# Motor de simulación del mundo de la Facultad de Informática

## Simulation engine of the world of the Faculty of Computer Science


AUTOR:

LLUÍS CAÑELLAS SALESA

DIRECTORES:

PABLO GERVÁS GÓMEZ-NAVARRO

GONZALO MÉNDEZ POZO

GRADO EN DESARROLLO DE VIDEOJUEGOS

TRABAJO DE FIN DE GRADO

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

# Resumen

En este proyecto se desarrolla una simulación donde se representa el comportamiento humano habitual en una situación social y espacio físico determinados. En concreto, la aplicación maneja un conjunto de personajes en la Facultad de Informática de la Universidad Complutense que seguirán los comportamientos de alumnos, profesores y otros trabajadores en su día a día en el edificio de la facultad. Esta simulación se animará a tiempo real en un motor de renderizado 3D con un modelo ya existente del edificio.

Este proyecto, a parte de servir para observar las interacciones y desplazamientos habituales en la facultad, también permitirá simular situaciones excepcionales, como el caso de un incendio, para probar protocolos de evacuación, o un brote de una enfermedad contagiosa como podría ser el covid-19.

Este programa puede llegar a ser una herramienta de gran utilidad ya que permite probar horarios de clases y otras actividades, ya sean nuevos o ya existentes, y encontrar problemas logísticos o de aforo en su fase de planificación. Además, también permite encontrar posibles problemas relacionados con la evacuación en el caso de incendios u otros eventos y encontrar las medidas eficaces para evitar contagios.  Partiendo de los resultados obtenidos, se podrán estudiar soluciones a esos problemas y otras mejoras de manera preventiva.

## Palabras clave:

- Simulación social
- Comportamiento humano
- Agente inteligente
- Unity
- Modelo 3D
- Motor 3D
- Componente

# Abstract

In this project, we will develop a simulation where common human behavior is represented in a given social and physical situation. More specifically, the application manages a set of characters in the Complutense University's Faculty of Computer Science who will follow the behaviors of students, teachers and other workers in their day-to-day life in the faculty. This simulation will be animated in real time in a 3D rendering engine using an existing model of the building.

This project, apart from allowing the observation of the usual interactions and behaviors in the faculty, will also allow simulating exceptional situations, such as the case of a fire, to test evacuation protocols, or an outbreak of a contagious disease such as covid- 19.

This program can be a very useful tool since it allows researchers to test class schedules and other activities, whether new or existing, and find logistical or capacity problems during the planning phase. In addition, it also allows you to find possible problems related to evacuation protocols in a fire situation and helps finding effective measures to prevent contagion in an outbreak situation. Based on the results obtained, solutions to these problems and other preventive improvements can be studied.



## Keywords :

- Social simulation
- Human behavior
- Smart agent
- Unity
- 3D Model
- 3D engine
- Component

# Índice

# Índice de figuras

# Capítulo 1 - Introducción

Desde hace ya unos años la inteligencia artificial ha empezado a introducirse progresivamente en muchos campos que antes eran impensables. Su capacidad de simular el comportamiento humano a muchos niveles ha hecho abrir los ojos a muchos expertos sobre su utilidad práctica. Además, el auge de los sistemas de aprendizaje automático y redes neuronales han permitido simular capacidades humanas que parecían imposibles de implementar a través de algoritmos clásicos.

Este proyecto gira alrededor de los nuevos usos de la inteligencia artificial y el renderizado 3D en un entorno nuevo como son estudios de carácter social. Hace unos veinte años la tecnología 3D estaba muy limitada sobretodo por la potencia del hardware por lo que no se usaba mucho para realizar estudios científicos u otras aplicaciones serias. Actualmente la situación es muy diferentes gracias a los grandes avances en la tecnología de las tarjetas gráficas que han permitido que conseguir computadores con gran capacidad de renderizado gráfico sea más fácil que nunca y, además, han aparecido varias herramientas de software que han hecho muy accesible el desarrollo de aplicaciones con renderizado 3D.

Estas tecnologías se han convertido en herramientos muy útiles en el campo de la sociología ya que las han podido aprovechar para la representación de simulaciones sociales consiguiendo, de esta forma, una simulaciones más realistas en todos los aspectos. En estas simulaciones, la inteligencia artificial se usa para simular el comportamiento humano en una situación social y localización concreta, característica de gran interés para este proyecto. Estos sistemas son muy prácticos si se implementan correctamente ya que permiten predecir y encontrar errores logísticos o de protocolo en varias situaciones. Algunos ejemplos donde se pueden aplicar sería el caso de una estación de metro donde se podrían identificar lugares en los que se producen mayores acumulaciones de personas en hora punta o donde hay más riesgo en el caso de una evacuación de la estación. También pueden servir a la hora de construir intersecciones en carreteras para comprobar qué solución es más eficaz y segura para los conductores. Como se puede ver, hay una gran variedad de situaciones donde estas simulaciones son de gran ayuda aunque la complejidad e implementación pueden variar mucho según el caso, eso sí, en la mayoría de casos la visualización en 3D añade toda una nueva capa de comprensión que no sería posible de otra forma.

Este proyecto se enfoca en el desarrollo de una aplicación que soporte una simulación social situada en la facultad de la que surge este trabajo: la Facultad de Informática de la Universidad Complutense (de ahora en adelante FDI). Esta

simulación tiene un gran interés aparte del puramente académico ya que se podría usar para encontrar problemas logísticos en los horarios de clases o en situaciones excepcionales como podría ser un simulacro de incendio o brote de una enfermedad. Gracias a este proyecto, se podrían realizar mejoras de estos aspectos en la FDI.

La herramienta escogida para el desarrollo del proyecto es Unity, un motor de videojuegos 3D, ya que anteriormente a este trabajo se desarrolló en la FDI un modelado 3D de la facultad en este motor que servirá como base de la simulación. Además este motor es de uso gratuito en aplicaciones en las que no se alcanza un beneficio mínimo en ventas. Como IDE se ha usado Visual Studio principalmente para editar código y manejar la solución.

El comportamiento de los personajes será el eje central de la simulación ya que a través de este comportamiento, que se verá por pantalla, se podrán identificar los problemas que pueden haber en la FDI. Para implementar los personajes se ha seguido un enfoque muy extendido en el campo de la inteligencia artificial conocido como el de agente inteligente. Este concepto se desarrollará en los capítulos posteriores.

Durante el desarrollo de este proyecto también se dará importancia a otros aspectos de la aplicación que benefician en gran medida la utilidad de la simulación. En vez de tener unos horarios y personajes preestablecidos en el código estos se podrán configurar según los intereses del usuario. Además, una simulación de este tipo no se siente completa si el usuario no puede interactuar con ella durante la ejecución, por este motivo, se han implementado varias formas de interacción.

Todo el código desarrollado en este proyecto junto con el proyecto de Unity se encuentra en el siguiente repositorio de GitHub:
https://github.com/LluisCS/SimulacionFDI

## 1.1. Motivaciones

El motivo principal de este proyecto es poder detectar posibles problemas de aspecto logístico en los horarios de clase y en el seguimiento de protocolos de seguridad en la facultad. Por ejemplo, se pueden simular situaciones como la evacuación en caso de incendio o el contagio de una enfermedad para intentar encontrar posibles mejoras durante su ejecución. Gracias a esta aplicación se podrán observar todo tipo de problemas que pueden ser muy complicados de detectar de otra forma.

Otra motivación importante es estudiar el potencial que aportan los modelos 3D en las simulaciones sociales y si se pueden aprovechar en este campo potentes herramientas como los motores de videojuegos.

Tener una aplicación que permita analizar el comportamiento e interacciones habituales en la propia facultad es también un motivo importante ya que aporta un novedoso enfoque al funcionamiento de la facultad. Además, tiene un gran potencial para extenderse en nuevos escenarios y todo tipo de situaciones donde esta visión puede ser de gran ayuda.

## 1.2. Objetivos

El primer objetivo es desarrollar una herramienta de software para probar y simular el comportamiento humano en una facultad universitaria, en concreto: la Facultad de Informática de la Universidad Complutense de Madrid (FDI). Este objetivo incluye tanto la simulación del comportamiento rutinario en la FDI como las situaciones singulares como el caso de un simulacro de incendio. El comportamiento de los personajes se implementará a través de agentes inteligentes por lo que la simulación será lo que se conoce como Simulación Social Basada en Agentes (con siglas en inglés ABSS).

Para extender las aplicaciones prácticas de este programa, es muy útil que se puedan introducir diferentes clases, horarios y personajes con facilidad. También será importante implementar una interfaz de usuario para el control de la simulación y de la visión durante la ejecución.

De manera detallada, los objetivos que se persiguen son:

- Crear una simulación social renderizada en 3D de la FDI
- Simular un comportamiento realista de los personajes presentes en la simulación
- Permitir simular eventos especiales en esta simulación
- La simulación tiene que ser fácilmente configurable
- Desarrollar una interfaz para controlar la simulación en ejecución

# Chapter 1 - Introduction

In the last years, artificial intelligence has been progressively entering many fields that were previously unthinkable. Its ability to simulate human behavior on many levels has opened the eyes of a lot of experts regarding its practical usefulness. Furthermore, the rise of machine learning and neural networks have made it possible to simulate human capabilities that seemed impossible to implement through classical algorithms.

This project revolves around the new uses of artificial intelligence and 3D rendering in a new environment such as social science. About twenty years ago, 3D technology was very limited, especially due to the power of the hardware, for that reason it wasn't used much for scientific studies or other serious applications. The current situation is very different thanks to the great advances in graphics cards technology making it easier than ever to get computers with a great graphic rendering capability and, in addition, developing applications with 3D rendering has become very accessible thanks to the appearance of new software tools.

These technologies have become very useful tools in the field of sociology because the representation of social simulations has improved considerably thanks to them, thus achieving more realistic simulations in every aspect. In these simulations, artificial intelligence is used to simulate human behavior in a specific social situation and physical location, this feature is of great interest for this project. These systems are very handy if implemented correctly since they allow predicting and finding logistical or protocol errors in a lot of situations. For example, they can be used to identify which places there is more accumulation of people in a subway station during rush hour or which are the places with a higher risk for passengers when an evacuation occurs. They can also be used when building road intersections to check the best solution for the drivers both in safety and effectiveness. As you can see, there are a wide variety of situations where these simulations are very helpful, although the complexity and implementation can vary greatly depending on the specific case. In all this situations, the 3D visualization adds a whole new layer of understanding that wouldn't be possible otherwise.

This project focuses on the development of an application that supports a social simulation located in the faculty from which this work arises: the Facultad de Informática de la Universidad Complutense de Madrid (the Faculty of Computer Science of the Complutense University of Madrid; hereafter FDI). Apart from the purely academic one, this application has other interesting uses since it could be used to find logistical problems in class schedules or in exceptional situations such

as a fire or infectious disease outbreak. Thanks to this project, improvements in these aspects could be made in the FDI.

The chosen main tool for the development of the project is Unity, a 3D video game engine, since prior to this work the3D model of the faculty that will serve as the basis of the simulation was developed at the FDI using this engine. Another important reason for using it is that this engine is free to use in applications that don't reach a minimum profit in sales. As for the IDE, Visual Studio has been used mainly to edit code and manage the program's solution.

The behavior of the characters will be the central point of the simulation since through this behavior, which will be seen on the screen, the problems that may be in the FDI can be identified. To implement the characters, a widely used approach has been followed in the field of artificial intelligence known as the intelligent agent. This concept will be developed in later chapters.

During the development of this project we've focused on other aspects of the application that greatly benefit the practical use of the simulation. Instead of having pre-established schedules and characters in the code, they can be configured by the user according to his objectives. Furthermore, this type of simulation seems incomplete if the user cannot interact with it at runtime, for this reason, various forms of user interaction have been implemented.

All the code developed in this project along with the Unity project can be found in the following GitHub repository:
https://github.com/LluisCS/SimulacionFDI

## 1.1. Motivations

The main motivation for this project is to be able to detect possible logistical problems during class hours and during the procedure of security protocols at the FDI. For example, situations such as the evacuation in a fire event or the spread of disease can be simulated to find possible improvements during its execution. Thanks to this application you can see all kinds of problems that can be very difficult to detect with other methods.

Another important motivation is to study the potential of 3D models in social simulations and whether powerful tools such as video game engines can be used in this field.

Having an application that allows analyzing the behavior and habitual interactions in the faculty itself is also an important reason since it provides a novel approach to the

faculty operation. In addition, it has great potential to spread in new settings and all kinds of situations where this vision can be of great help.

## 1.2. Objectives

The first objective is to develop a software tool to test and simulate human behavior in a university faculty, specifically the FDI. This objective includes both the simulation of the usual behavior in the FDI and unique situations such as fire event. The behavior of the characters will be implemented through intelligent agents, so the simulation will be what is known as Agent-Based Social Simulation (ABSS).

The ability to introduce different classes, schedules and characters with ease will extend the practical uses of this program. It is also very useful to implement a user interface to control the simulation and the viewpoint at runtime.

In detail, the objectives pursued are:

- Create a 3D rendered social simulation of the FDI
- Simulate realistic behavior of the characters present in the simulation
- Allow the simulation of special events in this simulation
- The simulation must be easily configurable
- Develop an interface to control the simulation at runtime

# Capítulo 2 - Estado del arte

Com ya se ha introducido anteriormente, el desarrollo de aplicaciones de simulaciones sociales con renderizado 3D ha pasado a ser mucho más accesible en estos últimos años. Por esta razón, han aumentado el número de productos comerciales y estudios académicos relacionados con este tema por lo que se han realizado grandes avances tanto en el tamaño como en el realismo de dichas simulaciones.

En este apartado hablaremos sobre algunos estudios, trabajos y aplicaciones que han fundamentado este trabajo de varias formas. Gracias a estos, se ha podido profundizar más en todos los aspectos de la simulación. Nos centraremos en los campos más centrales para el funcionamiento de este programa: los agentes inteligentes, los sistemas multiagente, las simulaciones sociales y, finalmente, las simulaciones sociales en 3D.

## 2.1. Agentes inteligentes

Empezaremos con la unidad básica sobre la que se construirá toda la simulación: los agentes inteligentes. ¿Qué es un agente? Es difícil encontrar una definición única de lo que es un agente ya que distintos equipos y autores proporcionan definiciones con ciertas variaciones (Franklin y Graesser, 1996) dando más o menos importancia a los elementos que los forman pero, de todas las diferentes definiciones, se pueden extraer unas características principales que debe tener todo agente además de otras que suelen estar presentes pero no son esenciales para ser clasificado así. Estas variaciones en la definición suelen estar determinadas por el uso concreto de cada autor.

La definición que seguiremos en este proyecto la formaremos a partir de la visión de varios autores y es la siguiente: Un agente es un sistema autónomo que se encuentra y forma parte de un entorno que puede percibir y actuar sobre él siguiendo sus objetivos. Es bastante común que los agentes tengan capacidad para aprender con el paso del tiempo y guardar conocimiento. Para que sea considerado inteligente, un agente tiene que realizar algún tipo de razonamiento interno con la información adquirida del entorno y/o su conocimiento anterior (Maes, 1995; Hayes-Roth, 1995).

Otro enfoque para entender lo que es un agente mucho más práctico y dirigido directamente al programador es el esquema del funcionamiento básico de un agente inteligente propuesto por Michael Wooldridge (1999). El esquema es el siguiente:

bucle while true
2. observar el entorno
3. actualizar el modelo interno
4. razonar el objetivo actual
5. razonar un plan para conseguir su objetivo
6. ejecutar la planificación

fin del bucle

Ya es una realidad el gran impacto y alcance de los sistemas basados en agentes inteligentes en el mundo de la informática. La existencia y la relevancia que han conseguido aplicaciones de software como JADE[1] y Cougaar[2] son claras pruebas de ello, ya que son dos herramientas para el desarrollo en java que permiten producir todo tipo de aplicaciones usando una arquitectura basada en agentes.

Tratando el tema de los agentes inteligentes en la actualidad es difícil no mencionar el impacto de la FIPA (Foundation for Intelligent Physical Agents), una organización cuyos principales objetivos han sido promover el uso de agentes y sistemas multiagente y han desarrollado y actualizado los estándares *de facto* de su uso en el mundo de la informática. Su labor destaca porque han conseguido crear un estándar para la comunicación entre distintos tipos de agentes y para el funcionamiento de sistemas basados en agentes.

## 2.2. Sistemas Multiagente

En este proyecto se usará el concepto de agente inteligente para implementar un sistema multiagente. Un sistema multiagente es un tipo de sistema formado por múltiples agente independientes que interactúan en un mismo entorno (Posland, 2007). El uso de este nuevo enfoque en la inteligencia artificial y otros campos de la informática ha permitido solucionar problemas muy difíciles de solucionar con métodos clásicos o sistemas con agentes individuales.

Este tipo de sistemas introducen un cambio relacionado con la perspectiva desde la cual se estudia el entorno que resulta de mucho interés en este proyecto. Gracias a este cambio, se consiguen dos perspectivas distintas difíciles de obtener de otra forma y son la perspectiva individual de cada agente y la perspectiva global de todo el sistema. En el caso de este trabajo, poder estudiar y contrastar estos dos puntos de vista del sistema permite generar un modelo de datos más profundo y, por lo tanto, conseguir unos mejores resultados.

Los sistemas multiagente se han extendido su uso progresivamente a cada vez más campos y áreas del conocimiento. El número de aplicaciones donde este enfoque

ha empezado a aplicarse no ha hecho más que multiplicarse demostrando una gran efectividad y es cada vez es más común encontrarlo en sistemas informáticos vitales tanto para empresas como para ciudades y países. Algunos de usos más prácticos que se ha estudiado son: en sistemas de control en el entorno industrial (Tapia, Corchado, Paz, Rodríguez y Bajor, 2008), en el manejo de las redes de señales de tráfico (Arel, Liu, Urbanik y Kohls, 2010) y en los sistemas de control de recursos automáticos en hogares (Conte, 2009). En campos más teóricos y académicos han servido para profundizar de nuevas formas en la inteligencia artificial (Davidsson, 2002) y, donde han tenido uno de sus mayores impactos, es en la investigación en ciencias sociales ya que han permitido usar metodologías antes inaccesibles para todo tipo de estudios económicos y sociológicos (Gilbert, 2007; Macy y Willer, 2002).

## 2.3. Simulaciones sociales

Uno de los ámbitos donde han tenido más repercusión los sistemas multiagente es en el de la simulación. Las ventajas que aporta en este contexto, cómo estar formado por unidades autónomas que se relacionan con el entorno a la vez que interactúan con el resto de unidades y poder estudiar el problema a la vez desde distintas perspectivas, han convertido a las simulación en una de las mayores aplicaciones de los sistemas multiagente. A raíz de esto, se han creado nuevos términos como Simulación Basada en Agentes (ABS, del inglés Agent Based Simulation) o Simulación Social Basada en Agentes (ABSS, del inglés Agent Based Social Simulation).

En este proyecto se desarrollará un ABSS por lo que entraremos más en detalle en este concepto. Un ABSS y un sistema multiagente tienen una colisión importante en su significado pero no tienen porqué siempre coincidir ya que ABSS indica el uso de agentes para la simulación social pero un sistema multiagente no tiene porque ser una simulación social. Como su nombre indica, en una simulación social se intentan reproducir determinados comportamientos e interacciones de una sociedad usando métodos computacionales con el fin de generar un conjunto de datos. Dependiendo del tipo de simulación y su funcionalidad estos datos pueden servir para distintos objetivos pero, en general, servirán para ser analizados y poder procesar unos resultados (Davidsson, 2002).

Las simulaciones sociales basadas en agentes son una de las nuevas maneras de estudiar problemas y conceptos en el campo de las ciencias sociales. Usando esta metodología, se aprovechan las capacidades y ventajas que aporta la inteligencia artificial y la computación en áreas del conocimiento relativamente distantes como sociología, psicología y economía. Ejemplos más concretos del uso actual de estas simulaciones son: el estudio sobre los efectos en la bolsa de las inversiones y otras

operaciones (Hoffmann, Jager y Von Eije, 2007), sus posibles usos en el campo del marketing (Jager, 2007), el estudio de la propagación de enfermedades (Gong y Xiao, 2007) y el análisis de los efectos de las decisiones tomadas por los grandes empresarios en la demanda de productos básicos (Moss, Downing y Rouchier, 2000).

## 2.4. Simulaciones en 3D

La simulación social que se desarrolla en este proyecto se representará usando un modelo tridimensional. Para conseguir este modelo se usará Unity, un motor 3D enfocado al desarrollo de videojuegos.

La capacidad de representar elementos del mundo real usando modelos renderizados tridimensionalmente por computación ha demostrado ser sumamente útil en muchas ramas del conocimiento. Crear y, posteriormente, estudiar este tipo de representaciones dentro de un ordenador han permitido importantes avances en una gran variedad campos como la medicina (Bajaj et al., 2014), microbiología (Creber et al., 2010), arquitectura (Clayton, Warden y Parker, 2002), física (Scheucher, Bayley, Guetl y Harward, 2009) o diseño (Schkolne, Pruett y Schröder, 2001). El rápido auge de esta tecnología se debe en gran parte a los avances en el renderizado gráfico que se ha producido durante las últimas dos décadas tanto en el software como en el hardware necesario.

En un marco más centrado en este trabajo, las simulaciones sociales se han beneficiado sin duda alguna de la tecnología 3D y, gracias a la creación de entornos tridimensionales, se ha impulsado el estudio de conceptos y problemas sociales. Usando estas técnicas se obtienen modelos más complejos y realistas que, a su vez, permiten un análisis más profundo de las interacciones de la simulación. La simulación de protocolos de evacuación de incendio (Cruz, Durbey y Sanz, 2011), de integración laboral de personas discapacitadas (Barriuso, De La Prieta y Li, 2015) o el estudio de comportamientos sociales (Pan, Han, Dauber y Law, 2006) son algunos de los usos cercanos de la tecnología en cuestión.

Creo que también es importante resaltar la relevancia de los videojuegos en la situación actual de las simulaciones y otros tipos de aplicaciones 3D. El gran peso económico de la industria del videojuego ha impulsado el avance de todas las tecnologías directamente relacionadas. Son uno de los principales motivos del gran progreso en las tarjetas gráficas y, en caso del software, muchas empresas de videojuegos han desarrollado motores gráficos cosa que han hecho más accesible el desarrollo de aplicaciones 3D. Hay gran variedad de estos motores pero algunos de los más destacados son Unity[3], UnrealEngine[4] y Frostbite[5].

Cabe mencionar que los videojuegos también tienen una gran importancia en el desarrollo de simulaciones sociales ya que hay categorías enteras que se centran en el manejo de una simulación de este tipo por parte del usuario. Una de las razones de su impacto es el número y variedad de videojuegos que se han desarrollado de esta categoría ya que se han hecho juegos con todo tipo de enfoque, escala y ambientación. Caesar 3[6] un juego sobre el manejo de una ciudad en la época romana producido por Sierra Entertainment; Los Sims[7], una de las sagas más conocidas desarrollada por Electronic Entertainment centrada en la simulación de la vida de unos personajes y en seguir y cumplir sus deseos y Two-Point Hospital[8], videojuego de Two Point Studios en el que se maneja un hospital con la simulación de todas las interacciones sociales que ocurren en este, son solo tres de los muchos videojuegos que se pueden destacar en este ámbito.

## Notas

[1]Página web de JADE: https://jade.tilab.com/
[2]Página web de Cougaar: http://www.cougaar.world/
[3]Página web de la FIPA http://www.fipa.org/
[4]Página web de Unity: https://unity.com/
[5]Página web de UnrealEngine: https://www.unrealengine.com/
[6]Página web de Frostbite: https://www.ea.com/frostbite
[7]Caesar 3 no tiene una página web oficial pero en esta página se puede ver un pequeño tráiler: https://www.youtube.com/watch?v=eq_wTyoLuOk
[8]Página web de Los Sims: https://www.ea.com/es-es/games/the-sims
[8]Página web de Two-Point Hospital: https://www.twopointhospital.com/

# Capítulo 3 - Herramientas usadas: Unity

En este apartado se introducirá la herramienta principal usada para el desarrollo del proyecto, sus características principales y las ventajas que aporta en este tipo de aplicaciones.

## 3.1. ¿Qué es Unity?

Unity es uno de los motores de videojuegos más populares actualmente desarrollado por *Unity Technologies*. Se trata de un motor multiplataforma que permite crear juegos tridimensionales, bidimensionales, en realidad virtual y realidad aumentada. También se ha empezado a usar en otros ámbitos como el cine, construcción o simulación.
El lenguaje de programación usado en Unity es C#, que se trata de un lenguaje orientado a objetos, esto es vital para entender la arquitectura y el desarrollo en Unity.

