Simulacija prodavaonice pomoću agenata temeljenih na uvjerenjima, željama i intencijama

Kiš, Martin

UNIVERSITY OF ZAGREB
FACULTY OF ORGANIZATION AND INFORMATICS
VARAŽDIN

Martin Kiš

SIMULATING A CONVENIENCE STORE THROUGH THE USE OF BELIEF-DESIRE-INTENTION AGENTS

MASTER'S THESIS

Varaždin, 2023

UNIVERSITY OF ZAGREB

FACULTY OF ORGANIZATION AND INFORMATICS

V A R A Ž D I N

Martin Kiš

Student ID: 0016136486

Programme: Information and Software Engineering (IPI)

SIMULATING A CONVENIENCE STORE THROUGH THE USE OF BELIEF-DESIRE-INTENTION AGENTS

MASTER'S THESIS

Mentor:

Bogdan Okreša Đurić, PhD

Varaždin, September 2023

Martin Kiš

Statement of Authenticity

Hereby I state that this document, my Master's Thesis, is authentic, authored by me, and that, for the purposes of writing it, I have not used any sources other than those stated in this thesis. Ethically adequate and acceptable methods and techniques were used while preparing and writing this thesis.

The author acknowledges the above by accepting the statement in FOI Radovi online system.

Abstract

The thesis focuses on combining Unity and Prolog to create a convenience store-like environment and populate it with agents that play the roles of customers and clerks. The theoretical part of the thesis describes the meaning of belief-desire-intention agents as well as the process of creating them and specifying what they need to be able to accomplish in the setting. The practical part consists of creating the desired environment and the agents that inhabit it. Agents make their decisions based on their knowledge bases.

Keywords: simulation, BDI agents, Unity, Prolog, Artificial Intelligence, agents

Table of Contents

1. Introduction

The following thesis explores the idea of using a multiagent system to simulate the work of a convenience store. Unity, an engine typically meant for game development, is used to create a virtual store environment that is populated by various types of agents. In order to expand upon the functionalities that Unity and the C# programming language offer, the project also incorporates the declarative language, Prolog. By utilizing the unique options the language offers, the project can be given another layer of complexity, allowing for the creation of Belief-Desire-Intent agents who offer a unique form of modelling that is explored within this thesis.

With artificial intelligence growing more and more popular each day, people are exploring new ways to incorporate it into various problem domains. It is a vast field that can offer a lot to anyone who chooses to employ the different tools it offers. A growing number of businesses and organisations are starting to build highly complex expert systems to aid in their decision-making processes, image and audio generation is becoming a widely-spread phenomenon, chatbots are taking over tech support positions and many more such examples can be seen in modern society.

Among the opportunities that artificial intelligence presents are simulations and agents, which can serve as simplified models of real-world scenarios and roles. Their utilization can offer valuable data to their designers and allow for a new bottom-up approach, where the entire model hinges on smaller roles played by the agents within the simulation. With a high number of studies already existing on the subject, this thesis tries to showcase that game engines can make use of these concepts too, making them apt for creating more realistic and decision-driven scenarios rather than the more commonly used flowcharts and finite state machines.

2. Artificial intelligence

Artificial Intelligence is a very popular and common subject, especially in recent years, with a lot of advancements and applications in a large number of scientific fields. From assisting in business decisions of large corporations to creating a fun experience for people playing video games, there are a lot of ways in which it can be applied or defined depending on what the user wants it to accomplish, allowing it to help in a wide variety of problem domains.

The use of AI allows people to outsource some of the workload onto the vastly superior capabilities of a computer. An AI never tires, does not make mistakes and its results will always be more consistent than ones that a human would produce. Complications start to arise when the user needs to let the AI know what its purpose is and how it should accomplish that purpose. A lot of time needs to be spent on properly conveying complex real-world tasks to a computer. A common way to do this is by either making a problem highly abstract, making it simpler for the computer to understand, or by giving the computer tools with which to decipher what the user wants it to do. By using some combination of these two processes, the user can let the artificial intelligence make decisions based on the tools it was provided.

The specific branch of AI that this thesis focuses on pertains to the use of agents and setting them up in environments to create simulations. The following few sections try to explain those concepts and how they are used in the practical part of the thesis.

2.1. Agents

There are a lot of different definitions for what an agent is, typically depending on what context the agents are used in and what their purpose within that context is. However, there are certain agreed-upon properties that all agents should have in some capacity, first mentioned by Wooldridge and Jennings[1, p. 116]:

- **autonomy** - agents can control their own behaviour to a certain extent, choosing what actions to take off their own accord when it benefits them the most and also changing their internal state when they need to. This differentiates them from objects found in object-oriented programming that are not able to choose what methods to call or when to change their internal state without outside influence.
- **social ability** - agents are capable of communicating with humans or other agents through some sort of agreed-upon language. This ability enables agents to work together towards accomplishing goals.
- **reactivity** - whereas objects tend to wait for their methods to be called and are not influenced by changes in the outside environment, agents are able to both perceive their surroundings and change their behaviour based on their findings.
- **pro-activeness** - in order to meet their goals, the agents need to be able to take initiative instead of merely waiting for things to happen to them.

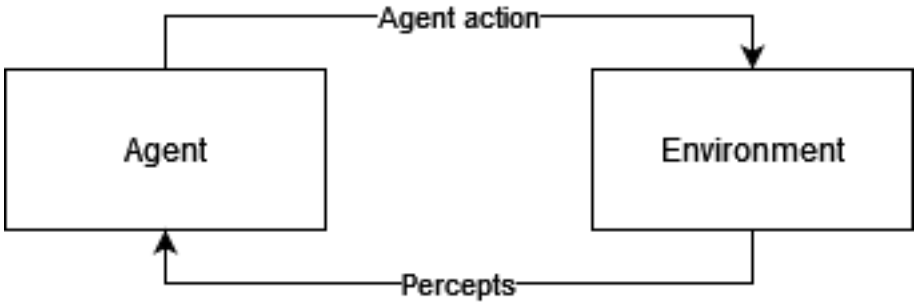

Figure 1: Interaction between an agent and its environment

To be able to have these properties, agents need an environment to exist in and interact with. Their actions need to leave visible results that change their surroundings or their internal state in some way, they need to make and carry out decisions that will help them accomplish their goals and, in the case of multiagent systems, they need to have a way of communicating. As can be seen in figure 1, it is a cycle of the environment being perceived by the agent and the agent reacting to what it sees.

With the purpose of enabling agents to achieve their goals, they are often designed as intentional systems (meaning, according to Dennet[2, p. 49], that their "behaviour can be predicted by the method of attributing belief, desires and rational acumen"). In their article, Wooldridge and Jennings [1, p. 120], claim that "being an intentional system seems to be a necessary condition for agenthood,...". Envisioning agents as intentional systems lets the user view them as a combination of information that the agent has and rules that the agent follows when coming to conclusions. Essentially, agents designed as intentional systems are given the tools they need to understand the world around them and accomplish their goals. It allows them to use the knowledge they have to come to conclusions using first-order logic, finite state machines, Belief-Desire-Intention models, production rule systems or any of the other available decision-making models. The choice of decision-making model depends on which one is best suited to the designer's needs, with new models and derivations being developed constantly.

2.1.1. BDI agents

This thesis focuses specifically on Belief-Desire-Intent agents who, on top of the aforementioned attributes, are built with a particular framework in mind. According to a scientific journal [3, p. 1055], "BDI agent programs are essentially a set of plan rules of the form G : $\Psi \leftarrow$ P, meaning that plan P is a reasonable plan for achieving the goal (or responding to the percept) G when (context) condition is believed true." They further explain that an agent's beliefs are usually pictured as some sort of database, their desires are goals and the plans are placed in some sort of plan library. Figure 2 demonstrates this, upon taking in facts from its environment, the agent uses its interpreter and the beliefs, desires and plans it contains to decide on an action to take.

Designing agents like this makes them look over their knowledge of the world and formulate the most efficient plan they can. Imagine a person who wants to purchase a new book

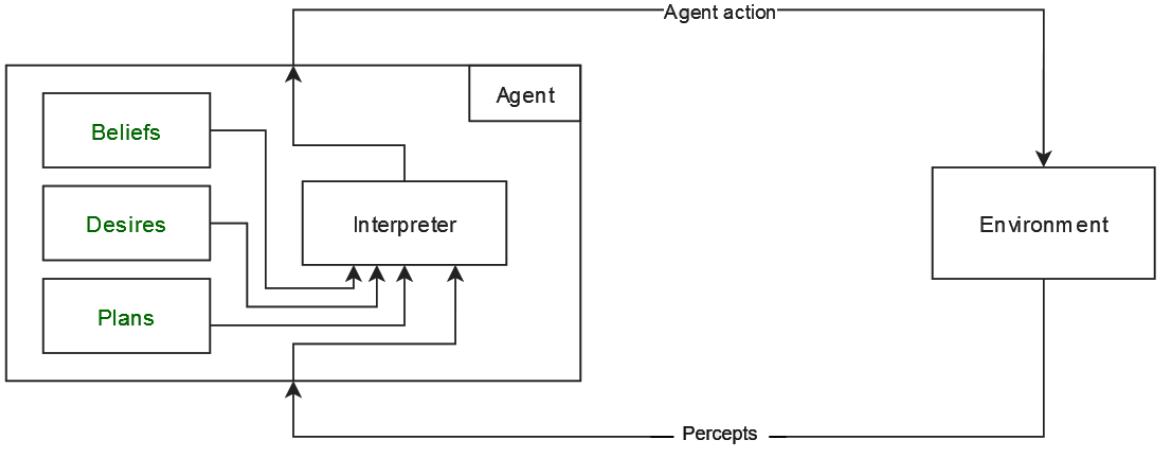

Figure 2: Interaction between a BDI agent and its environment

and the book is available in a nearby bookstore, but the bookstore also offers the option of buying the book through their web shop. Both of these options have their upsides and downsides and it is up to the agent to decide what is more important to them: having the book as soon as possible, saving time, saving money or something else entirely. On top of that, each decision carries with it a few steps that need to be completed before the book can be bought. Buying it in person means taking the time to drive to the bookstore, find it, wait in line and drive back home, with there being the possibility that the book is not even in stock currently. On the other hand, the person could buy the book online and have it delivered to them, but that would mean paying delivery fees and waiting a few days for the book to arrive. A BDI agent would need to go through a similar decision-making process before choosing what it believes to be the best option.

Seeing as they take in their surroundings, Belief-Desire-Intent agents are designed to be used in dynamic environments that shift constantly. Before planning and committing each step, the agents need to reassess their position and rethink their options to see if a potentially better way of achieving a goal exists.

