POLITECNICO DI TORINO

Collegio di Ingegneria Informatica, del Cinema e Meccatronica

Corso di Laurea Magistrale
in Ingegneria del Cinema e dei Mezzi di Comunicazione

Tesi di Laurea Magistrale

DEVELOPMENT OF A LIBRARY OF INTELLIGENT AGENTS FOR VR APPLICATIONS

Relatore

prof. Andrea Bottino

Candidato

Edoardo Battegazzorre

Aprile 2019

0. Riassunto in Lingua Italiana

I professionisti nel campo della medicina hanno mostrato grande interesse, in tempi recenti, nel campo delle simulazioni in Realtà Virtuale, specialmente nelle applicazioni per la cura dei disturbi d'ansia.
L'obbiettivo di questa tesi, nel contesto di una tale simulazione, è creare una libreria di agenti intelligenti autonomi che popolino l'ambiente in una maniera credibile.
Gli obbiettivi principali del lavoro di tesi sono stati focalizzati su: realismo, efficienza e modularità. Il realismo nei movimenti e nelle animazioni è necessario per aumentare il senso di immersione dei pazienti. L'efficienza è intesa in termini di utilizzo delle risorse computazionali, ed è un requisito avere una buona frame rate in real time anche quando un alto numero di agenti deve essere gestito. La modularità riguarda l'abilità degli agenti di adattarsi a qualsiasi tipo di ambiente e situazione, visto che l'applicazione in Realtà Virtuale in questione è dotata di un editor che permette ai medici e agli psichiatri di personalizzare impostazioni via cloud, generare eventi e modificare l'ambiente durante la sessione.

Per affrontare il problema abbiamo adottato l'approccio bottom-up tipico del stile di modellazione ad agenti, impiegato nella maggior parte delle simulazioni virtuali.
La classe principale di agenti della libreria è quella dei clienti, che sono caratterizzati da un comportamento abbastanza complesso. Questo comportamento è stato modellato come una Macchina a Stati Finiti Gerarchica (HSM: *Hierarchical State Machine*). Le HSM sono uno sviluppo del concetto di Macchina a Stati Finiti tradizionale, permettendo di unire gruppi di stati in "super-stati". Abbiamo optato per questo modello perché, nonostante la sequenza di stati degli agenti sia molto lineare, essa è composta da numerose micro-operazioni, con un certo grado di interconnessione. Le HSM permettono di gestire un grande numero di stati con relativa facilità, grazie alla loro natura gerarchica, e di mantenere il numero di archi sotto controllo.
Come prima operazione abbiamo cercato di identificare gli stati che rispecchiano il comportamento dei clienti di un supermercato.
La sequenza di "super-stati" identificati è la seguente grafo di alto livello:

Entrare nel Negozio → Comprare gli oggetti desiderati → Pagare alla cassa → Uscire

Ognuno di questi "super-stati" è gestito internamente come una macchina stati finiti, che definisce nel dettaglio tutte le micro-operazioni di livello più basso che devono essere compiute dagli agenti per completare il loro ciclo.
Ognuno di questi stati implica concetti complessi come: ricevere e dare informazioni con l'ambiente esterno, muoversi nell'ambiente, *path finding*, evitare le collisioni, gestione delle animazioni e fisica.
In primo luogo gli agenti devono poter comunicare col mondo esterno, raccogliendo informazioni riguardo alla disposizione degli ostacoli dell'ambiente, degli scaffali, dei prodotti e la posizione e lo stato di altri agenti e dell'utente.

Il *path finding* e la *collision avoidance* sono probabilmente gli elementi più critici, considerando il fatto che gli agenti devono muoversi in maniera autonoma, da soli e spingendo un carrello in maniera realistica, cercando di evitare ostacoli e raggiungere la loro destinazione, adottando tecniche di *path finding* dinamico.
Molto importante per il senso di immersione e per il realismo degli agenti sono le animazioni. I movimenti degli agenti non devono apparire troppo robotici, per esempio nella maniera in cui parcheggiano il loro carrello o nella maniera in cui si mettono in fila alle casse, in cui devono apparire abbastanza "disordinati" da sembrare i loro corrispettivi umani.
Per migliorare ancora di più il realismo delle animazioni, è stato creato ad hoc un sistema di gestione delle animazioni che integra le animazioni pre-renderizzate (che sono state ottenute dai movimenti di una persona vera registrata tramite tecniche di motion capture) con animazioni gestite a runtime tramite cinematica inversa. Un esempio di impiego di questo sistema è quando un agente deve prendere un oggetto specifico da uno scaffale, dove il sistema permette di raccogliere l'oggetto con un movimento naturale che risulta in un effetto piacevole, specie se confrontato con un'animazione di raccolta generica.
E' inoltre stato adottato un motore fisico per gestire i prodotti e le loro collisioni, permettendo agli agenti di disporre e impilare gli oggetti nel loro carrello in maniera credibile e naturale.

La prima parte del lavoro è stata focalizzata sullo sviluppare un sistema ad agenti a sé stante, mentre la seconda ha previsto di implementare questo sistema su un'applicazione pre-esistente, visto che questo progetto ha coinvolto più persone, ognuna focalizzata su un sistema in particolare. Questa seconda parte ha testato la modularità e adattabilità del lavoro svolto.

L'efficienza del sistema è stata inoltre sottoposta a dei test il cui scopo era verificare i valori medi di frame al secondo tramite varie telecamere posizionate in punti strategici del supermercato, in posizioni significative dal punto di vista dei poligoni renderizzati e dei tasso di traffico degli agenti.
Nonostante cali di frame rate si siano verificati (come ci si aspettava), i valori non sono quasi mai scesi al di sotto della soglia critica di 60 FPS, anche quando il numero di agenti era molto alto, alle soglie se non oltre della capienza massima dell'ambiente di test.
Se consideriamo inoltre che ci sono ampi spazi per ottimizzazioni varie, possiamo dire che i risultati ottenuti sono molto promettenti.
Possiamo concludere dicendo che il sistema di agenti che abbiamo sviluppato è un solido punto di partenza per futuri sviluppi. Una prima area di miglioramento possibile è l'implementazione di più complessi modelli comportamentali basati su studi effettuati su clienti di supermercati. La seconda area riguarda il campo delle interazioni con gli utenti e tra agenti stessi, che possono portare ad un ancora più alto livello di realismo ed immersione.

Index

1. INTRODUCTION

The thesis work is part of a larger project which is being worked on by multiple people. The objective of the project was to create a virtual reality simulation aimed for psychiatric/medical therapists and aid them in studying and eventually treating patients with anxiety disorders.
The simulation takes place in a supermarket/grocery store, a typical environment which may cause anxiety and/or fear for patients to be in.
In this simulation, created using the free game-engine Unity, the patient takes control of an avatar of himself via a VR headset and controllers (Oculus Rift, HTC Vive, etc.) and is free to move in the store, interact with other customers and cashiers, and also with various inanimate objects such as products on sale or shopping carts.
The patient's goal is to complete the simple task of shopping for groceries while in this VR simulated safe space, so he can be more at ease before moving on to more “shocking” stages of treatment such as actually going to a supermarket in real life.