## 3.2. Desarrollo en Unity

Unity usa una arquitectura orientada a componentes por lo que, para desarrollar juegos u otras aplicaciones, se manejan principalmente dos tipos de objetos: las **entidades** ( en Unity *Game Objects*) y los **componentes**.

Las entidades son objetos prácticamente vacíos con la capacidad de contener componentes. Estas entidades siempre tienen un componente llamado *transform* que da información sobre su posición, rotación y escala además de métodos para la gestión de sus componentes como añadir, eliminar o buscar. Por otra parte, los componentes se encargan de toda la lógica concreta del programa pudiendo realizar todo tipo de funciones: renderizado, input, física, lógica, etc.

Unity tiene disponible una gran cantidad de componentes de todo tipo para la mayoría de funciones básicas en todo juego, por ejemplo, renderizar un cubo o esfera, tener todo tipo de mallas, interactuar con el motor de física para simular gravedad o colisiones, reproducir sonido y muchos más. Pero la característica más interesante es que el desarrollador puede implementar componentes adicionales para poder controlar las entidades según sus intereses.

El ciclo de generación de contenido general consiste en crear entidades en las que se añaden componentes que determinarán su funcionalidad. Para programar correctamente los componentes hay que saber su funcionamiento y el flujo de ejecución usado por Unity, pero primero se deben tener conocimientos sobre lo que es bucle principal de un juego. En general, en los videojuegos siempre se ejecuta un bucle donde se tienen que realizar todas las funcionalidades distintas del videojuego. La mayoría de estas tareas se agrupan en tres principales grupos que se llaman siempre en el mismo orden. Primero se ejecuta todo el código relacionado con la obtención de input, tanto por parte del usuario como recibir información a través de la red en juegos multijugador. A continuación, se ejecuta toda la lógica propia del juego, como la inteligencia artificial o movimiento de personajes,en esta parte se suele procesar primero la simulación física y las colisiones en caso de estar presentes. Finalmente, se realiza el renderizado gráfico del videojuego y otros tipos de retroalimentación para el jugador, por ejemplo, la vibración de un mando. El propio motor de Unity ya se encarga de gestionar el bucle de juego además del ciclo vital de la aplicación, las entidades y los componentes.

Todos los componentes en Unity heredan de una clase padre Component que tiene algunos métodos ya implementados para controlar su ciclo de vida pero tiene muchos otros que se encuentran vacíos preparados para que el programador los pueda rellenar libremente en cada componente distinto.

Usando estos métodos se puede elegir en qué momento del bucle de juego y cuantas veces se ejecuta el código dependiendo de las necesidades de cada componente. Los dos métodos más básicos son el *Update* y *Start*: el primero es llamado por Unity en cada vuelta del bucle de juego y el segundo solo se llama una vez al crearse su entidad. Igual que estos, hay muchos otros métodos que se llaman en diferentes momentos del ciclo de vida del componente, por ejemplo, cuando la entidad colisiona con otra, cuando se activa o cuando se destruye.

Siguiendo este esquema de desarrollo, la mayoría del código de este proyecto se encuentra en los nuevos componentes implementados, además de algunas clases contenedoras de datos.

## 3.3. Editor

Una de las características más útiles de Unity es su editor. Gracias a su interfaz visual que permite una gestión eficaz y más accesible de las escenas del proyecto. Una escena en Unity es un estado del juego independiente que contiene un conjunto de entidades repartidas en el espacio distanciadas, lógica y físicamente, de cualquier otra escena.

En el editor se puede ver y modificar en tiempo real el estado de la escena actual; permite crear entidades nuevas y interactuar con las ya existentes pudiendo modificar su posición, rotación y escala. También permite toda la gestión relacionada con los componentes: se pueden añadir, eliminar y modificar su configuración a través de su interfaz.

Para acceder cómodamente a las entidades hay una pestaña con una lista completa de las entidades presentes en la escena, aparte, también se puede navegar por las carpetas donde se encuentran todos los ficheros del proyecto. Asimismo, hay más ventanas desplegables disponibles para la configuración general del proyecto y algunos componentes específicos.

Otra característica especialmente potente del editor en cuestión es la de poder ejecutar la escena activa en cualquier momento para probarla, aparte de poder pausarla y ejecutarla paso a paso, eso siempre que no haya ningún error de compilación en el código. Esto permite probar rápidamente cualquier cambio en la escena y poder reaccionar rápidamente si hay algún error.

## 3.4. Carga de datos

Los desarrolladores de Unity han dado mucha importancia a la carga de datos porque su eficacia es muy importante en los videojuegos de gran tamaño. La mayoría necesitan cargar grandes cantidades de modelos, texturas y sonidos por lo que siempre se intenta optimizar al máximo este proceso.

En este proyecto también es una parte vital ya que, a parte de los modelos y texturas de la facultad, hay que cargar toda la configuración de la simulación. Parte de esta configuración se puede realizar a través de los componentes que se encuentran en las entidades de la escena usando el editor, pero toda la información sobre asignaturas, alumnos y otros personajes de la simulación no se puede configurar de esta forma.

A todos los archivos de recursos en el campo del desarrollo de videojuegos y aplicaciones similares se les suele llamar *assets*, que es también como se llaman en Unity. Todos los archivos de este tipo se tienen que colocar en una carpeta llamada **Assets** en el directorio del proyecto para que Unity tenga acceso a ellos. Unity cargará automáticamente todos los assets referenciados en las entidades de cada escena pero, para cargar los datos sobre los horarios, personajes y asignaturas, hay otra manera más eficaz ya que esta información tiene la peculiaridad que puede variar la cantidad de alumnos o asignaturas que hay que cargar en la simulación y tener que modificar la escena cada vez que se quiere hacer un cambio de este tipo sería un gran estorbo y poco escalable. Para solucionar esta clase de problemas,

en Unity hay una clase llamada **ScriptableObject**, esta clase está preparada para hacer clases hijas personalizadas con campos de datos; como un número, palabra o imagen; que se quieren poder configurar desde el editor pero se quieren cargar como assets.

Luego de crear una clase hija de *SciptableObject* se podrán crear instancias de la clase desde el propio editor que se guardarán como archivos en el proyecto y se podrán modificar sus parámetros usando el editor en cualquier momento. Se puede acceder en cualquier momento desde el código a estos recursos a través de la clase **Resources**, que además, permite cargar a la vez todos los recursos con un mismo tipo, una capacidad muy útil para el proyecto en cuestión.

## 3.5. Navegación por mallas

El desplazamiento de los actores por el escenario de la simulación es también una fracción importante en este proyecto ya que hay que controlar los movimientos de muchos personajes simultáneamente y hacer navegable un edificio complejo. En el campo de la inteligencia artificial el tema de la búsqueda de ruta, mejor conocido como pathfinding, siempre ha sido un tema recurrente y aún más en el ámbito de los videojuegos donde se ha trabajado mucho en su desarrollo.

Unity, al ser un motor de videojuegos, tiene ya implementado un sistema para realizar *pathfinding*. Esta herramienta funciona a través de mallas de navegación a las que llama NavMesh. Para crear una NavMesh primero hay que seleccionar entidades con mallas de colisiones ya existentes y, a continuación, usar la ventana de la herramienta para configurar y crear la malla deseada.

Para poder navegar por esta malla hay que usar un componente ya incluido en Unity llamado NavMeshAgent. Aparte de poder configurar muchas opciones como velocidad y tamaño tiene métodos para controlar su comportamiento que se pueden llamar desde otros componentes. El método más importante es el SetDestination donde se le introducen unas coordenadas y el componente intentará acercar lo máximo posible su entidad a esa posición usando el camino más corto, siempre limitándose por la malla de navegación creada anteriormente.En el componente NavMeshAgent también se puede consultar algunas variables para saber su estado además de tener varios métodos más ya sea para reiniciar el destino actual o parar su movimiento completamente.

## 3.6. Interfaz

Otro de los objetivos planteados inicialmente para que el proyecto final sea satisfactorio era poder interactuar con la simulación en tiempo de ejecución de

manera cómoda y directa. Con este fin en mente, se puede usar una interfaz de usuario gráfica o GUI, esta forma de interactuar con el entorno virtual a través imágenes es muy común en los videojuegos, sobretodo en menús de juego y configuración.

El motor de Unity tiene varios componentes y entidades preestablecidas para facilitar en gran medida la creación de este tipo de interfaces. Permite crear un entidad Canvas donde ya se encuentran la mayoría de elementos configurados para desarrollar una interfaz. Luego de tener un Canvas ya se pueden crear otras entidades como botones, textos y imágenes que, simplemente usando el editor, se pueden configurar y colocar en la posición deseado de la pantalla. De esta forma se puede extraer información de la simulación eficazmente además de hacer sistemas de control usando botones.

La información expuesta en este apartado se trata solamente de una breve y general introducción a Unity y los sistemas que aporta más usados en el proyecto. Se puede acceder a una explicación más completa y extensa a través de la página web oficial de Unity y su página de documentación del motor. Para otras dudas más específicas se puede usar el foro online de Unity.

## Notas

Manuel de Unity https://docs.unity3d.com/Manual/index.html
Referencia para programar en Unity https://docs.unity3d.com/ScriptReference/

# Capítulo 4 - Diseño e Implementación

## 4.1. Estructura de una simulación social 3D

¿Cómo se crea una simulación social en un espacio 3D? Para responder a esta pregunta primero habrá que analizar las partes en las que se dividen estas simulaciones y así conseguir entender su funcionamiento interno y saber cómo se coordinan todos sus elementos.

Como ya se ha explicado anteriormente, en este proyecto se desarrolla una simulación social, en este tipo de simulaciones se presenta un conjunto de personajes situados en un espacio limitado que, a través de inteligencia artificial, intentan imitar el comportamiento humano en una situación social concreta.

Lo primero que necesitamos para dar vida a una simulación son los **personajes**. Cada personaje se tiene que encargar de simular su comportamiento social de forma autónoma. Como consecuencia, tendrá que guardar su estado y interpretar el mundo desde su propio punto de vista. Con esta finalidad se usará lo que se conoce en IA como agentes inteligentes. Dependiendo de cada caso concreto de simulación, puede haber una variación importante en el nivel de complejidad de cada personaje, además del de sus capacidades, pero siempre tienen que poder interactuar con su entorno y con otros personajes de alguna forma. Los personajes deben tener una representación visual en el escenario aunque sea muy simple.

El segundo elemento fundamental es el **espacio** ya que determina gran parte de las característica de la simulación. Los personajes tendrán que desplazarse y interactuar en un espacio concreto ya sea una habitación, edificio o población. El escenario estará formado por dos partes: la visual y la lógica. La parte visual sirve para que el usuario pueda percibir el espacio físico en el que se sitúa la simulación y la parte lógica servirá para que los personajes puedan interpretar ese escenario paro moverse e interactuar a través de este.

Otro componente esencial para una simulación es el **tiempo**. Es común no dar gran importancia a la gestión del tiempo pero es, sin duda, uno de los elementos cruciales. El tiempo hará que la simulación avance y cambie el estado continuamente, además, los personajes pueden tener horarios, como en este caso, o comportamientos afectados por el tiempo. Aunque la simulación sea independiente de la fecha actual, el tiempo transcurrido entre cada cambio o suceso en la simulación es muy importante para analizar los resultados.

Finalmente, el último elemento que debe tener toda simulación social es un **gestor** de la simulación. El gestor se tiene que encargar de coordinar el resto de elementos que forman la simulación aparte de controlar el inicio, final y estado general. Tendrá que controlar la carga de recursos y los cambios significativos en su situación. El tamaño y cantidad de responsabilidades pueden variar mucho en cada caso pero siempre tiene que haber algún tipo de gestor. También se puede usar para recopilar y extraer datos y estadísticas útiles para el usuario.

Aunque no tienen porque encontrarse en todas, las simulaciones también suelen tener otros elementos que mejoran su utilidad y flexibilidad. Los elementos descritos a continuación, están incluidos en este proyecto y ayudan a proveer una mejor experiencia global en el uso de la aplicación.

El primero de estos elementos es alguna herramienta para **configurar el contenido** de la simulación fácilmente. Esto da mucho potencia al simulador ya que posibilita simular variedad de situaciones con una misma aplicación. Poniendo como ejemplo este proyecto, se ha implementado una herramienta para configurar los personajes y está preparado para ser accesible para cualquier usuario con un mínimo conocimiento del funcionamiento del programa. Gracias a esta herramienta se pueden probar todo tipo de horarios distintos con diferentes cantidades ya sea de alumnos, profesores u otro personal.

El segundo y último de los elementos adicionales es la **interacción con el usuario**. La mayoría de simulaciones permiten a los usuarios interactuar en cierto nivel con la simulación durante su ejecución. En este caso se ha implementado una interfaz gráfica de usuario(GUI) que posibilita alterar su estado cómodamente, además, de permitir al usuario acceder a información sobre los personajes y el estado de la simulación actual. De esta forma actúa como canal de comunicación entre el usuario y la máquina.

## 4.2. Diseño

### 4.2.1. Visión general

Hasta ahora se han descrito los componentes que forman una simulación social genérica relacionándolos brevemente con este proyecto pero, a partir de este momento, ya entraremos en más detalle. ¿Cómo funciona la simulación de la FDI?

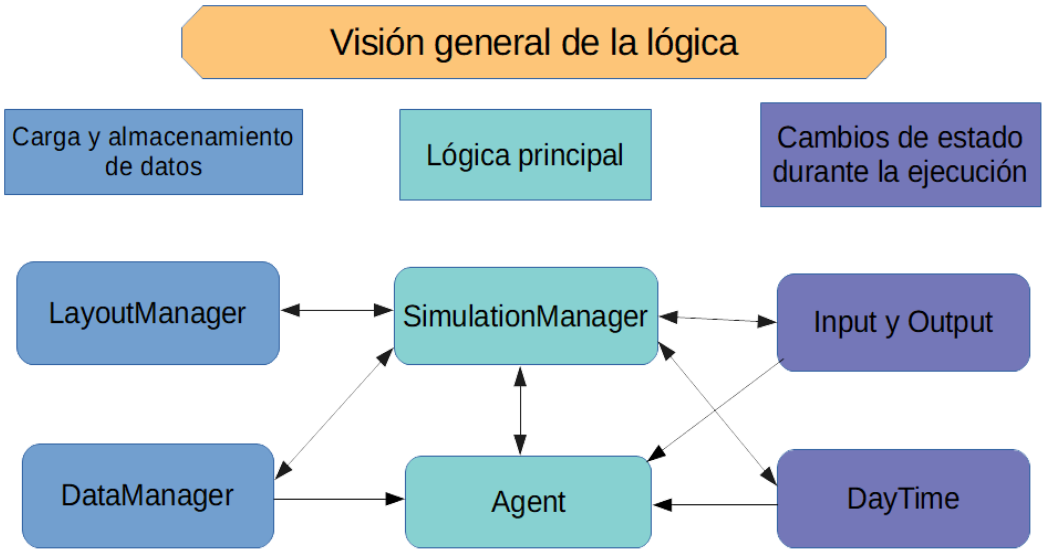


Figura 1: Diagrama de la visión general de la lógica

En la figura 1 se puede observar una visión general de la organización y funcionalidad básica de los módulos que forman la lógica de la aplicación, además de con qué otros módulos interactúa principalmente cada uno de estos.

En la sección de la izquierda se encuentran los dos elementos que se ocupan de cargar los datos necesarios para configurar la simulación. Su función principal es la reestructurar los datos introducidos por el usuario de tal forma que la lógica pueda usarlos cómodamente y dar acceso a estos datos al resto de elementos.

En el centro del diagrama se encuentran los módulos más importante para la lógica ya que son los que interactúan con más elementos del sistema y los que se encargan, en última instancia, del funcionamiento de la simulación. Todos los elementos del sistema se encargan de proporcionar los datos y procesos necesarios para que el módulo Agent pueda simular el comportamiento humano deseado y este pueda ser observado.

Los elementos de la región derecha se encargan principalmente de controlar los cambios en el entorno de la simulación. Gracias a estos la simulación avanza y cambia durante la ejecución. En el módulo del Input y Ouput la parte de output no realiza esta función sino que se encarga de extraer y procesar la información de la simulación para que el usuario la pueda aprovechar pero en este esquema se mantienen juntos para mantener su simplicidad y porque el input y el output están muy interconectados en su funcionamiento.

Gracias a este diagrama se puede observar que los elementos de cada región, diferenciadas por su color, solo interactúan directamente con los módulos de las secciones contiguas, de esta forma, los elementos que se encuentran en el centro suelen relacionarse con las otras dos regiones pero los elementos relacionados con la carga de datos nunca interactúan directamente con los que encargan de los cambios en el entorno. Ya se entrará más en detalle en las interacciones entre los elementos de la lógica en los apartados de diseño e implementación.

A continuación, se entrará en más detalle en cada uno de los elementos que forman el sistema, con este objetivo, se empezará con los módulos que se inician primero durante la ejecución del programa, además de centrarse primero en los elementos más simples para terminar con los más difíciles de abordar.

### 4.2.2. LayoutManager (Gestor de escenario)

El primero de los módulos que dirige el sistema es el **Gestor de escenario**( a partir de ahora **LM** por su nombre en inglés *Layout Manager*), éste es el primero en iniciarse cuando se introduce la escena. Su función principal es la gestión del escenario de la simulación, por lo tanto, se encarga de crear y guardar en memoria un modelo lógico del escenario además de habilitar formas para que el resto de elementos puedan obtener la información que necesitan de este. Concretamente, se guarda información referente al estado y posición de cada piso, aula y asiento de la facultad. En este caso, nos referimos como asiento a las posiciones que pueden ser asignadas como el destino de un personaje.

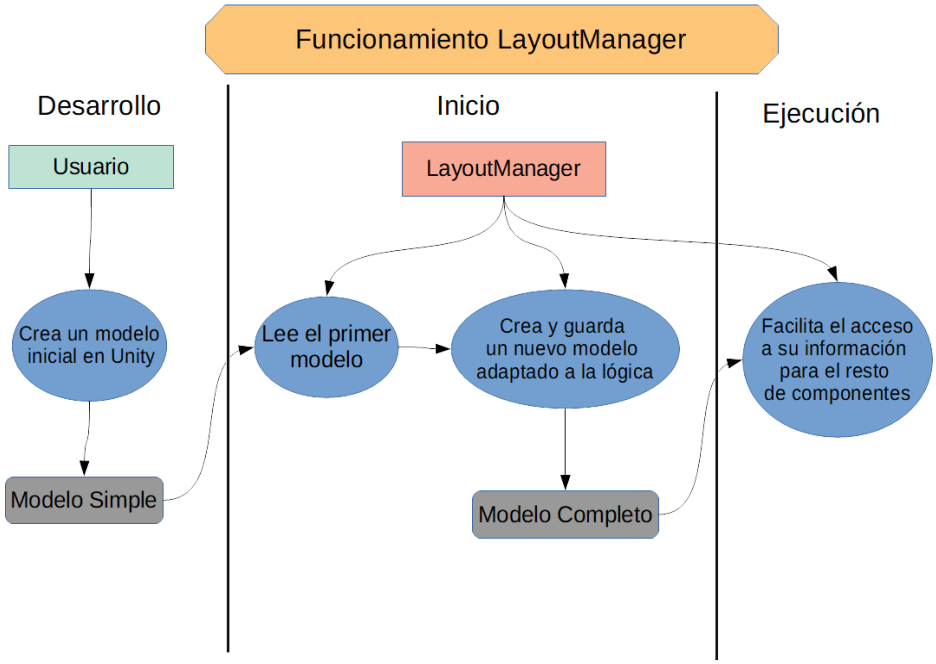


Figura 2: Diagrama del funcionamiento del LayoutManager

En la figura 2 se puede ver un esquema del funcionamiento básico del LM. Para empezar, el LM solo necesita un modelo lógico simple del escenario en la escena de Unity y no depende de ningún otro módulo de la lógica. En el momento que se inicia la ejecución de la escena lo primero que debe hacer este gestor es leer el modelo inicial y, a continuación, generar un nuevo modelo completo que pueda ser usado por el resto de la lógica. Durante el resto de la ejecución, el LM servirá para almacenar este modelo y proveerá métodos de consulta que usarán el resto de elementos de la escena para obtener la información que necesiten, por ejemplo, los personajes tendrán que acceder al LM para obtener la posición concreta a la cual dirigirse cuando deban desplazarse por el escenario.

### 4.2.3. Simulation Manager(Gestor de la simulación)

A continuación, se inicia el **Gestor de Simulación** (**SM** a partir de ahora por su nombre *Simulation Manager*), esta unidad es la piedra angular del sistema y sus funciones principales son las de coordinar y manejar el resto de elementos además de guardar y actualizar el estado de la simulación. De esta manera, el SM proporciona una visión global de la parte lógica de la simulación.

Este gestor está formado por una instancia global que será accesible por el resto de elementos de lógica. Gracias a esta característica y que se trata de la pieza central de la arquitectura funcionará como intermediario cuando un módulo necesita interactuar con otro módulo distinto del sistema, por ejemplo, cuando un agente necesita saber la posición de una habitación se comunicará con el LM, que tiene este dato, usando como intermediario al SM. También se encarga de manejar la velocidad de la simulación: cuando se produce un cambio de este tipo se llama al SM y este se encarga de procesar el cambio, guardar su nuevo estado y notificar a todos los elementos interesados sobre dicho cambio. Se puede aumentar y reducir la velocidad de la simulación además de ponerla totalmente en pausa.

En la figura 3 se puede ver el funcionamiento del SM de forma esquemática. Se pueden ver también las otras funciones que realiza: guarda toda la información relacionada con los horarios de clase y se encarga de actualizar el estado actual de cada asignatura y de que los agentes interesados en estos cambios reciban la información necesaria. Otra de sus funciones es informar a todos los personajes de la simulación de los cambios globales que los afectan a varios agentes, por ejemplo, cuando se debe iniciar un nuevo día en la simulación.

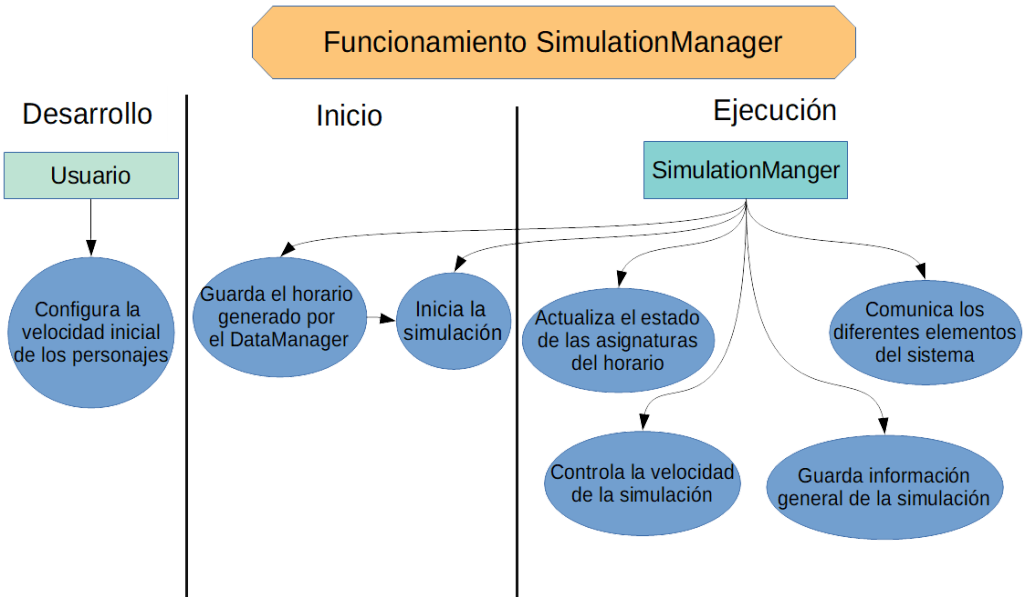


Figura 3: Diagrama del funcionamiento del SimulationManager

### 4.2.4. Data Manager (Gestor de Datos)

El **Gestor de datos** (se referirá a este como **DM** por *Data Manager*) es iniciado por el SM ya que este se encarga de comunicarlo con el LM. Este módulo se ocupa de cargar el resto de datos de la simulación: toda la información relacionada con los personajes y los horarios que deben seguir. Esta información se encuentra guardada fuera de la escena como recursos externos y se puede modificar por el usuario usando el editor de Unity. El DM necesita la información que contiene el LM sobre el escenario para poder crear un horario completo preparado para la simulación ya que en la información de cada clase se indicará la posición del aula en la que se imparte.

