
MAS-Hospital: Modeling Healthcare Through a Multi-Agent System in Unity

Angelo Galavotti
University of Bologna
Master's in Artificial Intelligence
angelo.galavotti@studio.unibo.it

Abstract

This project presents a multi-agent hospital simulation developed in Unity which models the complex interactions between healthcare staff and patients in a dynamic medical environment. The system features autonomous agents representing doctors, nurses, janitors, and patients, whose behaviours collectively maintain hospital operations.

The simulation implements realistic healthcare workflows including patient diagnosis and treatment, equipment maintenance, infection control, and staff fatigue management. Key features include a queue-based task system where medical staff handle patients based on disease severity, dynamic equipment failure and repair mechanisms, disease transmission modeling with protective measures, and resource allocation constraints reflecting real-world hospital limitations.

1 Introduction

Multi-agent systems (MAS) constitute a sophisticated computational paradigm for modeling complex environments where autonomous entities collaborate to achieve both individual objectives and collective goals. Each agent operates independently with distinct capabilities and priorities, yet must coordinate effectively with others to maintain overall system functionality. This distributed approach naturally reflects real-world organizational structures where specialized roles and interdependent responsibilities drive institutional operations, making MAS particularly well-suited for simulating complex environments such as manufacturing systems, transportation networks, and of course healthcare facilities.

In particular, hospital environments exemplify the ideal application domain for multi-agent modeling due to their intricate operational complexity and clearly defined roles of the employees. Healthcare facilities operate through the coordination of personnel (i.e. doctors, nurses, administrative staff, and other support personnel) each with distinct responsibilities, priorities, and constraints. At the same time, patients flow through the system with varying needs and urgency levels, while resources such as equipment, beds, and medications must be allocated efficiently. Additionally, unexpected events like equipment failures, staff shortages, or disease outbreaks require adaptive responses that emerge from the collective behavior of individual employees.

This project aims at building a comprehensive hospital simulation system using the Unity engine (2), designed to capture the complex interdependencies between healthcare professionals (doctors, nurses, and janitors) and patients in a realistic medical environment. The simulation encompasses authentic healthcare workflows including patient diagnosis and treatment protocols, equipment maintenance procedures, infectious disease transmission dynamics, and staff energy management systems.

Ultimately, the goal is to build a working hospital system using multi-agent principles, creating something that accurately represents how these environments actually function while being interesting to watch and interact with.

2 Background

2.1 Original game

The main inspiration for this project came from the game “Theme Hospital”, released by Bullfrog Productions in 1997. It is a simulation game that allows players to design and manage their own hospital. Players build facilities, hire staff, treat patients with humorous diseases, and ensure the hospital runs efficiently. It features a variety of challenges, including managing resources, dealing with patient needs, and maintaining staff morale. At the time, it garnered high critical acclaim, and it is now considered a timeless classic in the space of simulation and strategy games. A screenshot of the game is shown in Figure 1.



Figure 1: Screenshot of Theme Hospital

Theme Hospital’s gameplay revolves around constructing and expanding medical facilities room by room, each of which serves a specific purpose. However, staff management is at the base of the core gameplay loop. In this context, non-playable characters (NPCs) perform the core operations of the hospital based on the player’s decisions: doctors with varying skill levels handle diagnosis and treatment, nurses assist with patient care and operate certain machines, handymen maintain equipment and clean up litter. Each staff member has individual attributes including skill level and stamina, requiring players to balance hiring costs with performance needs.

2.2 Goals

The primary goal of this project was to create a multi-agent system that accurately models and simulates the complex interactions between healthcare staff and patients in a hospital environment. As previously mentioned, many of the characteristics of the system were inspired by the original game; however, the focus was not on replicating its mechanics (i.e. a strategy-based gameplay loop, with the main objective of managing the resources correctly), but rather on implementing a realistic simulation that operates without the need for user interaction. As such the assignment of tasks and resources is completely automated, and so are many other gameplay aspects previously managed by the user.

In other words, the connection to the original game is more thematic than functional: the roles of the agent roles and separation of diseases are indeed inspired by the original game, with the added characteristics of multi-agent systems (e.g. coordination, individual and collective goals, resource sharing).

2.3 Why Unity?

Unity proved to be the ideal platform for implementing this multi-agent hospital simulation primarily due to its Entity-Component-System (ECS) architecture (3), which organizes software objects into three distinct categories:

- **Entity:** represents a general-purpose object identified by a unique ID. In the hospital simulation, each agent (doctor, nurse, janitor, or patient) is first instantiated as a `GameObject`,

which constitutes an entity that serves as a container for various behavioral and data components.

- **Component:** Defines specific aspects or capabilities of an entity by holding the relevant data. For example, worker agents possess a *Worker* component, which implements energy management, the task queue, as well as movement components. On the other hand, patients have a *Patient* component which for example implements the state machine (for more details, see Section 4.1).
- **System:** Processes entities that contain specific component combinations. All Unity components inherit from the *MonoBehaviour* base class (4), and the engine executes the *Awake()*, *Start()* and *Update()* methods which are implemented in all components and are defined in the *MonoBehaviour* specification.