In order to know what the agent needs to be able to do, it is also important to know what kind of environment the agent will find itself in. Understanding all the points that need to be taken into account enables designers to prepare agents more specialized for their target environment. Seeing as this thesis focuses on a convenience store, the following chapters explore what kinds of agents find themselves in such an environment.

2.1.2. Clerks

In order for a store environment to be able to run smoothly, it needs to be manned by a certain amount of clerk agents. These clerks would need to be able to do the following:

- **locate points of interest** - the clerk agent needs to know where specific points of the store are. Specifically, in this envisioned environment, it needs to know

where the cash register, inventory, exit and product containers are. Knowing these locations will enable it to complete its other actions that require interacting with the points of interest.

- **move around the environment** - once the clerk knows where it needs to go, it needs to be able to reach their destination.
- **detect shelves that need stocking** - while moving around, the clerk needs to detect what products are not currently available to the customers, allowing it to note it for later and decide to restock a particular station.
- **know what they are carrying** - the clerks need to be aware of what product they are carrying so that they bring it to the right station.
- **help customers** - the environments are designed to allow for special stations to exist, ones in which customer agents can pick up products that are not merely placed on shelves. A lot of convenience stores have special stations for bread and pastries, dairy products, ham, precooked meals or other such items. Clerks need to detect when a customer wishes to get a product from one of these stations and help them acquire it. The same goes for the cash register, a clerk needs to be there to check out the customers.
- **be aware of other clerks** - in the case of there being more than one clerk agent, it can happen that both of them detect that a customer might need help or that a shelf might need stocking, the clerks need to somehow notify each other of what their intentions are in order to keep others from starting to do the same action.

Outside of what they should be able to do, the clerk agents need to have some sort of plan for how to handle the store environment and be efficient. In order to be able to complete most of their tasks, they need to know the locations of all points of interest, so their first priority should be learning the layout of the store. The second priority should be helping out customers in need, seeing as that way they enable the customer agents to keep performing their own tasks. Lastly, they need to keep the store stocked, meaning that if they find any empty shelves during their patrol, they should fill them up. Essentially, the clerks need to follow a set of guidelines that will help them cover their most important priorities first.

2.1.3. Customers

Customers are agents that want to acquire things throughout the store as quickly as possible and then leave. Their abilities are a bit more simple than those of clerks, but there is some overlap:

- **locate points of interest** - like the clerks, customers need to know where certain points of interest are, specifically the exit, the cash register and the various products.

- **move around the environment** - to be able to reach the points of interest, the agents need to be able to move around the environment.
- **create shopping lists** - customers need to have a way of generating lists of what items they want to pick up from the store, giving them clear ideas of what to look for.
- **mind what they have** - in order to not accidentally take too many of the same item, customers need to be aware of what they are currently carrying on their person.
- **checkout and leave** - once they have found everything they were looking for, the customers need to head for the register and leave the store after being checked out, having achieved their goal.

The first priority of the customers should be knowing what they came in for, the second priority should be leaving upon acquiring all the wanted products. If a certain product is not currently available, the customer should not waste their time by waiting around for it to get restocked and should instead try to find some other items from the list. If a certain product requires the assistance of a clerk, the customer should make it known that they need help and wait at the proper station for a clerk agent to show up. Once everything is acquired, a customer should head for the register and leave the store.

2.2. Environments

Agents need environments to inhabit, interact with and complete their tasks in. To understand how to properly prepare an agent for its environment, the environment itself must be understood first. There is a multitude of properties that can be viewed when analyzing an environment. According to Russell and Norvig [4, p. 41-45], these properties are:

- **observability** - environments can be either fully or partially observable, depending on whether the agent has the ability to sense all aspects of its surroundings, fully observable environments are a lot easier to make agents for due to the agents being able to access more information. Most environments are partially observable, meaning that agents are never fully aware of everything that is going around them. For instance, an agent playing poker knows what cards it holds and knows how powerful they are, but it cannot observe what cards the other players have.
- **agent count** - the amount of agents that populate the environment, it is either a single agent or multiple. The number usually depends on the complexity of the task and what they need to achieve, but increasing the agent count can sometimes lead to its own problems, like needing to find a way to have agents communicate and organize or having them argue and fight over the same resources or conflicting goals.

- **deterministic vs. stochastic** - whereas deterministic environments are only influenced by the actions of an agent, a stochastic environment contains more uncertainty and variables that need to be accounted for, agents in these environments need to be prepared for unexpected changes. If an agent is playing a game like Sudoku, the state of the board will not change until the agent acts. On the other hand, an agent that is merely walking around town can be influenced by traffic, the weather and a lot of other variables that are outside of its control.
- **episodic vs. sequential** - based on whether an agent's actions depend on its past experience or not, an environment is either episodic or sequential. Agents do not need to be aware of their past or future actions in episodic environments, whereas their actions in sequential environments can reflect on all their future decisions. An agent that paints does not need to worry about its previous or future paintings whilst working on the current piece of artwork, but an agent that works in some sort of construction company might cause a building to collapse if it skips a crucial step in the process.
- **static vs. dynamic** - static environments do not change themselves without the agent's influence, an agent can take as much time as it wants to make a decision, formulate a plan and go through with it. Dynamic environments progress even if the agent chooses to do nothing. If an agent is deliberating on whether or not it wants to buy a product at the store, it is possible that some other agent shows up and takes the product before a decision is made, changing up the first agent's variables and making it have to reconsider its plans.
- **discrete vs. continuous** - whereas discrete environments have a preset set of precepts and actions, continuous environments are harder to predict and have way more variables that need to be accounted for, making them harder to prepare agents for. Like in the earlier Sudoku example, the agent only has a set number of moves it can make to progress the game if it is following the rules, whereas the agent moving around town constantly needs to make different decisions relating to where it is moving and what it is choosing to do in relation to the other agents and obstacles.
- **known vs. unknown** - known environments are ones that the agent and their designer know all the rules and possible outcomes of, while unknown environments are not fully known to either the agent or their designer, meaning that the agent will need to adapt as best it can to an unknown environment. As in the poker example, the agent cannot observe the opponents' cards, thus it does not know whether it will win or lose the game if it decides to play. If the agent was fully aware of what would happen in both scenarios, the environment it finds itself in would be fully known.

By assessing all of these properties, along with the problem that needs to be solved in the given situation, it can be determined what kinds of agents are needed. Various combinations can offer varying degrees of difficulty when it comes to designing agents so it is important to try and get as many details as possible before beginning the agent design process.

2.2.1. Convenience store

The practical part of the thesis simulates a simplified version of a convenience store and its usual routine. Some parts are less complex than what they are like in a real-world scenario, but the general idea of a store being a place where two types of agents exist (customers and clerks) is still present. The agents and their necessary abilities were mentioned in the earlier sections.

The convenience store needs to have certain stations that allow it to operate properly and enable the agents to accomplish their tasks:

- **entrance/exit** - both clerks and customers need to appear in the environment and leave the environment from some spot.
- **cash register** - the cash register is the spot at which clerks can check customers out, letting them leave the store. It is the last station customers must visit before leaving.
- **containers for products** - the store needs to have shelves, freezers, boxes or other types of containers that can store items.
- **inventory** - once customers empty out the containers, they need to be restocked. An inventory needs to exist to enable clerks to restock the store.
- **unique stations** - some products are not readily available for purchase and require the help of a clerk agent.

Now that the requirements of the store are known, it is possible to deduce what kind of an environment the simulated convenience store is. Using the earlier attributes, the convenience store is:

- **fully observable** - all of the environment is observable, both the clerks and customers should be able to locate all the points of interest and be able to understand what they are.
- **multiagent** - in order for the convenience store routine to be played out normally there need to be multiple agents in the environment.
- **deterministic** - the store environment can only be changed by the actions of the agents, there are no outside factors that can potentially influence the outcome.
- **sequential** - depending on what actions an agent does, they will need to complete some other actions afterward. For instance, if a clerk agent decides to restock a shelf, they will need to go to the store's inventory, find the proper box, return to the product's shelf and then restock it.

- **dynamic** - the environment can change while the agents are making decisions, a shelf might run out or get restocked, a customer might appear at a station or at the register, a new customer may appear and so on. All of these moments can change whilst someone is deciding what to do.
- **discrete** - the environment is made from scratch and all the agents that populate it only have a set amount of actions they can take. While a real convenience store might be continuous, the one described here would be discrete.
- **known** - with the environment being made in a game engine, the designer is be aware of its rules. The agents have a set amount of actions they can take and are prepared for all the outcomes. Again, it is different than a real-world version of a convenience store.

The agents can be designed with a lot of knowledge of this environment as there are not a lot of unknown variables in it. The agents' efficiency will mostly rely on what they decide to do. If customers require help often at special stations, the clerks will take more time between restocking, which in turn might make some customers look around for the products they wish to collect longer. The environment is pretty simple and easily observable overall, with the agents only needing to be able to "see" to be able to find their way in it.

2.3. Agent-Based Models

As mentioned earlier, a common use for agents is playing roles in simulations, offering their designer a simplified version of some real-world actor in a scenario. As as a journal notes [3, p. 1054], "In agent-based modelling (ABM), the system of interest is conceptualised in terms of agents and their interactions." They go on to explain that the model evolves as the internal states of agents change, pushing the simulation further. Every agent that participates in the model has to make a decision at every step of the simulation based on what they know about the world and what they are able to accomplish. One agent's decision might influence the next decision of all other agents, thus impacting the next step of the simulation and heavily changing the entire model.

Using agents in this way allows the user to, for instance, take an environment such as traffic. It contains roads, traffic lights, crossing walks and other obstacles. The roles of pedestrians, cyclists and drivers can be played by agents, which all have their own goal of reaching some destination. Traffic has certain rules that need to be followed, like waiting at a stop sign or a red light, letting pedestrians pass or deciding who has the right of way. The agents that play roles in this model need to be able to perceive the environment to understand what position they are in, what position the other agents are in and to read the current state of their surroundings. Using this type of model, the user can measure its efficiency and adjust some parameters. The ultimate goal is for each agent to reach their target location as quickly as possible. By adjusting timers on traffic lights, adding lanes, potentially banning certain types of vehicles or changing speed limits, the simulation tests the customized environment and allows

its users to compare it to the existing results. Finding a more efficient setup can potentially allow the user to apply the changes to some real-life roads.

One real world example of Agent-Based modelling being used is SEARUMS [5], a software environment built to model and simulate outbreaks of Avian Influenza. In this project, agents played the roles of waterfowl flocks, poultry flocks and humans. SEARUMS lets their users configure certain attributes of the agents (geographic, migratory and statistical) and then play out simulations that display likely scenarios and potential outbreaks of the virus. Using the information gained from this model, researchers could predict potential outbreaks and be more readily prepared for them.

