tener que modificar las cámaras en el editor.
- Agregar **nuevos tipos de sensores** a los agentes, como el sensor de distancia.
- Establecer el **comportamiento de los agentes** en el archivo de configuración, a través de una referencia al programa que contiene el código.

El código fuente completo de SASAMAS se encuentra en el siguiente enlace de la plataforma github: **https://github.com/FranEnguix/sasamas**

Bibliografía

[1] Michael Wooldridge and Nicholas R Jennings. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(2):115–152, 1995.

[2] Michael Wooldridge. *An introduction to multiagent systems*. John wiley & sons, 2009.

[3] Michael Luck and Ruth Aylett. Applying artificial intelligence to virtual reality: Intelligent virtual environments. *Applied artificial intelligence*, 14(1):3–32, 2000.

[4] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

[5] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[6] Uri Wilensky. Netlogo (and netlogo user manual). *Center for connected learning and computer-based modeling, Northwestern University. http://ccl. northwestern. edu/netlogo*, 1999.

[7] JA Rincon, Emilia Garcia, Vicente Julian, and Carlos Carrascosa. The jacalive framework for mas in ive: A case study in evolving modular robotics. *Neurocomputing*, 275:608–617, 2018.

[8] A Barella, A Ricci, O Boissier, and C Carrascosa. Mam5: multi-agent model for intelligent virtual environments. In *10th european workshop on multi-agent systems (EUMAS 2012)*, pages 16–30, 2012.

[9] Sean Luke, Gabriel Catalin Balan, Liviu Panait, Claudio Cioffi-Revilla, and Sean Paus. Mason: A java multi-agent simulation library. In *Proceedings of Agent 2003 Conference on Challenges in Social Simulation*, volume 9, 2003.

[10] YiJi Zhao, Farnoosh Fatemi Pour, Shadan Golestan, and Eleni Stroulia. Bim sim/3d: Multi-agent human activity simulation in indoor spaces. In *2019 IEEE/ACM 5th International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*, pages 18–24, 2019.

[11] Unity automotive uses tech from gaming to aid automakers | digital trends. https://www.digitaltrends.com/cars/unity-automotive-virtual-reality-and-hmi/, 2018. Consultado en [10 Junio 2022].

[12] Soluciones | unity. https://unity.com/es/solutions. Consultado en [11 Junio 2022].

[13] Marcus Toftedahl and Henrik Engström. A taxonomy of game engines and the tools that drive the industry. 08 2019.

[14] Javier Palanca, Andrés Terrasa, Vicente Julian, and Carlos Carrascosa. SPADE 3: Supporting the new generation of multi-agent systems. *IEEE Access*, 8:182537–182549, 2020.

[15] Michael Bratman. *Intention, Plans, and Practical Reason*. Cambridge: Cambridge, MA: Harvard University Press, 1987.

[16] Hafiz Tayyab Rauf, Basharat Ali Saleem, M Ikram Ullah Lali, Muhammad Attique Khan, Muhammad Sharif, and Syed Ahmad Chan Bukhari. A citrus fruits and leaves dataset for detection and classification of citrus diseases through machine learning. *Data in brief*, 26:104340, 2019.

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

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

Objetivos de Desarrollo Sostenible	**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 vinculados con la herramienta de trabajo desarrollada son el ODS 9 (industria, innovación e infraestructuras) y el ODS 17 (alianzas para lograr objetivos), y en menor grado el ODS 4 (educación de calidad).

El ODS 9 está relacionado con SASAMAS porque precisamente el marco de trabajo desarrollado es innovador, ya que actualmente no existe una herramienta que conste de un sistema distribuido, compuesto por un simulador y agentes inteligentes SPADE que habitan el entorno tridimensional simulado, y que crear o modificar dicho entorno sea una tarea muy sencilla y rápida. Además, tiene una aplicación directa sobre la investigación científica de sistemas multi-agente, ya que se utilizará en una línea de investigación activa del VRAIN (Valencian Research Institute for Artificial Intelligence), ahorrando tiempo y esfuerzo a los investigadores.

El ODS 17 está relacionado con el marco de trabajo porque gracias a la capacidad de desacoplar los agentes del simulador y de poder distribuir los agentes en diferentes máquinas y redes, SASAMAS permite la colaboración de equipos de investigación aunque estos estén en distintas partes del mundo. Es por este motivo que es una herramienta idónea para que distintos grupos de investigación puedan colaborar en el mismo proyecto.

El ODS 4 está relacionado en menor grado que los dos ODS anteriores porque puede ser usado por el equipo docente para formar al alumnado que curse la asignatura de agentes inteligentes, y también puede ser usada por el alumnado en los laboratorios gracias a su fácil instalación y la sencillez de programación que proporciona el marco.

Los ODS 12 y 13 están ligeramente relacionados porque gracias al uso de la herramienta, se puede detectar que el experimento no está listo para su despliegue en el mundo real, por lo que se ahorrarían los costes de transporte, construcción y puesta en marcha del experimento.