The ECS approach allows each agent to be treated as an independent entity with modular components that define their capabilities and behaviors, making it straightforward to create the diverse agent types required for the hospital simulation.

Additionally, Unity's built-in NavMesh (5) system also provided essential pathfinding capabilities that would have been complex to implement from scratch, allowing agents to navigate the hospital environment autonomously while avoiding obstacles and finding optimal routes to their destinations.

Finally, from a practical standpoint, my previous experience developing games in Unity as a hobbyist made this choice even more logical.

3 Design and features

3.1 Workflow

The workflow of the hospital simulation is designed to reflect the typical operations of a healthcare facility, albeit simplified. The main steps in the workflow are as follows:

1. **Patient Arrival:** Patients enter the hospital through the entrance and are automatically assigned to a diagnosis desk, where they wait for a free doctor for the diagnosis.
2. **Diagnosis:** At the diagnosis desk, a doctor examines the patient and determines their condition. Based on the diagnosis, the patient is assigned a treatment path.
3. **Treatment:** Depending on the diagnosis, the patient may be directed to a hospital bed for serious conditions, or to a pill machine for minor diseases. Nurses or doctors assist in administering treatment based on the gravity of the illness (at their respective stations).
4. **Paperwork:** After treatment, nurses file reports at the file cabinet to document the patient's treatment outcome.

However, the workflow is not linear, and it can be interrupted by various disrupting events which can occur at any time during the simulation. These events include:

- *Equipment Failure:* Machines, in particular pill machines, can break down after a certain number of uses, requiring janitors to repair them.
- *Infection Spread:* A patient with an infectious disease can enter the hospital, and infect other patients and, most importantly, the hospital staff, leading to potential outbreaks.
- *Staff Fatigue:* Staff members can become exhausted after their energy is depleted, requiring them to rest at designated stations before returning to work, and cancel their currently assigned task.
- *Patient Death:* At any time during the simulation, a patient may die randomly if the treatment is not effective enough or if they are just very unlucky. After a patient dies, a nurse must file a report at the file cabinet.

3.2 Agents

At the core of the system lie the agents, which represent the various roles within the hospital. By definition, each agent is an autonomous entity that can perform specific tasks and interact with other

agents, stations and the environment in general. While they have individual goals, their management is also supported by an Hospital Manager background process, which aids in the coordination of agents and distribution of tasks.

The agent can be divided into two main categories: *workers* and *patients*. These two types of entities share some characteristics, but they are very different in their behaviour and how they are managed.

At their core, agents share the capability of moving around the hospital. In fact, whenever an agent is idle, it will move around the hospital to a random location until it is assigned a task or station. This small addition to the system prevents the agent from being static and makes the simulation more fun and dynamic. In addition, agents all share a `cleanliness` statistic, which indicates how dirty the agent can be and as such how likely it is to spawn puddles (as well affecting the agent's contagion risk and death chance).

3.2.1 Workers

Workers maintain the hospital functionality through specialized roles and responsibilities. They differ from the patients in that they implement a task-based behaviour. The workforce is designed around three distinct professional categories, each with unique capabilities that mirror real-world healthcare hierarchies.

- *Doctors*: Responsible for diagnosing patients and determining their treatment paths. They can also treat patients at hospital beds, and are the only agents that can perform diagnoses.
- *Nurses*: Assist doctors in treating patients, dispense medication from pill machines, and file reports at the file cabinet.
- *Janitors*: Maintain the hospital by cleaning up puddles, repairing broken machines, and retrieving tools from the janitor closet. They are essential for keeping the hospital running smoothly.

A collection of screenshots of each agent can be found in Figure 2.

The worker design incorporates realistic limitations that affect their availability. In particular, energy depletion creates natural work-rest cycles where staff members must periodically step away from duties to recover.

The design emphasizes resource sharing and sequential handoffs. In particular, an agent cannot be assigned a station that has already been assigned, and has instead to wait their turn (this is also true for patients, when they are waiting for treatment or diagnosis). This can lead to complex situations, especially when considering janitors, as they need to equip the correct tool for the task.

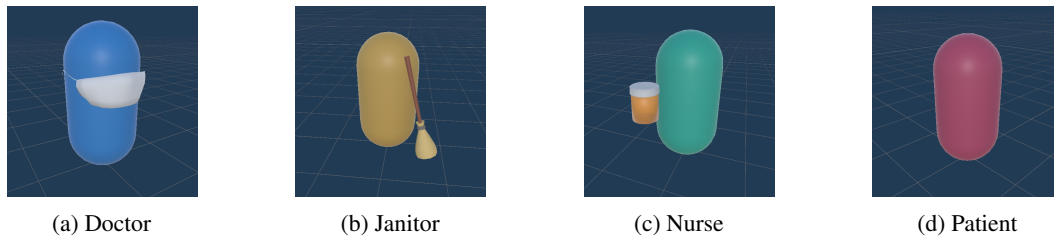


Figure 2: Screenshots of the agents in the game. The doctor is wearing a mask, the janitor has a broom in his “hands”, and the nurse is grabbing a pill box. Normally (e.g. when idle), each agent does not have an accessory item with it.