Examples such as these showcase why agent-based models are useful and why they should be employed more frequently in the future, they allow designers to focus on smaller roles that play key roles in the entirety of the model. With this approach, it is easier to play out more complex scenarios through agent interaction.

3. Convenience store simulation

The following chapter describes the technologies used in implementing the convenience store environment, the implementation itself, the results of some of the ran simulations and potential improvements that could be added to the project in the future.

3.1. Technologies used

The following sections list the technologies that were used in the creation of the virtual store environment.

3.1.1. Unity

Unity[6] is a cross-platform game engine. It is free to use and very accessible, leading to it being a popular choice for entry-level programmers who are trying to learn game development. Its non-existent price tag and popularity also lead to it having a large community and a wide range of tutorials. It comes packaged with a lot of tools and potential extensions that can help in development. The source code of the engine is written in C#, meaning that all of the usual programming paradigms that the C# language offers (object-oriented programming, encapsulation, etc.) are available in the engine.

The engine offers tools that allow the creation of game objects for the agents and environments. Each game object can be further customized with various components that enable them to perform their actions. The developers are given a lot of freedom in how they want to address or combine certain game objects.

This thesis uses the basic tools available in the Unity editor (version 2021.3.28f1) and a free package named "Cinemachine" which offers the user a lot more options when it comes to using and controlling cameras.

3.1.2. Prolog

Prolog is a logical and declarative programming language that is based on predicate logic. According to Clocksin and Mellish [7, p. 3], computer programming in Prolog consists of:

- specifying facts about objects and their relationships
- defining rules about objects and their relationships
- asking questions about objects and their relationships

In the case of a convenience store, facts can be asserted about where products are in the environment, what agents are looking for, which customers are waiting at some spot and

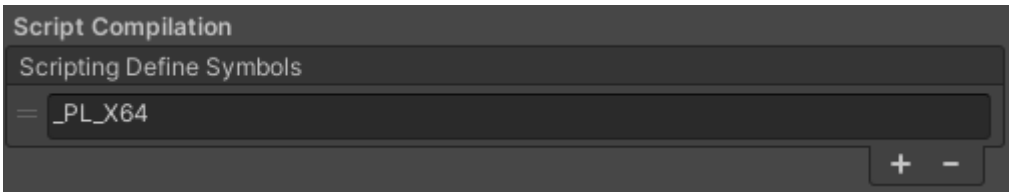

Figure 3: Defining a variable in the Unity project

other facts or happenings in the environment. When choosing what to do, an agent should need to ask questions about the environment.

To be able to use Prolog in Unity, this project employs a bridge that connects C# with SWI_Prolog's engine.

3.1.3. SwiPIC

This thesis utilizes a SWI_Prolog bridge named SwiPIC, created by Uwe Lesta[8]. The bridge has last been updated to work with SWI_Prolog version 8.0.3., which needs to be installed on the PC of the project. According to the official documentation[8], which can be reached through the repository of the project, "The described interface provides a layer around the C-interface for natural programming from C#. The interface deals with automatic type-conversion to and from Prolog, mapping of exceptions and making queries to Prolog in an easy way."

In order for the bridge to work, the proper version of Prolog needs to be fully set up, the SWIPIC folder from the repository needs to be added to the Unity project's assets and, as seen in figure 3, a variable needs to be defined in the environment of the Unity project.

After implementing it, the interface lets the developer use the Prolog engine directly in C# code.

There were a few complications and bugs that appeared throughout development, with the engine returning errors during assertion or querying instead of the expected results. The engine is also active from first initialization so it is important to clean up between starting the simulation over. This also means that any game object can access the Prolog engine and knowledge base if the proper library is being used in the script.

3.1.4. Blender

Blender[9] was used to create some simple models for the environment. It is a free modelling tool that allows the user to create models. The models can later be exported and used in Unity projects.

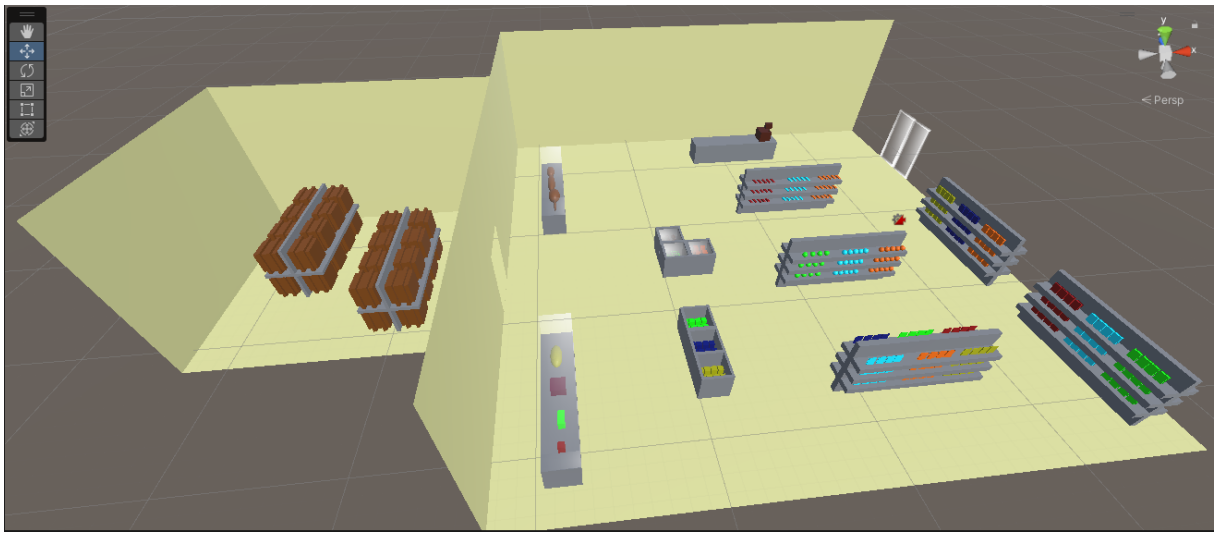

Figure 4: Image of the store

3.2. Implementation

The following sections cover the implementation of the virtual store environment, as well as the agents and the scripts that were created to make the multiagent system function.

3.2.1. Store environment

The models for the building, the furniture, the products and the agents were created in Blender and then later imported into the Unity project. The store has two rooms: the main room and the inventory. The inventory is made up of two shelves filled with boxes. The main room has sliding doors that are used as the entrance and exit, a checkout table that holds the cash register, two special stations that require clerks to assist customers when acquiring a product and containers for products that hold items that can be freely picked up. Figure 4 shows what the store looks like in the Unity project.

3.2.1.1. Store setup

The store itself has a script component that allows the user to define some variables through serialized fields that will impact how the store operates. The variables, as seen in figure 5, are:

- **number of clerks** - this setting influences how many clerks will spawn at the start of the simulation
- **number of customers** - sets how many customers need to complete their shopping before the simulation ends
- **view distance** - decides how far the agents can see

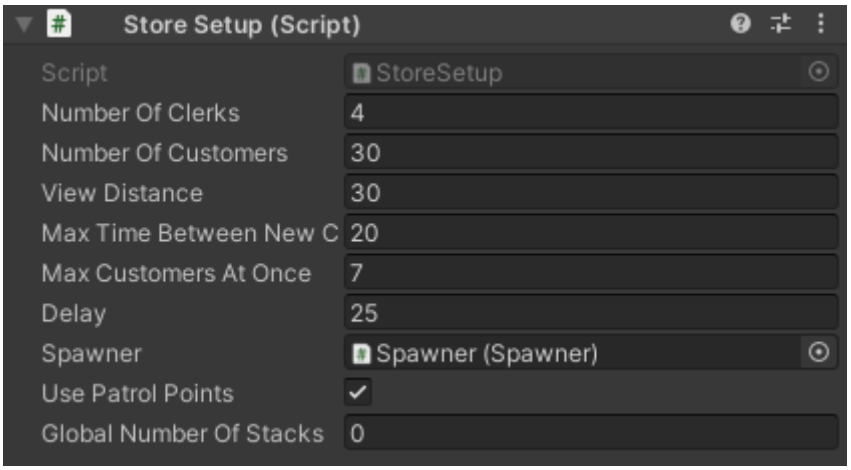

Figure 5: Store variables

- **max time between new customers** - sets how long the spawner will wait before creating a new customer agent
- **max customers at once** - sets how many customers can be in the store at the same time
- **delay** - seconds that need to pass before customers start spawning in, giving clerks some time to prepare
- **spawner** - the spawner game object that is required for the simulation to work
- **use patrol points** - if this field is checked, the agents will use points around the store that were preset for moving around, otherwise they will choose random points
- **global number of stacks** - influences the number of items of the product at all places in the store. If set to 0, every container controls its own maximum number of items

Some of the variables will be described in more detail as their game objects are being covered.

The store's script starts the simulation and passes all the serialized fields along to the GameObjects and their components. During the Awake() method of the script, visible in listing 1, the Prolog engine is initialized and all of the locations of the various items in the store are asserted into the knowledge base. Once the knowledge base is loaded, the spawning process can begin.

Listing 1: Store's script's Awake() method

```
1      private void Awake()
2      {
3          bool everythingOkay = SetKnowledge();
4
5          if (everythingOkay && viewDistance >= 20 &&
6              viewDistance <= 50)
```

```
7           {
8               everythingOkay = true;
9           } else
10          {
11              everythingOkay = false;
12          }
13
14          if (everythingOkay)
15              StartSpawningProcess();
16      }
```

The SetKnowledge() function, as seen in listing 2, initializes and cleans up the Engine if there was a previous session of the simulation. The Unity engine allows the user to find GameObject objects by their components, meaning that any object can be accessed by a script component they have.