Como se puede ver en la figura 4, lo primero que hace el DM es cargar los recursos generados por el usuario previamente y, seguidamente, genera el horario de clases procesando toda la información presente en esos recursos. Después de crear el horario, inicia a todos los personajes y configura el estado y propiedades de cada uno de estos. Mientras se realiza este proceso, completa la información de cada asignatura añadiendo a sus datos las referencias a todos los personajes que participan en sus clases. Cuando ya se han creado todos los personajes, el DM guardará una referencia global que permitirá que otros los otros módulos puedan acceder a estos. Finalmente, se almacenará el horario generado en el SM.

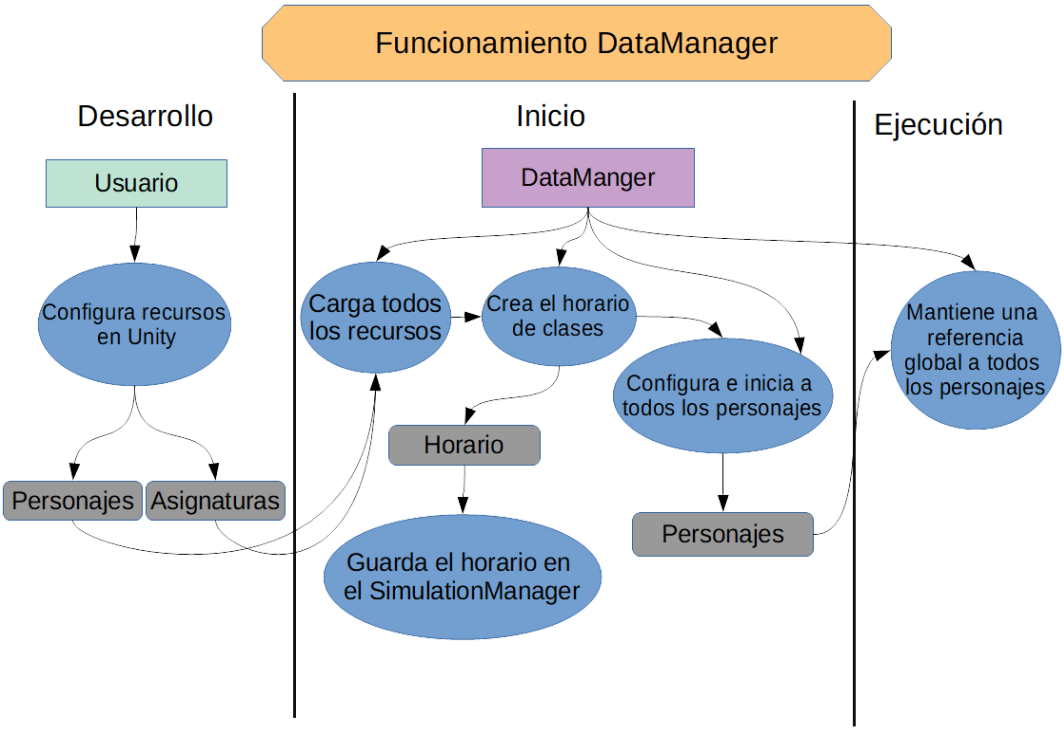


Figura 4: Diagrama del funcionamiento del DataManager

### 4.2.5. DayTime (Gestor de tiempo)

Hay otro componente que se inicia antes que el SM: el **módulo del tiempo**(de ahora en adelante **DT** por el nombre que recibe en el proyecto, *Day Time*). Comparado con el resto, este módulo es bastante simple y se encarga de guardar y actualizar la fecha y hora actual de la simulación . Principalmente, se encarga de transmitir el paso del tiempo de la ejecución de Unity a la simulación, con las modificaciones necesarias. El DT funciona a través de una instancia global para que todos los elementos de la simulación interesados puedan a acceder a la información que este suministra.

### 4.2.6. IO (Entrada y salida)

Uno de los elemento restantes es el de **Entrada y Salida (IO** por Input y Output), su funcionalidad está repartida entre distintos tipos de entidades y en varios componentes en la escena pero en conjunto permite al usuario interactuar con la simulación y extraer información sobre el estado de la simulación para el usuario. A través de este módulo pasan todos los cambios externos que afectan a la simulación y todos los datos que se presentan al usuario durante y luego de su ejecución. En esta aplicación la entrada y salida están muy relacionados ya que, para interactuar con la simulación, se usa mayoritariamente una interfaz gráfica donde se enseña información de la simulación pero también tiene botones que el usuario puede pulsar usando el ratón.

Los elementos concretos que forman el sistema de entrada y salida son: la interacción directa con la simulación a través del ratón y el teclado, la interfaz gráfica presente durante la ejecución de la simulación, el resto de la interfaz formada por un menú de inicio y un menú desplegable y el sistema de registro eventos mediante un archivo de texto. A continuación se entrará en más detalle en cada uno de estos elementos.

Para recibir el input de teclado y ratón, se usa el sistema de eventos integrado en Unity que ya incluye la gestión de este tipos de eventos. El input de teclado se usa principalmente para controlar la cámara, para interactuar con agentes concretos de la simulación y para iniciar eventos especiales. La cámara es la forma de que tiene Unity para poder manejar la visión de la escena durante la ejecución. Hay diversos tipos de cámara y el usuario puede alternar entre ellos, más detalles sobre la cámara se explicarán en el capítulo de la implementación. Se puede consultar el uso concreto de cada tecla en un menú de controles presente en la pantalla de inicio y en el menú desplegable de la escena principal.

El resto de la interacción se hace a través de los botones que se encuentran en la interfaz de usuario. Hay dos tipos de botones: los primeros se encuentran en la parte de la interfaz que siempre es visible durante la simulación y sirven para controlar la velocidad de la simulación y saltar el día actual, estos botones se comunican con otros elementos de la lógica como el SimulationManager que se encargarán de aplicar estos cambios al resto de elementos de la simulación. Los otros botones suelen estar poco tiempo visibles y no interactúan con la lógica de la simulación ya que están más relacionados con el manejo general de la aplicación. Aparecen en el menú principal y el menú de la simulación y sirven para cambiar entre la escena de la simulación y la del menú principal, para salir de la aplicación y para mostrar el panel donde se enseñan los controles del programa. En la figura 5 se puede ver el menú principal que se ha mencionado anteriormente con sus botones en el centro.


Figura 5: Imagen del menú principal

La interfaz gráfica sirve para mostrar información sobre el estado de la simulación y sus personajes y está formada por varios paneles repartidos por el contorno de la ventana y el texto que contienen. En esta interfaz se enseñan los siguientes datos organizados entre los distintos paneles que forman esta interfaz: el día y la hora actuales de la simulación, la velocidad actual de la simulación, el nombre y aula de las clases que se imparten en este momento y la información específica del personaje seleccionado. Para poder sacar toda esta información por pantalla se usan varios componentes que se encargan de leer, interpretar y transmitir los datos de la lógica a la interfaz gráfica.


Figura 6: Imagen de la interfaz gráfica

En la figura 6 se puede observar la parte de la interfaz gráfica descrita anteriormente dentro del programa en ejecución. Además se observa el menú desplegable durante

la simulación, que pertenece a la parte de la interfaz que no está presente en todo momento y está formada por este mismo panel y la pantalla de inicio.

La última herramienta para extraer información para el usuario se llama en el proyecto LogSystem y se encarga de generar un archivo de texto en el que se registran todos los eventos relevantes de la simulación con su marca de tiempo. De esta forma, el usuario tendrá un registro guardado con toda esa información luego de finalizar la ejecución del programa. Para construir este archivo, el resto de componentes de la escena llaman a una instancia global de LogSystem cuando les interesa registrar un evento y este se encargará de pasar esa información al formato adecuado, escribirlo en el archivo y guardarlo en el disco. Se crea un archivo diferente por cad día de simulación para tener toda la información registrada bien organizada.

### 4.2.7. Agentes (Agent)

Para finalizar, el último de los componentes del proyecto son los propios personajes a los que llamaremos **Agentes** ya que siguen el enfoque con ese mismo nombre. El resto de módulos, a parte del IO, son gestores con una única instancia, en cambio, la funcionalidad de los agentes está repartida entre varios personajes independientes que se encargan de controlar su propio estado y acciones.

Como agentes que son los personajes tienen la capacidad de percibir y interactuar con el entorno siguiendo sus propios objetivos además de guardar determinados conocimientos. Los agentes reciben una parte de la información a través del SM como los cambios globales de la simulación y los cambios relacionados con el horario de sus asignaturas. Ellos mismos se encargan de recoger la información sobre el estado de sus actividades y sobre los personajes que se encuentran cerca de ellos.

La mayor parte del peso lógico de los agentes se dirige al razonamiento, tienen que procesar toda la información externa juntamente con su conocimiento sobre su estado actual y obtener una respuesta acorde a sus objetivos. Estos objetivos dependen principalmente de su estado actual y pueden ser seguir su horario de clases y actividades habitual, intentar abandonar el edificio ante alguna emergencia o hasta perseguir a otros personajes de la escena. Los personajes pueden seguir cuatro situaciones de simulación distintas: rutina habitual en la FDI, evacuación en caso de una emergencia, contagio de enfermedad infecciosa y plaga zombie.

Para poder seguir sus objetivos los personajes necesitan tener una forma de actuar, en este caso, pueden realizar dos tipos principales de acciones: desplazarse por el escenario hacia un objetivo o interactuar directamente con otros personajes. Para

poder moverse por la facultad se usará una herramienta de navegación de mallas de Unity llamado NavMesh que se controlará desde la lógica del agente. Para relacionarse con los otros personajes se hará usando la propia lógica del agente pero ya hablaremos más en detalle posteriormente.

También mencionar que los personajes deben tener una representación visual en el modelo 3d, en este caso, se usarán modelos de personas y un cubo con un color determinado por su estado actual para que el usuario pueda identificar el estado en el que se encuentra cada personaje de la simulación.

En este momento ya tenemos una idea general de las funciones de las partes que componen esta  simulación y de cómo interactúan entre sí, ahora vamos a ver más detalladamente cómo se han implementado.

## 4.3. Implementación

### 4.3.1. LayoutManager (Gestor de escenario)

Éste módulo está implementado usando solamente un componente en Unity que se encuentra en una entidad de gestión, esto es básicamente una entidad que solo contiene componentes con este tipo de funciones y no interactúa directamente con el escenario ya que no es visible ni tampoco se desplaza por el mapa. El LayoutManager está implementado a través de un componente para que lo llame Unity automáticamente al iniciar la escena y para poder configurar sus parámetros desde el editor.

Entrando en más detalle, en este componente se guarda la posición de todos los lugares que pueden ser útiles a los agentes de una manera estructurada, de esta forma, se crea y almacena un modelo lógico de la facultad. Este modelo tiene que describir físicamente el edificio de la FDI de una manera lo más simplificada posible y tiene que estar preparada para que los agentes puedan leer y interpretar la información que necesitan para su comportamiento.

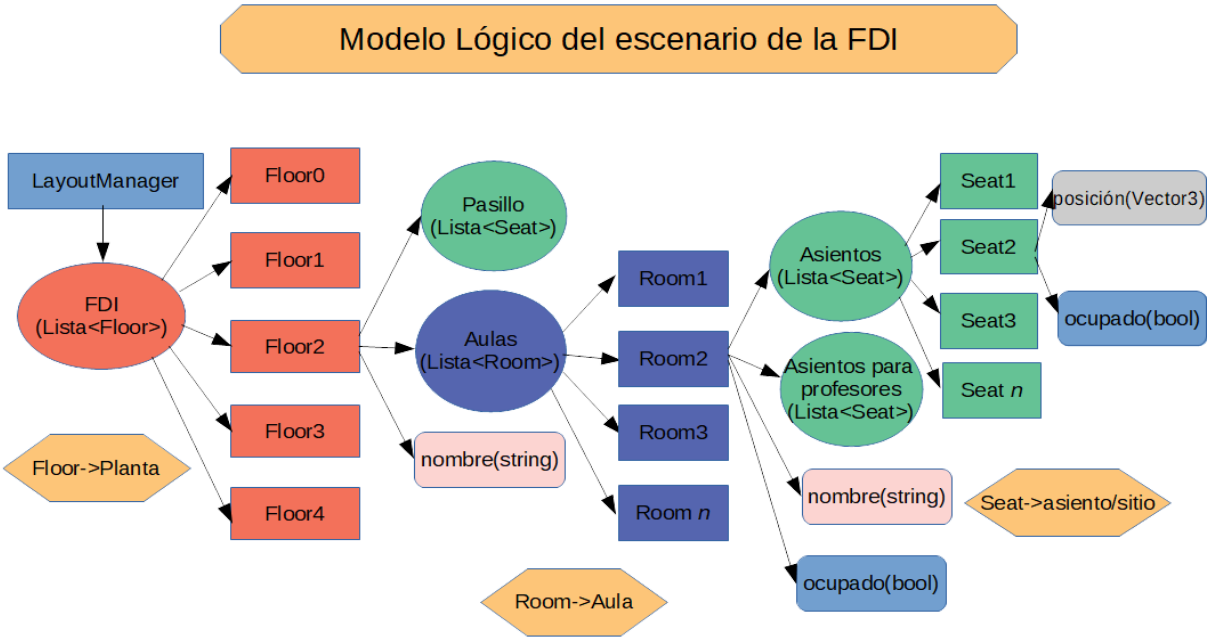


Figura 7: Diagrama del modelo lógico del escenario de la FDI

El modelo en cuestión está implementado usando una lista de una clase llamada **Floor** que representa una planta del edificio que, a su vez, está formada por una lista de posiciones en sus pasillos y otra lista de la clase **Room**. La clase Room es la representación de una habitación que guarda información sobre su nombre, entrada y una lista de **Seat**, la clase que representa una silla o cualquier sitio en el que se puede posicionar un agente como resultado de un desplazamiento. Esta clase almacena su posición y un booleano que informa si el sitio ya está ocupado. Toda esta información se puede ver de una forma más clara y organizada en la figura 7.

Las tres clases descritas anteriormente son clases pequeñas que no heredan de ninguna otra y su función principal es la de guardar información de manera organizada para que otros componentes puedan acceder a ella. También tienen algún método para acceder a posiciones de formas específicas, por ejemplo, acceder al sitio libre de un pasillo más cercano a una posición determinada. En el LayoutManager también se guardan las entradas del edificio separadas del resto del modelo para tenerlas perfectamente diferenciadas, además, no es muy útil incluirlas en la clase Floor ya que siempre estarán en la planta baja.

Para poder cargar todo este modelo, el LayoutManager usa una técnica muy común en Unity bastante sencilla pero muy útil para los diseñadores que no deben tocar el código. Se trata de crear entidades vacías por la escena a las que este componente tiene acceso y usar su posición como indicador de la posición lógica deseada, así, para diseñar el escenario solo hay que crear entidades vacías y colocarlas en la posición que se desea usar en la lógica. Para hacer esto posible, se aprovecha que las entidades pueden tener entidades hijas y, de esta forma, se crea una estructura de entidades vacías anidadas a diferentes niveles repartidas por todo el modelo de la facultad que el componente puede interpretar durante el inicio y crear el escenario lógico que ya podrán usar los agentes.

En este modelo solo se guardan de forma organizada los puntos de interés del escenario para los agentes, la otra parte del mapa lógico es la que se encarga de que los personajes se puedan desplazar entre estos puntos y será el propio Unity el que se encargue de generarlo y almacenarlo, ya se hablará más en detalle de esto en el apartado sobre el desplazamiento de los agentes.

En primera instancia se planteó usar el propio modelo gráfico de la facultad para crear el lógico pero resultaba bastante ineficiente ya que igualmente se tenían que designar muchos puntos de entrada y, en muchos casos, había modelos que agrupaban varios elementos que debían estar diferenciados para los agentes, por ejemplo las sillas de las aulas.

### 4.3.2. Data Manager (Gestor de Datos)

El DataManager es bastante similar al LayoutManager ya que también se encuentra implementado en un solo componente que se encuentra en la misma entidad de gestión que este, además, también se ocupa de cargar datos, procesarlos y guardar sus referencias en memoria. Sin embargo, la diferencia es que el DataManager carga los datos sobre los agentes y el horario y depende del Layout Manager para realizar esta tarea.

Data Manager está implementado a través de un componente de Unity para poder configurarlo desde el editor y mantener la cohesión en el diseño porque, a diferencia del LayoutManager, el Simulation Manager se encarga de llamarlo para que se inicie. Esto es así porque el DM necesita el acceso al mapa lógico del LM y el SM se lo puede conceder.

¿Por qué es necesaria toda esta complicación? Porque el Data Manager se encarga de crear el horario de clases que usará la simulación  y, en la información sobre cada asignatura, se tiene que guardar la posición de las aulas en las que se imparten esa asignatura. Este horario será de vital importancia porque todo el

modelo está centrado en la interpretación de un horario de clases ya que este será el principal motor que guíe las acciones de la mayoría de los personajes de la simulación. El horario se guarda en la clase SubjectSchedule, una clase bastante compleja por la cantidad y variedad de información que contiene. Se describe su especificación completa en el apartado sobre el SimulationManager ya que es este el que hace su mayor uso de la clase SubjectSchedule.

Para poder cargar todos los datos que necesita el DataManager se aprovecha la clase de Unity Scriptable Object que se ha explicado anteriormente. En el proyecto se han implementado dos tipos principales de Scriptable Object:

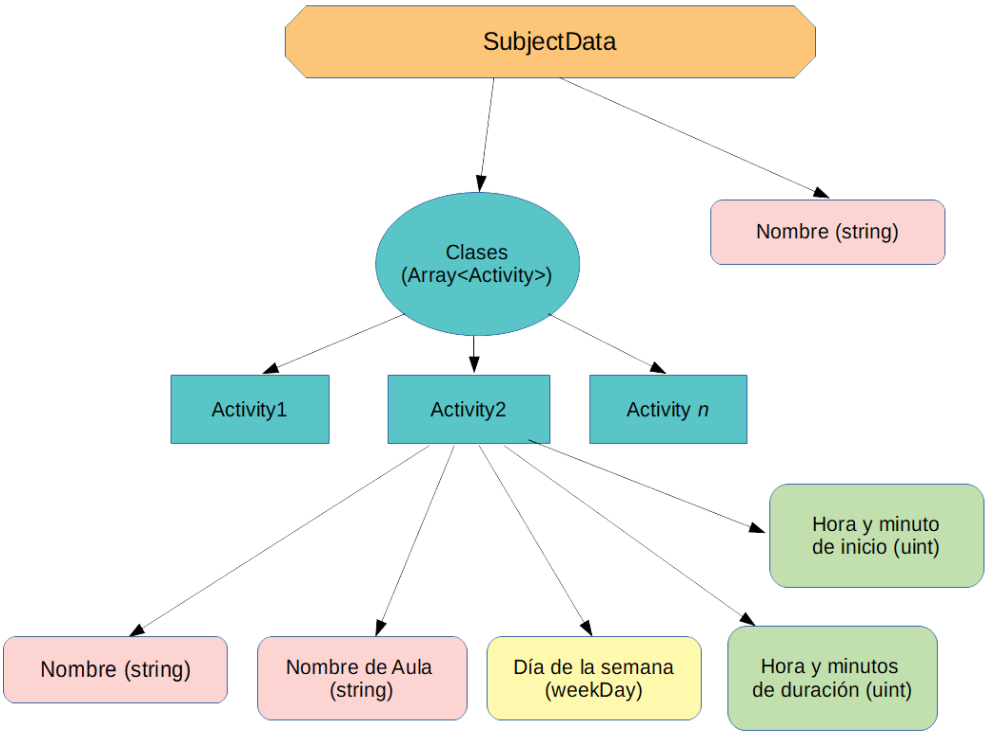


Figura 8: Diagrama de la estructura de SubjectData

**-SubjectData**: cada instancia de esta clase guardada en los recursos representa una asignatura y, en esta, se guarda toda la información necesaria para crear una asignatura lógica en la simulación: el nombre, el número de clases semanales y, de cada una de ésas, el nombre del aula, el día, la hora de inicio y su finalización. En la figura 8 se puede ver de forma esquemática la información que contiene esta clase.

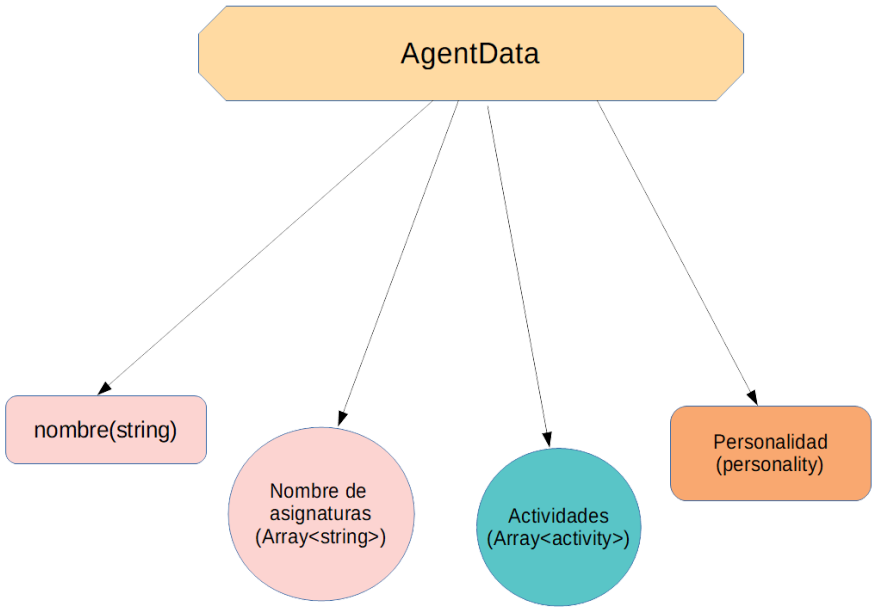


Figura 9: Diagrama de la estructura de AgentData

**-AgentData**: esta clase contiene toda la información necesaria para crear un personaje de la simulación: el nombre, la personalidad, el número de asignaturas que cursa y, de cada una, su nombre y una lista de otras actividades que tiene en la facultad. Para cada actividad hay que proporcionar un nombre, aula, hora de inicio y duración. En la figura 9 se observa un diagrama donde se muestra toda la información contenida en AgentData.

Dentro del proyecto hay dos clases llamadas StudentData y TeacherData a las que aquí se referirá conjuntamente con el nombre AgentData ya que solo se diferencian en que la primera sirve para crear personajes normales y la segunda solo para profesores, esta división existe porque inicialmente se habían planteado mayores diferencias en su funcionalidad.

La ventaja que aporta este sistema de carga de recursos es que, en cualquier momento, se puede añadir otro alumno o asignatura sin necesidad de modificar ningún otro archivo o el código de la aplicación, de esta forma, personas que no saben programar o no conocen el código pueden aprovechar el potencial del programa.

Lo primero que hace el Data Manager al iniciarse es cargar todos los Scriptable Objects en memoria y, a continuación, genera el horario juntamente con la información del mapa lógico de la facultad. Esta tarea se divide en dos fases:

Primero se guarda la información de cada asignatura en un diccionario que usa su nombre como clave (string) y el contenido es el resto de información sobre la

asignatura (subjectInfo). En cada entrada se guarda información similar al Subject Data pero estructurada de otra forma y añadiendo una referencia a la localización de cada clase. Este diccionario se guarda en el SubjectSchedule que se le devolverá al SimulationManager.

La segunda fase consiste en crear todos los agentes y, para conseguirlo, se ha usado lo que se denomina en Unity **Prefabs**: un prefab es una entidad que se guarda fuera de una escena y se puede usar para instanciar copias en cualquier escena. En el proyecto hay un prefab de agente que contiene todos los componentes necesarios para que funcione un personaje en la simulación ya configurados. Por cada AgentData cargado se instancia una copia del prefab y se configura su componente Agent usando la información del AgentData en cuestión.

Para finalizar esta fase se introduce en la información de cada asignatura (SubjectInfo) el nombre de cada alumno que la cursa y el de los profesores que la imparten. Adicionalmente, el Data Manager también tiene la capacidad de añadir una actividad adicional a los personajes que representa la comida en la facultad. En cada AgentData se puede elegir si se quiere que se haga este cálculo, de ser así, comprobará por cada día si tiene clases o actividades antes y después de la hora de comer. Si se cumple esta condición añadirá esta nueva actividad a su lista.