3.3 Stations

During its operation, an agent can be assigned to a station, which is a specific location in the hospital where it can perform its tasks. Each station is related to a specific task or state of a task, and some stations may require the presence of a patient in order to be used. Finally, each station has some specific characteristics, and may be broken or require the use of a specific tool in order to be used.

Here are the main types of stations that were implemented in this systems:

- *Diagnosis desk*: Newly arrived patients must first visit an available diagnosis desk where a doctor determines their condition and assigns the appropriate treatment path.
- *Pill machine*: Patients diagnosed with minor diseases receive medication from the pill machine, operated by a nurse who dispenses the necessary pills. After a set amount of uses, the machine will break and it will need to be repaired by a janitor.
- *Hospital beds*: Patients with serious conditions require intensive care and treatment from a doctor at a dedicated hospital bed, where they remain until fully recovered.
- *File cabinet*: Whenever a patient dies or is cured, a nurse must complete the necessary paperwork by filing a report at the cabinet.
- *Janitor closet*: Janitors retrieve the appropriate tools from this station before cleaning puddles or repairing broken equipment.
- *Rest sofa*: When staff members become exhausted, they must temporarily abandon their duties to restore their energy at this rest area before returning to work.

More details about the implementation of the station and how they are integrated in the environment can be found in the Paragraph 4.2.

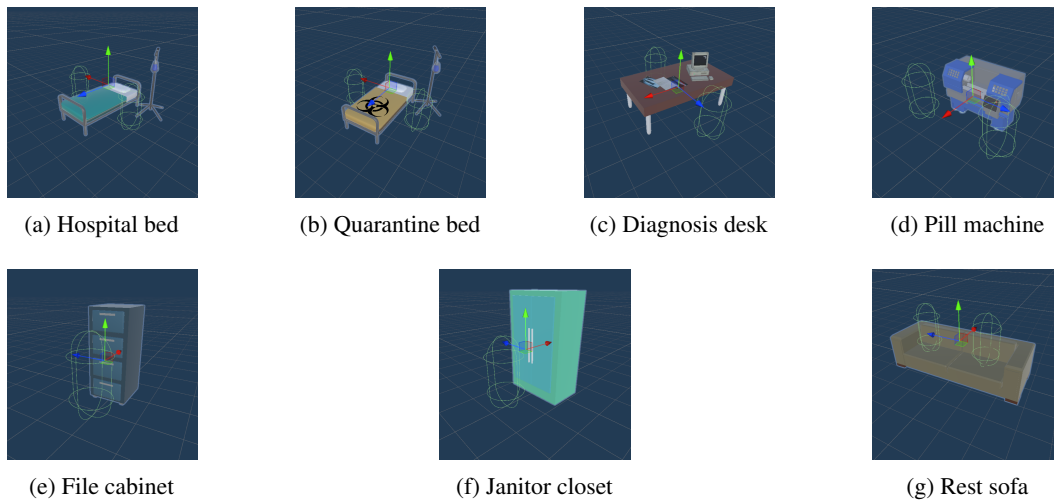


Figure 3: Hospital stations and equipment used throughout the simulation. In each image, a green outline can be seen. Each of them represents either a patient or worker position when interacting with the station.

3.4 Diseases

There are three main types of disease variations that patients can contract (*mild*, *serious* and *contagious* diseases), each with its own characteristics and treatment paths. In particular, the mild diseases require a patient to visit the pill machine, where a nurse will hand a pill to the patient. On the other hand, the serious diseases require a patient to be treated at a hospital bed, where a doctor will take care of the patient until they are cured. After that, they will be able to leave the hospital, and they'll hopefully be satisfied and without the need for further medications.

Finally, some diseases are infectious, meaning that they can spread to other patients and staff members. For simplicity, all infectious diseases are treated as serious diseases. In this case, the patient will be quarantined to a variation of an hospital bed (i.e. a quarantined hospital bed, which has a different design from the regular one, as shown in Figure 3b), and during treatment, the doctor will be required to wear a mask. This is not just for show: it will reduce the contagion probability of the doctor by a large percentage (but not completely eliminate it). Whenever a staff member gets infected, they will fundamentally turn into a patient, and as such they will require to undergo diagnosis and treatment process, and may die at any time. If their treatment is successful, they will be able to return to work. Walls also prevent the spreading of the infection, adding even more realism to the simulation.

4 Implementation

4.1 Agent implementation

At the foundation, the Agent base class defines essential behaviors shared across all entities in the simulation, including autonomous movement through Unity's NavMesh (5) system. The patrol system enables agents to move randomly within a configurable radius when idle, preventing static behavior and maintaining visual dynamism. This autonomous movement is constrained by the `CanPatrol()` abstract method, which each agent subclass implements according to its specific behavioral requirements.

Each agent also implements a cleanliness system where they can spawn puddles based on individual cleanliness statistics. Additionally, all agents maintain contagion susceptibility values that influence disease transmission rates. An abstract method `DismissAgent()` is also specified in the base Agent class, which is then implemented by the Worker and Patient classes. This function handles the release of stations and tasks (depending on the type of agent) whenever an Agent is deleted by the user through the UI (see Section 4.4).