Listing 2: The method that sets the store's knowledge

```
1           private bool SetKnowledge()
2       {
3           bool everythingOkay = true;
4           PrologInitialize();
5           RetractExisting();
6           if(everythingOkay)
7               everythingOkay = SetCashRegisterLocation();
8           if (everythingOkay)
9               everythingOkay = SetEntranceLocation();
10          if (everythingOkay)
11              everythingOkay = SetDispensers();
12          if (everythingOkay && usePatrolPoints)
13              everythingOkay = SetPatrolPoints();
14          return everythingOkay;
15      }
```

Listing 3 is an example of how an object's location is asserted into the knowledge base:

Listing 3: Example of a method that asserts a location

```
1           private bool SetEntranceLocation()
2       {
3           Spawner spawner = GameObject.FindObjectOfType<Spawner>();
4           if (spawner != null)
5           {
6               string location = spawner.transform.position.ToString();
7               PlQuery.PlCall("assert(location(exit, " +
8                   location + "))");
9               return true;
10          }
11          else
12          {
13              UnityEngine.Debug.Log("Missing entrance and exit");
14              return false;
15          }
16      }
```

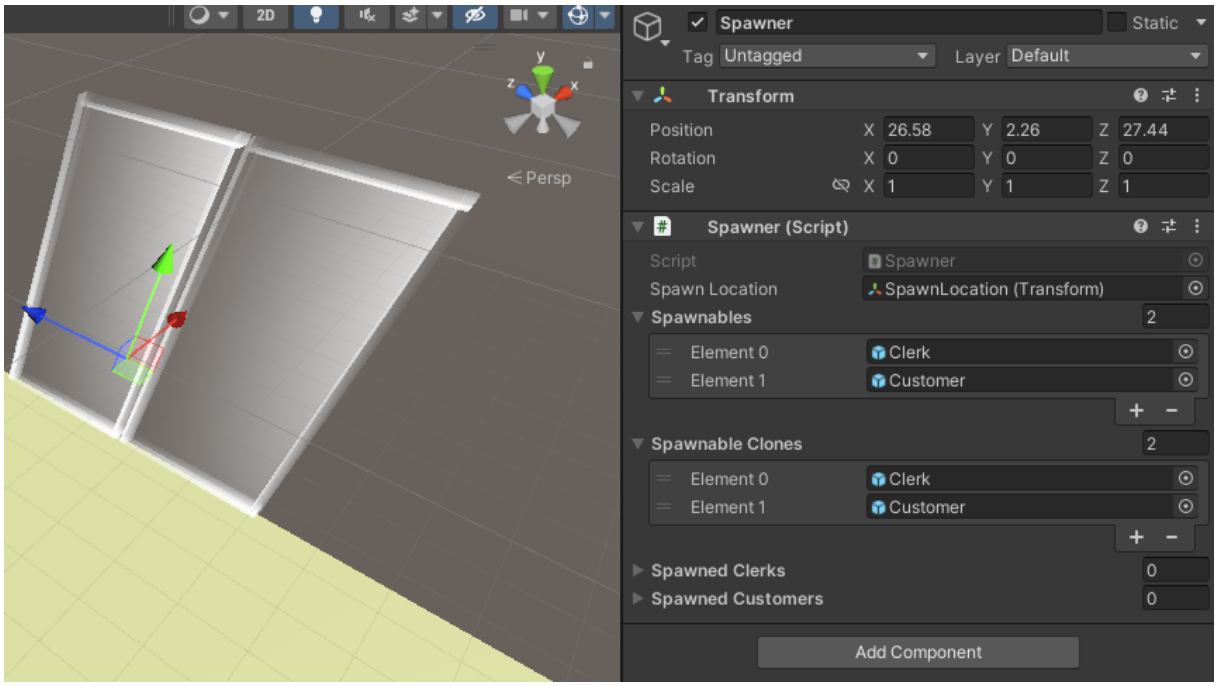

Figure 6: Spawner game object

Along with the entrance location, the locations of all the objects that agents can interact with are asserted as well, setting up all the current facts about the environment and making it easier later on to give agents that knowledge. If all the required facts are asserted and the conditions are met (there needs to be an entrance, a cash register and every product needs to have a box in the inventory), the store starts the spawning process, which is handled by the Spawner game object.

3.2.1.2. Spawner

The store has a GameObject, Spawner, that is used to spawn and destroy agent objects. The spawner needs to be assigned a Transform object that will serve as the location in which the agents will be spawned. In this scenario, the Transform component of the GameObject is placed in front of the sliding door model, as seen in figure 6. Along with a spawn location, the spawner has two GameObject arrays that both contain two prefabs for the agents, one is for the clerk agents and the other is for the customer agents. There are also two arrays for the already spawned clerks and already spawned customers. The spawning process was made following a guide[10].

The Spawning function is a coroutine, which means that the engine will allow the user to pause the process for certain amounts of time, letting the user add delays when necessary, like giving clerks a head start before customers start entering the store or randomizing the time before a new customer shows up. Once the set amount of customers visited and left the store, all the clerks leave the store and the simulation ends. This process is visible in listing 4.

Listing 4: The Spawner's main method

```
1     IEnumerator Spawning (int numberOfClerks, int delay,
```

```
    int numberOfCustomers, int maxTimeBetweenNewCustomers,
    int maxCustomersAtOnce, int viewDistance)
{
    yield return new WaitForSeconds(2);
    SpawnClerks(numberOfClerks, viewDistance);
    yield return new WaitForSeconds(delay);
    for (int i = 0; i < numberOfCustomers;)
    {
        if(spawnedCustomers.Count >= maxCustomersAtOnce)
        {
            yield return new WaitForSeconds(20);
        } else
        {
            SpawnCustomer(i, viewDistance);
            i++;
            yield return new WaitForSeconds(
                RandomTimeToWait(
                maxTimeBetweenNewCustomers));
        }
    }
    while (spawnedCustomers.Count > 0)
    {
        yield return new WaitForSeconds(10);
    }
    DestroyClerks();
    while (spawnedClerks.Count > 0)
    {
        yield return new WaitForSeconds(10);
    }
    Application.Quit();
}
```

As shown in listing 5, each clerk or customer is assigned its own identification number and receives the view distance that was assigned during the store setup phase.

Listing 5: The method that spawns a customer agent

```
    internal void SpawnCustomer(int id, int viewDistance)
{
    spawnableClones[1] = Instantiate(spawnables[1],
        spawnLocation.position, Quaternion.Euler(0, 0, 0));
    spawnedCustomers.Add(spawnableClones[1]);
    spawnedCustomers[spawnedCustomers.Count - 1].
        GetComponent<Customer>().SetId(id);
    spawnedCustomers[spawnedCustomers.Count - 1].
        GetComponent<Customer>().SetViewDistance(viewDistance);
}
```

The Spawner class also has two public methods that allow agents to destroy themselves when their goals are complete, shown in listing 6. The agents themselves call these methods and send their IDs to the Spawner so it can know which particular agent it wants to remove from its array.

Listing 6: The method that destroys a customer based on the customer's id

```
1      public void DestroyCustomer(int id)
2  {
3      int index = -1;
4      foreach(var c in spawnedCustomers)
5      {
6          if (c.GetComponent<Customer>().id == id)
7          {
8              index = spawnedCustomers.IndexOf(c);
9          }
10     }
11     GameObject customer = spawnedCustomers[index];
12     Destroy(customer);
13     spawnedCustomers.RemoveAt(index);
14 }
```

3.2.1.3. Dispensers

Dispensers play an important part in the store's routine as most of the objects that the customers and clerks interact with are extensions of this class. The base Dispenser class only has the product field, which dictates what product it stores. The class has three extensions:

- **stock dispensers** - these hold the product stock in the inventory and are the simplest extension. They only have a single function that returns the product that had been assigned to them.
- **product dispensers** - the containers that can hold products are the product dispensers, they have a max amount of items they can hold when they are fully stocked. This can either be assigned globally or each dispenser can have its own fully stocked amount. Along with attributes that keep tabs on how many items there are in the current stack, product dispensers have a function that allows clerk agents to restock them, a function that lets customers take one item off of the stack and a function that enables renders for the same number of products that are currently on the stack. Figure 7 demonstrates the look of a product dispenser and its variables.
- **special station dispensers** - these are similar to product dispensers, but they are meant to be used on special stations that require the assistance of some clerk, meaning that the customers cannot get the product on their own.

The amount of stock dispensers needs to be equal to the amount of product dispensers plus special station dispensers, as every product needs a place to take stock from.

3.2.1.4. Cameras

The environment has three cameras set up throughout the store that can be operated to get different angles. There is a simple script that lets the user switch between views by using the left and right arrow keys.

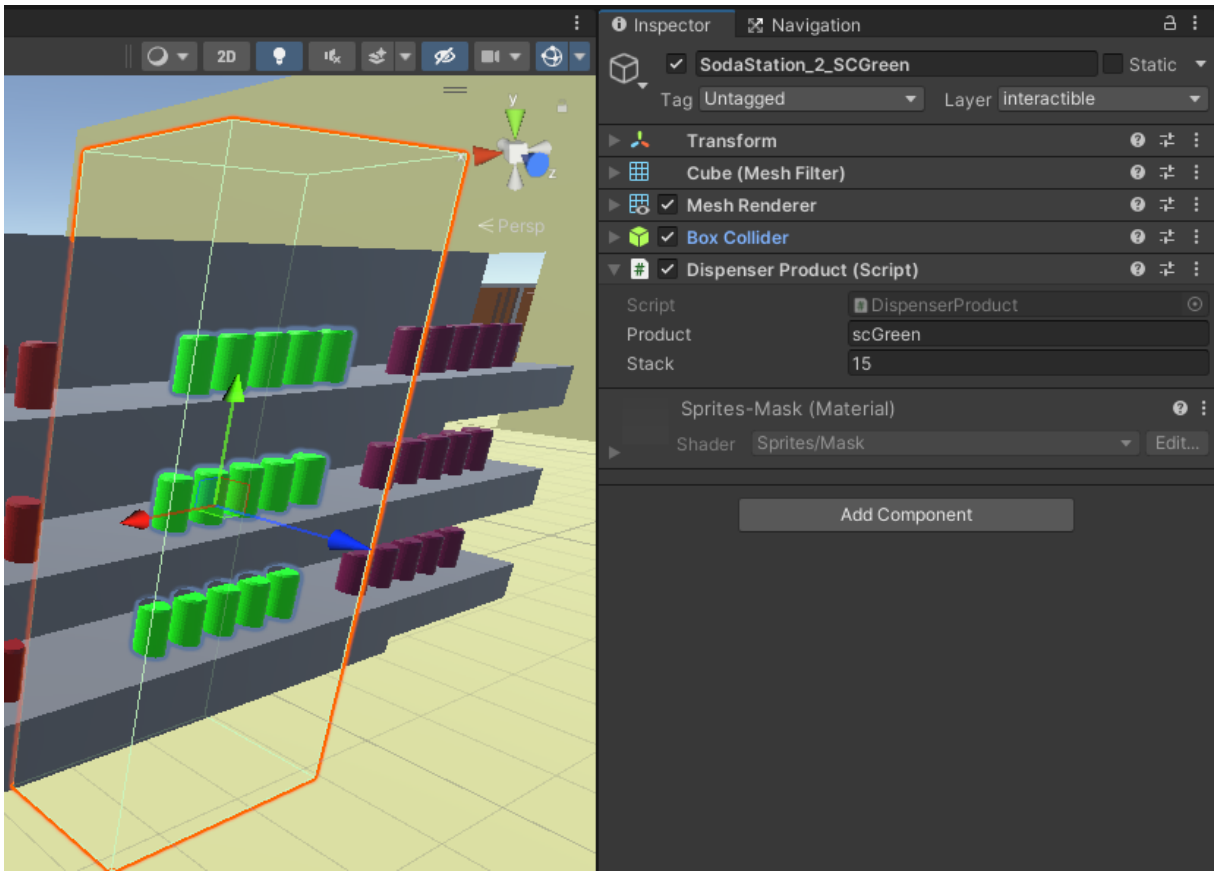

Figure 7: A dispenser for green soda products, applied to a shelf model

3.2.2. Agents

The rest of the simulation is handled by the agents and the choices they make. Agents are saved as prefabs, meaning that they are a "GameObject complete with all its components, property values, and child GameObjects as a reusable Asset"[11]. Using prefabs allows the designer to easily instantiate and destroy new objects as necessary. As mentioned earlier, the Spawner uses these as it completes its methods.