Con todos los agentes ya creados, el DataManager devuelve al SimulationManager el horario configurado y, a partir de este momento, su función será dar un acceso global a todos los agentes a través de la entidad padre de todas los agentes.

### 4.3.3. DayTime (Gestor de tiempo)

El gestor de tiempo está implementado en un componente llamado DayTime y usa el patrón de diseño singleton para poder ser accesible en todo momento en cualquier componente de la escena y asegurarse de que solo pueda haber una instancia.

Este componente guarda la fecha actual de la simulación y se encarga de actualizarla, ésto lo hace a comprobando el tiempo que ha pasado entre cada llamada al método Update de Unity. Para obtener este número se usa el parámetro **deltaTime** de la clase de Unity **Time**. El resto de elementos de la simulación accederán a este componente cuando necesiten saber la hora actual. También está diseñado para controlar la velocidad de la simulación, ya que es posible alterarla en tiempo de ejecución y será el Simulation Manager que se encargará de notificar esas variaciones. Finalmente, puede ser consultado con ciertas funciones para que calcule si una actividad ya ha empezado o terminado.

### 4.3.4. Simulation Manager(Gestor de la simulación)

Este gestor es la pieza central de la arquitectura ya que se encarga de comunicar y coordinar el resto de módulos y controla el estado general de toda la simulación. Se ha implementado a través de un componente y usando el patrón de diseño singleton, de esta forma lo llama Unity en su bucle de juego y se puede acceder desde el resto de elementos sin tener que guardar una referencia a una instancia concreta.

Para controlar la simulación, este gestor guarda el horario generado por el DataManager usando la clase **SubjectSchedule**, esta clase contiene los datos del horario ya preparados para ser usados y no tiene lógica propia sino que se delega a otra clase llamada **Subject**. SubjectSchedule usa internamente también la clase llamada **SubjectInfo** para guardar información sobre cada asignatura. A continuación se presenta una descripción de cada una de estas clases.

-**SubjectInfo**: sirve para guardar toda la información de una asignatura, ésta consiste en: su nombre, la lista de profesores, la lista de alumnos, el horario y el estado. Hay cuatro posibles estado para una asignatura que se guardan usando una variable enumerada (subjectState): empezando, activa, terminando o finalizada.

-**Subject**: es la representación lógica de una clase de una asignatura por lo que contiene el día de la semana, la hora de inicio y final, el aula donde se realiza y una referencia a la información de esta asignatura (subjectInfo). Tiene integrada la lógica para actualizar el estado de la clase según varía el tiempo y para informar a los agentes interesados en esos cambios.

-**SubjectSchedule**: esta clase contiene un diccionario de nombres con la información asociada a la asignatura (SubjectInfo) con dicho nombre, un array de cinco listas donde se guardan todas las horas de clase (Subject) divididas entre los días lectivos de la semana y una lista con las clases (Subject) activas actualmente.

En la figura 10 se muestra toda la información descrita anteriormente de una forma más concisa y esquemática desde el punto de vista de la mayor clase, SubjectSchedule.

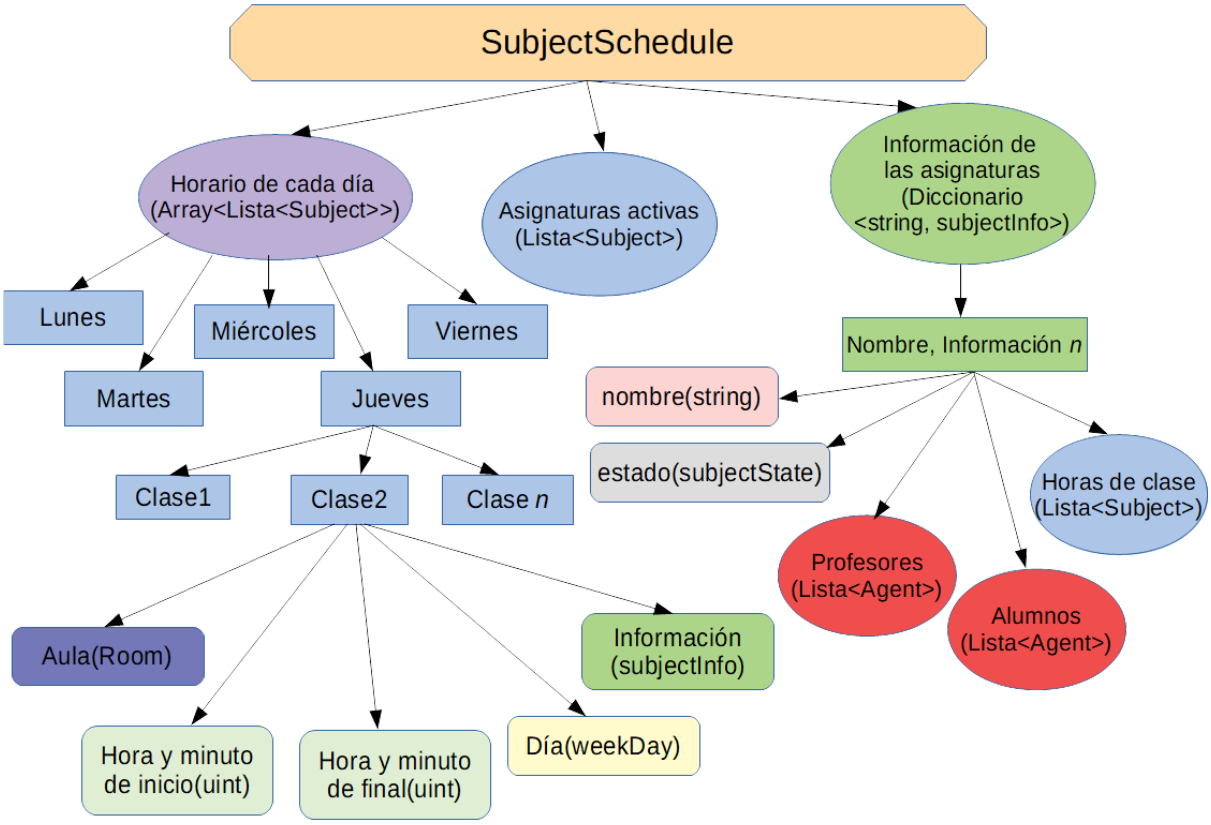


Figura 10: Diagrama de la estructura de SubjectSchedule

¿Cómo utiliza el SimulationManager toda esta estructura de datos y lógica? Pues en cada llamada del método Update de Unity pide la hora al gestor de tiempo y actualiza el estado de todas las clases del día presente usando la hora obtenida. Los objetos de la clase Subject se ocuparán individualmente de comprobar si, a causa del cambio en la hora, su estado debe cambiar y lo notificarán al gestor por si se tienen que añadir o quitar en la lista de asignaturas activas. Además los Subject se encargan de informar a los profesores y estudiantes listados en su asignatura de ese cambio de estado para que actúen consecuentemente.

Otra cosa que hace este gestor en el Update es comprobar si ya ha terminado el día porque también se ocupa de notificar a todos los agentes para que se preparen para empezar el siguiente día. En este momento, también se encarga de limpiar el estado de todas las asignaturas para prepararlas correctamente y no dejar ninguna habitación ocupada o asignatura activa cuando no debería.

El SimulationManager guarda referencias públicas a los otros gestores, como el LayoutManager y el DataManager, que se encuentran en la simulación para que el resto de elementos de la lógica tengan una forma fácil de acceder a estos usando la referencia global al SimulationManager, de esta manera actuará como un intermediario en gran parte de las comunicaciones entre diferentes módulos de la

lógica. También guarda información global útil para las simulaciones especiales, por ejemplo, guarda el estado y configuración de la simulación de la enfermedad usando la clase InfectiousSimulationInfo.

Para finalizar, el SimulationManager procesa y coordina los cambios de velocidad en la simulación y el estado de pausa: guarda el multiplicador de la velocidad global y hace que todos los agentes, el gestor de tiempo y la interfaz se actualicen cuando este es modificado. Hace lo mismo para cuando se activa o desactiva la pausa de la simulación.

### 4.3.5. IO (Entrada y salida)

Este módulo es el que se encuentra más repartido entre diferentes entidades y componentes de todos y se le pueden atribuir **cuatro funciones** principales: control de la cámara, sacar información por pantalla a través de la interfaz visual, manejar la pulsación de teclas y botones de la interfaz para interactuar con la simulación y crear archivos de texto para registrar la sucesión de eventos de la simulación.

#### 4.3.5.1. Cámara

Para el **control de la cámara** se han implementado dos componentes distintos que se encuentran en la entidad que también contiene el componente de Unity de la cámara. Para poder visualizar una escena en Unity durante la ejecución siempre hay que tener un componente de cámara, este componente se asigna a una entidad y se puede mover por toda la escena y añadir todo tipo de lógica como cualquier otra entidad. Esta cámara tiene varias opciones de configuración para modificar la visualización de la escena como cambiar de vista ortogonal a perspectiva, mover el plano cercano y lejano o modificar la apertura de la lente.

Cada uno de los dos componentes implementados sirve para una configuración de la cámara: la libre y la de seguimiento. La primera está implementada en el componente FreeCamera y permite el control libre de la cámara por el escenario usando el ratón y teclado, en cambio, la segunda sigue automáticamente a un agente seleccionado por la escena y usa el componente FollowCamera. Dentro de la simulación se puede cambiar en cualquier momento entre estos dos tipos de cámara.

La cámara a la que llamamos **FreeCamera** se asemeja a una cámara en primera persona que se podría encontrar en un videojuego pero bastante simplificada. En Unity es relativamente sencillo implementar una cámara así por las funciones que permiten fácilmente acceder al desplazamiento del ratón y el vector director donde mira la cámara en la escena.

El otro modo de visualización, **FollowCamera**, permite seguir cómodamente con la cámara a un personaje mientras sigue su rutina sin tocar ningún botón. Este efecto se consigue replicando la posición del agente seleccionado con un cierto retardo y distancia de compensación en cada llamada al Update. Ahora bien,¿cómo se consigue seleccionar al personaje? Este componente tiene un método Select que, internamente, usa otro método de Unity llamado Raycast que se trata de una función de su motor de físicas. Esta función simula la proyección de un rayo y devuelve la colisión con el primer objeto con el que colisiona. Hay muchos parámetros para controlar su comportamiento específico pero, en este caso, se usa una configuración que permite acceder a la entidad que se encuentra donde mira la cámara.

Aprovechando esta tecnología se ha podido implementar que, usando el ratón, se pueda seleccionar el personaje al que está mirando la cámara. Para terminar este proceso, falta un paso más: identificar que la entidad seleccionada es realmente un personaje de la simulación y no otra entidad que puede formar parte del escenario. Con este fin se ha usado una técnica conocida como Duck Typing, esta técnica consiste en identificar que es un objeto según sus capacidades y propiedades y no directamente conociendo su tipo. En el caso que nos ocupa se sabrá que una entidad es, efectivamente, un agente si tiene presente en su lista de componentes el componente Agent. Conocer si una entidad tiene un componente es muy sencillo en Unity: todas las entidades tienen el método GetComponent qué retorna el componente del tipo especificado de su entidad. Simplemente usando este método y comprobando que no devuelve el valor *null* se puede identificar si una entidad es un agente.

Por defecto, la cámara será libre pero en cualquier momento se puede seleccionar a un personaje para seguirlo, por otra parte, cuando la cámara está en modo seguimiento se puede pulsar otra tecla para liberar la cámara. También se pueden desactivar las dos cámaras para poder desbloquear el cursor y usar los botones de la interfaz gráfica. Para agilizar el desplazamiento de la cámara por el escenario se pueden usar dos teclas adicionales para mover la cámara directamente entre una lista de posiciones de cámara determinadas. Se pueden crear más posiciones y modificar las ya existentes desde el propio editor de Unity de una forma similar a la configuración del modelo lógico del escenario.

#### 4.3.5.2. Input

El que se encarga de manejar el input de la aplicación es el componente InputComponent, que actúa como un **gestor de input** simplificado. Es el componente encargado de recibir toda la entrada de teclado y ratón y transmitir la información adecuada a los elementos que lo necesitan. Aparte del input relacionado con las cámaras también maneja el que se encarga de modificar el

estado de agentes seleccionados y el que sirve para abrir o cerrar ciertas interfaces. La información del estado del teclado y el ratón se obtiene a través de Unity usando la clase Input que permite verificar la pulsación de cada tecla individualmente, por ejemplo, usando su método GetKeyDown con el parámetro KeyCode.D permite detectar el momento en el que se empieza a pulsar la tecla *d*.

Aparte del InputComponent hay otra manera de interactuar con la simulación y esto es a través de los **botones de la interfaz**. Todos los botones están implementados a través del componente de Unity llamado Button que permite crear y hacer funcionar un botón rápidamente sin tener que modificar el código ya que permite directamente seleccionar una entidad de la escena y llamar a cualquier método público de sus componentes y hasta deja configurar algunos parámetros. Aunque esto no es suficiente en algunos casos y se han creado algunos componentes sencillos que se encargan de que algunos botones realicen su función deseada.

Estos botones tienen una parte visual y otra funcional, en este apartado se describe la parte funcional de estos botones y, en el siguiente, se trata su apartado visual. Primero trataremos los botones que se encuentran siempre visibles durante la simulación. Los botones de este tipo son los que listan a continuación:

- Dos botones que permiten mostrar y ocultar los paneles donde se muestra la información del personaje seleccionado y la de las clases impartidas actualmente.
- Dos botones que permiten al usuario aumentar y disminuir la velocidad de la simulación.
- Un botón que permite acabara el día actual y pasar al siguiente día de la simulación directamente.

Los tres últimos botones descritos interactúan con la simulación a través del SimulationManager ya que este es el que se ocupa de controlar la velocidad de la simulación y los cambios de día. El SM realiza este proceso se realiza de la siguiente forma:

1. Recibe la notificación de uno de estos cambios desde un botón
2. Procesa el cambio alternado su estado interno
3. Notifica a todos los agentes y gestores que necesiten actualizar su estado

El resto de botones se encuentran en los menús de la aplicación, concretamente, en la escena de inicio y en un menú desplegable durante la simulación. Estos botones se encargan de cambiar de escena, salir del programa o enseñar y ocultar el panel de controles. Para hacer funcionar estos botones se han implementado dos componentes sencillos llamados ExitButton y SceneSwapButton que tienen un solo método preparado para ser llamado desde el componente Button. Los dos

componentes usan internamente funciones de Unity que manejan la salida de la aplicación en ExitButton y el manejo de la escena activa en SceneSwapButton. Para enseñar y ocultar el panel de controles simplemente se activa o desactiva la entidad de ese panel.

#### 4.3.5.3 Interfaz Gráfica

La última parte de este módulo la forma la retroalimentación visual a través de una **interfaz gráfica**. Anteriormente ya hemos explicado que Unity da muchas facilidades para crear interfaces visuales ya que siempre está presente, de una forma u otra, en todos los videojuegos. También se ha introducido brevemente a cómo se crean estas interfaces por lo que nos centraremos en la parte específica de este proyecto. Igual que en la parte del input, la interfaz también está dividida en dos partes: la HUD y los menús de control.

Los menús son bastante simples: hay un menú principal con una imagen de fondo y algunos botones interactuables (ver figura 5) y, ya en la escena principal, se puede desplegar un panel con más botones (ver figura 6). Tanto en el menú principal como en el menú desplegable se puede usar un botón para enseñar un panel adicional donde se enseñan los controles específicos para el manejo de la simulación. En la figura 11 se puede ver este último panel concreto.

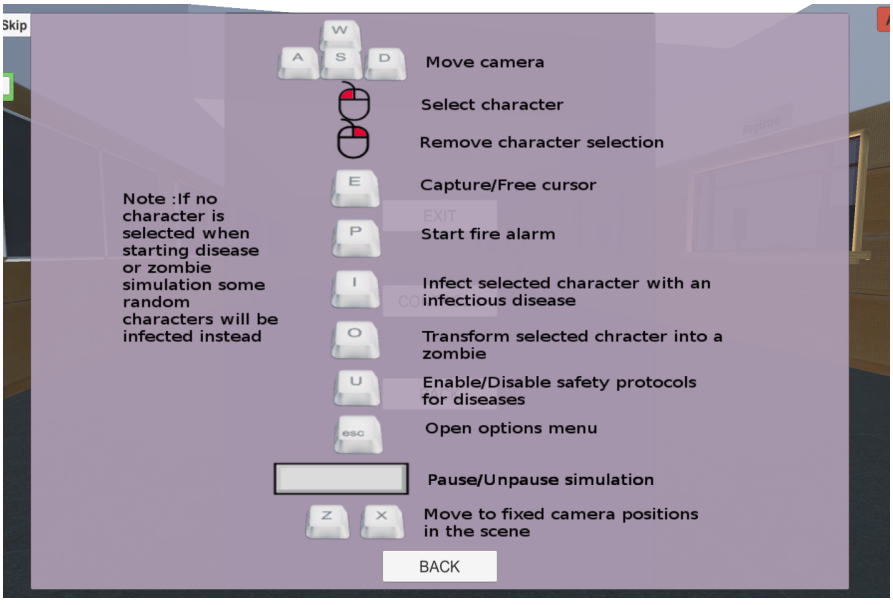


Figura 11: Imagen del panel de controles

La otra parte de la interfaz la forma la HUD. La HUD o barra de estado es un concepto usado frecuentemente en el campo de los videojuegos que se refiere a toda la información que se enseña en la interfaz al usuario durante la partida y le permite conocer el estado del sistema. En este caso usamos este mismo concepto para enseñar ciertos elementos relevantes del estado de la simulación. Esta interfaz

está formada por varios paneles con texto y números que están repartidos por la ventana. Algunos de estos paneles se pueden ocultar para evitar sobrecargar la pantalla cuando no son de interés. A continuación se detallarán todos los elementos que lo forman y su posición en pantalla usando como referencia la figura 12:


Figura 12: Imagen de la interfaz gráfica

1. En este panel pone el día y hora actual de la simulación que se actualizarán en tiempo real usando el componente TimeUI. Justo a la derecha está el botón que permite pasar al día siguiente directamente.

2. En este otro se puede ver el multiplicador a la velocidad de la simulación, aquí también se encuentran los dos botones para modificarlo. Es el propio SimulationManager el que se encarga de actualizar el texto de este panel.

3. Este panel muestra la información sobre el agente seleccionado. Es un panel bastante complejo y, para su manejo, usa el componente AgentUI. Concretamente, muestra el nombre, tipo, acción, personalidad, simulación y asignatura actual del agente. Se puede mostrar u ocultar parcialmente usando el botón que se encuentra en su interior.

4. Funciona de forma similar al panel anterior pero enseña una lista con el nombre y aula de todas las clases que se imparten en este momento. Usa el componente SubjectUI para su funcionamiento y también se puede ocultar.

5. Aparece cuando se pone pausa a la simulación para poder diferenciar los dos estados fácilmente y en cualquier momento.

Los componentes SubjectUI, AgentUI y TimeUI tienen una funcionalidad parecida ya que todos se encargan de recoger la información presente dentro de los gestores para transformarla en texto plano, sacarla por pantalla y actualizarla constantemente. Los componentes AgentUI y SubjectUI necesitan otros componentes de Unity que permiten reescalar automáticamente elementos de la interfaz según su contenido.


Figura 13: Imagen del panel de *feedback*

También se ha añadido un panel adicional para dar retroalimentación de los eventos que suceden en la simulación que se puede ver en la figura 13. Por defecto es invisible pero cuando se inicia alguna simulación especial, cambia el día o se modifica cierta configuración global se vuelve visible durante unos segundos. El texto que aparece en este panel está determinado por el último cambio ocurrido en la simulación. El componente FeedbackUI es el que se encarga de todo el control de este panel.

#### 4.3.5.4. Sistema de registro

Se ha implementado un componente llamado LogSystem que se encarga de registrar todos los eventos relevantes de la simulación y guardarlos de manera ordenada en archivos de texto. Su implementación sigue el modelo singleton, con una instancia única y global que puede ser accedida desde cualquier otro componente, de esta forma, cualquier elemento de la simulación puede pedir al LogSystem registrar información usando su método llamado Log donde solamente hay que indicar el texto que se quiere registrar.

Cada vez que se inicia un día se crea un archivo nuevo por lo que se guarda un archivo independiente por cada día que se simula consiguiendo un registro completo bastante estructurado. Para la creación y escritura de archivos se usa la clase StreamWriter de la librería estándar de C#. No se escribe directamente en el archivo cada vez que se llama a Log sino que se guarda en una lista cada escritura que se quiere realizar y, cada vez que pasa un periodo de tiempo determinado, se hace un volcado a archivo de todo su contenido a la vez que se vacía la lista. El método WriteLogs es el que hace este volcado.

En un sistema de registros, es de vital importancia saber el momento concreto en el que sucede cada evento por lo que el propio LogSystem se encarga de añadir al texto de cada Log la marca de tiempo actual para guardarla posteriormente en el archivo.

Finalmente, el LogSystem también puede hacer el recuento de ciertos eventos donde este valor sea de interés. El método Log puede identificar el tipo de evento con un parámetro opcional del enumerado logType. Cuando se registra un evento con un tipo especial, el LogSystem, aumenta el contador de ese tipo y añade automáticamente al texto original el nuevo valor del contador además de un texto adicional para describir el evento. Esto se usa para el número de contagios que ocurren, personas que evacúan el edificio o personas transformadas en zombie y todos los contadores se reinician cuando empieza un día.

### 4.3.6. Agentes (Agent)

Los agentes, similarmente al SimulationManager, se encuentra en una zona central de la arquitectura pero con una gran diferencia: en vez de coordinar y manejar el resto de elementos, los agentes se encuentra en el centro porque el resto de módulos se encargan conjuntamente de aportar todos los datos y procesos necesarios para que realicen su función principal. Está función es la de simular el comportamiento humano en una situación social por lo que es una de las funcionalidades esenciales del proyecto. Su implementación se encuentra en un componente llamado Agent pero necesita otros componentes ya existentes de Unity para funcionar correctamente. Usa varios componentes estándar para poder renderizar y permitir las colisiones además de otros dos componentes vitales para su IA: el NavMeshObstacle y el NavMeshAgent.

Estos componentes están relacionados con la herramienta de pathfinding que tiene Unity y permite que las entidades en las que se añade aprovechen esta tecnología. **NavMeshObstacle** permite que el sistema de navegación interprete esa entidad como un obstáculo y los otros personajes lo puedan evitar automáticamente cuando se desplacen. Por otra parte, **NavMeshAgent** es el componente que permite a su entidad desplazarse por la malla de navegación. Para poder aprovecharlo hay que llamar a sus métodos desde otros componentes.

**Agent** es el componente que tiene más líneas de código de todos ya que tiene que controlar todo el estado y comportamiento de los personajes. Sigue la estructura de agente inteligente por lo que percibe los cambios en el entorno a través de receptores que le permite interpretar la información en base a su conocimiento previo y realizar una acción a través de unos actuadores.

La implementación que siguen los agentes de esta simulación no sigue exactamente la estructura estándar indicada en modelos formales como el de la FIPA de un agente inteligente sino que su estructura se ha basado en ese modelo pero se ha tomado alguna pequeña libertad de diseño. Estas modificaciones sirven, principalmente, para conseguir un sistema más centralizado y sincronizado dando un mayor peso al gestor de la simulación, cosa que es de gran interés en este caso concreto ya que permite mejorar el rendimiento del programa porque en esta simulación pueden llegar a haber muchos personajes presentes simultáneamente cada uno con su propio razonamiento.

Gracias a esta centralización se puede aprovechar el hecho de que muchos agentes tienen que realizar habitualmente el mismo razonamiento, como es el caso de todos los alumnos que cursan una misma asignatura y tienen que comprobar si ya ha empezado la clase, y hacer esa parte del razonamiento que es común en todos esos agentes una única vez en el gestor de la simulación para luego comunicar el resultado a los agentes interesados.