The Worker class extends the base Agent class with an energy system: whenever an agent is executing a task, its energy is gradually reduced (based on the effort required by the task) until it is completely depleted. In this event, the `ForcedRest()` function gets called. This essentially creates a new Rest "task", and assigns the agent a rest station (Figure 3g), where the agent will be finally allowed to recharge its energy. Before the `RestTask` can be executed, the current task and all the pending tasks in the agent's queue are canceled. In this way, they can be redistributed to other workers while the agent is taking a nap. The `RestTask` is implemented in the base Worker class, as it is shared by all of them.

The state of the Worker is modeled through a `StatePair` system, which consists on a data structure that stores the current task that the agent is doing, and the action related to it (e.g. moving towards the station of the task, working on the task...). This information is then used in order to do the correct action related to the job at the proper time (see Section 4.2 for more details).

4.1.1 Doctors

Doctors implement two core task classes: `DiagnoseTask` and `TreatTask`, which implement the patient evaluation and serious condition treatment respectively.

The `DiagnoseTask` is assigned to the doctor whenever a patient reaches the diagnosis desk station (Figure 3c). The first step of the task consists on the movement to the desk's operator position, then the agent's `StatePair` transitions to a working state for a configurable diagnosis duration. During execution, the task continuously validates patient status using `IsPatientValid()` and automatically releases the assigned desk station upon completion or cancellation. The diagnosis process concludes by calling `patient.ReceiveDiagnosis()`, which triggers patient state transitions and disease severity determination.

The `TreatTask` handles serious patients with serious diseases. Upon task initialization, doctors automatically check patient contagion status and activate protective masks via `SetMask(true)` when treating infectious patients. The treatment workflow monitors patient state transitions, specifically detecting when patients enter the Leaving state to determine successful treatment completion. During active treatment, doctors call `patient.ReceiveTreatment()` with their effectiveness parameter to reduce patient death probability. After their treatment is completed, the patient transitions to the Leaving state, and will be de-spawned as soon as they reach the hospital's main entrance.

4.1.2 Nurses

Nurses handle mostly support services through two primary task, which consist on treating mild diseases through medication dispensing and managing documentation workflows. In addition, they are also the only agents whose tasks completely depend on other agent's tasks or events.

The `TreatModerateTask` manages pill machine operations for patients with minor conditions, featuring visual feedback systems where nurses display pill box accessories during treatment procedures via `setPillBoxVisibility()`. This task directly influences the usage value of a machine, by calling `pillMachineComponent.Use()` upon successful treatment completion to increment machine

wear and potentially trigger breakdown events. For the most part, however, this task is similar to its doctor counterpart (see Section 4.1.1): the agent calls `patient.ReceiveTreatment()` and changes their status to `Leaving` whenever the process is complete.

The `FileReportTask` handles the hospital’s administrative documentation, by simulating the hospital record keeping through filing procedures at file cabinets (Figure 3e). This particular task is assigned after a patient’s successful treatment or death, after which the agent needs to request a free file cabinet station from the Hospital Manager. Based on the event, the task causes the hospital’s statistics to change, by increasing either the cured or dead patients counters.

4.1.3 Janitors

The janitor implementation showcases the most complex task coordination system among all Worker types, featuring tool management and multi-agent resource sharing. In general, Janitors have to deal with repairing machines and cleaning puddles (the `CleanPuddleTask` and `RepairMachineTask` classes are associated to these corresponding jobs). However, their implementation gives priority to repairs, and only afterwards it scans for cleaning tasks. This discrimination is due to the fact that machines are indeed very important for the hospital workflow to proceed correctly, since without them the nurses cannot handle mild diseases.

Janitors are the only agent type that require the right tool whenever they need to work on a task. The tool management system requires multi-step task sequences where janitors must first execute `GetToolTask` to retrieve appropriate equipment (brooms for cleaning, wrenches for repairs) from storage closets before proceeding with their primary objectives. In addition, multiple janitors cannot go to the same closet at the same time; as such, they can complete both their `GetToolTask` and the following one only if they have access to a janitor closet (Figure 3f). Nevertheless, if the agent already has the correct tool for the job, they won’t need to perform a `GetToolTask` first, and they can instead get to work without a care in the world.

4.1.4 Patients

Unlike Workers, Patient agents implement a Finite State Machine (FSM) that guides them through the hospital experience using the `PatientState` enumeration, rather than a `StatePair`. As such, their behaviour is more deterministic than the Workers; however, they can still experience random events, most notably their death. Each patient maintains an individual `deathChance` probability that is checked at regular intervals defined by `deathCheckInterval` (typically every 15 seconds). When the random death check triggers, the patient immediately transitions to the `Dead` state, visually indicated by rotating the patient mesh 90 degrees and lowering its position to simulate lying on the ground. The death event automatically releases any assigned stations (diagnosis desks, treatment beds, or pill machines) and will trigger a `FillReport` task assignment. Treatment effectiveness can reduce a patient’s death probability over time, but never eliminates it entirely, maintaining the inherent uncertainty that characterizes real medical environments. A schematic representation of the Finite State Machine is shown in Figure 4.