The focus of this thesis' work was to create and model the behaviour of other non-human actors: the aforementioned store clients and workers.

1.1 Objectives

The objective of our work was to develop intelligent autonomous agents in the Unity framework, with particular focus on their behaviour patterns and animations, to achieve the final goal of maximizing the patient's immersion and presence in the simulation.
The intelligent agents must perform as faithfully as possible what their real life counterparts would do in that context, avoiding unnatural/impossible actions.
It must be noted that the VR application comes with an environment editor, so the therapist can place crucial elements (general layout, shelves, products, cashier counters) in any way they like, for example to recreate a particular store that the patient may visit in real life, or to make the patients "work their way up" to bigger and bigger stores that may increase the anxiety levels in them.
This means that, while modelling the elements of the simulation, a modular approach must be observed and we can't "cheat" and make the agents follow a simple, pre-written script: they must adapt to and act in any possible environment allowed by the editor.

There are two classes of intelligent agents:

- Customers
- Cashiers

The most complex of the two is the former, customers must:

- move around somewhat realistically
- interact with the environment (shelves, items, shopping carts)
- interact with other agents (other customers, cashiers)
- interact in some way with the user
- complete their task (enter the store → shop for groceries → check out → exit the store)
-

On the other hand the latter class, cashiers, are much simpler, since they don't move around and mainly sit at the counter waiting for customers to check out.

2. STATE OF THE ART

Since the main goal of the project was to program agents with an Artificial Intelligence, several modes of implementation have been considered.
We will now present different approaches to deal with AI.
In this chapter we will also discuss techniques for computer animation and the uses of VR for medical practice.

2.1 Agent-based model

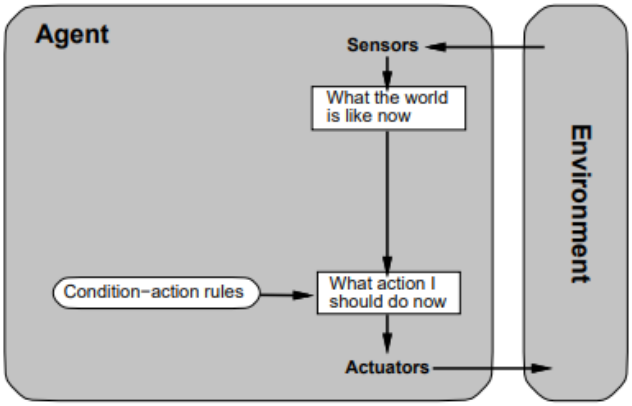
While the theory behind Agent-Based model is very wide and complex, we will briefly mention it because most simulations, including our own, are based on this style of modelling.
An Agent-based model is a broad category of modelling styles where we represent the relation and interactions between individuals and environment. They start off modelling the properties and behaviour of single entities (agents) and only successively consider the high level behaviour of the whole system. This is a type of bottom-up approach, opposed to the more traditional Equation-Based modelling, where we model first the system as a whole and doesn't aim to consider the behaviour at a lower-level. This is, by contrast, a top-down approach.
For further reading on the subject of Agent-Based modelling versus Equation-Based we remand to the paper by H.V.D. Parunak et al. [1] and the article by K. Singh [2].
If we take as an example the behaviour of a gas in a box, the Agent-based approach is interested in modelling the single particles and their properties, while an equation-based approach models the behaviour of the gas as a whole.
Agent-based modelling is becoming increasingly popular because the ever-increasing computational power of computers allows to run more and more complex simulations.
And the key is here: ABM is very useful to observe non-linear (and often even stochastic) phenomena which are not very easily representable with ordinary physics-based equations.
Also ABMs allow much more easily to represent space (i.e. the environment of a simulation) thereby offering the possibility of considering topological particularities of interactions and information transfer.
This is relevant in our problem, because movement and behaviour of clients in a grocery store is impossible to reduce to one or more equations (it contains many stochastic elements / variables) so the bottom-up approach of the Agent-Based model is ideal in our case, more so because of the behaviour of the clients (agents) MUST be dependent on the layout of the environment, which is not fixed and can be arranged by the user in any way they like.
Further applications of Agent-Based Modelling can be found in fields such as flow simulation, organizational simulation, market simulation, diffusion simulation [3].

Also store clients are not only “agents”, they are Intelligent Agents.

2.2 Intelligent Agents

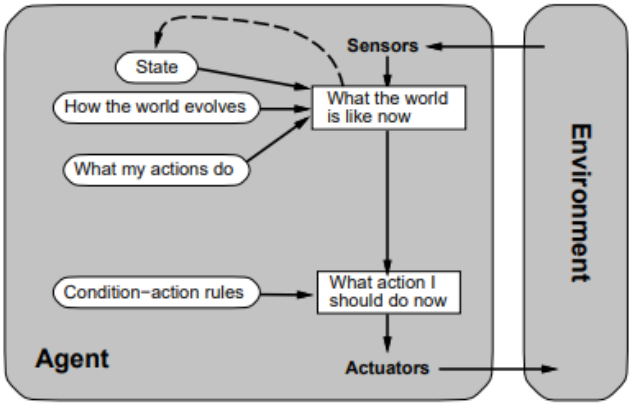
The most comprehensive definition a Intelligent Agent (IA) is: "[...]An autonomous entity which acts […] (i.e. it is an *agent*) upon an environment using observation through sensors and consequent actuators (i.e. it is *intelligent*)" (S.J. Russel, P Norvig [4]).
This is a very broad term, to the point that even a simple thermostat is technically considered a (reflex-based) Intelligent Agent.
Intelligent Agents are classified in five categories [4] based on their perceived intelligence and potential capability.

-Simple Reflex Agents
-Reflex Agents with State
-Goal-Based Agents
-Utility-Based Agents
-Learning Agents



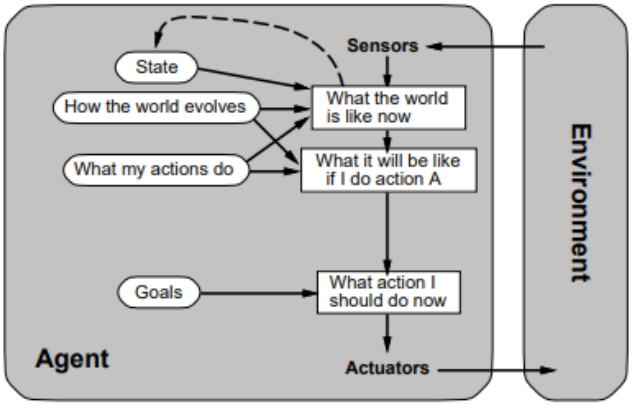
Img 2.1) Simple Reflex Agent

Simple Reflex Agents: The simplest kind of agent, they act only considering the current local/global state, and they have no memory. They act on simple *Condition → Action* rules, where the condition is fully dependent on the environment, which must be always observable by the agent's sensors. Infinite loops are often unavoidable when dealing with this type of agents.