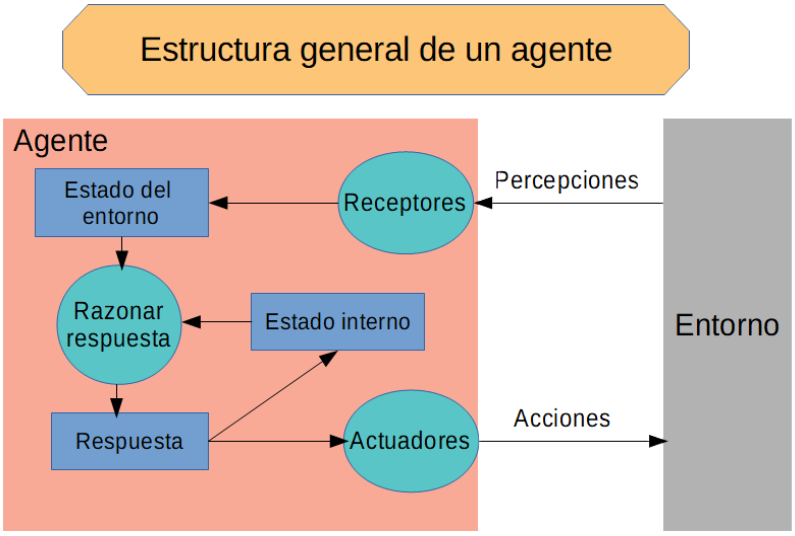


Figura 14: Diagrama de la estructura general de un agente

Para entrar más en detalle en la implementación del componente Agent dividiremos este tema en varios apartados siguiendo la estructura de los agentes inteligentes. En la figura 14 se puede ver un esquema general de esta estructura basado en la definición seguida en este proyecto. Empezaremos tratando su estado interno y continuaremos con los las tres partes fundamentales para el funcionamiento de un agente: los receptores, el razonamiento de la respuesta y los actuadores, en ese orden.

#### 4.3.6.1. Estado de un agente

El estado de un agente se encuentra dividido entre un conjunto de variables pero la información principal se encuentra concentrada en un struct llamado **agentState**. En este struct se guarda la acción que está realizando actualmente (agentAction), el tipo de agente (agentType), el tipo de evento que está simulando(simulation), la personalidad (personality), si se está desplazando y si tiene actividades pendientes. AgentState es bastante útil porque unifica el acceso a toda la información relativa al estado general de un agente y permite que otros elementos de la simulación accedan a esta información cómodamente.

- **AgentAction** permite diferenciar qué tipo de acción realiza un agente, éste siempre realiza una de las siguientes opciones: dirigirse a una habitación donde debe realizar una actividad, realizar la actividad, salir luego de terminar una actividad, descansar o no estar realizando ninguna de las anteriores.
- **AgentType** distingue los personajes entre estudiantes, profesores u otro personal de la facultad.
- **Simulation** informa sobre si un agente sigue la rutina habitual, un simulacro de incendio, una enfermedad contagiosa o una infección zombie.
- **Personality** simplemente diferencia entre personajes muy puntuales, impuntuales, con puntualidad estándar o muy variables.

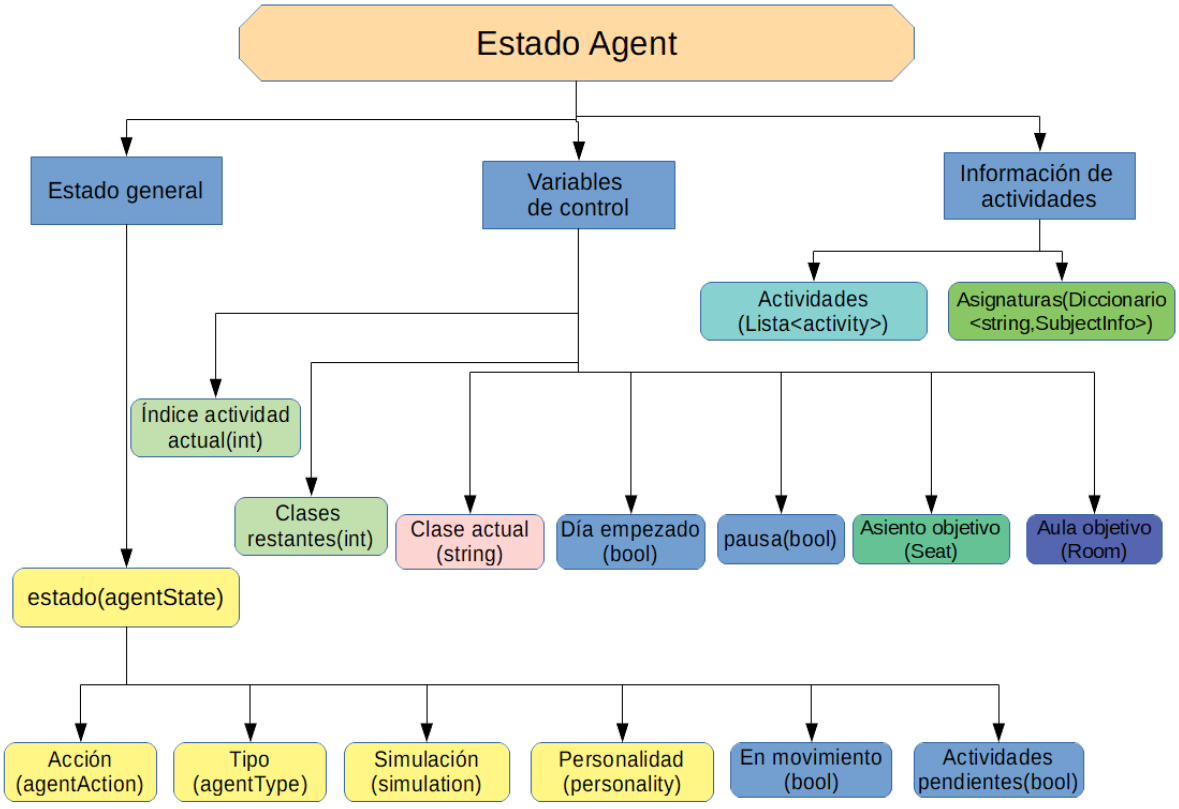


Figura 15: Diagrama del estado de Agent

Como se puede observar en la figura 15 hay muchas más variables que se encargan de controlar el estado de una gente. Las variables que no se encuentran en agentState se pueden dividir en dos categorías: variables de control y información de actividades.

Las variables de control, como su nombre indica, se usan para el manejo interno del componente pero no sirven para representar el estado actual del personaje de la simulación. Algunos datos que se encuentran en esta categoría son: el nombre de la asignatura que está realizando, la habitación y asiento en el que se encuentra o se está dirigiendo y el número de asignaturas restantes a las que debe acudir durante el día actual.

Finalmente, cada agente guarda una lista con la información de sus actividades y un diccionario con el nombre y información de cada asignatura que cursa. Estas variables no se modifican durante toda la ejecución del programa y representan el conocimiento del agente sobre su horario de clases.

#### 4.3.6.2. Receptores

Para que un agente pueda interactuar con el mundo primero debe conocerlo y, por eso, necesita los **receptores**.¿Cómo reciben los agentes esta información?

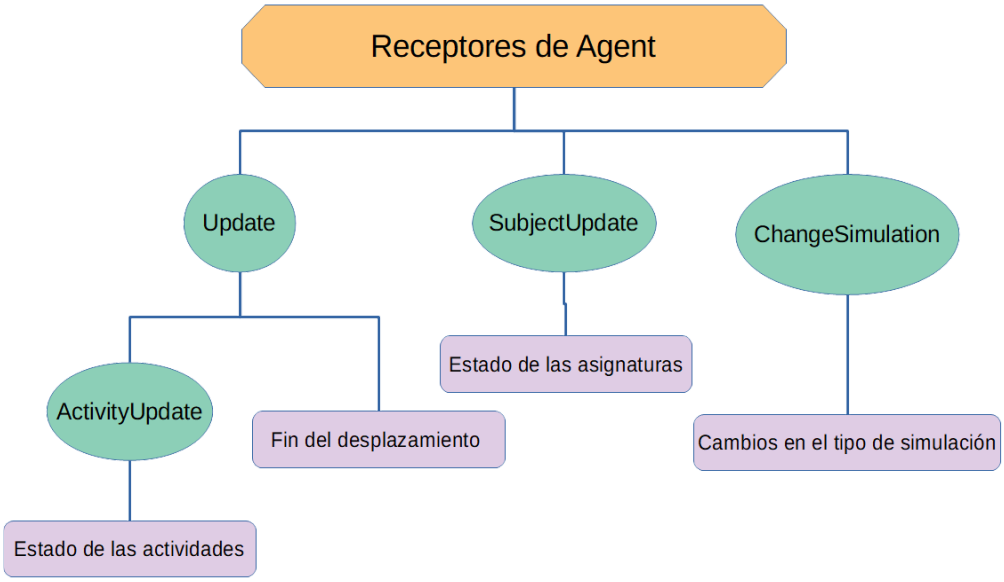


Figura 16: Diagrama de los receptores de Agent

En el componente Agent la recogida de información está bastante unida al razonamiento ya que, al no ser muy extensa, se suele realizar en un mismo método donde primero se capta el estado y, a continuación, se razona la respuesta. Los métodos donde se recoge información del entorno son: ActivityUpdate,

SubjectUpdate, ChangeSimulation y en el propio Update. En la figura 16 se muestra un diagrama que resume la información que recibe cada método.

**ActivityUpdate** se llama en cada vuelta del bucle principal y se encarga de detectar cambios relacionados con las actividades que no forman parte de una asignatura. Primero comprueba si ya está realizando alguna actividad y, si es así, tiene que comparar la hora actual con la hora de finalización para detectar si ya termina, sin embargo, si no está realizando ninguna actividad tendrá que comprobar si ya es hora de empezar alguna de sus actividades previstas para ese día. El cambio de estado detectado en las actividades determinará la respuesta del agente.

**SubjectUpdate** realiza una función equivalente al ActivityUpdate pero, a diferencia de éste, actualiza el estado relacionado con sus asignaturas. Otra diferencia fundamental es que no se llama a través del Update del agente sino que se llama desde el SimulationManager.

Con el motivo de centralizar la gestión de los horarios de las asignaturas y evitar que todos los agentes que atienden a una misma asignatura tengan que comprobar constantemente la misma condición, el Simulation Manager se encargará de que solo se compruebe el cambio de estado de una asignatura una única vez por vuelta del bucle y, solamente cuando haya cambios en su estado, informará a todos los participantes llamando a su SubjectUpdate.

Como se ha explicado en la implementación del SimulationManager, la actualización del estado de las asignaturas se realiza dentro de la clase Subject que realiza una comparación simple entre la hora actual y las horas de inicio y final de la clase y comprueba si el intervalo en el que se encontraba antes es distinto al nuevo. Cuando Subject detecta un cambio se actualiza el estado de la asignatura y informa únicamente a los agentes guardados en su lista de participantes.

A parte de cambios en sus actividades y asignaturas, los agentes también pueden obtener otra información del mundo que les rodea. Pueden recibir órdenes desde los gestores o de otros agentes para cambiar su estado a través del método **ChangeSimulation**. Un ejemplo práctico sería si en una simulación de incendio un agente que ha detectado el incendio informa a otro que lo desconocía y éste tiene que cambiar su estado para reaccionar al incendio.

Para terminar, los agentes también perciben los cambios en el estado del componente NavMeshAgent : en el método **Update** consultan si ya se ha alcanzado el destino establecido y, si es así, razonan una respuesta.

#### 4.3.6.3. Razonamiento

Luego de recibir los cambios en el entorno los agentes deben usar esa información junto con su estado interno para realizar un paso más antes de actuar: **razonar la respuesta**. Ahora veremos cómo relacionan toda la información obtenida para poder efectuar una acción acorde.

En principio, los agentes realizan un razonamiento por vuelta del bucle principal, esto ocurre a través del método de Unity Update. La primera comprobación que realizan antes de computar una respuesta es la del tipo de simulación que está siguiendo el propio agente ya que determina en gran medida el razonamiento posterior. Esto es así porque es muy posible que, si un agente no se encuentra en su estado de simulación estándar, ignore todo cambio relacionado con su horario de clases habitual ya que podría estar evacuando durante un incendio u otras situaciones. Por lo tanto, el estado de simulación guardado en el agente es lo que determinará en mayor medida el tipo de comportamiento de un cada agente. En la figura 17 se puede observar el flujo de ejecución que sigue el agente para realizar todo este proceso.

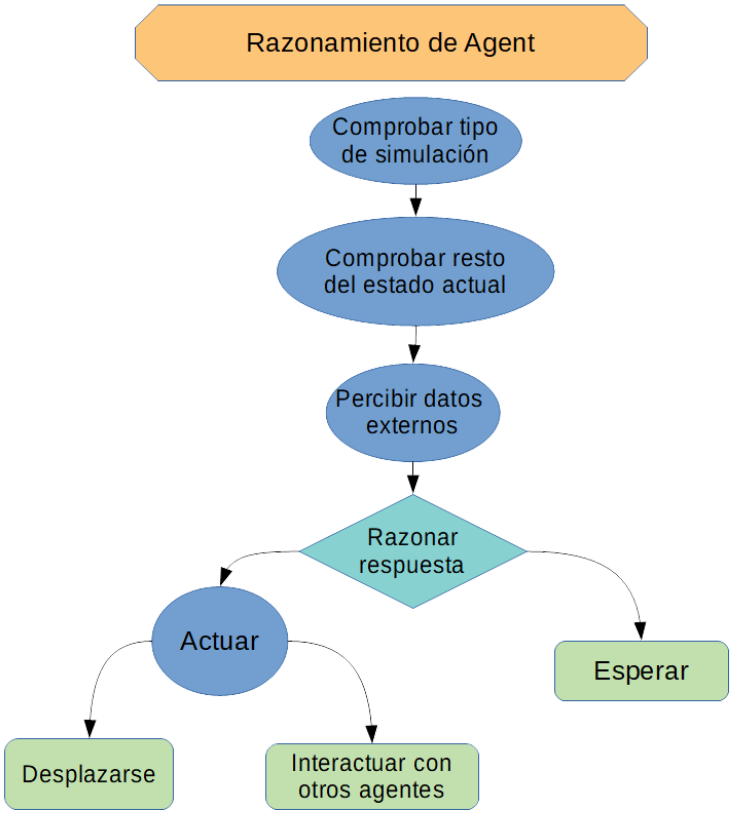


Figura 17: Diagrama del razonamiento de Agent

Es en el método **SimulationUpdate** donde ocurre el razonamiento relacionado con el tipo de simulación que se encuentra un agente por lo que es el responsable de la mayor parte del comportamiento de los personajes. Se llama a este método en el

Update de Unity y se encarga de ejecutar el código correspondiente al estado en el que se encuentra el agente en este momento. Las simulaciones más complejas usan métodos adicionales para su funcionamiento pero la mayoría se llaman desde aquí. A continuación, se describe todo el razonamiento que realizan los agentes dividido entre los cuatro eventos posibles que pueden simular: estándar, incendio, enfermedad y zombie. Si no se especifica lo contrario, el razonamiento se realiza dentro del propio SimulationUpdate.

#### 4.3.6.3.1. Simulación estándar (regular)

Durante la mayor parte del tiempo, los agentes siguen la simulación **estándar**, en este estado su objetivo principal es cumplir los horarios de clases por lo que su razonamiento les servirá principalmente para saber dónde y cuándo deben desplazarse. En general, en este estado no tienen que razonar hasta que sus receptores perciban alguna de las siguientes señales: un cambio de estado de una actividad o asignatura, la llegada al destino del desplazamiento o un cambio en el tipo de simulación.

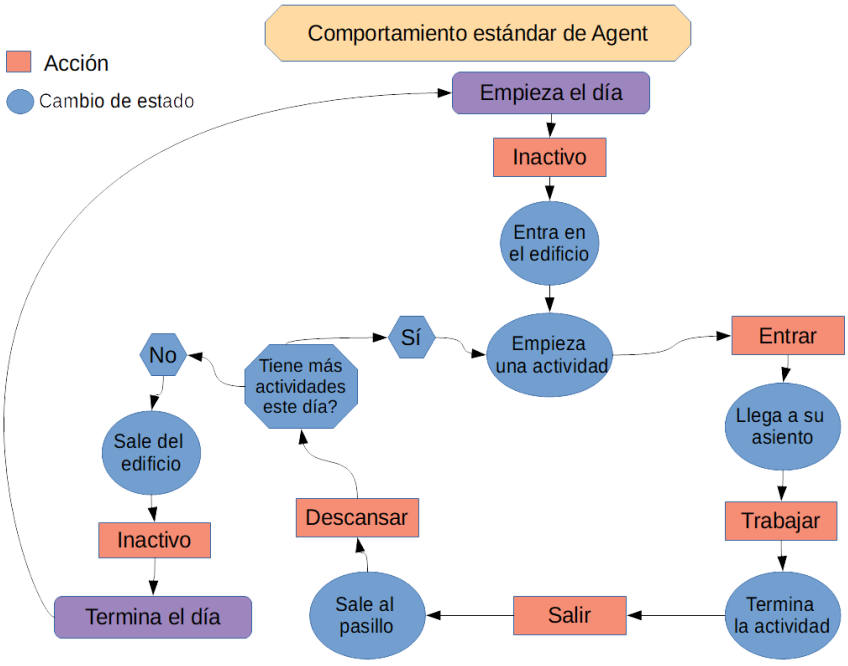


Figura 18: Diagrama del comportamiento estándar de Agent

En la figura 18 se muestra cómo funciona el comportamiento estándar de los agentes, en este se puede observar el orden en el que se realizan las distintas acciones posibles del agente y la información que necesita recibir para pasar al siguiente. A continuación se entrará en detalle en la implementación de todos los elementos de este diagrama.

El método **ActivityUpdate** es el que se encarga de procesar todos los cambios en las actividades de un agente. Este método se llama desde SimulationUpdate y también se ocupa de recibir información externa necesaria para el razonamiento.

Este método funciona de la siguiente forma: primero comprueba si hay alguna actividad en proceso, si es así, solo tiene que comprobar la finalización porque los agentes no pueden empezar una actividad distinta hasta que termine la actual. Cuando el receptor detecta el final de una actividad cambia su estado para indicar que ya no está ocupado y que está saliendo de un aula y, a partir de este momento, ya puede empezar otra actividad. Luego de actualizar su estado le toca decidir su nuevo destino. Con este objetivo se ha implementado una función que permite a todo agente conocer la posición libre más cercana a la salida de su aula actual, para obtener esta información interactúa con el LayoutManager ya que es este quién tiene acceso a toda la información del escenario.

Mientras no se encuentra realizando ninguna actividad el ActivityUpdate comprueba si llega la hora de inicio alguna de sus actividades pendientes del día, cuando esto ocurre, cambia su estado para indicar que se encuentra realizando una actividad y que está de camino a una habitación concreta y, al igual que antes, fija un nuevo destino al que desplazarse. En este caso, tiene que obtener un asiento vacío del aula indicada por la información que tiene sobre la actividad. Este asiento lo obtiene a través de la clase Room, la representación lógica del aula, ya que tiene un método que permite obtener una referencia a un sitio vacío. Luego de obtener su ubicación deseada ya puede empezar a desplazarse.

Los agentes guardan un índice que indica qué actividad de su lista está activa actualmente. Para mejorar el rendimiento también incluyen una variable booleana que indica cuando un agente ha completado todas las actividades del día actual, gracias a esto, puede evitar seguir comprobando si empieza alguna actividad cuando ya no tiene ninguna actividad pendiente.

El método **SubjectUpdate** es bastante similar al ActivityUpdate pero siendo específicamente para los cambios en las asignaturas. Otra diferencia fundamental es que las comprobaciones temporales ya se realizan dentro de la clase Subject por lo que solo se llama a este método cuando hay un cambio de estado en una de sus asignaturas. Cuando se llama a este método el agente debe realizar un razonamiento adicional al que realiza normalmente una vez por llamada en el método Update.

Una clase de una asignatura no se interpreta igual que una actividad ya que presentan las siguientes diferencias:

- Las asignaturas tienen prioridad sobre las actividades, por lo tanto, un agente no empezará una actividad mientras está atendiendo a una clase pero puede abandonar una actividad para empezar una clase.

- Las actividades solo están activas o inactivas, en cambio, las asignaturas contemplan cuatro diferentes estados ordenados en el tiempo: empiezan con el estado *start*, luego pasan a *active*, siguen con *end* y terminan con *inactive*.

Cada vez que se llama al SubjectUpdate el agente recibe el nombre, habitación y estado actual de la asignatura, por lo tanto, la respuesta del agente se podrá clasificar en cuatro categorías respectivas a los estados posibles de una asignatura.

- ❖ Inicialmente el agente recibe el mensaje **start** de una de sus asignaturas y, si no está ocupado en otra tarea, se dirigirá a la clase designada. Para encontrar su destino seguirá el mismo proceso usado con las actividades y obtendrá una silla vacía del aula indicada por la información de la asignatura.

- ❖ A continuación, llega el mensaje con el estado **active**, esto indica que la clase ya ha empezado. Si no está ocupado con otra asignatura o está de camino a ésta el personaje tendrá que acudir al aula designada igual que con el mensaje anterior. Este mensaje sirve para asegurarse que todos los agentes han recibido correctamente la información sobre el inicio o por si alguno estaba ocupado con otra tarea y había ignorado el primer aviso. En este caso, el agente puede interrumpir una actividad para poder atender a esa clase.

- ❖ En el siguiente paso la asignatura ya notifica a los agentes sobre el final de clase, con el estado **end**, por lo que todos los personajes que atendían tendrán que abandonar la sala. Cada agente seguirá el mismo proceso usado al terminar una actividad: se dirigirá al espacio vacío más cercano en el pasillo.

- ❖ Para terminar, hay un último aviso para asegurarse que ningún agente se quede asignado a esa clase y se indica que la clase ya ha terminado con el estado **inactive**.

Cada agente guarda el nombre de la asignatura que atiende actualmente y la variable **agentAction** que indica la acción que realiza el agente en este momento. Estas dos variables son las que condicionan principalmente la respuesta del agente ante los cambios de sus asignaturas. Si un agente está entrando o trabajando en una asignatura ignorará las llamadas de cualquier otra asignatura hasta que no haya empezado a salir de clase. Cuando un agente debe actuar ante una llamada a SubjectUpdate o ActivityUpdate cambia el tipo de acción que realiza.

Para poderse desplazar y conocer su posición el componente Agent guarda el sitio en el que se encuentra (Seat) o se dirige junto con la habitación designada (Room).

Relacionado con esto, el tipo de un agente puede determinar la posición que se le asigna en una habitación, esto se debe a que a los agentes del tipo alumno se le asigna un asiento estándar pero si está marcado como profesor recibe la posición de un asiento marcado para profesores. De esta forma, aunque dos agentes vayan al mismo aula pueden tener posiciones muy diferentes.

Otro tipo de razonamiento importante que falta por explicar es el encargado de controlar la entrada y salida de un agente del edificio. Cuando empieza un nuevo día todos los personajes reinician su estado y se colocan fuera del escenario. El método StartDay es el que se encarga de esto además de contar el número de asignaturas a las que tiene que acudir ese día. El agente guardará este número que servirá posteriormente para computar la salida. Es el SimulationManager el que detecta el cambio de día y lo notifica a todos los agentes de la simulación.

Un personaje no entra en el edificio hasta que tiene que realizar una actividad o una clase y, cuando esto pase, aparecerá por una de las entradas disponibles. Un método llamado SetUp comprueba si el personaje aún no ha entrado en escena y, si es así, se encarga de pedirle una entrada al LayoutManager para posicionarlo ahí y empezar a dirigirse a su destino.

Se ha explicado cómo se inicia un desplazamiento por el escenario pero falta tratar como se controla la llegada al destino de un agente. Los receptores comprueban en el Update, independientemente de la simulación actual, si el agente ha terminado su movimiento. Cuando recibe este cambio tendrá que modificar su estado para indicar que ya no se está desplazando y para cambiar la acción que realiza. El nuevo estado dependerá de la acción que realizaba anteriormente: si un agente está entrando en una habitación se pondrá a trabajar, en cambio, si estaba saliendo de alguna habitación se pondrá a descansar.

Entrando ahora en el tema de la salida del edificio, los agentes abandonan el escenario cuando ya no tienen ninguna tarea pendiente, tanto actividad como asignatura. Esto se comprueba en el propio Update de Unity independientemente de lo que esté simulando actualmente el agente. Para saber si ya no tiene que realizar ninguna actividad se usa el booleano que indica si tiene alguna actividad pendiente, a esta variable se le asigna el valor de falso la primera vez que en el ActivityUpdate se da cuenta que ninguna actividad empieza más tarde de la hora actual. Por otro lado, para saber si no debe acudir a más clases se usa la variable asignada al iniciar el día que indica el número de clases restantes. Cada vez que termina una clase ese número disminuye.