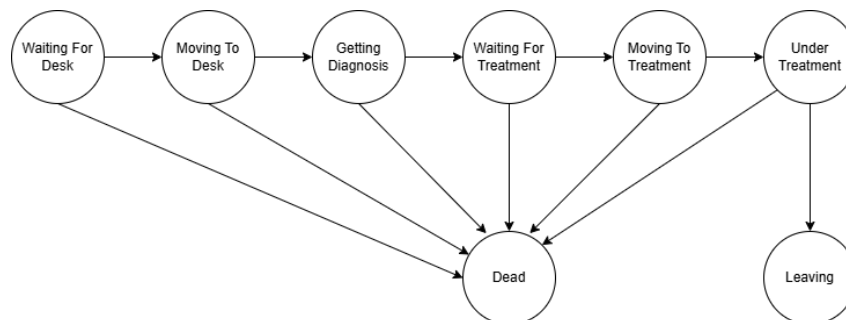


Figure 4: The patient’s FSM which dictate its behaviour. This simplistic implementation is the one that makes most sense in the context of the patients, since their behaviour is more deterministic than workers and they do not possess the concept of “task”.

While the “puddles spawning” system is implemented for all types of agents, in practice only the patients are able to use it, based on their cleanliness. This decision was made in order to reduce the event of Janitors being overwhelmed with work. Keeping the number of puddles low is important, as they influence the cleanliness of the hospital, and consequently the contagion and death probability of patients. This mechanic is dictated by the `CheckPuddleSpawn()` function, which spawns a puddle periodically based on the value of `puddleSpawnCheckInterval`.

Patients are also at the core of the infection disease system. Whenever a patient has a contagious disease, a rotating indicator is displayed on the top of its mesh, as seen in Figure 5. As previously mentioned, patients can infect the workers, and when they do so, all their tasks are canceled and they become effectively patients. In order for this mechanic to be implemented, the Patient class store the `GameObject` reference of the Worker instance before the infection (i.e. the `originalWorkerGameObject` variable); thus, when their treatment is complete, the Patient agent is removed and instead replaced with a new instance of the older Worker object. In this way, the values and characteristics of the agent before this event remain intact, and easily restored once they have been cured.

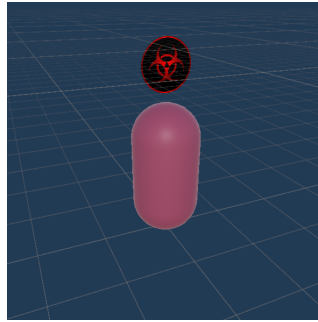


Figure 5: A contagious patient. The symbol over its head is the infection indicator.

The infection mechanic is handled on a similar as the puddle spawn and the death mechanic, where a function `CheckForContagion()` is called whenever a timer is triggered. Through this procedure, infectious patients periodically scan their surrounding area using `Physics.OverlapSphere()` to detect nearby agents within a configurable contagion radius. For each potential target identified, the system employs `HasLineOfSight()` to perform raycast validation, ensuring that walls can block disease transmission by checking specifically for objects on the “Wall” layer. Once line-of-sight is confirmed, `TryInfectAgent()` calculates the actual transmission probability by combining the patient’s base contagion chance with the target agent’s individual susceptibility rating. Figure 6 shows a screenshot of the simulation during an infectious disease outbreak.

4.2 Task system

The task system represents the core mechanism of the Worker agents, while patients simply do not make use of it. In the previous paragraph, it was mentioned how task execution operates through a state-pair mechanism where each task maintains both a worker state (e.g. Moving, Working, Resting, Idle) and a specific task type (e.g. Diagnose, Treat, Clean, Repair). This dual-state approach enables a more precise behavioral characterization, and allows to model each task execution flow as a very simplistic FSM where only the Moving and Working states need specific implementations.

Each task is built around a common base class `HospitalTask` that defines the essential interface for all worker activities throughout the simulation. Every task implementation must provide three fundamental methods:

- `Execute()`, which performs the core task logic during each update cycle. It is called in the `Update()` method of the Workers whenever their current task is defined. This is also where the FSM is defined.
- `Complete()`, which performs the final procedures before completing the tasks.
- `Cancel()`, which is called whenever the task is interrupted abruptly (e.g. infection or rest).
- `isComplete()`, which simply returns “true” if the task has been completed.

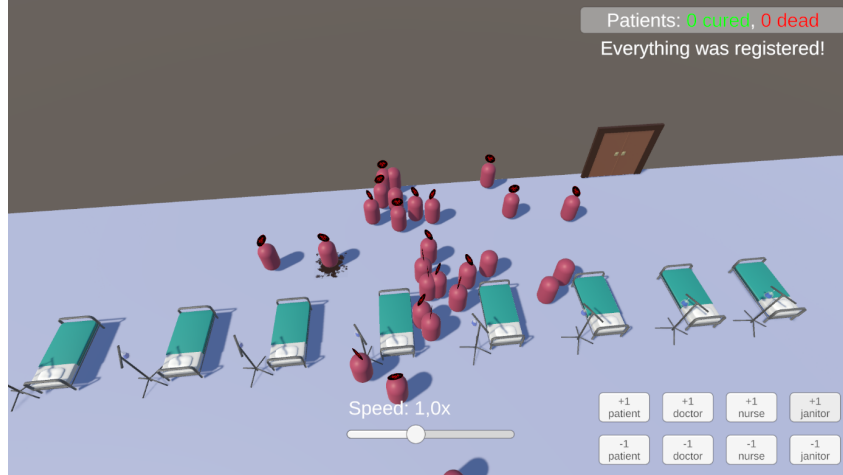


Figure 6: A picture showing a contagious disease outbreak. We can see how almost all agents have been infected, and there are no healthy workers left.