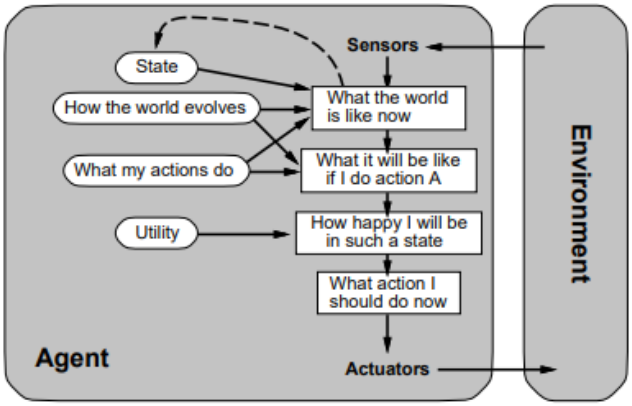
Img 2.2) Reflex Agents with State

Reflex Agents with State: Also called Model-Based Reflex Agents, because they have some kind of knowledge about "how the world works" (a *model* of the environment). They store the current state of the environment and their actions are based on the evolving history of that on top of the current input from their sensors.



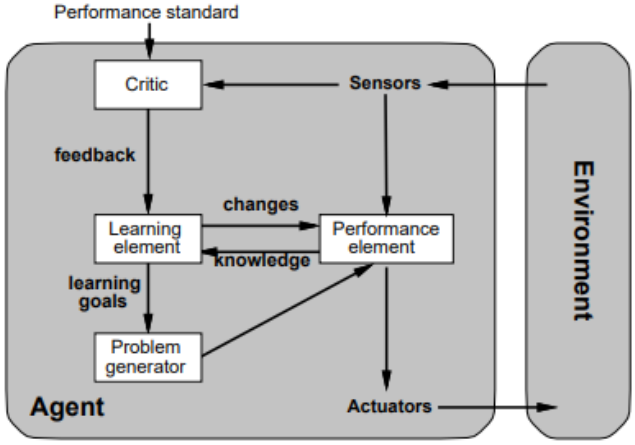
Img 2.3) Goal-Based Agents

Goal-Based Agents: They expand upon model-based agents adding the notion of *Goal*. This means that they can plan ahead one or more sequences of actions to achieve their set goal. They also can chose the most optimal path to achieve a goal. More concepts on Goal Based Agents and Goal Oriented Action Planning will be explained in a subsequent chapter (XYZ Goal Oriented Action Planning).



Img 2.4) Utility-Based Agents

Utility-Based Agents: Further expanding upon goal-based agents, here is introduced the concept of *Utility*. This transcends the simple duality of the goal achieved/not achieved state, but adds on top of that a Utility Function used to measure "how happy will the agent be" in different world states.



Img 2.5) Learning Agents

Learning Agents: The most complex of the group, these agents have the advantage that they can make their own model of the world, and so they can operate in an initially unknown environment. They have a learning element and a performance element. The first one makes improvements on the agent's behaviour, while getting feedback form the second "critic" element which determines how well the agent is doing. A problem generator element is also present, its function being directing the agent towards new behaviours that may lead to new learning experiences.

2.3 Models for Artificial Intelligence

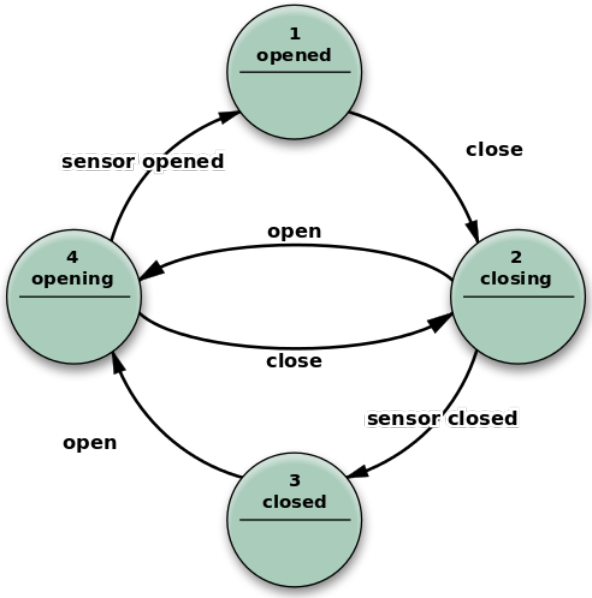
Used in most games and simulations, where the patterns of behaviour are mostly fixed and set by the programmer, the following models are considered to be part of "Classic" AI, to differentiate them from the newer practices related to machine learning. The agent's actions are determined by any number of different inputs, and based on these inputs the AI decides its next state/behaviour.
This approach is ideal where the number of situations/environments/stimuli is limited and can be controlled and predicted easily.

2.3.1 Finite State Machines

FSMs used for applications in the field of computational linguistics are actually called Finite State Transducers (we will refer to them as FSM anyway throughout the text) and can be distinguished in two types:

-*Moore Machines*: For Moore machines the output is based only on the current state, with no external input.
-*Mealy Machines*: In contrast with Moore machines, here the output is based both on the current state and on an input command. This makes that they usually have less states than a corresponding Moore counterpart.

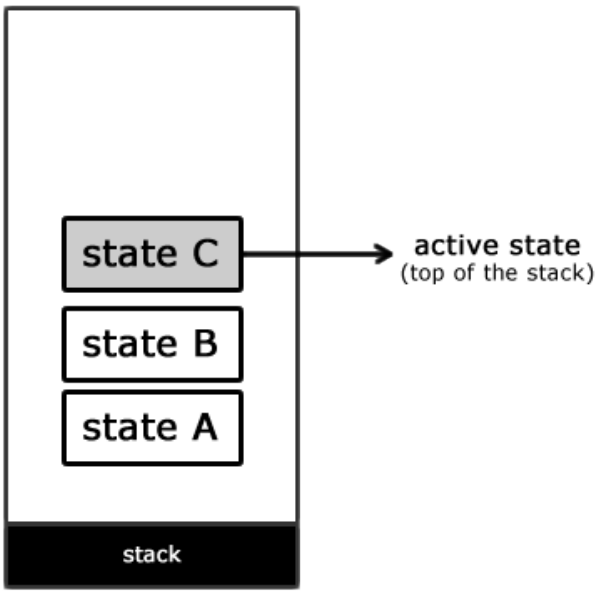
A FSM is usually represented by a graph where the nodes are the states and edges are transitions between states.