Por lo tanto, cuando el valor que indica si aún quedan actividades es falso y el número de clases restantes es zero el agente se dispone a abandonar el edificio.

El desplazamiento lo realiza otra vez pidiendo una salida y luego se dirige a ella. Cuando colisiona con un objeto que está identificado como salida y el estado del agente indica que está intentado salir, este abandonará el escenario. El método llamado EndDay configurará al agente de la manera adecuada para que termine el día.

#### 4.3.6.3.2. Simulación enfermedad (infection)

La simulación enfocada al contagio de una **enfermedad** se ha implementado de tal forma que pueda simular aproximadamente una situación real de brote de una enfermedad infecciosa como podría ser el coronavirus. En este modo de simulación los personajes actuarán de forma estándar, siguiendo su horario de clases con normalidad, con la diferencia que pueden contagiar a las personas cercanas. Este evento se inicia por parte del usuario a través del input de teclado creando casos activos en el personaje seleccionado o, si no hay agente seleccionado, se eligen unos cuantos personajes aleatorios de la simulación para ser infectados. A partir de ese momento, la enfermedad ya podrá empezar a expandirse por la facultad.

Hay dos tipos de infectados: los casos activos con la enfermedad desarrollada que actúan como el foco de los contagios y los nuevos contagiados, que acaban de entrar en contacto con el virus y aún no se ha manifestado en su cuerpo. Los primeros solo pueden ser creados por el usuario ya que todos los contagios nuevos que ocurran en la facultad solo crearán casos pasivos sin capacidad de contagio directa. De esta forma, se representa mejor la situación real ya que estas enfermedades tardan un tiempo en empezar a manifestarse. Para identificar el tipo de simulación de un agente se usan diferentes colores en su modelo pero, en este caso, los casos activos y pasivos también tendrán diferentes colores para una mejor identificación visual. Los dos tipos se encuentran en el mismo tipo de simulación pero una variable booleana adicional se encarga de diferenciar los dos estados.

El contagio se puede realizar de dos maneras: por cercanía directa con el infectado o por entrar en contacto con las partículas infecciosas que libera. La primera se ha implementado comprobando cada un periodo de tiempo todos los agentes que se encuentran a menos de una distancia determinada del infectado y, por cada uno de estos, habrá una probabilidad de que se infecte. Los valores de tiempo entre cada comprobación, el rango y la probabilidad del contagio son configurables desde Unity para poder adaptar fácilmente enfermedades con diferentes niveles de transmisión.

El segundo tipo de contagio se ha implementado con un nuevo componente que simula la presencia de esas partículas infecciosas llamado InfectiousRemains. Este componente hará algo bastante similar a los personajes infectados ya que cada un tiempo tiene la posibilidad de contagiar a los personajes que se acerquen

demasiado. Estos restos por lo general tienen una menor área que los personajes infectados además de un tiempo de vida limitado. Cuando un personaje infectado activo abandone un sitio en una clase o pasillo siempre creará una entidad con el componente InfectiousRemains que podrá contagiar a cualquier otro personaje que se acerque hasta que finalice su duración.

Para hacer aún más útil esta simulación el usuario tiene la capacidad de activar el protocolo de seguridad contra la transmisión de enfermedades en la facultad en cualquier momento. Cuando esta función está activa, se simula que cada personaje lleva mascarilla, se desinfecta contínuamente todo el material de la FDI y se mantiene la distancia de seguridad. Para representar esto, el rango y posibilidad de contagio se reducen considerablemente y los casos activos dejan de liberar partículas contagiosas en el entorno. Además se dejará un asiento de separación libre en todas las aulas. Gracias a la capacidad de alternar entre estos dos estados se pueden realizar comparaciones del número y de la velocidad de contagios con o sin el protocolo. La figura 19 muestra un diagrama del comportamiento que añade esta simulación enfocado a los tipos de agentes enfermos.

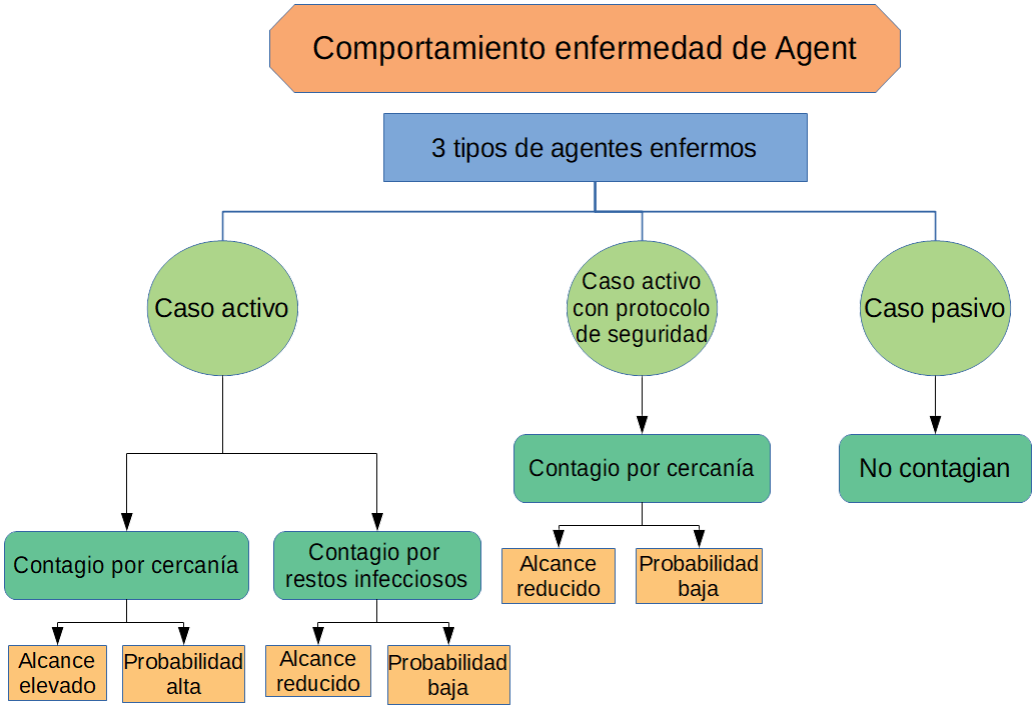


Figura 19: Diagrama del comportamiento enfermedad de Agent

El componente llamado InfectionSimulationInfo se encuentra en la entidad de gestión donde se encuentran el resto de gestores de esta simulación y sirve para configurar el funcionamiento de la simulación de la infección. Contiene tanto las variables del rango, tiempo y posibilidad de contagio como un booleano que indica si están activados los protocolos de seguridad. Todos estos valores se pueden editar

desde el editor de Unity para que, durante la ejecución, los agentes infectados acceden a esa información a través del SimulationManager.

#### 4.3.6.3.3. Simulación incendio (fire)

Otra simulación de interés es la de una **situación de incendio** ya que se puede usar para detectar fallos y mejoras en el protocolo de evacuación usado. Esta simulación también la puede activar el usuario pero su funcionamiento es algo distinto: se pueden activar las alarmas de incendio del edificio para que todos los personajes cercanos a una alarma cambien su estado para afrontar el incendio. Todos los agentes que perciban el incendio procederán a dirigirse a las salidas del edificio de manera controlada avisando a cualquier otra persona que se encuentren en el camino y no se haya enterado del incendio. En la figura 20 se muestra el comportamiento en este estado en forma de esquema.

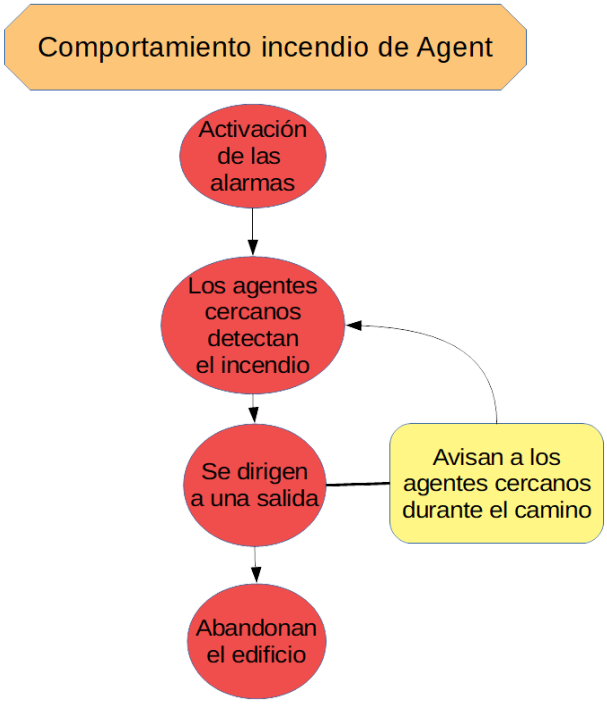


Figura 20: Diagrama del comportamiento incendio de Agent

Cuando un personaje sigue esta simulación ignora totalmente su horario de asignaturas y actividades hasta que abandona el edificio. Posteriormente, al empezar un nuevo día todos los agentes volverán a su estado habitual y seguirán su horario con normalidad. Este tipo de simulación usa el mismo mecanismo usado en la simulación estándar para que los personajes salgan del edificio, es decir, el agente pide al LayoutManger una salida y usa su posición como destino de su movimiento, cuando colisione con esta salida abandonará el edificio.

#### 4.3.6.3.4. Simulación zombie (zombie)

Finalmente, se ha implementado un último evento en la simulación diseñado para demostrar el potencial de la simulación además de probar ideas más creativas. Se trata de lo que se suele denominar como **plaga zombie**. Este evento se activa de la misma forma que la enfermedad: el usuario usa una tecla que transforma al personaje seleccionado en zombie o, si no se ha seleccionado ninguno, se transforman varios personajes elegidos de manera aleatoria.

Cuando un agente se encuentra en este estado deambula por el edificio en busca de personas para poder transformarlo en zombies. Gracias al aumento de su velocidad de desplazamiento, en este estado pueden perseguir y convertir eficazmente a los otros personajes. Cualquier agente cercano a un zombie intentará huir y abandonar el edificio. Como es obvio, tanto los agentes que huyen como los que persiguen ignoran cualquier horario mientras se encuentren en estos estados.

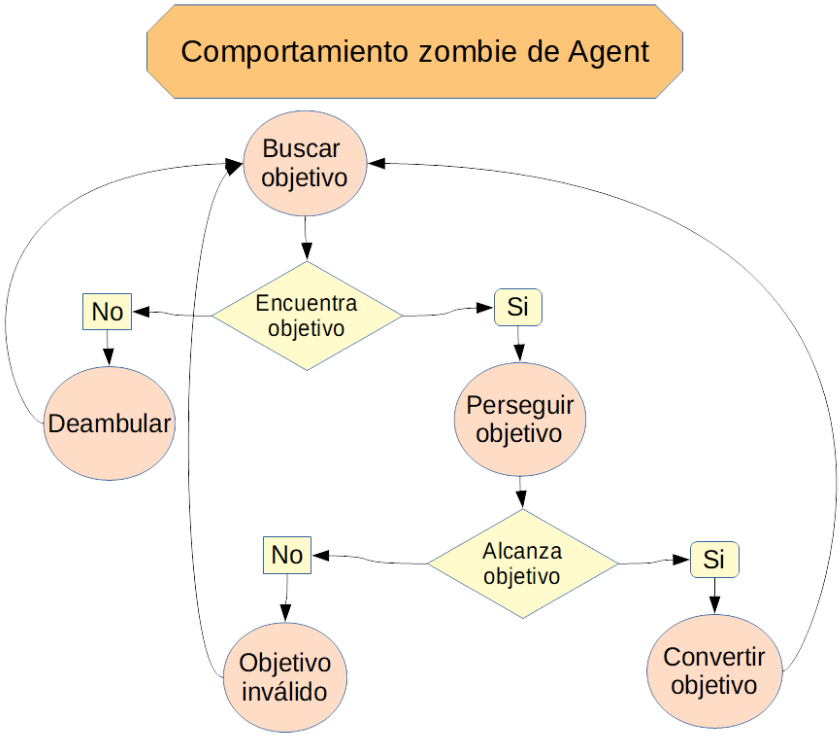


Figura 21: Diagrama del comportamiento zombie de Agent

El comportamiento de los zombies presenta un modelo distinto al usado hasta ahora ya que tienen que llegar a un destino que se desplaza constantemente por lo que tiene que actualizar el objetivo dinámicamente. Todos los zombies tienen la capacidad de transformar en zombies a los personajes con los que colisionan. Inicialmente un zombie no tiene objetivo, por lo que empieza buscandolo en sus proximidades. Para conseguir su objetivo buscará entre todos los agentes que se encuentren en un rango determinado y elegirá el que se encuentra a menor

distancia y no sea un zombie. Si no encuentra a ningún objetivo viable se dispondrá a deambular por el escenario mientras sigue buscando manteniendo el mismo estado de búsqueda.

Cuando finalmente encuentra un objetivo guarda la referencia a su posición y pasa a la fase de seguimiento. Durante esta fase en cada vuelta de bucle actualizará la posición de destino a la posición actual del agente que está siguiendo. El agente continuará en este estado hasta que su objetivo se vuelve inválido ya sea porque lo ha transformado en zombie o ha conseguido escapar del edificio. En ese momento volverá a la fase de búsqueda de objetivos. En la figura 21 se puede ver el comportamiento descrito resumido en forma de diagrama.

#### 4.3.6.4. Actuadores

Para terminar de tratar el funcionamiento de los agentes falta explicar cómo transforman su razonamiento en acciones a través de sus **actuadores**. Los agentes realizan principalmente dos tipos de acciones: desplazarse por el escenario y comunicarse con otros agentes. En la figura 22 se muestra la clasificación de todas las formas de actuar de Agent, a continuación, se describe su implementación y funcionamiento en más en detalle.

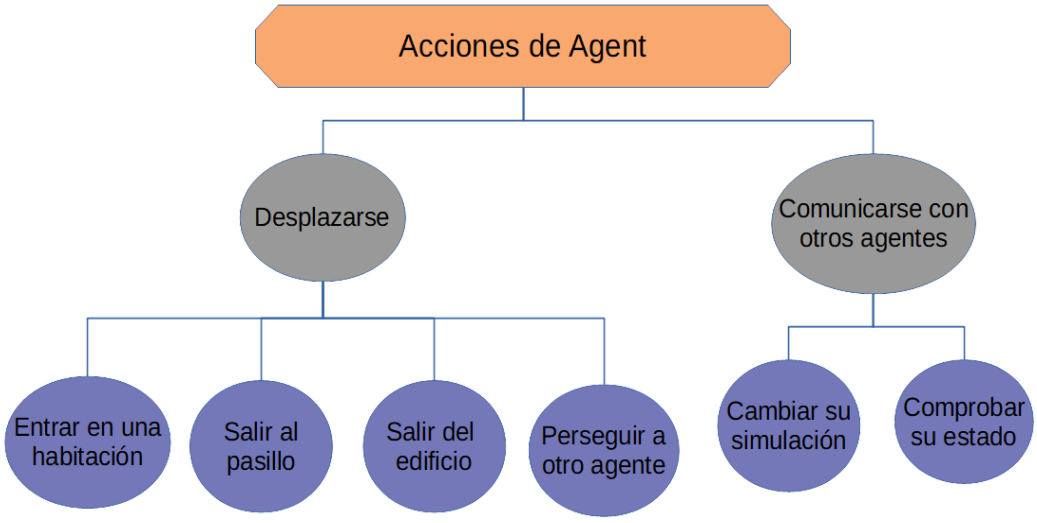


Figura 22: Diagrama de las acciones de Agent

El **movimiento** se ha implementado usando el componente **NavMeshAgent** desde el propio componente Agent. Pero, para conseguir una simulación más realista, no se usan directamente sus funciones sino que se hace a través de otros métodos que permiten conseguir una mayor versatilidad.

Cuando un agente tiene que desplazarse siempre llama al método MoveToDestination o MoveToRandomExit. El primero se usa en todos los casos de

desplazamiento menos cuando abandona el edificio, que se usa el segundo. En los dos métodos se cambia el estado del agente para indicar que se está desplazando y se llama a la función SetDestination del componente de navegación. Los métodos difieren internamente porque el primero usa una posición de destino que se configura durante el razonamiento, en cambio, el segundo no necesita configuración previa y se pide la posición de salida al LayoutManager dentro del propio método. Siempre se llama a alguno de estos métodos cuando un agente se debe desplazar en su simulación estándar, en estados donde debe actuar con prisa, como en un incendio, se llama directamente al SetDestination sin pasar por estos métodos.

Para controlar el movimiento también es importante limpiar los objetivos anteriores para poder empezar uno nuevo. Se ha implementado el método ResetTarget que realiza esta función: devuelve todas las variables usadas para guardar su destino y tarea actual al estado por defecto además de indicar que el sitio que ocupaba vuelve a estar vacío. También hace que el componente de navegación se pare y reinicia el camino que tenía previamente. Se llama a este método antes de realizar un movimiento y así se puede estar seguro que no hay problemas de sincronización, por ejemplo, que una silla siga ocupada luego de que un personaje la abandone o que un agente esté realizando una actividad y una asignatura a la vez.

Siempre que un agente se quiere desplazar durante su comportamiento estándar se le aplicará un tiempo de retardo antes de que este se realice. Este tiempo permite que la simulación sea mucho más realista ya que los personajes no saldrán ni se dirigirán a una clase en el momento exacto en que les llegue el mensaje. Además, el tiempo de retardo es un número aleatorio dentro de un rango predefinido por lo que no todos los agentes se dirigirán a la vez a una clase o actividad, algunos pueden llegar más tarde o temprano dependiendo de este valor. Para que esta característica sea aún más interesante se ha añadido una variable de personalidad en el estado de todos los agentes que determina el rango del valor de retardo, de esta forma, se puede configurar desde el propio editor de Unity si se quiere que un personaje sea más puntual , impuntual, neutral o muy variable. El retardo se aplica usando un método de Unity llamado Invoke que permite llamar a un método con el tiempo de retardo indicado. Esto se usa tanto en los desplazamientos entre aulas como para entrar y salir del edificio.

Hay una última cosa que el agente debe controlar sobre el movimiento y se trata de la velocidad. El componente de navegación guarda una variable que indica la velocidad máxima a la que se desplazará el personaje, siempre hay aceleración cuando se inicia un movimiento y desaceleración cuando se llega al destino. Como esta velocidad puede cambiar en medio de la ejecución ya sea porque el usuario aumenta o disminuye la velocidad de la simulación o el agente pasa a un estado en el que debe ser más veloz, se tiene que manejar adecuadamente para que todos los

agentes siempre usen la misma velocidad base. El SimulationManager es el que guarda la velocidad actual de la simulación y se encarga de que todos los agentes la actualicen cuando esta se modifica. Pero son los agentes los que acaban de manejar su velocidad final ya que en el método que se usa para modificarla llamado UpdateSpeed antes de aplicar el cambio de velocidad pueden modificar ese valor si se encuentra en un evento que lo requiera como sería el caso un zombie.

La **comunicación** entre agentes permite simular comportamientos más interesantes y profundos pudiendo crear situaciones sociales nuevas. Un agente puede tener diversos motivos para comunicarse con otros agentes cercanos pero la forma de conseguir el canal de comunicación es el mismo en todos los casos. Esta capacidad se suele usar en las simulaciones de eventos especiales como en un incendio o infección. El objetivo es que un agente pueda acceder de forma eficaz a todos los agentes que se encuentren dentro de un radio determinado de distancia de este.

Para la implementación de esta característica primero se optó por usar una función de Unity preparada principalmente para estas situaciones pero al final se ha desestimado por su gran impacto en el rendimiento de la aplicación. Este método proyecta una esfera en una posición con un radio determinado y devuelve una lista con todas las colisiones obtenidas. De esta forma es fácil acceder a todos agentes cercanos comprobando qué entidades tienen el componente agente. El problema de este método es que no está preparado para usarse muchas veces en un mismo Update ya que simular la proyección de una esfera y sus colisiones es bastante costoso. Por esta razón, es totalmente ineficaz por la gran cantidad de agentes que pueden estar a la vez comprobando si tienen a otros agentes cercanos. El problema en cuestión se ha solventado aprovechando la referencia a la entidad padre de todos los agentes que se encuentra en el DataManager. En vez de usar la función de Unity, se llama a una función del SimulationManager que se comunica con el DataManager y permite acceder a todos los agentes, a continuación, se calcula la distancia con cada uno. Todos los agentes que se encuentran a una menor distancia que la especificada son sobre los que se interactúa.

Esta comprobación sigue siendo bastante costosa por lo que, para asegurar un buen rendimiento, se ha implementado que cualquier escaneo de este tipo tenga un tiempo de retardo hasta la siguiente comprobación. El tiempo de retardo entre comprobaciones se puede configurar desde el editor y, introduciendo un valor adecuado, se consigue que solo se ejecute dos o tres veces por segundo en vez de sesenta o incluso más si no se hubiera creado esta restricción.

Hay dos tipos de interacciones entre agentes pero la más extendida es la modificar el comportamiento de otro agente cambiando el tipo de simulación que sigue. Esta es la interacción que ocurre cuando en un incendio un agente avisa a otro para que

también evacúe o cuando un personaje enfermo contagia a otro. Esta acción se realiza a través del método **ChangeSimulation**, un método público de Agent donde se especifica la simulación a la que debe cambiar el agente seleccionado. Dentro del método se modifica el estado de acuerdo con el parámetro introducido y, si es necesario, se configura para que el agente pueda empezar a seguir su nuevo comportamiento.

El otro tipo de comunicación que usan los agentes sirve para conocer el estado de otro personaje y actuar en consecuencia. Los agentes que siguen la simulación zombie usan este tipo de interacción. Cuando un zombie encuentra a otro agente que cumple sus requisitos guardará la referencia a su posición para ponerla como el destino de NavMeshAgent en cada vuelta de bucle.

# Capítulo 5 - Resultados

Este programa se puede usar directamente o extenderlo para simular todo tipo de situaciones útiles donde se necesita poder observar el comportamiento de los personajes y las interacciones que ocurren durante el día a día en la facultad. En este apartado se expondrán dos ejemplos sencillos de uso práctico de la aplicación con los resultados que se pueden obtener y una demostración a gran escala probando los límites de la simulación.

Personas con más conocimiento sobre psicología, sociología o otras ciencias sociales podrían aprovechar aún más el potencial del programa ya que pueden identificar mejor las interacciones más relevantes además de realizar un análisis más completo y profundo de todos los datos que se pueden extraer de la simulación y, de esta manera, conseguir unos resultados aún más fiables. En el caso de la simulación del contagio de una enfermedad, un epidemiólogo puede determinar qué configuraciones o modificaciones pueden hacer que esta simulación sea más realista.

Para estas pruebas se han usado horarios basados en los horarios reales de algunos grados universitarios impartidos en la facultad además de usar como hora libre para la comida la misma que se sigue en la realidad (de 13 a 14). Hay alumnos con horarios de mañana, de tarde y de mediodía. En el caso de los docentes tienen configuradas actividades que representan sus horas de tutoría en sus despachos coherentes con su horario de clases. De esta forma, intentamos que la simulación tenga un parecido importante con la situación real.

Los dos primeros ejemplos son a pequeña escala y usan la misma configuración tanto de horarios de clase como de personajes, concretamente, tienen 46 estudiantes y 18 profesores presentes en la simulación que se dividen en 5 horarios de clases diferentes. Cuatro de estos son horarios habituales en un cuatrimestre de un grado universitario, siendo dos de mañana y dos de tarde, y, el último, es un horario cruzado que combina clases por la mañana y la tarde por lo que los alumnos que lo siguen se queda a comer en la facultad. Esta configuración se ha usado frecuentemente durante el final del desarrollo para comprobar que todas las funcionalidades implementadas funcionan adecuadamente.

## 5.1. Simulación de horarios

El primer uso que se le puede dar a este programa es el de simular diferentes horarios, ya sean los actuales o planificados para aplicar en el futuro, para encontrar posibles problemas y defectos que pueden ser difíciles de identificar de otra forma.

Se pueden observar posibles aglomeraciones en los pasillos y accesos al edificio o, el caso contrario, periodos en los que se mueven pocas personas por la facultad y se pueden aprovechar para inicar más clases. También se pueden identificar problemas de aforo en sitios de acceso libre como la cafetería.

Combinando la representación 3D y el registro de eventos en texto que aporta, esta aplicación permite una revisión exhaustiva del funcionamiento de un horario en la FDI. Además, gracias a su capacidad de configuración se puede adaptar a diferentes situaciones fácilmente y alternar entre ellas.