Each task also contains a `requiredEffort` property, which defines the amount of energy consumed when working on it.

A key feature of this project is its ability to handle task interruptions gracefully, using a system which is, in a way, similar to Linda’s Tuple-based coordination model (6). When workers become exhausted and require rest, or when they get infected, current tasks are automatically cancelled. When nurses and doctors do so, the function `SuspendTask()` is called on the task that is being interrupted. This function wraps the task using a `SuspendedTask` data structure, which encapsulates the information related to the task, such as the task itself, the reference to the patient and the worker type, thus building what is essentially a tuple. Every 60 frames, the suspended tasks are then processed and reassigned based on the worker availability¹ (see Section 4.3.1).

Task are canceled in specifically when the worker is exhausted or is infected. On the other hand, the janitor’s tasks are not reassigned, since they scan the level for broken machines and puddles by themselves. Before assigning tasks requiring specific locations, availability is first verified through the Station Manager. This prevents conflicts such as multiple workers attempting to use the same diagnosis desk simultaneously (more details in Section 4.3.2). This resource-aware approach ensures that the simulation maintains realistic operational constraints while providing the flexibility needed for complex multi-agent coordination.

4.3 Managers

The “Managers” are processes that support the coordination of different functions of the simulation. They are implemented as a single `GameObject`, with five components attached to it, with the Hospital Manager being the main one.

4.3.1 Hospital Manager

The Hospital Manager is implemented as a Singleton (7), and it is responsible for holding the references to (most of) the other components. One of the its most important functions is the task suspension system. Whenever an agent calls the `SuspendTask()` function, it gets stored into a queue, and every 60 frames it attempts to reassign a suspended tasks to another agent. Two different criterias for choosing the agent for reassignment are implemented:

- *Random assignment*: the task is assigned to a random worker compatible with it.

¹Of course, in the Linda system the situation is different, as the agents usually processes the tuples by themselves. However, this implementation was more compatible with the task queue system and allows suspended tasks to be processed much faster.

- *Least-tasks assignment*: the task is assigned to the relative worker with the least task. This is the default selection method for task assignment.

The Hospital Manager also plays a part in handling the death of patients. It removes the tasks related to the entity from the `suspendedTasks` queue, as well as the queues dedicated to the patients waiting for diagnosis or treatment. Additionally, it handles part of the logic responsible for infecting patient. In particular, it spawns the `Patient GameObject` when a Worker-type agent is infected, and of courses handles the removal of patients from their respective queues.

4.3.2 Station Manager

The Station Manager handles the discovery, tracking, and allocation of the station, which are essentially all the physical locations where agents perform their tasks. At startup, the component scans the scene for stations using Unity's tag system, automatically building internal registries for each station type. This dynamic discovery approach means new stations can be added to levels without requiring code changes: the system simply searches for objects with specific tags like `DoctorDesk` or `PillMachine` and registers them appropriately.

Each station is represented by a `StationInfo` object that maintains three key transform references: the main station object, an operator position where workers stand, and optionally a patient position for medical procedures. The availability tracking system prevents resource conflicts by marking stations as unavailable when assigned to agents. When a doctor begins diagnosis at a desk, that station becomes locked until the task completes. When a task is cancelled, however, the station is not released, in order to ease redistribution.

Although puddles can be considered as stations in this context, they are treated in a much different way by the system. The puddle management subsystem dynamically tracks the environment to acquire the object's references; this happens periodically, not just at the startup. In this way, a janitor can simply ask this component for a puddle to clean (if there are any). In order to avoid janitors cleaning the same puddles, a separate lists for assigned ones is kept, and obviously if the puddle is in both lists it won't be assigned. When puddle counts exceed the configured maximum, the system destroys excess puddles in FIFO order, preventing performance degradations and the overwhelming of the environment.

4.3.3 Patient Manager

The Patient Manager orchestrates patient flow through the hospital by maintaining queues for patients awaiting diagnosis and treatment while coordinating with the Station Manager to match patients with appropriate medical facilities. When new patients arrive, the system attempts immediate desk assignment, but falls back to queue management when resources are unavailable. To do so, the component maintains waiting queues and checks them every second to reassign patients when stations become available. In fact, it is responsible for the patient's station assignation in general, based on the disease type and the current state of the patient's FSM.

4.3.4 Stats Manager

The Stats Manager (or Statistics Manager) component is dedicated to the assignation of values of the properties of each agent. In particular, a random number in a predefined range is assigned for the most important values of each agent; as such, when an Agent spawns, its characteristics are always unique. The range for each value can be changed in engine, as well as the Start Menu UI (Section 4.4).

In addition, this module contains the name generation logic for the agents, which procedurally generates a name using data from a list of first and last names.