Img 2.6) Example of a Moore State Machine modelling the functions of an elevator

In the FSM model an automation based on the machine can only have one state at a time, and transitions occur when certain conditions are met (the origin of these transitions may vary, as seen in the difference between Moore and Mealy State Machines).

In Agent based applications, a useful variation on the FSM model are the so-called Stack-Based FSMs. In this model the states are placed in a stack, and transitions are handled by pushing or popping states in the stack.

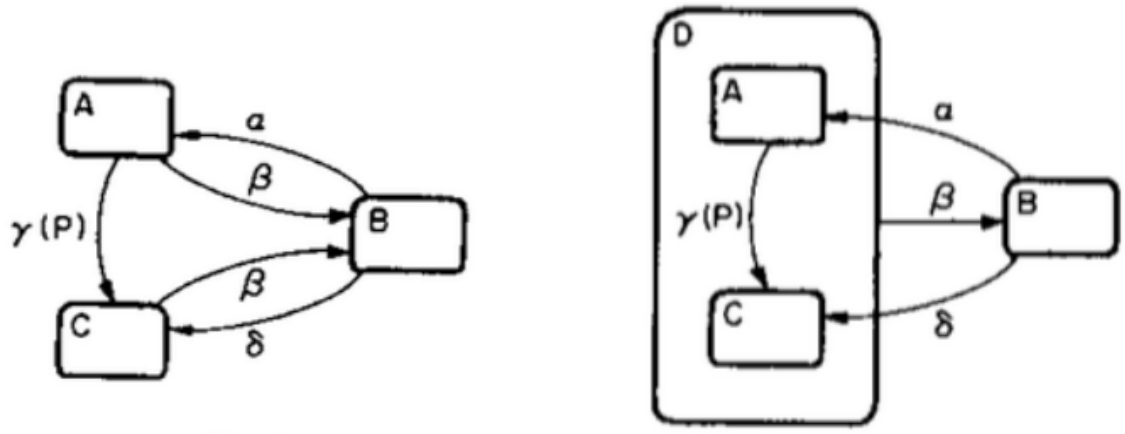


Img 2.7) A Stack-Based Finite State Machine

The active state is the one placed on top of the stack, and when a transition occurs it can:

-*Pop itself*: this means that the state subsequent in time is the one previously assumed (in this case if C pops itself, the machine will go back to the B state it before going to C).
-*Push a new state*: This means that the new state (it would be a "D" state in this instance) will remain active for a while, but the machine has stored in memory its previous state, to which it can revert at any time.
-*Pop itself+Push*: This is a full transition akin to the one seen in a traditional non-stack-based FSM.

As we will discuss later, a huge drawback of finite state machines is their tendency to quickly become a nightmare of states and transitions as their complexity grows.
To ease this tendency, the visual formalism of Hierarchical States Machines was proposed. Also known as State Charts, they were first introduced by David Harel (for more information see this paper "Statecharts, a visual formalism for complex systems" [5]).



Img 2.8) State aggregation in Hierarchical State Machines

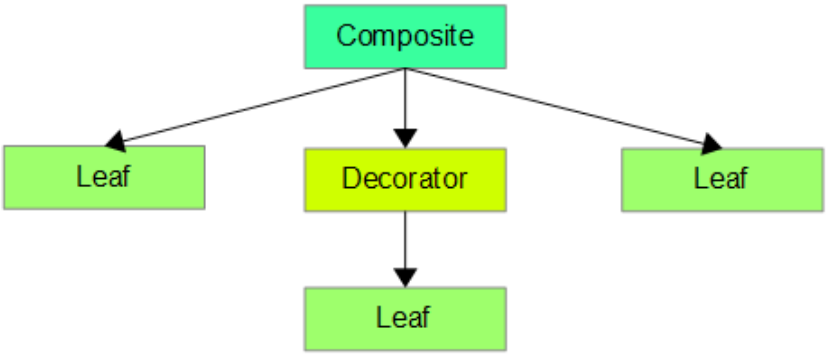
The idea here is to aggregate groups of states together in clusters called *super-states*, with the objective of reducing the overall number of transitions (with *generalized transitions*) and facilitate reading graphs of complex systems.
This procedure also introduces the concept of *behavioural inheritance*, allowing nested states to mutate and have new behaviours.

2.3.2 Behaviour trees

A behaviour tree is similar to a hierarchical state machine, with the key difference that the nodes represent tasks instead of states.
The leaves at the extent of the tree are the actual actions or tests the controlled agent performs, usually implemented as functions when coding.
Intermediate "utility" nodes along the branches represent the logic of the decision making process.
Each node returns a value to its parent, this value is usually one of tree possible states: *success*, *failure* or *running*.

Behaviour trees have 3 main node archetypes:

-Composite
-Decorator
-Leaf

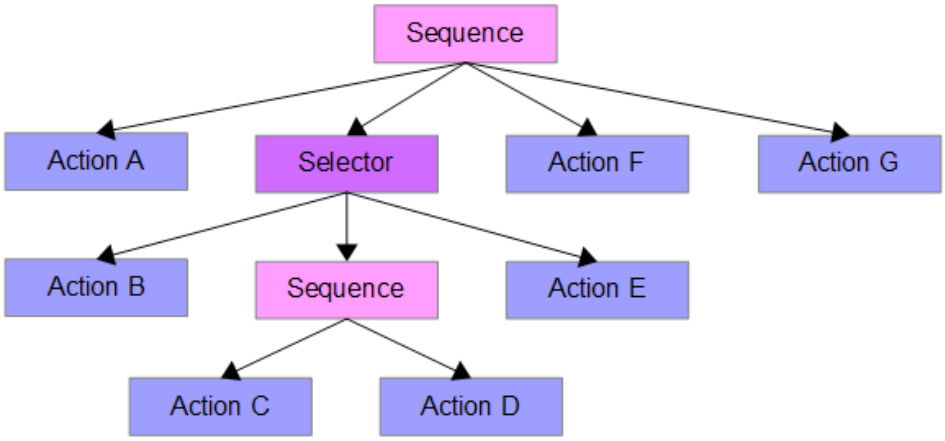


Img 2.9) BT node archetypes

Composite Node: a composite node is a node that can have multiple children, children can be accessed in a defined sequence or randomly.
The two main types of Composite Node are:

-*Sequence:* works in a similar fashion as an AND logic gate: returns a success if all its children return a success, otherwise returns a failure.

-*Selector:* works as an OR gate: returns a success if any of its children returns a success.



Img 2.10) An example of a tree with nested Sequences and Selectors.