Figura 23: Imagen simulación de horarios 1

En la figura 23 se observa un grupo de estudiantes entrando por la puerta principal de la FDI y dirigiéndose a clase. Los estudiantes y profesores llegan por alguna entrada al edificio antes de que empiece su primera actividad del día. El color del cubo que tienen los personajes a sus pies sirve para identificar la simulación que sigue un agente, aquí se puede ver que son de color azul, color que indica se encuentra siguiendo la simulación estándar.

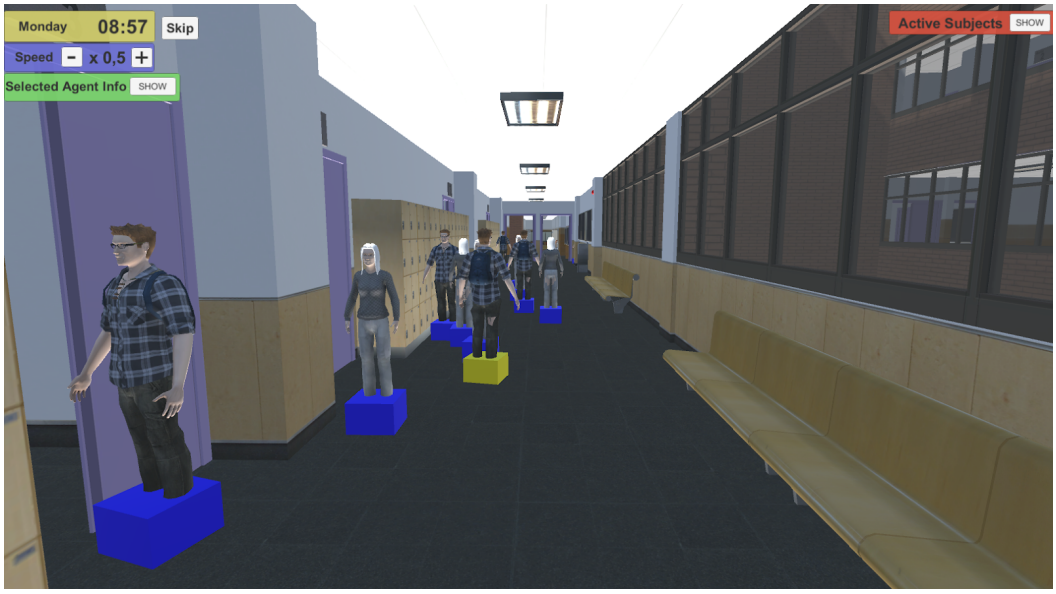


Figura 24: Imagen simulación de horarios 2

En la imagen de la figura 24 se pueden ver a varios alumnos desplazándose hacia diferentes aulas ya que están a punto de empezar sus clases. Se observa como no hay ningún problema logístico ya que sigue habiendo espacio de sobra en el pasillo. También se puede reconocer un agente con el color amarillo, este color indica que ese personaje se trata de un profesor.


Figura 25: Imagen simulación de horarios 3

En la figura 25 se presenta la impartición de una clase con todos los alumnos en sus asientos y el profesor delante. Se puede ver como se respeta la ocupación de sillas, antes de cualquier desplazamiento a un aula el asiento al que un agente se dirige se

marca como ocupado para que no haya ningún tipo de colisión, además, se priorizan siempre los asientos de la parte posterior de la clase.


Figura 26: Imagen simulación de horarios 4

La figura 26 representa una clase en un laboratorio de la FDI con todos los alumnos colocados en los ordenadores. Las clases de laboratorio son de casi dos horas siguiendo el modelo de clases real de la facultad.

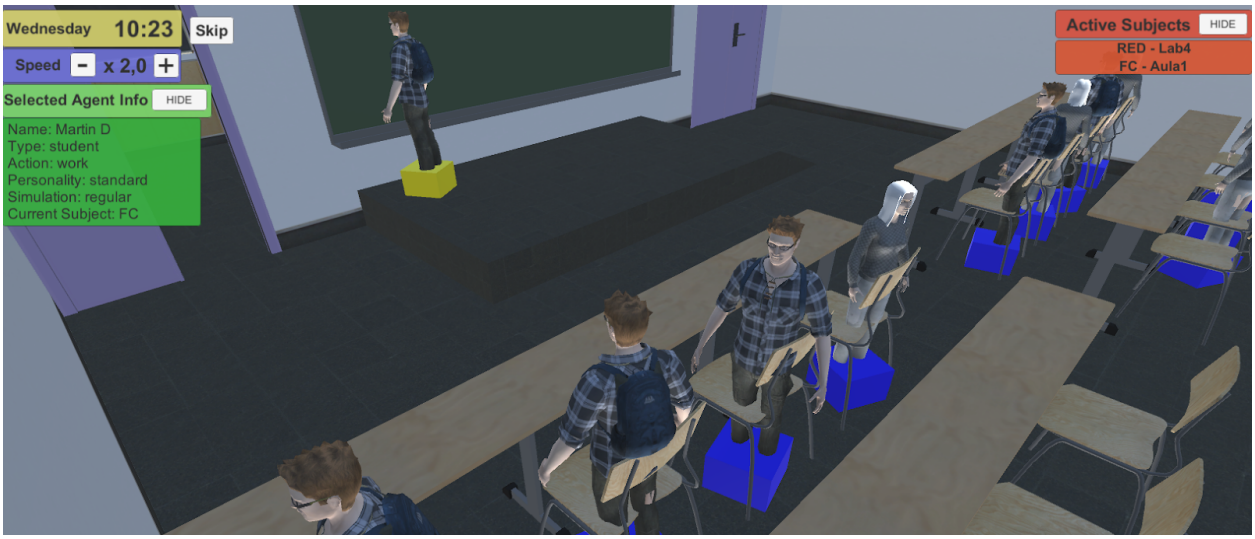


Figura 27: Imagen simulación de horarios 5

En la figura 27 se observa la información representada en la interfaz de la aplicación donde se puede ver información variada sobre el agente seleccionado a la izquierda y, a la derecha de la ventana, se ven los nombres de las clases activas con el nombre del aula en el que se imparten.


Figura 28: Imagen simulación de horarios 6

En la figura 28 se observan personajes de la simulación interactuando en la cafetería de la facultad. Los diferentes agentes comen en distintas horas y pueden tardar más o menos dependiendo de su horario de clases. Dependiendo del número de personajes en la simulación y la configuración del horario pueden haber problemas de aforo en la cafetería como también pasa a veces en la realidad,

## 5.2. Simulación enfermedad infecciosa

Otro caso donde esta aplicación puede ser una herramienta muy útil es un estudio de los efectos de un brote de una enfermedad muy infecciosa en la facultad y ver las diferencias que ocurren si se siguen los protocolos de seguridad contra estas situaciones. Un claro ejemplo donde aplicar este tipo de prueba es en la crisis actual causada por el covid-19 ya que se puede comprobar, a cierto nivel, el verdadero efecto que proporciona la aplicación de los protocolos de seguridad.

En esta prueba se estudiarán los contagios producidos en la facultad durante el transcurso del día empezando con 4 personajes contagiados activos y usando la misma configuración de personajes y horarios que en la demostración anterior. Se hará una primera prueba sin seguir ninguna norma de seguridad y , a continuación, se probará en un nuevo día cumpliendo el protocolo de seguridad. Al finalizar, se compararán los resultados obtenidos en cada ejemplo.

Empezaremos este ejemplo analizando los contagios producidos durante la simulación de la enfermedad sin ningún tipo de seguridad.

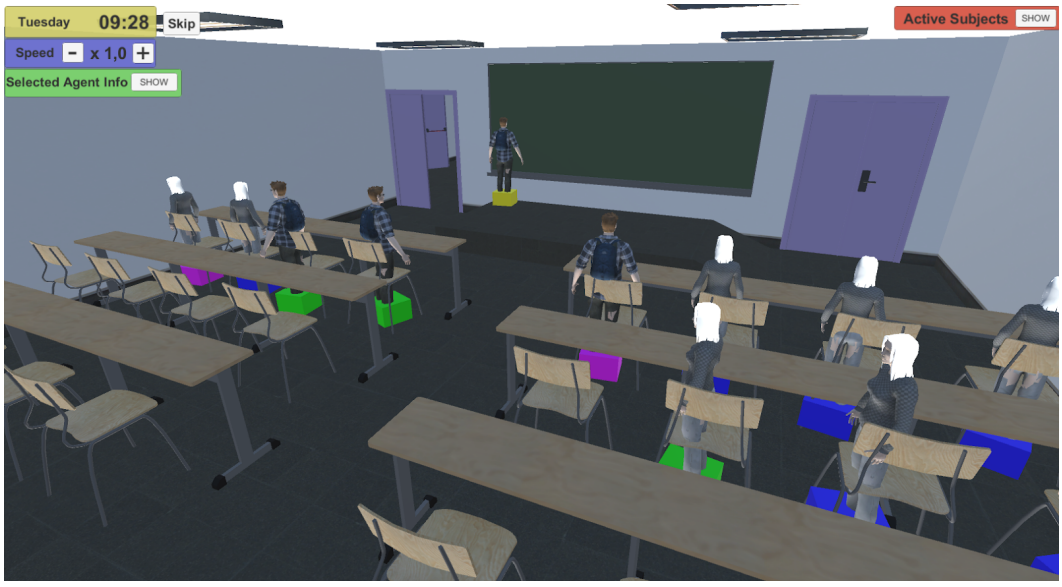


Figura 29: Imagen simulación enfermedad infecciosa 1

En la figura 29 se puede observar la situación en una aula luego de casi media hora de la primera hora de clase del día de estos personajes. Se identifica con un cubo de color magenta en sus pies los personajes que están infectados y pueden transmitir la enfermedad. Un personaje solo se transforma en un caso activo cuando el usuario lo indica a través del input. Los tres personajes con su cubo de color verde han sido infectados durante el tiempo transcurrido en la simulación por los dos casos originales. Con esta primera imagen ya se puede empezar a ver la gran facilidad de contagio de esta enfermedad.

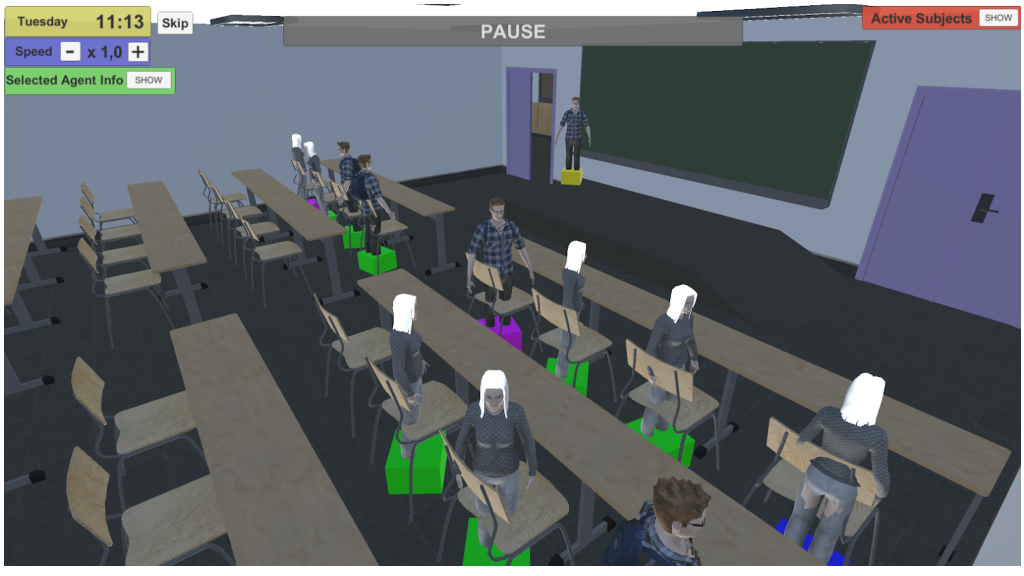


Figura 30: Imagen simulación enfermedad infecciosa 2

En la figura 30 se puede observar, luego de más de una hora, al mismo grupo de alumnos atendiendo a otra asignatura y se puede ver cómo sigue aumentando el número de nuevos casos a gran velocidad.

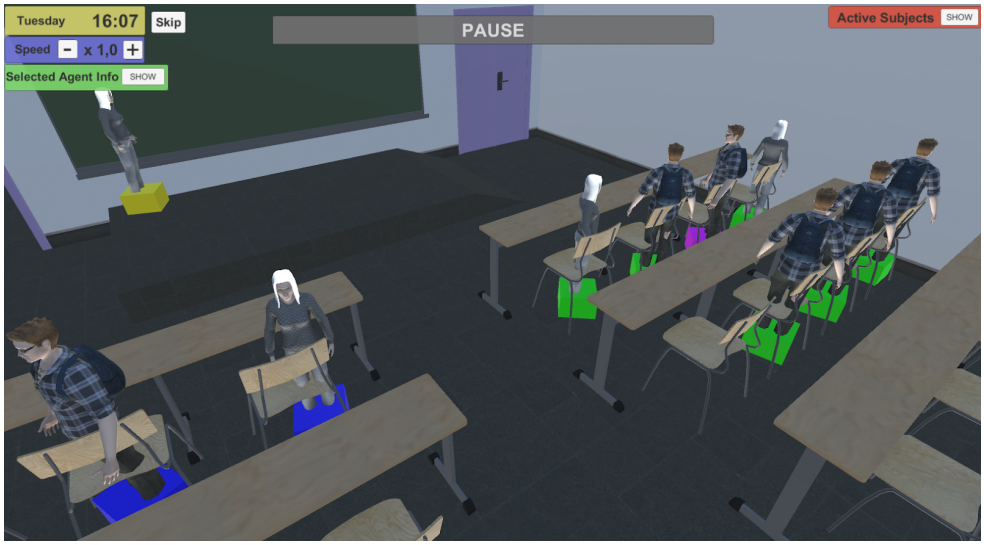


Figura 31: Imagen simulación enfermedad infecciosa 3


Figura 32: Imagen simulación enfermedad infecciosa 4

Las figuras 31 y 32 se sitúan en el mismo día que las anteriores pero, las dos, son de grupos de alumnos distintos que tienen clase por la tarde. Las imágenes en cuestión se han tomado cerca del final de la clase y, en ambas, se puede observar un elevado número de contagios. Analizando conjuntamente todas estas imágenes se puede empezar a deducir que los personajes cercanos a los casos activos tienen una posibilidad muy elevada de ser contagiados aunque también se pueden infectar otros personajes cuando entran en contacto en los pasillos y entradas.

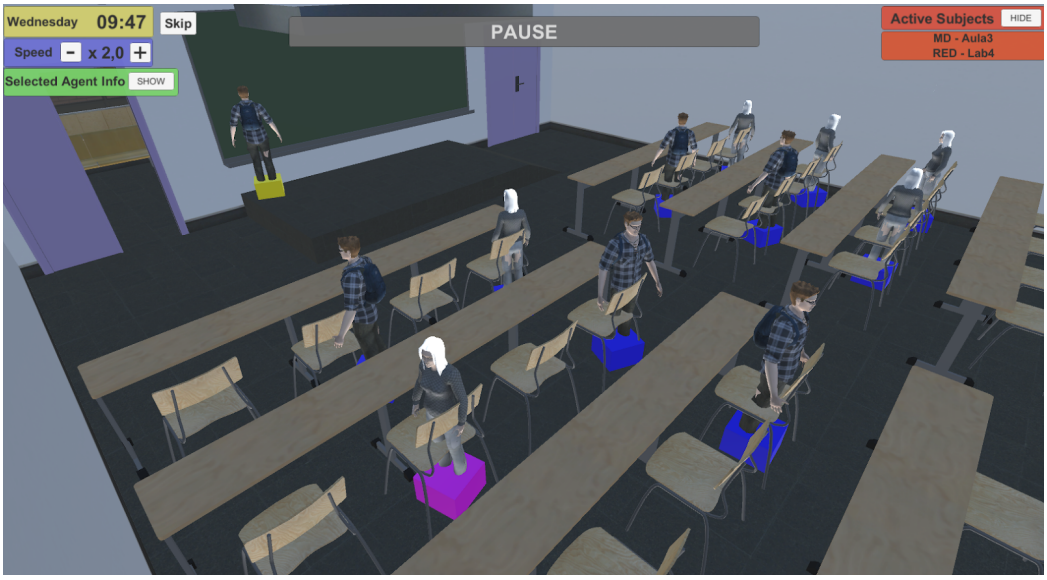


Figura 33: Imagen simulación enfermedad infecciosa 5

La figura 33 muestra un nuevo día en la facultad iniciado con todos los personajes de vuelta a su estado inicial. La simulación se ha configurado exactamente igual que la anterior con la diferencia que, esta vez, se han activado los protocolos de seguridad contra el contagio de enfermedades. En la imagen se puede observar que se dejan asientos vacíos para respetar el distanciamiento social y que en esta clase hay un caso activo iniciado por el usuario. Los infectados iniciales se crean en el inicio del día por lo que, en esta imagen, ya se puede ver eficacia del protocolo ya que, luego de casi una hora de clase, no ha habido ningún nuevo contagio.


Figura 34: Imagen simulación enfermedad infecciosa 6

En la figura 34 se puede observar otro caso similar al anterior pero localizado en una laboratorio de la facultad y con dos casos activos. La imagen se ha tomado

luego de las casi dos horas de clase y no ha ocurrido ningún nuevo contagio por lo que se vuelve a observar un resultado muy positivo del protocolo.

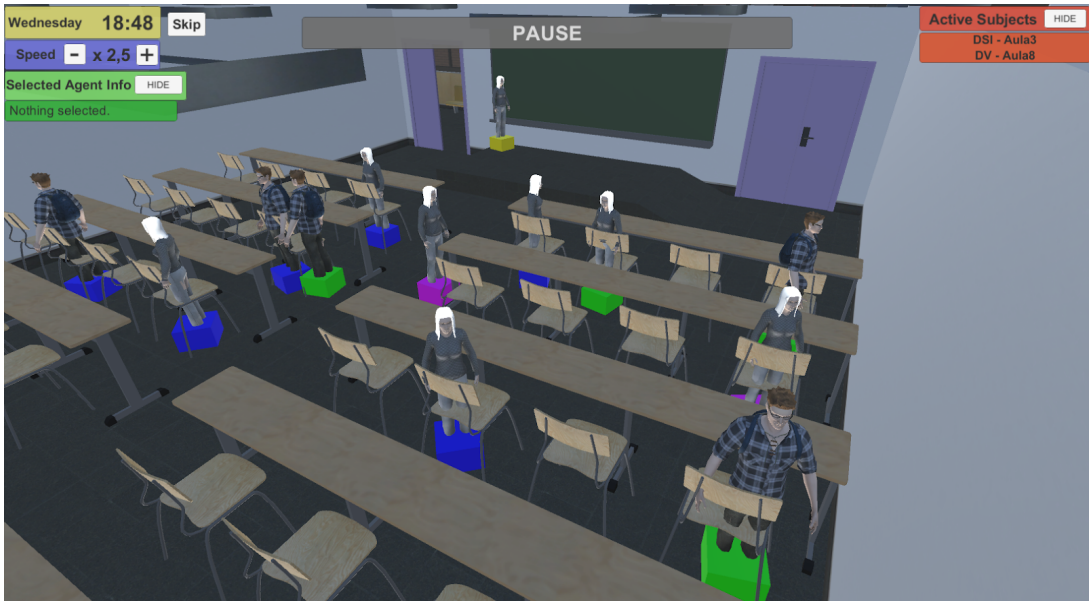


Figura 35: Imagen simulación enfermedad infecciosa 7

En la figura 35 se puede observar una situación bastante distinta a las anteriores ya que han ocurrido cuatro nuevos contagios luego de un día entero de clases en la FDI aunque el protocolo de seguridad esté activado. Usando solamente la información extraída de la representación visual de la simulación se puede deducir que seguir el protocolo de seguridad reduce el número de contagios pero no es infalible y pueden seguir apareciendo nuevos casos.

El registro en texto del programa es una herramienta de gran utilidad en este caso ya que puede ayudar a confirmar o rechazar las posibles conclusiones derivadas de la simulación visual. Gracias a la marca de tiempo añadida a todos los eventos registrados se puede identificar el momento exacto de cada contagio y, de esta forma, sacar nuevos datos y métricas interesantes. El texto siguiente está extraído de los archivos de texto creados por el sistema de registro durante la ejecución de esta prueba.

*PE is a about to begin in Aula3 at 18:01*
*Mercedes R abandoned the faculty at 18:05*
*Ignacio A abandoned the faculty at 18:09*
*Rafael S got infected at 18:17 30 character/s infected by the disease.*
*Jorge P abandoned the faculty at 18:20*

*DV is a about to begin in Aula8 at 18:01*
*Santiago G abandoned the faculty at 18:10 29 character/s in the building.*
*Pablo T abandoned the faculty at 18:19 28 character/s in the building.*

*Javier S got infected at 18:35  9 character/s infected by the disease.*
*Antonio A got infected at 18:45  10 character/s infected by the disease.*
*DSI has just started at 18:47*

En estos fragmentos se pueden observar los registros de distintos eventos ocurridos en la simulación pero, en este caso, el dato que nos interesa es el número de contagios. Lo que se muestra en cada fragmento es el último registro de un contagio en cada uno de los dos archivos de texto. El primer fragmento es de la simulación sin seguir los protocolos de seguridad con 30 contagios totales ocurridos durante todo el día, en cambio, en el segundo fragmento se han registrado solo 10 contagios por lo que se cumple la hipótesis que, como ya era de esperar, seguir el protocolo de seguridad reduce el número de contagios.

Con este pequeño estudio no se pueden sacar conclusiones significativas ya que, para mantener su simplicidad, se ha usado una configuración con pocos personajes, se han realizado pocas simulaciones y el resto de datos usados para su configuración son parcialmente arbitrarios. Pero, siguiendo un enfoque similar y con un trabajo más exhaustivo, se puede llegar a extraer información muy útil sobre diferentes situaciones representables en la facultad. En este mismo ejemplo, si se usa una configuración para los contagios lo más cercana posible a la realidad, se aumenta considerablemente el número de personajes, se realiza un gran número de experimentos y se hace un análisis estadístico completo con los datos extraídos por el programa se podría llegar a conocer los peligros reales de contagio en la facultad y la eficacia de los protocolos de seguridad.

## 5.3. Simulación a gran escala

En los anteriores ejemplos se ha usado una configuración con pocos personajes con el fin de tener un entorno más fácil de manejar pero en la realidad hay muchas más personas que pasan por la FDI a diario. En este apartado se realizará un experimento usando muchos más personajes con el fin de comprobar los límites de la simulación y verificar si puede soportar una configuración mucho más cercana a la real.

Para que todos los nuevos personajes sigan horarios de clases realistas se ha aprovechado el horario usado en los ejemplos anteriores para crear tres nuevos horarios similares pero que usan aulas distintas y con otras pequeñas variaciones. Una forma fácil de entender cómo funcionan estos nuevos horarios es pensar en el primer horario como el horario de clases de cada uno de los cuatro años de un grado universitario y los tres nuevos son diferentes grupos de ese mismo grado. De esta forma, la simulación supera los 400 personajes y ya se acerca bastante más a la situación real. A continuación se explican los resultados obtenidos usando esta configuración.


Figura 36: Imagen simulación a gran escala 1

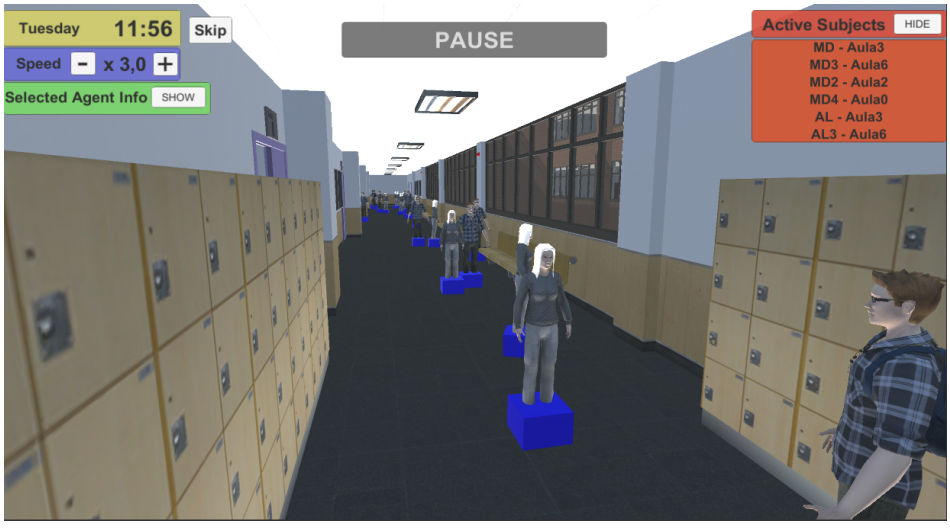


Figura 37: Imagen simulación a gran escala 2

En las figuras 36 y 37 se observa el pasillo de la planta 0 y el de la planta 1, respectivamente, durante el cambio de clase. Las dos imágenes se han tomado en el mismo instante de tiempo de la simulación, de esta forma, se destaca el elevado número de personajes que se mueven y interactúan simultáneamente en el mismo escenario.