The agents use the Prolog database to assert their beliefs. With it, they know where objects in the environment are, relative information about other agents and relative information about the store. Using this information, they select one of their plans, which are implemented in the form of switch methods. Each step of the plan is one case in the switch method, with the agents keeping track of what plan they are executing and what step they are on with internal strings and counters respectively, seen in listing 7. After they complete a step of the plan, the counter is increased, leading to the next switch case, which is the next step in the plan they are executing. The agents' decisions on what to do and which plan to select rely on their existing knowledge and what they see within their view distance. These switch case methods are shown in the agents' subsections.

Listing 7: strings and a counter that help the agent in knowing what it is doing

```
1     private string currentTask;
2     private string lookingFor;
```

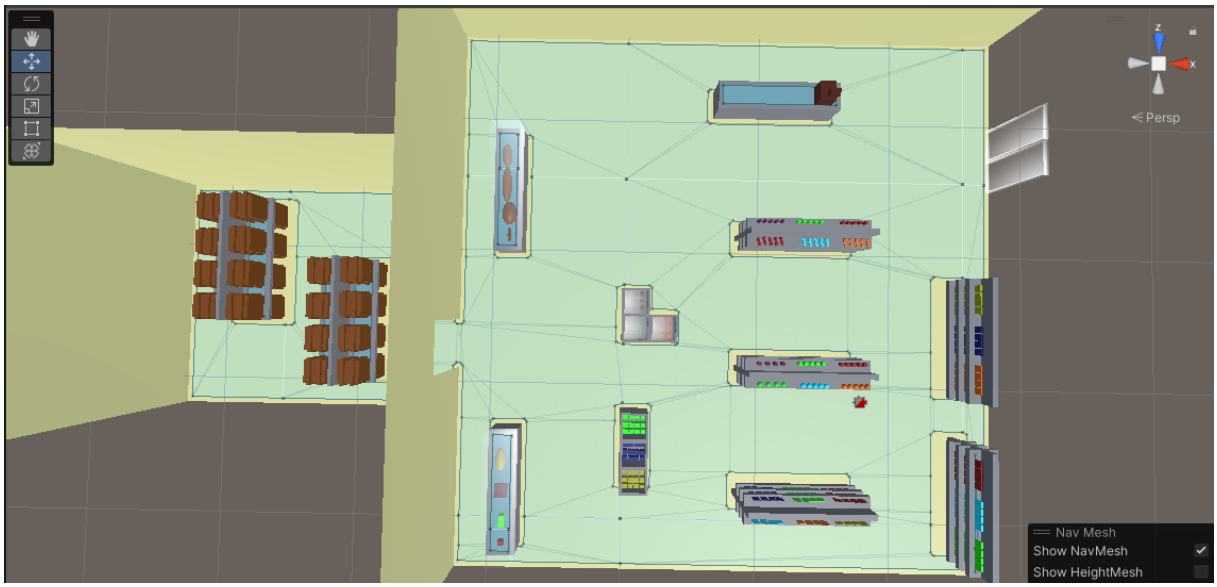

Figure 8: Navigation mesh of the store

```
3       private int step;
```

In the case of clerks, their first plan is to find the points of interest that they need to be able to complete their other plans. Knowing where the inventory is allows them to restock, knowing where the cash register is allows them to check customers out and knowing where the special stations are allows them to serve customers. After that, their main and secondary goals are to make sure that customer agents are not waiting at any stations and that all the product containers are stocked. This can be seen through their patrolling plan, which serves to show what their priorities are.

Customer agents' main goal is to get all the items from their list and leave. They only need to learn where the cash register is, which enables them to check out and leave. After learning where the register is, they walk around the store, looking for containers of items they want. Upon finding a container that holds one of the items they want, they will start executing a plan to try and get the item.

3.2.2.1. Navigation

Unity has a built-in system for navigation. With the help of a tutorial[12] and the manual [11], a Navigation Mesh of the store was created, "a data structure which describes the walkable surfaces of the game world and allows to find a path from one walkable location to another in the game world. The data structure is built, or baked, automatically from your level geometry." To create a NavMesh, all the static objects in the environment had to be marked as such before the baking process. The result of the baking is a "map" that shows where the future Navigation Mesh Agents will be able to walk, as can be seen in figure 8.

Once a Navigation Mesh is created, the game objects that are going to interact with it need to have a NavMeshAgent script component added to them. These agents can avoid obstacles and each other while moving throughout the environment.

The NavMesh Agent script contains methods needed for the agents to move around the environment, visible in listing 8, both the clerk and the customer prefab come with it as a component. NavMeshAgents have a "destination" property that dictates what location they are trying to reach. When they have a destination that they are not currently standing on, they start moving towards it using the NavMesh to navigate the environment. The destination needs to be a Vector3 type.

Listing 8: The method that sets an agent's destination

```
    public void SetDestination(string position)
{
    position = position.Replace("(", "");
    position = position.Replace(")", "");
    string[] positionParts = position.Split(',');
    Vector3 v3 = new Vector3();
    v3.x = float.Parse(positionParts[0].Trim(),
        CultureInfo.InvariantCulture.NumberFormat);
    v3.y = float.Parse(positionParts[1].Trim(),
        CultureInfo.InvariantCulture.NumberFormat);
    v3.z = float.Parse(positionParts[2].Trim(),
        CultureInfo.InvariantCulture.NumberFormat);
    agent.destination = v3;
}
```

There is also a method that checks whether the agents are close enough to their destination to interact with the object at the end. Agents cannot do anything while they are moving and the distance between them and their destination cannot be 0, as that would mean that they are inside the object. Due to this, if the agent is within 3 distance units of their destination, it counts as them reaching it. This process is shown in listing 9.

Listing 9: The method that checks if an item has been found

```
    public bool CheckIfIFoundIt()
{
    bool stillLooking = true;
    float dist = Vector3.Distance(agent.destination,
    agent.transform.position);
    if (dist < 3)
    {
        stillLooking = false;
    }
    return stillLooking;
}
```

Lastly, there are a few methods that get random positions around the store, letting agents change their positions to try and find what they need. If patrol points are not enabled, these positions are randomly selected by a built-in NavMesh function that returns a random destination from the mesh. The distance that the agents can see is used in calculating the destination. The method that calls uses these random destinations is visible in listing 10.

Listing 10: The method that uses the NavMesh to get a random destination

```
    public void GoToRandomLocationInStore(float viewDistance)
{
    if (patrolPointsExist)
    {
        SetDestination(GetSomePatrolPoint());
    } else
    {
        Vector3 randomDirection = UnityEngine.Random.insideUnitSphere
            * viewDistance;
        randomDirection += transform.position;
        NavMeshHit hit;
        NavMesh.SamplePosition(randomDirection, out hit,
            (viewDistance * 2) - (viewDistance / 2), 1);

        agent.destination = hit.position;
    }
}
```

On the other hand, if patrol points are enabled, one of them will be selected randomly as the destination using the method in listing 11. This usually results in the simulation speeding up a bit because the agents do not get stuck due to having bad luck with the random point selection. The patrol points are saved in the Prolog knowledge base, so they need to be retrieved from it.

Listing 11: The method that uses patrol points to get a random destination

```
private string GetSomePatrolPoint()
{
    string returnMe = "";
    int patrolPoint = 0;
    using (PlQuery q = new PlQuery("patrolPoint(_,_)"))
    {
        int numberOfPatrolPoints =
            q.SolutionVariables.Count<PlQueryVariables>();
        patrolPoint = new System.Random().Next(
            1, numberOfPatrolPoints+1);
    }
    using (PlQuery q = new PlQuery("patrolPoint("
        + patrolPoint + ",C)"))
    {
        foreach (PlQueryVariables v in q.SolutionVariables)
        {
            returnMe = v["C"].ToString();
        }
    }
    return returnMe;
}
}
```

The rest of the agents' behaviours are unique to each type, so they will be covered in their own subsections.

3.2.2.2. Clerks

Clerks are tasked with keeping the store running. Upon entering the store, their first priority should be learning where everything is. They automatically learn where the exit is seeing as they come in from it. Once it starts, it calls an Update() function, visible in listing 12, every frame.

Listing 12: The Clerk's Update() method

```
1      private void Update()
2  {
3      if (currentTask == "")
4      {
5          StartDoingSomething(DecideWhatToDoNext());
6      }
7      else if (!moving)
8      {
9          KeepDoingSomething();
10         moving = true;
11     }
12     else
13     {
14         moving = agent.CheckIfIFoundIt();
15     }
16 }
```

The agent checks whether it has a plan it is following currently. If there is no plan currently being followed, the agent will decide to do something. If the agent has a plan but is currently moving, it will not do anything seeing as it needs to reach its destination first. If the agent has a plan and is not moving, it is going to take the next step of the plan it is carrying out.

The DecideWhatToDoNext() function, as shown in listing 13, just checks whether the agent knows the store. If the clerk knows the store, it is capable of patrolling it, otherwise, it needs to learn the store first.

Listing 13: The method that helps the clerk agent decide what to do

```
1      private string DecideWhatToDoNext()
2  {
3      string choice = "";
4      if (DoIKnowTheStore())
5      {
6          choice = "patrol";
7      } else
8      {
9          choice = "learn";
10     }
11
12     return choice;
13 }
```

The function StartDoingSomething() prepares the plan that the agent decides to use. Based on what the choice is, a different starting function is called that prepares the agent's

internal state for the execution of the plan. The function is visible in listing 14, with an example of one of the subfunction being shown in listing 15.