Decorator: this kind of node can have one child only, and it modifies the value returned by the leaf.
The most important Decorator node types are:

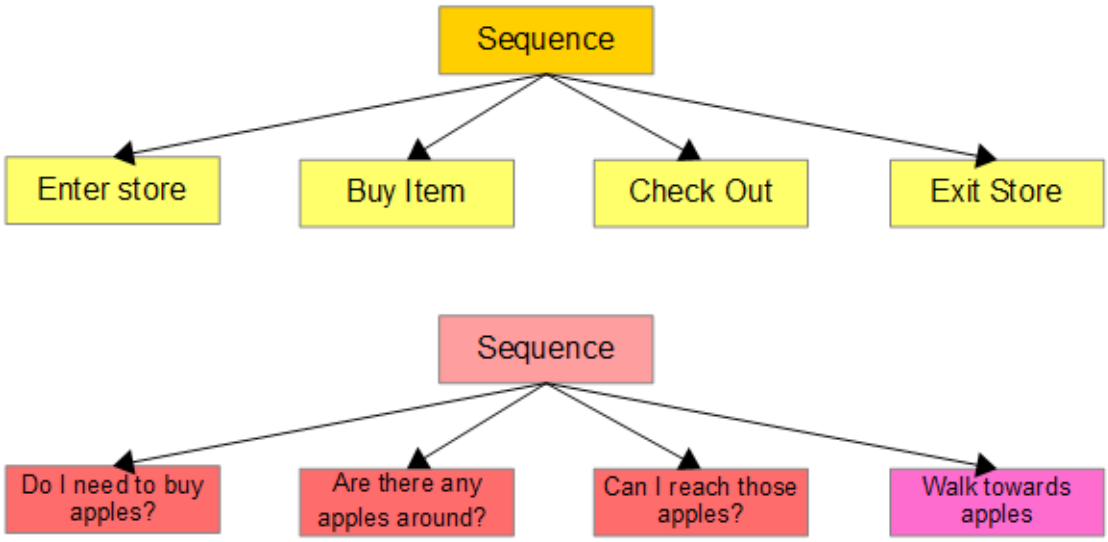
-*Inverter*: inverts the value returned by its child. Basically a NOT logic gate.

-*Succeeder/Failer*: they always return a success or failure respectively, with no regard to the actual value returned by the child. They are useful when we want to still process a branch whatever the returned results may be.

-*Repeater*: keeps accessing the child as long as a value is returned. They usually

are conditional, for example *Repeat Until Fail*.

Leaf: This nodes are the actual functions or tests performed when the node is called. As an example of an action, we can examine an hypothetical node that contains the function “Walk To Position (position)” (by the way, this is very similar to the actual function used by Unity's *NavMeshAgent*, SetDestination(*Vector3* position)). This node, the first time it's accessed, will compute a path to the desired position and start actually moving the character towards that location's coordinates. If the node is accessed while the character is walking, it will return the value *running* to its parent. If at any point the destination becomes unreachable or the path is blocked, it will return a *failure*. Finally, when the destination is actually reached, a *success* Is returned.
Leaves can also be a simple test, as in “Can I Walk?”. In this instance, the node will return a *success* or a *failure* immediately.



Imgs 2.11 & 2.12) Examples of leaf nodes as actions and as tests, parented to a Sequence Composite node.

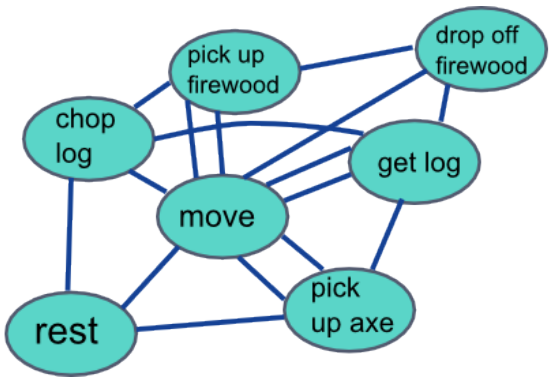
2.3.3 The Pros & Cons of FSMs and BTs

While Finite State Machines and Behaviour Trees may seem very simple, they assume a whole new layer of complexity it we organize them in hierarchies or layers.
For example, a leaf of a BT may call a whole other Behaviour Tree, or maybe a node in a State Machine can contain one or more Behaviour Trees in itself.
Also, while an agent based on a FSM can be in only one state at a point in time, multiple FSMs can be layered on top of each other, thus generating exponentially more complex behaviours, since every layer has it's own state. Unity uses this approach in its Animator, where we can create an arbitrary number of layers, assign masks and/or weights to them and achieve a much more complex animation system.
One drawback of FSM is that, while they can be as complicated as they need to be, they become very messy and hard to read very fast. BTs suffer the same problem but tend to be a little less confusing due to their hierarchical nature.
Another intrinsic downside is that rarely the programmer can foresee any possible situation, so the simpler is the environment, the better the FSM/ BT approach works (also from a computational performance perspective).
The behaviour of the agents in this project has been represented as an hybrid between a SM and a BT.
The overall behaviour states are represented by a quite simple sequential state machine, and each of these “super-states” has its own behaviour tree.

2.3.4 Goal Oriented Action Planning

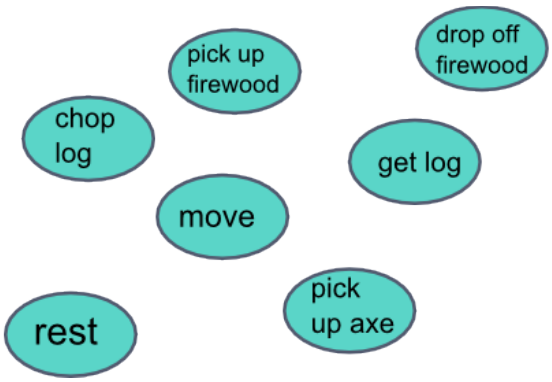
In addition to Behaviour Trees and State Machines, there are other models alternative or derived from them, employed to model or describe more complicated behaviours.
One of these is the so called Goal Oriented Action Planning (GOAP), introduced to simplify the process of adding new states to an Intelligent Agent, as one of the major problems intrinsic in State Machines is the issue that any new state introduced must be considered in relation to every other pre-existing state.
GOAP is defined as follows: “Goal Oriented Action Planning is an artificial intelligence system for agents that allows them to plan a sequence of actions to satisfy a particular goal. The particular sequence of actions depends not only on the goal but also on the current state of the world and the agent. This means that if the same goal is supplied for different agents or world states, you can get a completely different sequence of actions, which makes the AI more dynamic and realistic. ” (https://gamedevelopment.tutsplus.com/tutorials/goal-oriented-action-planning-for-a-smarter-ai-cms-20793).