El sistema de registro detecta hasta 248 personajes activos al mismo tiempo dentro del edificio y, en todo momento, estos realizan su comportamiento esperado y siguen sus horarios correctamente por lo que se puede afirmar que esta aplicación puede simular configuraciones de horarios y personajes similares a los reales.


Figura 38: Imagen simulación a gran escala 3

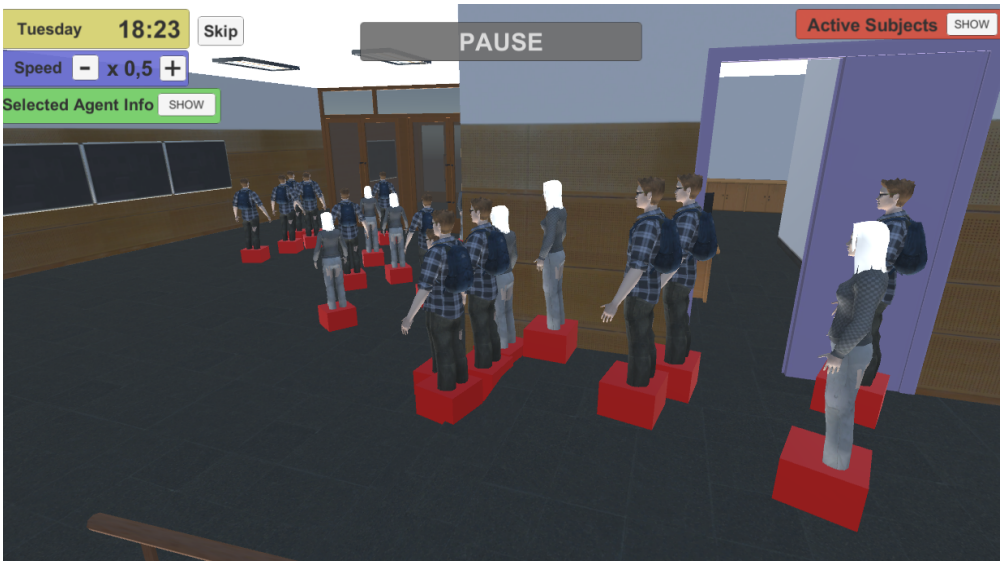


Figura 39: Imagen simulación a gran escala 4

En este experimento también se ha probado el funcionamiento de los eventos especiales disponibles. En la figura 38 se muestra la situación en la cafetería a la hora de comer y se puede ver que está abarrotada de gente. En esta misma imagen también se observan a bastantes personajes infectados. Tras varias pruebas usando esta configuración y como era de esperar, se ha observado que el número de contagios aumenta mucho cuando hay más personajes presentes en la simulación. En el caso concreto de este experimento, se han producido unos 140 contagios en solo medio día con 8 personajes infectados inicialmente.

En la figura 39 se muestran a los personajes abandonando la FDI durante un simulacro de incendio. Con este número de personajes aún no hay muchos problemas durante la evacuación ya que aún hay espacio suficiente en los pasillos y

las escaleras por lo que no se producen grandes atascos. Analizando los resultados de este experimento se puede afirmar que los eventos especiales siguen funcionando de la manera esperada aún con grandes cantidades de personajes.

Con cantidades de personajes iguales o superiores a los usados en este ejemplo, el principal problema que surge no es de la lógica sino que está relacionado con el renderizado gráfico ya que, renderizar a tantos personajes, empieza a tener un coste muy elevado. Por esta razón y como ocurre en muchas aplicaciones 3D, para usar este programa para simular cantidades aún más elevadas de personajes y mantener un buen rendimiento se necesita usar un computador con una tarjeta gráfica bastante potente.

# Capítulo 6 - Conclusiones y trabajo futuro

## 6.1. Conclusiones

En este proyecto se unen dos ramas del conocimiento que se suelen ver como muy distintas y con poca relación como son la informática y las ciencias sociales. Gracias a las grandes capacidades de la inteligencia artificial y el renderizado gráfico se pueden observar y estudiar de formas novedosas conceptos y problemas de gran interés relacionados con el comportamiento social y sus interacciones.

La versión final de la aplicación permite la observación del comportamiento humano habitual en la FDI además de la capacidad de crear y configurar a voluntad nuevas asignaturas, personajes y horarios con relativa facilidad. Asimismo, se pueden simular situaciones y eventos excepcionales como un incendio o brote de una enfermedad contagiosa. El usuario del programa, aparte de poder desplazarse por el escenario, también tiene la capacidad de interactuar con la simulación durante su ejecución modificando su velocidad, desplazándose en el tiempo entre los diferentes días de la semana o activando los eventos especiales. También se registran todos los eventos relevantes en un archivo para poder revisar y analizar posteriormente los resultados de cada simulación.

Siguiendo lo explicado anteriormente, se puede decir que se han conseguido cumplir la mayoría de los objetivos planteados inicialmente, a continuación se revisará el cumplimiento de cada objetivo concreto.

- Crear una simulación social renderizada en 3D de la FDI

Este objetivo es fundamental para el proyecto ya que es necesario completarlo para seguir con el resto de objetivos. Se considera que se ha cumplido de manera adecuada ya que se ha implementado en el programa una simulación que cumple todos los requisitos de una simulación de este tipo: se simula el comportamiento de personajes, estos personajes interactúan entre ellos de forma social, la simulación está representada mediante un modelo 3D y se encuentra localizada en la facultad tanto en el apartado físico, visualmente igual a la facultad, como en el lógico, los personajes actúan siguiendo comportamientos similares a los que ocurren realmente en la FDI.

- Simular un comportamiento realista de los personajes presentes en la simulación

Se ha cumplido una parte importante de este objetivo pero no completamente ya que algunas características que se plantearon inicialmente en este apartado al final no se implementaron con la profundidad deseada. Los personajes de la simulación siguen horarios realistas y un comportamiento general parecido a la realidad pero se podrían haber implementado más interacciones, como que los personajes se paren a hablar entre ellos o vayan al baño, que aumentarían el realismo general de la simulación.

- Permitir simular eventos especiales en esta simulación

Este objetivo se considera superado porque en la aplicación se pueden simular tres eventos distintos en la facultad: incendio, contagio enfermedad infecciosa y plaga zombie. Estos eventos tienen un nivel de complejidad en su simulación bastante dispar, siendo el más simple el incendio y el más complejo el contagio.

A causa de la crisis del covid-19 ocurrida durante el desarrollo del proyecto se optó por desarrollar más la simulación del contagio de una enfermedad ya que tiene una mayor utilidad y interés actual en vez de otras opciones como la simulación de incendio. Inicialmente, la simulación de una enfermedad se había planteado como algo bastante sencillo y genérico pero, finalmente, se ha intentado reproducir con cierto realismo el contagio del coronavirus.

- La simulación tiene que ser fácilmente configurable

Este objetivo también se ha cumplido y con un mayor nivel del planteado inicialmente. Gracias en gran parte a Unity y a la potencia de los ScriptableObjects, una persona con muy pocos conocimientos de programación puede usar esta aplicación para simular una gran variedad de horarios y situaciones en la FDI configurando el mismo los horarios y personajes que aparecerán en cada prueba. Además, un usuario con algunos conocimientos de Unity puede adaptar, sin muchas complicaciones, esta simulación a otros edificios similares en cuanto al tipo de actividades que se realizan en su interior.

- Desarrollar una interfaz para controlar la simulación en ejecución

Se ha completado con creces el último objetivo planeado ya que la aplicación cuenta con una interfaz gráfica durante la ejecución de la simulación que muestra información relevante sobre su estado, como la hora actual y las clases que se están impartiendo, además de permitir que el usuario interactúe con la simulación

usando botones. También se puede interactuar con la simulación a través del uso del ratón y el teclado directamente en la simulación. El programa cuenta también con un menú de inicio y otro desplegable durante la ejecución en los que, aparte de viajar por las dos escenas de la aplicación, se pueden comprobar los atajos de ratón y teclado usados para interactuar con la simulación.

## 6.2. Trabajo futuro

Este proyecto puede servir como punto de entrada para el desarrollo de varias extensiones o aplicaciones similares ya que este tipo de simulaciones tienen una gran capacidad para aumentar su contenido y capacidad. Aquí se recogen algunas ideas que se pusieron sobre la mesa durante la fase de planificación del proyecto y otras que aparecieron ya durante el propio desarrollo de la aplicación. Si se continúa expandiendo este proyecto se puede conseguir una herramienta muy potente que podría ser utilizada para crear horarios reales y protocolos en situaciones de emergencia basándose en los resultados de la simulación.

### 6.2.1. Personalidad de los agentes

Actualmente los agentes simulan una pequeña parte del comportamiento humano en la FDI, centrándose en el desplazamiento por el escenario y realización de tareas. Una mejora que se planteó es la de implementar agentes de mayor complejidad añadiendo una capa de profundidad en su comportamiento. Los personajes podrían tener rasgos específicos que determinarán cómo interactúan con los otros personajes de la simulación, de esta forma, se relacionarían más con agentes con intereses afines y intentarían evitar .

Se implementaría en un componente similar a Agent que se podría llamar AdvancedAgent que extiende el componente base y añade nuevas capacidades a estos agentes para poder interactuar con el resto de personajes. De esta forma se consigue que la simulación se acerque más al día a día real en la FDI y permite el estudio de otros sucesos sociales.

### 6.2.2. Generación de horarios

El proyecto está diseñado y preparado para que un usuario pueda abrir el proyecto en Unity y, sin tocar ninguna línea de código, pueda crear nuevos personajes y asignaturas por lo que puede generar un horario de clases completamente nuevo.
Pero esta capacidad está limitada ya que solo se puede usar el editor de Unity para crear y configurar individualmente cada nuevo personaje o asignatura.

Una extensión muy útil sería tener la capacidad de leer un horario generado con otro programa que se use normalmente para esta tarea y poder aplicarlo directamente a la simulación, de esta forma, se podrían probar diferentes horarios sin tener que configurar absolutamente nada, ni siquiera sería necesario abrir el editor. Un componente nuevo en Unity que lee un archivo con un formato concreto de datos donde se encuentra el horario y lo carga en memoria al iniciar el programa sería una buena mejora sin un coste de desarrollo muy elevado.

Para mejorar la generación de horarios también se puede tomar otro punto de vista: se puede implementar un pequeño programa que sirva para crear horarios de manera simple para un usuario y que, automáticamente, los pase al formato de este proyecto. Siguiendo este método, un usuario puede crear cómodamente el horario en la nueva aplicación y automáticamente ya se encontrará preparado en la simulación.

### 6.2.3. Simulación de eventos

Se han implementado varios eventos para probar el potencial de la simulación desarrollada pero aún hay un gran espacio para expandir este aspecto de la aplicación.

Un buen punto de entrada para mejoras son los eventos ya funcionales porque se podrían mejorar añadiendo mayor profundidad en su ejecución. En el caso de incendio se podrían crear diferentes tipos de incendios que harían reaccionar a los personajes de distinta forma y siguiendo distintos protocolos. También se puede aumentar la complejidad del comportamiento ante estas situaciones consiguiendo que este varíe un poco dependiendo de su personalidad.

Añadir nuevos eventos es una manera práctica de mejorar la calidad de la simulación ya que se podrán realizar más tipos de pruebas utilizando la arquitectura ya implementada. Se han ideado algunos eventos adicionales que serían muy útiles si se implementan en el futuro como podría ser la simulación de un congreso importante con muchos participantes, del foro del empleo que se realiza cada año en la facultad o de la usual visita de los estudiantes de 4º de la ESO. Esta aplicación se puede usar para implementar la simulación de todos estos ejemplos y muchos más consiguiendo que sea una herramienta cada vez más útil.

### 6.2.4. Nuevos escenarios

El código de la aplicación se puede reutilizar casi íntegramente para crear simulaciones de otros edificios con función educativa. Está preparado para simular

horarios de clases con alumnos y profesores pero también soporta la realización de actividades adicionales además de personajes de otros tipos ya sean visitantes u otros trabajadores del edificio. Gracias a esto, se puede reutilizar para simular otras facultades universitarias y colegios. En cada caso se tendrían que generar unos nuevos horarios y conjuntos de personajes, además de crear un nuevo mapa lógico del escenario. Con un modelo 3D relativamente detallado soportado por Unity y algunos ajustes en la escena se puede adaptar a muchas situaciones distintas.

Otra posible manera de expandir la simulación sería conseguir un escenario más complejo y de mayor escala uniendo varios edificios relacionados en una misma escena. De esta forma, se podrían analizar los caminos más usados para desplazarse entre los edificios y encontrar los momentos en los que hay acumulaciones de personas en esos caminos. Aumentar el tamaño de la simulación de esta manera añade una nueva capa de complejidad ya que permite el estudio de comportamientos fuera de un edificio y de nuevos elementos como sería el acceso por vehículo al escenario.

# Chapter 6 - Conclusions and future work

## 6.1. Conclusions

This project it's a case where two branches of knowledge which are usually seen as very distant and with almost no relation come together, such as computer science and social sciences. The extraordinary capabilities of artificial intelligence and graphic rendering allowed new ways to observe and study concepts and problems of great interest related to social behavior and their interactions.

The final version of the application allows the observation of common human behavior in the faculty and has the ability to create and set up new subjects, characters and schedules at will without much work. It also allows the simulation of exceptional situations and events such as a fire or an infectious disease outbreak. The user can move around the stage and also has the ability to interact with the simulation in various ways during at runtime by modifying its general speed, jumping between the different days of the week or initiating special situations. All relevant events are also registered in a log file, this way, the results of each simulation can be reviewed and subsequently analyzed.

Following this brief explanation, it can be said that most of the objectives that were set initially have been met, the fulfillment of each specific objective will be reviewed below.

- Create a 3D rendered social simulation of the FDI

This objective is essential for the rest of the project since it must be completed before continuing with the other objectives. We consider this objective has been fulfilled properly since the simulation implemented meets all the requirements of this type of simulations: it simulates the behavior of human characters, these characters interact with each other in a social way, the simulation is shown using a 3D model and it's located in the faculty both physically, visually equal to the faculty, and logically, the characters act following conducts similar to those that happen in the FDI in the real world.

- Simulate a realistic behavior of the characters present in the simulation

A big part of this objective has been met, but it can't be considered completed because some features that were initially planned in this section couldn't be

implemented in the desired depth. The simulation characters follow realistic schedules and a general realistic behavior but it's missing some interesting interactions, such as characters interacting and talking with each other or going to the bathroom, this interactions would increase the overall realism of the simulation.

- Ability to simulate special events in this simulation

This objective is considered to be completed suitably because there are three different events that can be simulated in the application: fire, infectious disease outbreak and zombie plague. The simulation of these events has quite an uneven level of complexity, with fire being the simplest and the disease outbreak being the most complex.

Due to the covid-19 crisis that occurred during the development of this project, we decided to develop further the simulation of the infectious disease outbreak since it's currently more useful and interesting than other options like the fire simulation. Initially, it was decided that the disease's simulation would be something very simple and generic but, at the end, we tried to reproduce with some realism the contagion of the coronavirus.

- The simulation has to be easily configurable.

This objective has also been met and even better than it was originally planned. Thanks to Unity and the potential of ScriptableObjects, a person with very little programming knowledge can use this application to simulate a wide variety of events and situations in the FDI just changing the classes and characters that will appear in each test or adding new ones. In addition, a user with some Unity knowledge can easily adapt this simulation to other buildings with a similar type of activities carried out inside.

- Develop an interface that allows the control the simulation at runtime

We consider the last planned objective has been fully completed since the application has a graphical interface during the simulation that shows relevant information about its state, for example, it shows the current time and the classes that are currently being teached. This interface also allows the user to interact with the simulation using buttons, moreover, the user can also interact with the simulation using the mouse and keyboard directly. The program also has a start menu and a drop-down menu that can be used to check the keyboard and mouse shortcuts used to interact with the simulation and to travel between the two scenes of the application.

## 6.2. Future work

This project can be used as an entry point to develop some extensions or similar applications since these types of simulations have a lot of options to be further developed, both in content and features. Here we show some ideas that were put on the table during the planning phase and some that appeared later during development time of the application. The continued development of this program can create a very powerful tool that could be even used to create real schedules and emergency protocols based on the results of its simulation.

### 6.2.1. Agents' Personality

Currently, agents can only simulate a small part of the human behavior that occurs in the FDI, focusing on moving around the scenario and performing simple tasks. An improvement we proposed was to implement more complex agents by adding a new layer of depth in their behavior. These characters could have specific traits that would define how they interact with the other characters of the simulation, following this method, they would interact more with agents with similar interests and would try to avoid those with higher conflict.

It could be implemented using a component similar to Agent called AdvancedAgent that extends the base component and adds new ways to interact with other characters. Thanks to this change, the simulation could get closer to the real world in the FDI and could allow the study of more social events.

### 6.2.2. Schedule generation

The project is designed so that a user can open the project in Unity and, without touching any line of code, can create new characters and subjects, thus giving the ability to create a completely new schedule. But the scope of this feature is limited because you can only create and configure each new character or subject individually using Unity.

The ability to read a schedule generated with another program that is usually used for this task and the ability to apply it directly to this simulation would be a very useful extension. This feature could be used to test different schedules without having to configure anything, it wouldn’t be even necessary to open the Unity editor. Developing a new component that reads a file with a specific data format which has the schedule information and loads it when starting the program could be a fine addition with a low development cost.

There's another possible approach to improve the schedule generation: implementing a small program that the user can use to create schedules in a simple way and that automatically converts them to this project's format. With this component, a user can comfortably adjust the schedule in the new application and it will be always ready to use in the simulation.

### 6.2.3. Event simulation

There are several events implemented to test the potential of the developed simulation but there is still a lot of room to expand further this aspect of the application.

A good place to start from are the current functional events because they can be easily improved by adding more depth to their behavior. For example, in the fire event some new types of fires could be added, these could make the characters react in different ways and follow distinct protocols. Making the behavior in these situations change a little depending on the character personality would be another way to increase its complexity.

Adding new events is another practical way to improve the utility of the simulation since the already implemented architecture can be used to simulate a wider variety of experiments. Some additional events that could be very useful to implement in the future are: the simulation of an important congress with many people, the employment forum that takes place every year in the faculty or the usual visit of the students from 4th of ESO. This application can be used to develop the simulation of all these examples and even more, thus making a tool with a high capacity to increase its usefulness.

### 6.2.4. New scenarios

The code of this application can be reused for the most part to create simulations of other buildings with educational purposes. It is prepared to simulate class schedules with students and teachers but also supports the performance of additional activities in addition to characters of other types, whether they are visitors or other building workers. Thanks to this, this program can be easily reused to simulate other university faculties or schools. In each case, a new set of characters and classes must be implemented and a new logical scenario too. With a relatively detailed 3D model supported by Unity and some modifications to the scene this application can be adapted to many different situations.

Another possible way to expand the simulation is by creating a more complex and larger scale scenario by joining several related buildings in the same scene. This way, the most used roads to move between buildings could be analyzed to find at which time there are accumulations of people on those roads. This way of increasing the size of the simulation adds a new layer of complexity because it allows the study of behaviors outside a building and other new elements such as the vehicle access to the scenario.

# Referencias


Gong, X., & Xiao, R. (2007). Research on multi-agent simulation of epidemic news spread characteristics. *Journal of Artificial Societies and Social Simulation*, *10*(3), 1.

Davidsson, P. (2002). Agent based social simulation: A computer science view. *Journal of artificial societies and social simulation*, *5*(1).

Mills, F., & Stufflebeam, R. (2005). Introduction to Intelligent Agents. *Ants NASA, Intelligent agent*.

Conte, G., Morganti, G., Perdon, A. M., & Scaradozzi, D. (2009). Multi-agent system theory for resource management in home automation systems.

Tapia, D. I., DePaz, J. F., Rodríguez, S., Pérez, B., & Corchado Rodríguez, J. M. (2008). Multi-agent system for security control on industrial environments.

Li, T., de la Prieta Pintado, F., & López Barriuso, A. (2015). An agent-based social simulation platform with 3D representation for labor integration of disabled people.

Moss, S., Downing, T., & Rouchier, J. (2000). Demonstrating the role of stakeholder participation: An agent based social simulation model of water demand policy and response. *CPM Report No. 00-76, Centre for Policy Modelling, The Business School, Manchester Metropolitan University, Manchester, UK*, *29*, 39.

Cruz Ruiz, F., Durbey Carrasco, A., & Sanz Briones, J. (2011). Entorno virtual 3D multiusuario para simulación de escenarios de evacuación.

Jager, W. (2007). The four P's in social simulation, a perspective on how marketing could benefit from the use of social simulation. *Journal of Business Research*, *60*(8), 868-875.

Hoffmann, A. O. I., Jager, W., & Von Eije, J. H. (2007). Social simulation of stock markets: Taking it to the next level. *Journal of Artificial Societies and Social Simulation*, *10*(2), 7.

Gilbert, N. (2007). Computational social science: Agent-based social simulation.

Arel, I., Liu, C., Urbanik, T., & Kohls, A. G. (2010). Reinforcement learning-based multi-agent system for network traffic signal control. *IET Intelligent Transport Systems*, *4*(2), 128-135.

Terna, P. (1998). Simulation tools for social scientists: Building agent based models with swarm. *Journal of artificial societies and social simulation*, *1*(2), 1-12.

Wooldridge, M., Jennings, N. R., & Kinny, D. (1999, April). A methodology for agent-oriented analysis and design. In *Proceedings of the third annual conference on Autonomous Agents* (pp. 69-76).

Conte, R., Gilbert, N., & Sichman, J. S. (1998, July). MAS and social simulation: A suitable commitment. In *International Workshop on Multi-Agent Systems and Agent-Based Simulation* (pp. 1-9). Springer, Berlin, Heidelberg.

Franklin, S., & Graesser, A. (1996, August). Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *International Workshop on Agent Theories, Architectures, and Languages* (pp. 21-35). Springer, Berlin, Heidelberg.

Poslad, S. (2007). Specifying protocols for multi-agent systems interaction. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, *2*(4), 15-es.

Macy, M. W., & Willer, R. (2002). From factors to actors: Computational sociology and agent-based modeling. *Annual review of sociology*, *28*(1), 143-166.

Pan, X., Han, C. S., Dauber, K., & Law, K. H. (2006). Human and social behavior in computational modeling and analysis of egress. *Automation in construction*, *15*(4), 448-461.

Bajaj, P., Schweller, R. M., Khademhosseini, A., West, J. L., & Bashir, R. (2014). 3D biofabrication strategies for tissue engineering and regenerative medicine. *Annual review of biomedical engineering*, *16*, 247-276.

Creber, S. A., Pintelon, T. R. R., Von Der Schulenburg, D. G., Vrouwenvelder, J. S., Van Loosdrecht, M. C. M., & Johns, M. L. (2010). Magnetic resonance imaging and 3D simulation studies of biofilm accumulation and cleaning on reverse osmosis membranes. *Food and Bioproducts Processing*, *88*(4), 401-408.

Clayton, M. J., Warden, R. B., & Parker, T. W. (2002). Virtual construction of architecture using 3D CAD and simulation. *Automation in Construction*, *11*(2), 227-235.

Scheucher, B., Bayley, P., Gütl, C., & Harward, J. (2009). Collaborative virtual 3d environment for internet-accessible physics experiments. *International Journal of Online Engineering*, *5*(REV 2009), 65-71.

Schkolne, S., Pruett, M., & Schröder, P. (2001, March). Surface drawing: creating organic 3D shapes with the hand and tangible tools. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 261-268).

Hayes-Roth, B. (1995). An architecture for adaptive intelligent systems. *Artificial Intelligence*, *72*(1-2), 329-365.

Maes, P. (1995). Artificial life meets entertainment: lifelike autonomous agents. *Communications of the ACM*, *38*(11), 108-114.