Listing 14: The method that prepares a plan depending on the agent's choice

```
1      private void StartDoingSomething(string choice)
2  {
3      switch (choice)
4      {
5          case "learn":
6              LearnStore();
7              break;
8          case "patrol":
9              PatrolStore();
10             break;
11         case "manCheckout":
12             ManTheRegister();
13             break;
14         case "stock":
15             StockShelf(lookingFor);
16             break;
17         case "manStation":
18             ManTheSpecialStation();
19             break;
20         default:
21             break;
22     }
23 }
```

Upon entering the store, the clerks will need to learn where all the points of interest are. As mentioned, the location of the exit is acquired right away, so learning the store means finding the cash register, the inventory and the special stations. It is assumed that all stock dispensers are in the inventory.

Listing 15: An example of a method that sets up a plan

```
1      private void LearnStore()
2  {
3      currentTask = "learn";
4      step = 1;
5  }
```

Each plan has a certain number of steps and a name. This allows the agent to know where to continue from during his next Update() method call. The stack of a plan's steps is represented by a switch function that directs the agent's actions. In the example of the learn task, the plan looks as follows, shown in listing 16:

Listing 16: An example of a plan realized with the use of steps and a switch case

```
1      private void KeepLearningStore()
2  {
3      switch (step)
4      {
5          case 1:
```

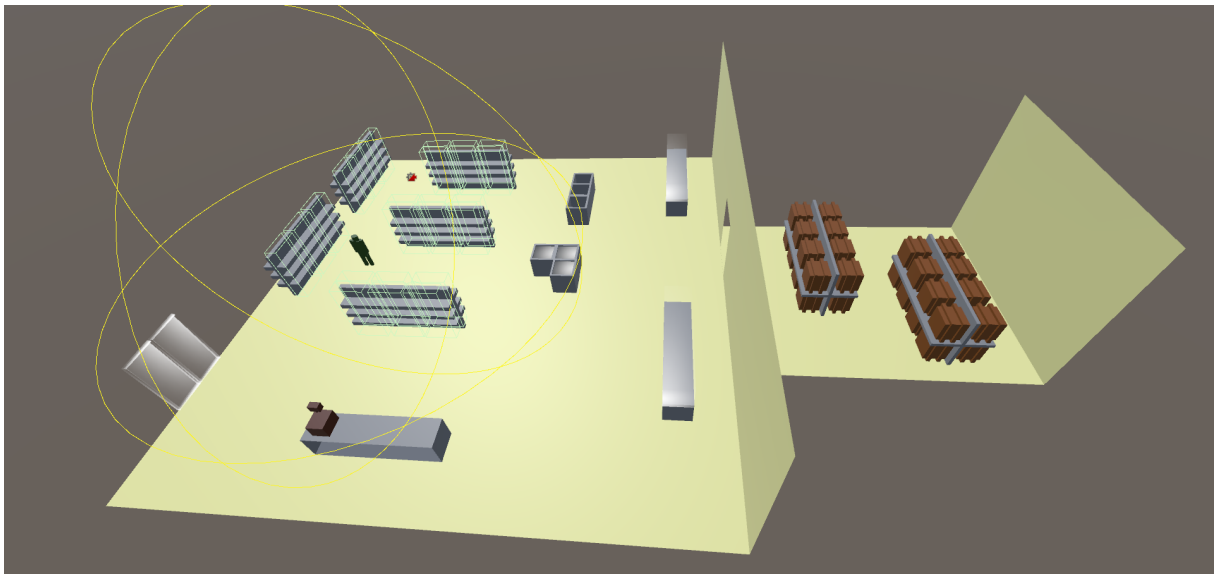

Figure 9: The agent's view distance is shown with a yellow sphere, it can only see things within the sphere

```
6               FindCashRegister();
7               break;
8           case 2:
9               FindInventory();
10              break;
11          case 3:
12              TryToFindSpecialStations();
13              break;
14
15          default:
16              break;
17      }
18  }
```

When **learning the store**, the agent has set priorities on what to find first. The cash register enables them to check out customers so it needs to be found first. The inventory enables them to restock the store, so it is the second thing they need to find. In the case they exist, the special stations are the last things agents need to find, seeing as some customers will not be slowed down by them not being known yet.

Finding things in the store is done by checking whether the object that is being looked for is within the agent's view distance. The view distance is a radius around the agent, visible in figure 9. Anything within the radius is visible to the agent and the agent can learn where it is. The location of the object being looked for is taken from the knowledge base that was filled up in the StoreSetup script. The agent also takes its own location and, using the Vector3 class' built-in function for checking the distance between two locations, it measures how close it is to the object, seen in listing 17.

Listing 17: An example of a method that locates an object

```
1       private void FindCashRegister()
```

```
{
    lookingFor = GetLocationOfCashRegister();
    Vector3 positionClerk = transform.position;
    Vector3 v3 = CreateVectorFromString(lookingFor);
    float distance = Vector3.Distance(positionClerk, v3);
    if (distance < viewDistance)
    {
        step = 2;
        lookingFor = "";
        GiveAgentKnowledge("register");
    }
    else
    {
        moving = true;
        agent.GoToRandomLocationInStore(viewDistance);
    }
}
```

When the agent finds what it is looking for, it acquires the knowledge about the point of interest's location and moves on to the next step of the plan. Otherwise, it keeps moving throughout the store until it finds the point of interest. The GiveAgentKnowledge() function, shown in listing 18, writes down what the agent knows to the knowledge base.

Listing 18: The method that gives agents specified knowledge

```
    private void GiveAgentKnowledge(string v)
{
    switch (v)
    {
        case "register":
            PlQuery.PlCall("assert(agentKnows("
                + id.ToString() + ",cashRegister))");
            break;
        case "exit":
            PlQuery.PlCall("assert(agentKnows("
                + id.ToString() + ",exit))");
            break;
        case "dispenserStock":
            PlQuery.PlCall("assert(agentKnows("
                + id.ToString() + ",dispenserStock))");
            break;
        case "specialStation":
            PlQuery.PlCall("assert(agentKnows("
                + id.ToString() + ",dispenserSpecialStation))");
            break;
        default:
            break;
    }
}
```

This process continues until the "learn" task is complete and the store is fully known to the clerk. Once that happens, it will begin to patrol the store.

When **patrolling** the store, the agent prioritizes checking the cash register first, as helping a customer that is waiting there will help that customer complete their goal, allowing them to leave and make room for a new customer agent. The simulation ends after a certain amount of customers visited the store so it is also in the clerk's interest to have as many customers pass as quickly as possible. After that, the clerk needs to check if someone is waiting at one of the special stations, seeing as that is yet another point where it can help another agent keep accomplishing their goal. Finally, it will check to see if any product in the vicinity needs to be restocked. If not, it will move to a different position in the store and start the plan over again. These steps can be seen as switch cases in listing 19.

Listing 19: Steps of the patrolling plan realized as a switch case

```
1     private void KeepPatroling()
2 {
3     switch (step)
4     {
5         case 1:
6             CheckRegister();
7             break;
8         case 2:
9             CheckSpecialStations();
10            break;
11        case 3:
12            CheckIfNeedsRestocking();
13            break;
14        default:
15            break;
16    }
17 }
```

The Check functions, one of which is shown in listing 20, help the agent realize whether it is needed at a certain station. If an agent is close enough to the station to see it, it will check if there is a customer that needs assistance and move on if there is none. If the agent cannot see the station yet, it will move closer to it.

Listing 20: An example of a method that checks a point of interest

```
1     private void CheckRegister()
2 {
3     if (AmIInRange("cashRegister"))
4     {
5         if (IsSomeoneWaitingAt("cashRegister")
6             && !IsSomeoneManning("cashRegister"))
7         {
8             Reset();
9             StartDoingSomething("manCheckout");
10        } else
11        {
12            step = 2;
13            lookingFor = "";
14        }
15    } else
```

```
16      {
17          MoveCloserToThing(lookingFor);
18      }
19  }
```

The AmIInRange() method measures the distance between the agent's location and the location of the point of interest. The two functions IsSomeoneWaitingAt() and IsSomeoneManning() check if there is someone at the station and if some clerk is already at the station respectively. If there is someone waiting, but no one is at the station, the clerk will start completing plan steps needed to help the customer out.

The MoveCloserToThing() function makes the agent move a bit closer to the station by finding a point between the agent's current position and the position of the point of interest and using it as a destination. To find the point between two locations, the built-in Lerp() method is used. The function can be seen in listing 21.

Listing 21: The method which moves an agent closer to its destination

```
1       private void MoveCloserToThing(string position)
2   {
3       Vector3 v3 = CreateVectorFromString(position);
4       agent.SetDestination(Vector3.Lerp(
5           transform.position, v3, 0.5f).ToString());
6   }
```

If the clerk does not need to man the cash register or a special station right now, it will finish its patrol by checking around itself to see if any nearby shelves need stocking. It will select the one closest to it and start restocking it. To accomplish this, it uses the method seen in listing 22.

Listing 22: The method that checks if a station needs restocking

```
1       private void CheckIfNeedsRestocking()
2   {
3       string toStock = GetNearestVisibleUnstocked();
4       if (toStock != "")
5       {
6           Reset();
7           lookingFor = toStock;
8           LetOthersKnowImStocking();
9           StartDoingSomething("stock");
10      } else
11      {
12          if(counter >= 3)
13          {
14              counter = 1;
15              Reset();
16              StartDoingSomething("patrol");
17          }
18          else
19          {
20              moving = true;
```

```
21              agent.GoToRandomLocationInStore(viewDistance);
22              counter++;
23          }
24      }
25  }
```

The function GetNearestVisibleUnstocked(), visible in listing 23, will grab all the currently unstocked products from the knowledge base and find the one closest to the clerk. It will return a blank string if there are no unstocked products or if they are not visible to the clerk currently.

Listing 23: The method that returns the nearest unstocked product if it is visible

```
1       private string GetNearestVisibleUnstocked()
2   {
3       string product = "";
4       using (PlQuery q = new PlQuery("product(D),
5           locationOfProduct(_,D,C), stocked(D,no),
6           gettingStocked(D,no)"))
7       {
8           Vector3 positionClerk = transform.position;
9           float currMin = -1;
10          foreach (PlQueryVariables v in q.SolutionVariables)
11          {
12              Vector3 v3 = CreateVectorFromString(v["C"].ToString());
13              float distance = Vector3.Distance(positionClerk, v3);
14              if (currMin == -1 || (distance < viewDistance
15                  && currMin > distance))
16              {
17                  currMin = distance;
18                  product = v["D"].ToString();
19              }
20          }
21      }
22      return product;
23  }
```

If the agent finds a product that needs stocking, it will let other clerks know that it is starting to stock a certain product and then start filling out the steps necessary to restock a dispenser. To skip the first few steps of checking both the cash register and the special stations and speed up the restocking process, the clerk will attempt to restock two other containers it sees nearby. The function LetOthersKnowImStocking(), seen in listing 24, is used to assert that some product is currently being stocked, avoiding the potential problem of two or more clerks restocking the same product.