The Goal Oriented Action Planning turns a State Machine like this:



Img 2.13) States and links of a traditional Finite State Machine

Into something like this:



Img 2.14) States of a Goal Oriented Action Planning model

This approach has the obvious advantage of decoupling states form each other, so the programmer can focus on each state separately, without worrying about their relationships and possible gaps.
The heart of this model is the *GOAP Planner*, a piece of code (we won't go into the implementation itself, as this can be obtained in many different ways) that queues *Actions* based on *Preconditions* and *Effects* to achieve a *Goal*.
The *Goal* is the desired final result we want the agent to fulfill. There may be many ways an agent achieves its goal and the task of the *Planner* is to define the best possible sequence of *Actions* to reach that goal, based on local (the agent's) and global (the world's) conditions and action costs.
An *Action* is defined as something the agent does (it usually corresponds to a State of a State Machine), and several *Actions* are queued by the GOAP Planner defining a behaviour towards the predefined goal.
Each *Action* has a cost. Costs are important because they help the *Planner* in comparing different routes that lead to a goal.
A *Precondition* is a state required for a particular *Action* to run and an *Effect* is a change to a state after the action has run. As an example we consider the *Action* “Get a shopping cart”. *Preconditions* for this could be “I don't have a shopping cart” and “A free shopping cart is available somewhere”, while an *Effect* of the “Get a shopping

cart” *Action* is the change from the state “I don't have a shopping cart” to “I have a shopping cart”.
As mentioned above, when the *Planner* has determined several sequences of actions that lead successfully to the desired goal, it sums the costs of the actions in each path, and the settles for the sequence with the lowest overall cost. With this the *Planner*'s task is finished, and is up to another part of the code to execute the final sequence of actions in the decided order.

However it should be evident that the GOAP is merely a substitute for a high-level decisional state machine, as it is not suited to manage low level actions such as moving, playing an animation or a sound effect, which still need to be enclosed in a traditional FSM. GOAP models the decision making process that plans ahead at a high-level, queueing “super-states” in an intelligent manner. Depending on the agent, this high level decision layer may be quite complex, and the more it is, the more one should reap the advantages of a Goal Oriented approach.
Using the situation of a shopper in a grocery store (coherently with the situation presented in our project) as an example, a Goal Oriented approach could be used by the agents to decide to take a shopping cart or not, based on preconditions such as: how many items do I need to buy? How many items can I carry by hand? Is there any cart left in the deposit area? The goal here being, obviously, to buy all items in my shopping list.
However the low level management of states such as planning a path, playing the “walking” and “take item” animations, go near a shelf containing a product are not part of what the GOAP does and still must be managed by a regular State Machine.
While certainly interesting, this Goal Oriented approach wasn't needed in our case. While some decisions are made at this high level, the overall state machine (see Image 3.1) is very linear and too poorly interconnected to meaningfully implement such a model.
Some elements amenable to a Goal Oriented Planning can however be found in the route planning algorithm for *Optimized/Anti-Optimized* Shoppers (see chapter 3.1), where the shelves to visit (that is, the “Walk To” actions) are queued using a cost-evaluating algorithm, in this case the cost being the length of the path connecting each shelf position.
In this case the goal could be formalized as “Buy everything from your list while walking the shortest overall path possible” and the given world information is the layout of the shelves and their relative distance.

2.4 Machine Learning

The second approach, Neural Networks and Machine Learning, is without any doubt very “trendy”, and potentially very powerful, since its aim is to develop a machine that simulates how the human brain works and learns to perform tasks.
Since we don't make use of machine learning in this project, only a short summary on this topic will be given, even though the literature is quite extensive on the subject.
Neural Networks must be “trained” on a set of data (the larger, the better), so they learn to recognise and replicate patterns in ways that can be defined “emergent” or even “creative”.
This kind of Artificial Intelligence has the upside that it can potentially adapt and produce results that were not foreseen by the programmer.
The downside is that the training process and which data are fed to the Neural Network are of critical importance to obtain the desired result. Also the training is extremely intensive from a computational standpoint, often requiring many hours if not days (on an average machine) to obtain somewhat acceptable results, and even then there can be undesirable side-effects, the most notable being *overfitting*.
Of the most notable categories of Neural Networks we want to cite are Recurrent Neural Networks (RNN).
Recurrent Neural Networks are very good at learning and reproducing simple data sets, such as text. There are many examples of people using RNNs to create computer-generated music, lyrics for songs, pages of books and even geometry texts *ex-novo* after training this kind of neural networks. For more information on this topic, we remand to the article by A. Karpathy “The Unreasonable Effectiveness of Recurrent Neural Networks” [6].
A particularly interesting sub-class of RNN are the so called Long Short Term Memory Networks (LTMS), introduced by S. Hochreiter at al.(in the article “Long Short Term Memory”[7]), capable of learning long-term dependencies, and several researchers are obtaining very interesting results using them. As an example, we cite K. Xu “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention ” [8], on the topic of automatically captioning an image, based not in its tags in text form, but on the actual contents of the picture.

It must be noted that the development team of Unity released in September 2018 the ML Agents toolkit, a framework designed to create, develop and train Intelligent Agents in the Unity engine itself (for more information, see this paper by A. Juliani et al. “Unity, a General Platform for Intelligent Agents” [9]).

2.5 Computer Animation Techniques

In the field of computer animation the most used procedures are based on hierarchical models.
In a hierarchical model a series of objects are connected in a structure of nodes and links, usually representing joints and limbs. Each joint may have multiple and varying degrees of freedom, for example a shoulder has 3 DOFs, those being the xyz rotations, while the elbow only has 1 DOF.
In the hierarchical model, a transformation applied to the root of the structures is propagated to every child node and link. Constraints can also be applied to each joint, to avoid unwanted movements (for example an elbow can bend more or less 150-160 degrees and only around one axis).
Human and animal skeletons are easily modelled using these hierarchical structures called Kinematic chains, while more complex animations, for example those involving facial muscles or amorphous objects or creatures may be more easily achieved with different methods not involving hierarchies, such as *blend shapes* or *morph targets*.

When talking about animation of a hierarchical structure, two different approaches can be employed: Forward Kinematics and Inverse Kinematics.