4.3.5 Machine Manager

The Machine Manager is a minor component used to keep track of the status of machines at runtime. In particular, it contains functions for handling machine assignment to janitors when they need to be repaired, and for initiating a repair when a machine is broken.

4.4 User interaction

The user can interact with the simulation in many ways, each of which is expressed through the user interface. In Figure 7, the various elements of the UI available at run-time can be clearly seen. Clicking on any agent displays a detailed information panel showing the agent's name, current status, assigned tasks, and relevant statistics. The interface enables the user to spawn additional agents of specific types or remove existing ones to test different scenarios and observe how the multi-agent system adapts to changing workforce compositions and patient loads.

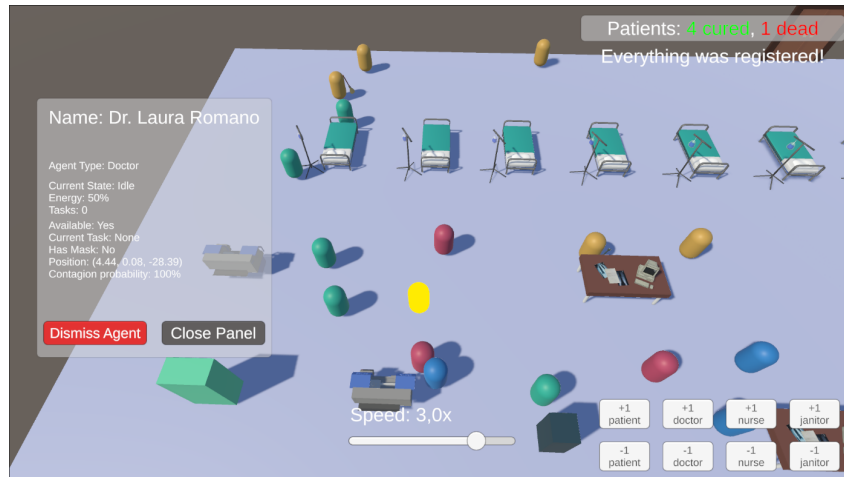


Figure 7: Screenshot showing an early level of the simulation running. The UI can be clearly seen here. At the center of the picture, the current highlighted agent can be seen colored in bright yellow. On the side, the panel with the agent information. On the bottom right, we have the buttons dedicated to adding and removing random agents. Finally, at the bottom center the simulation speed slider.

The main menu features a comprehensive parameter configuration UI, as shown in Figure 8, which allows users to adjust simulation variables including patient mortality rates, agent cleanliness levels, infection transmission parameters, worker skill ranges, and environmental factors like maximum puddle counts and spawn timing. These sliders allow the user to tune the simulation and observe how specific situations are handled by the agents.

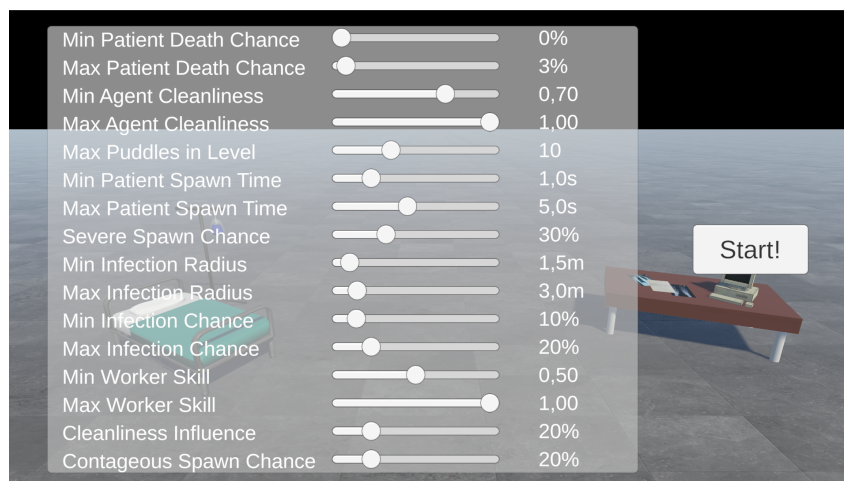


Figure 8: Screenshot showing the options that the user can tune in the main menu, to alter specific aspects of the simulation.

5 Final Remarks

5.1 Conclusions

MAS-Hospital demonstrates the fundamental principles of multi-agent systems through a practical implementation that showcases how autonomous agents can coordinate effectively to achieve complex collective objectives.



Figure 9: Screenshot of the final result.

The agent-based architecture successfully models the heterogeneous nature of multi-agent environments, where different agent types possess distinct capabilities and goals, yet must collaborate to maintain system functionality. The task-based coordination mechanism exemplifies how local agent decisions can produce global system behaviors, with individual choices about task prioritization (for janitors in particular) and resource allocation creating emergent patterns of hospital operation.

From a personal perspective, this project has been highly successful in deepening my understanding of multi-agent system design. Working through the complexities of concepts such as agent coordination and resource management has provided invaluable insights into the practical challenges of MAS development. In addition, this project has significantly enhanced my appreciation for the power and elegance of agent-based approaches. As such, MAS-Hospital not only validates multi-agent systems as an effective paradigm for modeling complex organizational structures such as medical centers, it also serves as a testament to the educational value of hands-on MAS implementation in understanding these sophisticated computational frameworks.