Listing 24: The method used when an agent starts stocking

```
1       private void LetOthersKnowImStocking()
2   {
3       PlQuery.PlCall("retract(gettingStocked(" + lookingFor + ",no))");
4       PlQuery.PlCall("assert(gettingStocked(" + lookingFor + ",yes))");
5   }
```

During its patrol, the agent can end up either continuing the patrol if it finds nothing to do, or it can start one of three actions:

- **man the register**
- **man a station**
- **restock**

When **manning the register**, the agent lets the other clerks know that the register is being manned so that someone else does not try to man it. Afterward, the agent will get the location of the register from the knowledge base and move toward it. Once at the register, the clerk will serve all the customers that were waiting at the register, let others know that no one is manning the register anymore and go back to patrolling the store. This plan can be seen in listing 25.

Listing 25: The method used for manning the register

```
1     private void KeepManningTheRegister()
2 {
3     switch (step)
4     {
5         case 1:
6             LetOthersKnowImManningStation();
7             step++;
8             break;
9         case 2:
10            agent.SetDestination(GetLocationOfCashRegister());
11            step++;
12            break;
13        case 3:
14            if (!IsSomeoneWaitingAt(lookingFor))
15            {
16                step++;
17            }
18            else
19            {
20                StartServing(lookingFor);
21            }
22            break;
23        case 4:
24            LetOthersKnowIStoppedManning();
25            Reset();
26            StartDoingSomething("patrol");
27            break;
28        default:
29            break;
30    }
31 }
```

The case of the agent **manning a station** is pretty similar to manning the register, the only difference is that the agent needs to grab the proper special station's location, seen in

listing 26 below.

Listing 26: The method used for manning a special station

```
    private void KeepManningTheStation()
{
    switch (step)
    {
        case 1:
            LetOthersKnowImManningStation();
            step++;
            break;
        case 2:
            agent.SetDestination(
                GetLocationsOfSpecificSpecialStation());
            step++;
            break;
        case 3:
            if (!IsSomeoneWaitingAt(lookingFor))
            {
                step++;
            }
            else
            {
                StartServing(lookingFor);
            }
            break;
        case 4:
            LetOthersKnowIStoppedManning();
            Reset();
            StartDoingSomething("patrol");
            break;
        default:
            break;
    }
}
```

The specific product that the station belongs to is stored in the "lookingFor" variable, allowing the agent to more easily grab it through a Prolog call, shown in listing 27.

Listing 27: The method that grabs the location of a specified special station

```
    private string GetLocationsOfSpecificSpecialStation()
{
    string position = "";
    using (PlQuery q = new PlQuery("product(" + lookingFor + "),
        locationOfProduct(dispenserSpecialStation,"
        + lookingFor + ",C)"))
    {
        foreach (PlQueryVariables v in q.SolutionVariables)
        {
            position = v["C"].ToString();
        }
    }
```

```
13      return position;
14  }
```

Lastly, when the clerk finds a container that needs to be **restocked**, it will go to the inventory, find the proper storage box that carries the same product as the container, take the product from the box, go to the container and fill it up. After filling the dispenser up, the agent will retract its statement that it is currently restocking something. All of these steps are visible in listing 28.

Listing 28: The realization of the restock plan

```
1       private void KeepRestocking()
2   {
3       switch (step)
4       {
5           case 1:
6               agent.SetDestination(GetPositinoOfDispenserStock());
7               step++;
8               break;
9           case 2:
10              TakeStock();
11              step++;
12              break;
13          case 3:
14              agent.SetDestination(GetPositionOfDispenserProduct());
15              step++;
16              break;
17          case 4:
18              FillDispenser();
19              LetOthersKnowIStoppedStocking();
20              Reset();
21              break;
22          default:
23              break;
24      }
25  }
```

The TakeStock() method that stock dispensers in the inventory have returns the name of the product to the clerk, setting the item that they are carrying as that product. On the other hand, the FillDispenser() is a method that fills up the container, setting the stack to max. It can be seen in listing 29.

Listing 29: The clerk's method for filling up a dispenser

```
1       private void FillDispenser()
2   {
3       DispenserProduct dp = FindClosestDispenserProduct();
4       if (dp != null)
5       {
6           SetAsStocked();
7           dp.Restock();
8       } else
9       {
```

```
10          DispenserSpecialStation dss = FindClosestDispenserSpecial();
11          if (dss != null)
12          {
13              SetAsStocked();
14              dss.Restock();
15          }
16      }
17  }
```

The FindClosestDispenserProduct() method, seen in listing 30, will return the GameObject closest to the agent, letting it call the dispenser's public Restock() method. The method will only return the GameObject if the product associated with it is also equal to the product that the clerk is currently carrying, meaning that the clerk will not accidentally refill the wrong station.

Listing 30: An example of a method that locates the closest object with a specific script

```
1       public DispenserProduct FindClosestDispenserProduct()
2   {
3       DispenserProduct[] dps;
4       dps = GameObject.FindObjectsOfType<DispenserProduct>();
5       DispenserProduct closest = null;
6       float distance = Mathf.Infinity;
7       Vector3 position = transform.position;
8       foreach (DispenserProduct dp in dps)
9       {
10          Vector3 diff = dp.transform.position - position;
11          float curDistance = diff.sqrMagnitude;
12
13          if (curDistance < distance && dp.product == carrying)
14          {
15              DispenserProduct dispenserProduct = dp;
16              closest = dispenserProduct;
17              distance = curDistance;
18          }
19      }
20      return closest;
21  }
```

Once the restock action is complete, the clerk will go back to patrolling.

Finally, at the end of the simulation, when all of the customers that were set to come to the store have left, the Spawner GameObject will send a signal to all of the clerks, making them start using their plan for leaving the store, visible in listing 31.

Listing 31: The plan that clerk's complete when the simulation is ending

```
1       private void KeepLeavingStore()
2   {
3       switch (step)
4       {
5           case 1:
6               agent.SetDestination(GetLocationOfExit());
7               step++;
8               break;
```

```
9           case 2:
10              GameObject.Find("Spawner").GetComponent<Spawner>()
11                  .DestroyClerk(id);
12              break;
13          default:
14              break;
15      }
16  }
```

The clerks merely go to the entrance of the store before sending a signal to the Spawner. The signal makes the Spawner Destroy the clerk and remove it from the array.

3.2.2.3. Customers

Customers work similarly to the clerks when it comes to pulling facts out of the knowledge base. Upon being spawned, the customers create a list of random items they will look for throughout the store. Unlike the clerks, customers only need to know where the cash register and the exit are, spending the rest of the time looking for the specific products they are interested in acquiring. Their plans, the selection of which is based on their choice, is visible in listing 32.

Listing 32: The customer's possible plans realized as a switch case

```
1       private void StartDoingSomething(string choice)
2   {
3       switch (choice)
4       {
5           case "remember":
6               CreateShoppingList();
7               break;
8           case "getStuff":
9               GetItemsFromShoppingList();
10              break;
11          case "getSpecificItem":
12              GetSpecificCloseItem();
13              break;
14          case "checkOut":
15              GoToCheckout();
16              break;
17          default:
18              break;
19      }
20  }
```

The process of creating the shopping list is shown in listing 33. Essentially, it is just the customer grabbing all of the available products and randomly asserting them as a shopping list item in the knowledge base. Each product has a 1/3 chance to be on the agent's list.

Listing 33: The method that creates a customer's shopping list

```
1       private void KeepRememberingShoppingList()
2   {
```

```
3       List<string> items = GetAvailableProducts();
4       bool atLeastOne = false;
5       foreach (string item in items)
6       {
7           if (randomGenerator.Next(3) == 0)
8           {
9               atLeastOne = true;
10              PlQuery.PlCall("assert(shoppingList("
11                  + id.ToString() + "," + item + "))");
12          }
13      }
14      if (!atLeastOne)
15      {
16          Reset();
17          StartDoingSomething("remember");
18      }
19      Reset();
20  }
```

Once the list is created, the customer will move around the store and look for items on its list. The random movements are the same as with the clerks, allowing the customers to also use patrol points if they are enabled. Upon reaching some destination, the customer will look around. Using the view distance of the customer, a function returns the item that the customer is looking for that is closest to the customer. If the function returns nothing, the customer will find a new location to move to and try again. This process can be seen in listing 34.

Listing 34: The method that customers use to find items in the environment if they are near enough

```
1       private void KeepGettingStuff()
2   {
3       string toGet = GetNearestItemFromShoppingList();
4       if (toGet != "")
5       {
6           Reset();
7           lookingFor = toGet;
8           StartDoingSomething("getSpecificItem");
9       }
10      else
11      {
12          moving = true;
13          agent.GoToRandomLocationInStore(viewDistance);
14      }
15  }
```

If there is an item that the customer wants nearby, the customer will start taking steps, seen in listings 35, 36 and 37, to retrieve the item. This is the most complex plan that the customer has. The customer will first move to the dispenser of the product that it wants to get. If the dispenser is empty, the customer will start looking for something else, restarting the loop from some new spot.

Listing 35: The first two steps of the item-acquiring plan

```
    private void KeepGettingSpecificItem()
{
    switch (step) {
        case 0:
            agent.SetDestination(GetPositionOfItem());
            step++;
            break;
        case 1:
            bool stocked = CheckIfStocked();
            if (stocked) {
                step++;
            } else {
                LookForSomethingElse();
            }
            break;
```

If the dispenser still has items in it, the customer will check whether it needs help from a clerk when retrieving the item. If the customer does not need help, it will immediately move to the 4th step. Otherwise, the customer will let it be known that it is waiting for help. Now the customer will wait at the station until a clerk shows up to serve it. Once the customer detects that it is being served, it will move to the 4th step.

Listing 36: The second two steps of the item-acquiring plan

```
        case 2:
            bool needsClerk = CheckIfNeedsClerk();
            if (needsClerk) {
                StartWaitingForClerk();
                step++;
            } else {
                step++;
                step++;
            }
            break;
        case 3:
            bool gettingServed = AmIGettingServed();
            if (gettingServed) {
                StopGettingServed();
                step++;
            }
            break;
```

In the 4th step, the customer will try to take the item. If successful, it will assert the item to its inventory, remove it from the shopping list and then decide whether it wants to go to the cash register or keep looking for items it still needs to get. If it does not manage to acquire the item, it will restart the loop of getting the item.

Listing 37: The final steps of the item-acquiring plan

```
        case 4:
            bool gotIt = TryToTakeItem();
            if (gotIt) {
                StopWaitingForClerk();
```

```
                    RemoveItemFromList();
                    AddToBackpack();
                    Reset();
                    if (GotEverything()) {
                        StartDoingSomething("checkOut");
                    } else {
                        StartDoingSomething("getStuff");
                    }
                } else {
                    step = 1;
                }
                break;
            default:
                break;
        }
    }
```

The TryToTakeItem() method works similarly to the way clerks restock dispensers, where the agent grabs the nearest GameObject with the item that the customer is looking for and calls the GameObject's public method for removing an item from the stack. The method is visible in listing 38.