Now let's discuss how exactly these two processes work in detail:

2.5.1 Forward Kinematics

Abbreviated as FK, in this process the position of the end-effector is computed by a series of kinematic equations applied to the joints of a hierarchy.
The end-effector is defined as the part of a robot arm that interacts with the environment, for example for a human arm the end-effector would be the hand or a finger.
The position of the end effector can be calculated as the product of a stack of matrices representing the rigid transformations of each joint, traversing the hierarchical tree from the root to the end-effector.
The convention of using translation and rotation matrices for each joint has been standardized in the '50s by Jaques Denavit and David Hartenberg, and the formula to obtain the link transformation matrix, known as the Denavit-Hartenberg matrix, is the following:

$$^{i-1}T_i = [Z_i][X_i] = \mathrm{Trans}_{Z_i}(d_i)\,\mathrm{Rot}_{Z_i}(\theta_i)\,\mathrm{Trans}_{X_i}(a_{i,i+1})\,\mathrm{Rot}_{X_i}(\alpha_{i,i+1}),$$

$$^{i-1}T_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_{i,i+1} & \sin\theta_i\sin\alpha_{i,i+1} & a_{i,i+1}\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_{i,i+1} & -\cos\theta_i\sin\alpha_{i,i+1} & a_{i,i+1}\sin\theta_i \\ 0 & \sin\alpha_{i,i+1} & \cos\alpha_{i,i+1} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

For simple systems, this analytic solution can be applied easily, but for more complex hierarchies, numeric methods are used instead.
One drawback of using Forward Kinematics is that a precise position of the end-effector is very hard to compute beforehand, and the difficulty increases exponentially as the kinematic chain's size is increased and as a result, FK can be a long trial-and-error process. Usually the preferred method, especially for human figures, is Inverse Kinematics.

2.5.2 Inverse Kinematics

Abbreviated as IK, this technique is the reverse process of Forward Kinematics, where the position of the end-effector is specified first, then the position and/or rotation of each other joint is computed afterwards.
There are several mathematical methods to solve Inverse Kinematics equations, typically the most flexible rely on iterative optimization of errors due to the difficulty of inverting a Forward Kinematics equation, and also because the solution space may be empty. On this note we must precise that usually there are multiple if not infinite solutions to an IK problem, so defining constraints for each joint or a plane on which all the joints must be aligned significantly improves the visual result and also simplifies the calculations. On the other hand the finial specified position of one or more end-effectors may be unreachable with the given constraints. In this case the strength of an error approximation method is obvious in that it returns the solution closest to the desired final position of the end-effector.
Most analytical methods for IK use a Taylor-type series to approximate the FK equation because it's a way easier function to invert, compared to the Denavit-Hartenberg functions mentioned above. The most common of such methods is the algorithm based on the inverse-Jacobian matrix method. While we omit the procedure itself, a detailed and extensive explanation can be found in this article by Luis Bermudez, “Overview of Jacobian IK” [10].
A similar algorithm based on the Hessian matrix is sometimes used, and can converge to the same error Δx in fewer iterations, but at the cost of more computational resources.
Also some heuristic algorithms are used to approximate IK problems. The most

notable of these methods are: *Cyclic Coordinate Descent* (CCD), as detailed in this paper by D. G. Luenberger et al. “Linear and non linear programming” [11], and the *Forward And Backward Reaching Inverse Kinematics* (FABRIK), explained by A. Aristidou et al. in their “FABRIK, a fast, iterative solver for the Inverse Kinematics problem” [12] document.

It must be noted that IK and FK are not mutually exclusive, on the other hand they are used in conjunction with each other. For example the position of the clavicle can be specified via FK, and then shoulder, elbow and wrist parameters are subsequently computed via IK based on the desired final position of the hand.

In the context of a game or a simulation, animations can be pre-recorded or procedurally generated at runtime.

> -*Pre-recorded Animations:* These are basically tables of positions, rotations and scales for each joint, keyframed in a dedicated program, and can be played if arbitrary conditions are met. While there are some parameters that can be modified, such as speed, offsets, loops, they are always the same.
>
> -*Procedural Animations:* These are computed at runtime, by controlling bones and joints in the skeleton hierarchy via code.

The pre-recorded approach is by far the most common in games, movies and simulations. Starting from a known kinematic chain, animators use the aforementioned IK or FK techniques or motion capture techniques to craft the desired animations in a dedicated program or a 3D modelling program (such as Blender, 3DS Max, Maya, etc.), and the export them as one or more files to be used in another environment (such as a game engine like Unity) more or less as-is.
Animations can be hand-crafted via keyframing methods. In this instance, the animator specifies the position of bones in the kinematic chain, via IK or FK, at given (key)frames, and the program automatically interpolates positions and rotations (and scale, in some cases) of joints and limbs between the specified frames.
Increasingly more common and less expensive is the method involving the so-called motion capture, where an actor performs the required animations and his/her movements are recorded and then transferred to a particular kinematic chain.
In optical motion capture systems, the actor wears a series of markers, one or more for each joint, and a system of two or more cameras triangulate the position of each joint in a three dimensional space.
Here is a non-exhaustive list of optical motion capture techniques:

> -Optical with active markers
> -Optical with passive markers
> -Optical with time-modulated active markers
> -Optical with semi-passive imperceptible markers

-Optical markerless

There are also other non-optical techniques sometimes used as an alternative, those comprehend:

-Inertial systems (gyroscopes, magnetometers, accelerometers)
-Mechanical systems (exoskeletons)
-Electro-magnetic systems
-Acoustic systems
-Radio-Frequency positioning systems

In this project we employed already-made open-source animations, downloaded mainly from the site Mixamo.com, but since they didn't fit perfectly our needs, we have implemented procedural system on their top, as will be explained later in the chapter about Agent Animations.

2.6 Use of VR in medical practice

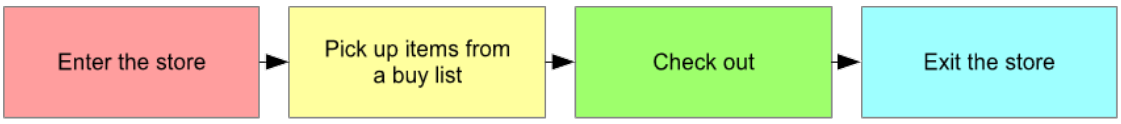
With the recently rekindled interest in Virtual Reality and its applications, due to the technological advancements in the field and the increased availability of consumer devices (most notably HMDs such as Oculus Rift, HTC Vive and PlayStation VR), medical practitioners are among the most interested in taking advantage of this technology.
Virtual Reality can be taken advantage of, for example, by surgeons or students to rehearse or learn surgical procedures (for more information on this topic and the actual usage of VR in this kind of medical practices, we remand to the paper "Recent Advancements in Medical Simulation: Patient Specific Virtual Reality Simulation" by W. I. M. Willaert [13]).
Not only practitioners can have benefit from VR but also patients. Several studies, (such as the one by HG. Hoffmann [16] "Virtual Reality Therapy", M.B. Powers "Virtual Reality Exposure Therapy for anxiety disorders: A meta-analysis" [15], B. K. Weiderhold "Virtual Reality Therapy for anxiety disorders: Advances in Evaluation ad Treatment" [16]), demonstrate that the use of VR simulations can be extremely effective in treating anxiety disorders of various nature. These procedures are known as VRET, Virtual Reality Exposure Treatment, the most famous example being the fear of spiders, as documented in these papers, also curated by HG. Hoffmann: "Virtual Reality in in the treatment of spider phobia: a controlled study" [17] and "Virtual Reality and tactile augmentation in the treatment of spider phobia: a case report" [18].
Other anxiety disorders for which VR therapies are being experimented are: fear of flying, fear of heights, fear of speaking in public and even Post Traumatic Stress Disorder for Vietnam veterans (the effectiveness of which must still be proved, but as stated by a case study by B. O. Rothbaum [19] "holds promises").
The same study [16] also shows that the use of VR can even ease physical pain in addition to psychological pain, as patients undergoing painful treatments report noticeably reduced levels of perceived pain when "distracted" with a VR application.
The immersiveness of a virtual reality environment really pushes this practices over the edge, as they proved much more effective when compared to similar "distraction" experiments done with traditional video games played on a screen.
The obvious advantages of using a VR therapy are that are easily customizable to be adapted to any patient, and can be also more time and/or cost effective than traditional practices (for patients with fear of flying, one of the final stages of treatment is to actually take a plane together with the therapist. If a VR treatment is used instead, it saves the cost of two airplane tickets!).