5.2 Future Work

While I am personally satisfied with the current feature set and implementation of MAS-Hospital, several areas could benefit from further development to enhance both functionality and realism.

- **Enhanced user interaction capabilities:** Future iterations could incorporate more interactive elements such as manual task assignment overrides, emergency response protocols that users can trigger, or real-time parameter adjustments for disease severity and equipment failure rates. Additionally, implementing a resource management interface would allow users to control hospital finances (e.g. equipment/station purchases and staffing decisions).
- **Expanded medical complexity:** The current disease model encompasses three basic categories (mild, serious, and contagious), which could be significantly expanded to include condition-specific treatment protocols and multi-stage diseases requiring sequential treatments. Additional station types such as radiology departments would create more diverse patient pathways and require expanded worker specializations.
- **Dynamic environment creation:** A comprehensive level editor would enable users to design, for example, custom hospital layouts and define the station placement. Such functionality

would significantly extend the simulation’s capabilities, while also bringing it even closer in function to the original inspiration. In addition, it would be easily compatible with the current station management system, as their implementation handles them dynamically (Section 4.3.2).

In addition, the project provides a solid foundation for implementing a gym for training agents using Reinforcement Learning, thus encouraging even more emergent behaviors that would make the simulation more dynamic and interesting. Although this was originally outlined as a stretch goal in the proposal, time and computational constraints prevented its inclusion in the current version. RL can be applied in numerous ways in this context:

- Patient and worker behaviors could be learned rather than hard-coded (i.e. learning agents). Defining reward/penalty structures would be straightforward since the main goal of the simulation is, in fact, to cure as many patients as possible. On the other hand, one challenge would involve extending the current system to ensure agents properly respect task durations, waiting times, and resource values.
- A management agent could oversee higher-level decisions like patient admissions (based on economic factors) and staffing changes (hiring/firing based on performance metrics). This agent would operate on regular time intervals, such as every few simulation ticks, essentially creating an AI that could play the original Theme Hospital game autonomously.

Finally, Unity provides the “MLAgents” components (8), which eases the introduction of learning agents to the scene. As such, instead of implementing QLearning algorithms from scratch, one could adapt the agents and environment to allow for the training and loading of models using the tools provided by the engine.

References

- [1] Electronic Arts (1997). *Theme Hospital*. Bullfrog Productions.
- [2] Unity Technologies (2025). *Unity Engine*. Retrieved from <https://unity.com/>
- [3] Bilas, Scott. *A Data-Driven Game Object System*, GDC Presentation 2002. Retrieved from <https://www.gamedevs.org/uploads/data-driven-game-object-system.pdf>.
- [4] Unity Technologies (2025). *Unity MonoBehaviour API*. Retrieved from <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>
- [5] Unity Technologies (2025). *Unity NavMesh API*. Retrieved from <https://docs.unity3d.com/ScriptReference/AI.NavMesh.html>
- [6] David Gelernter. *Linda*. Retrieved from <https://lindaspaces.com/>
- [7] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994
- [8] Unity Technologies (2025). *Unity MLAgents*. Retrieved from <https://github.com/Unity-Technologies/ml-agents>

A Guide

A.1 Compiling the project

In order to compile the project, you will to install the “Unity Hub” application on your machine, and then download a version of the engine using the launcher (by clicking “Installs > Add”). The project was developed using Unity 2022.3.21f1, and it is recommended to use the same version to avoid compatibility issues.

To load the project, press “Add > Add project from disk” dropdown on the top-right of the launcher, and select the project’s folder.

The code of the project is written in C#, and can be found in the “Assets/Scripts” folder of the project.

A pre-compiled version of the project is also available at the following link: https://liveunibo-my.sharepoint.com/:f:/g/personal/angelo_galavotti_studio_unibo_it/EtQYFa9YrfdKrdMcRaej7rkBoCNo6NOENlvUp103tsN-4w.

A.2 User interface

Whenever the software starts, the user is presented with a main menu, which displays buttons to decide the level to load and the specific type of mode for the simulation. In particular, there are two modes:

- *Sandbox mode*: in this mode, the user can decide which and how many agents to spawn or remove in real-time in a random fashion, thanks to a button panel on the bottom-right.
- *Simulation mode*: the worker agent are already spawned, and the patient will spawn randomly from the entrance of the hospital (although the user can still add and remove agents).

In order to return to the main menu, the user can press the “ESC” key, and here they will be able to chose to exit the application or run the simulation in another level or mode.

When the simulation is running, the user can interact with the agents by clicking on them, which will display a small panel with some information about the agent, such as its name, type, and current task. The user can also the decide to dismiss the specific agent by clicking the relative button.

The camera can be moved around the scene using the mouse, and the user can zoom in and out using the scroll wheel (alternatively, they can use the left mouse button and move it left or right for a faster zooming experience). The camera can also be rotated by holding the right mouse button and moving the mouse, and moved around by holding the middle mouse button and moving the mouse.

B Video demonstration

A video demonstration of the simulation can be found at the following link: <https://youtu.be/PENMUTSiL1E>.