Listing 38: The method that helps customers mark a station as empty

```
    DispenserProduct dp = FindClosestDispenserProduct();
if (dp != null)
{
    int currentStack = dp.TakeOneFromShelf();
    if (currentStack < 1)
    {
        SetItemAsNotStocked();
    }
    if (currentStack + 1 <= 0)
    {
        return false;
    }
}
```

The public function from the dispenser will return the leftover number of items in the dispenser. If the returned number is 0, the customer will assert that the dispenser is no longer stocked.

Once the customer has acquired all the items that were on its shopping list, it will start the process of checking out. The process is pretty similar to the process of waiting at a special station. The main differences are the 4th and 5th steps where, after getting served, the customer will start heading for the exit. Upon arrival, the customer will make the Spawner destroy them, removing them from the store and the Spawner's customer array. These last two steps can be seen in listing 39.

Listing 39: The final two steps of the customer's leaving the store method

```
    case 4:
```

```
2       StopWaitingForClerk();
3       LeaveStoreAndRemoveMyBackpack();
4       agent.SetDestination(GetLocationOfExit());
5       step++;
6       break;
7   case 5:
8       GameObject.Find("Spawner").GetComponent<Spawner>()
9           .DestroyCustomer(id);
10      break;
```

The customers also remove all the facts that were related to them in the knowledge base upon leaving the store. When the last customer leaves, the Spawner calls the clerks back too and ends the simulation.

3.3. Testing the simulation

The following chapter runs the simulation a number of times with various variables changed up, with the goal of finding the most time-efficient variables in the current system. There are some constants, however:

- there are always 30 customers that need to complete their shopping, with only 10 allowed in the store at any moment
- the clerks always get a 30 second head start before the first customer shows up
- the view distance of all agents is 30 units
- patrol points are being used for added consistency
- each combination will be ran 5 times

The goal is to find the optimal number of clerks for the store, with the simulation running with 2 to 4 clerks. Along with the number of clerks, the amount that can be stored in a container will be changed. During the first run, it will be set to 9 items in each container globally and then increased to each container's maxiumum during the second run. The time it takes to serve 30 customers is being measured.

The results of the tests are depicted in the table below.

As can be seen by the results in table 1, as the number of clerks in the store increases, the time it takes to serve 30 customers decreases on average. With just two clerks in the store, especially if the stacks are smaller than their maximums, it takes too long for just two agents to run the environment, with the average completion time being 9:23.44. If the stacks are at their maximum, they have to restock less often so they can increase the time they spend serving the customers, averaging 08:47.65 minutes per simulation. The results would be even worse if manning stations and cash registers used up some of the clerks' time.

When there are three agents working as clerks, the results improve drastically. On average, with the stacks globally being set to 9, the average completion time is 7:53.24. Increasing

# of clerks	Stack size	Attempt	Duration
2	9	1	09:45.40
2	9	2	10:05.39
2	9	3	09:27.40
2	9	4	08:52.39
2	9	5	08:48.43
2	max	1	08:38.75
2	max	2	08:49.40
2	max	3	09:29.38
2	max	4	08:05.38
2	max	5	08:57.52
3	9	1	07:34.53
3	9	2	08:13.35
3	9	3	08:47.36
3	9	4	07:27.35
3	9	5	07:25.36
3	max	1	07:24.36
3	max	2	07:43.33
3	max	3	07:11.43
3	max	4	06:51.44
3	max	5	06:52.43
4	9	1	07:17.45
4	9	2	05:59.44
4	9	3	06:51.45
4	9	4	07:04.44
4	9	5	07:19.48
4	max	1	07:17.35
4	max	2	07:10.39
4	max	3	07:33.40
4	max	4	06:22.39
4	max	5	07:05.39

Table 1: Recorded simulation durations

the stack sizes makes about as much of an impact as it did with two clerks, reducing the time on average by about 40 seconds, leading to the new average being 7:12.24. During the runtime, there were fewer huge lines too, so adding a bit more logic by making interactions last would not have as bad of an effect this time around.

Further increasing the number of clerks returns better results again, but it is not as noticeable as earlier. The average simulation time with 4 clerks and non-maximum stacks is 6:54.45 minutes. Interestingly, when maximum stacks were being used, the average time was slightly worse at 7:05.44. Both times are still better than when there are only 3 clerks in the store, but, especially when using maximum stacks, the improvement in average time may not be worth hiring a new agent. When running the simulation with 4 clerks, there were even fewer times with more than one or two customers waiting at a register or station, meaning that the customer clerks were able to achieve their goals with less waiting. The inconsistency where increasing the stacks to their maximums yielded worse results than if they were smaller can be attributed to the fact that the sample size is pretty small and that some of the more random variables of the simulation could have influenced the results negatively. If customers generated smaller shopping lists or if they all chose wildly different items to purchase, thus there being less of a need for restocking, it could have lessened the time that the customer agents spent at the store.

Overall, using two clerks for a store this size, especially if the store is as busy as in the simulation, would not be efficient. However, whether to use three or four clerks depends on what the priorities of the store owner are. While having four clerk agents available at the same time speeds up the the simulation, it leads to situations where all the product containers are stocked and all the stations are manned, leaving some agents to patrol without any tasks to accomplish. Having only three clerks still gives quick results, but it leads to fewer situations where agents have nothing to do.

3.4. Shortcomings and possible improvements

The simulation has a lot of room for improvement. There are some minor problems that can arise as the simulation progresses and some logical fixes that could be implemented, improving the simulation overall:

- **building** - the project currently only works as a Unity project, building it seems to break the Prolog bridge. Also, the project requires an old version of SWI_Prolog to run, switching to some other software might enable the project being built and remove the prerequisite of having SWI_Prolog installed.
- **agents running into each other** - all of the agents have a capsule collider component and if they run into each other head-on during movement, they can be stuck moving into each other until some other agent bumps into them and makes them continue moving towards their destination.
- **spacing with stations** - currently, both the clerks and the customers move to the

same location. Implementing it so that the clerk moves behind a counter or behind a station would make more sense logically.

- **customers asking for aid** - customers do not have a way of letting the clerks know that an item that they are looking for is unstocked, forcing customers to wander around the store until a clerk notices the container that needs to be restocked.
- **having the interactions last longer** - currently all of the customers that are waiting at a station or at the register get served at once and instantaneously. Logically, they should be served one at a time and each interaction should last a certain amount of time.
- **implementing a player option** - allowing the user to enter the simulation as a clerk or a customer could open up some new possibilities for the simulation.
- **availability of info** - having the simulation display more info about what is going on would help improve readability. Currently, there is no easy way to tell what a customer is looking for or carrying.
- **adding new problems** - making it so that the inventory runs out of a certain product could add more variables to the simulation, making it so that customers are not able to fully reach their goal or giving clerks a new action of ordering supplies.
- **manning stations for longer** - making it so that certain clerks are stationed for longer would make logical sense. In the real world, clerks wait at stations if they see a customer nearby or if there is nothing else for them to do at the moment.
- **currency** - introducing money into the simulation could add a layer of complexity and realism to the simulation, offering up opportunities for customers not having enough money to pay or clerks being able to tell how much money they made during the shift.

A proper simulation needs to be as close to the real-life counterpart that it is modelled after to give the most accurate and useful results. Still, the current iteration offers a solid base that can be built upon in future iterations.

4. Conclusion

Simulations are a common and useful tool when it comes to modelling real-world scenarios. They offer a lot of insight to their users, whether it is predicting potential future scenarios or testing out potential changes before putting them into practice. Due to their high flexibility and various methods of approaching them, they can be used in many unique fields. However, depending on which design philosophies are used, they can be complex and expensive to create, often requiring a lot of knowledge and preexisting data when being built. This can often make the investment seem as an unnecessary risk, leading to organisations ignoring this potentially highly useful tool.

Using agent-based models offers a different design solution that allows the creator to make their model from simpler, smaller parts, focusing mostly on the agents. Designing simulations in this way reduces complexity during the creation step without much cost to the final results. Agents themselves play a key role in changing the model and ultimately have a huge impact on its final state by merely trying to achieve their own goals. Belief-Desire-Intent agents specifically enable the users to create highly dynamic scenarios with intelligent participants, leading to more realistic and informative results.

The initial costs can also be lowered by using already existing tools and expanding upon their functionalities, making them become suitable to the user's needs. The Unity engine is highly flexible in this regard, letting the programmers modify it with various outside packages that enable it to create more complex GameObjects than its basic tools allow. As technology and artificial intelligence progress, it is important to keep trying to combine different pieces of software to achieve something greater than the sum of its parts.

Bibliography

[1] M. Wooldridge and N. R. Jennings, “Intelligent agents: Theory and practice,” *The knowledge engineering review*, vol. 10, no. 2, pp. 115–152, 1995.

[2] D. C. Dennett, “Précis of the intentional stance,” *Behavioral and brain sciences*, vol. 11, no. 3, pp. 495–505, 1988.

[3] D. Singh, L. Padgham, and B. Logan, “Integrating bdi agents with agent-based simulation platforms,” *Autonomous Agents and Multi-Agent Systems*, vol. 30, pp. 1050–1071, 2016.

[4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach* (Prentice Hall Series in Artificial Intelligence), 3rd ed., red. by S. Russell and P. Norvig. New Jersey, USA: Prentice Hall, 2010, 1132 pp., ISBN: 978-0-13-604259-4.

[5] D. M. Rao, A. Chernyakhovsky, and V. Rao, “Searums: Studying epidemiology of avian influenza rapidly using modeling and simulation,” *Lecture Notes in Engineering and Computer Science*, vol. 2167, 2006.

[6] Unity Technologies, 2023.

[7] W. F. Clocksin and C. S. Mellish, *Programming in PROLOG*. Springer Science & Business Media, 2003.

[8] U. Lesta, *Interface from c# to swi-prolog*, `https://github.com/SWI-Prolog/contrib-swiplcs`, 2019.

[9] B. O. Community, *Blender - a 3d modelling and rendering package*, Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[10] iUnity3Dtutorials, *Unity spawn prefab at position tutorial c# easy*, Youtube, 2015.

[11] Unity Technologies, “Unity user manual 2022.3 (lts),” *Manual/index. html (Eng.)*, 2023.

[12] Code Monkey, *How to use unity navmesh pathfinding! (unity tutorial)*, Youtube, 2021.

List of Figures

List of Tables

List of Listings