3. METHODS AND IMPLEMENTATION

3.1 Approach

The approach that was chosen was to go with the simple “classic” AI based on Finite State Machines, since it was deemed sufficient for the task at hand.
Besides, the Unity Engine offers a very solid (though NOT perfect) framework for navigation and collision avoidance (*NavMeshAgents*) to base our work on.
This approach is also much simpler and easier for the computer to manage, compared to the massive training time and effort that a Neural Network approach would have demanded, with no guarantee of a satisfactory result, on top of that.
While the behaviour of our agents is quite linear overall, the number of states in very high, also having a certain degree of interconnectedness.
To model the behaviour model of the client/agents we employed a top-down approach, first defining a sequence of super-states at a high-level, and then expanded each of those in more detailed intermediate-level nested state machines.
This segmentation is necessary because an all-encompassing low-level diagram, FSM or BT, would be too complicated, unreadable and effectively unusable for practical purposes.
Due to their size and poor readability, the detailed low-level state machines that mirror exactly the code implementation are omitted due to their size and poor readability, while in the actual chapter we will focus only on the intermediate-level state machines because they are far better for conveying and understanding the concepts we will talk about.

To define the high-level behaviour model we answered the simple question: “What do clients in a grocery store do?”.
The following diagram is used as a base to define the series of *super-states* or “phases” the client agents go through:



Img 3.1) Agents' high-level state machine

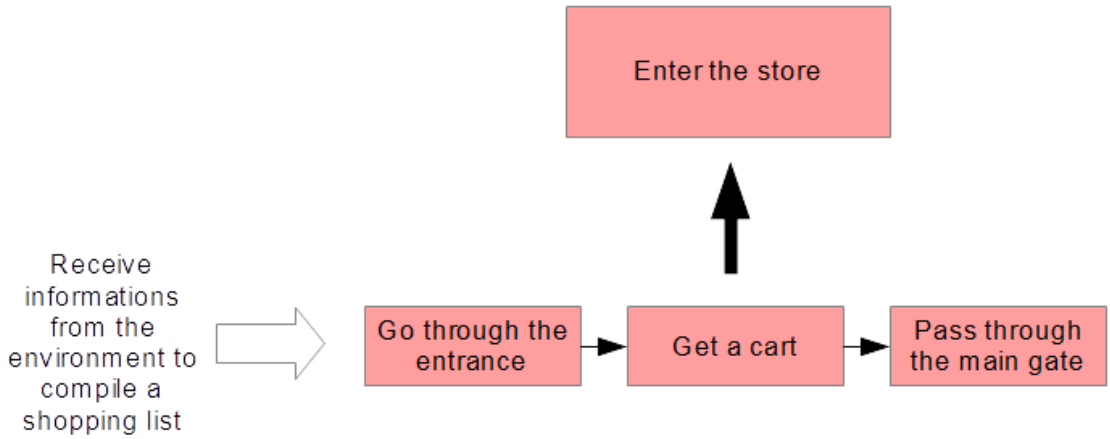
The most obvious consideration about this sequence of super-states is that it is completely linear, devoid of any branching path. This suggests that a FSM approach is sufficient to handle it, and a more complicated model such as a Goal Oriented Action Planning is not necessary at this level.
Obviously this is extremely generic and doesn't help at all in defining an actual

practical implementation.
In each sub-chapter we will see how each of these super-states contains expanded intermediate-level nested FSMs.

3.2 Enter the store

3.2.1 Logic

This super-state generalizes the phase where the customers spawns and goes through the main door, then completes all the tasks needed to setup the next phase. All of this is more complicated than it seems on the surface.
First of all we must define a point where the agent spawns. This place must be outside the store entrance, so the user can witness the agent actually entering the store through the main door.
The next fundamental step is to define the shopping list for the newly spawned agent. One may argue that since the agent must complete other sub steps before actually searching items in the shelves, this process could be postponed, but since the list is static (i.e. it does not change during the sequence of states) we might as well do it now when the agent is created.
It is obvious that clients of a grocery store are usually equipped with a shopping cart, so to define how, where and when they get one is an important issue.
It would make no sense to have agents spawn with a shopping cart already, because this is an highly unlikely occurrence, making it seem like the person brings their own personal cart from home, and that would make the simulation a lot less realistic.
So next the agents need to navigate towards a previously defined cart deposit area. As of now, for each newly spawned agent a shopping cart also spawns in the deposit area. This was done for practical reasons, even though a cart popping into existence at an arbitrary moment in time is not particularly realistic.
Since a deposit area is defined and also information on the maximum number of clients is available, an appropriate number of carts could also be spawned when the scene is loaded with very minor tweaks to the code.
The last step of the “Enter store” super-state is to make the agent, now equipped with a cart, go through the internal entrance gate. While this may seem obvious on the surface, it's not. When agents navigate in the simulated environment they use a pathfinding algorithm that enables them to reach their destination via the shortest possible path. The problem is that sometimes this path, especially in this beginning phase, goes through areas that in a real setting a normal person wouldn't pass through, that is: sometimes the shortest path passes through a passage reserved for exiting the store or for queueing at the cash counter. In this instance a real store client would take the gate labelled as “entrance” even though this means to lengthen their path, instead of passing through a “forbidden” area.
That means that we must set an artificial waypoint past the entrance gate to force agents to navigate towards that instead of taking the optimal path (“optimal” by following their logic based on taking the shortest possible route).
The individual sub-states discussed until now are represented in the following intermediate-level FSM:



Img 3.2) Intermediate-level state machine of the "Enter the store" phase

As we can see again this intermediate-level sequence of states is completely linear but perfectly adequate in describing the behaviour of a reasonably realistic store client.
The arrow on the left represents the agent gathering meta-informations from the outside world, in this case the list of available items in the store.

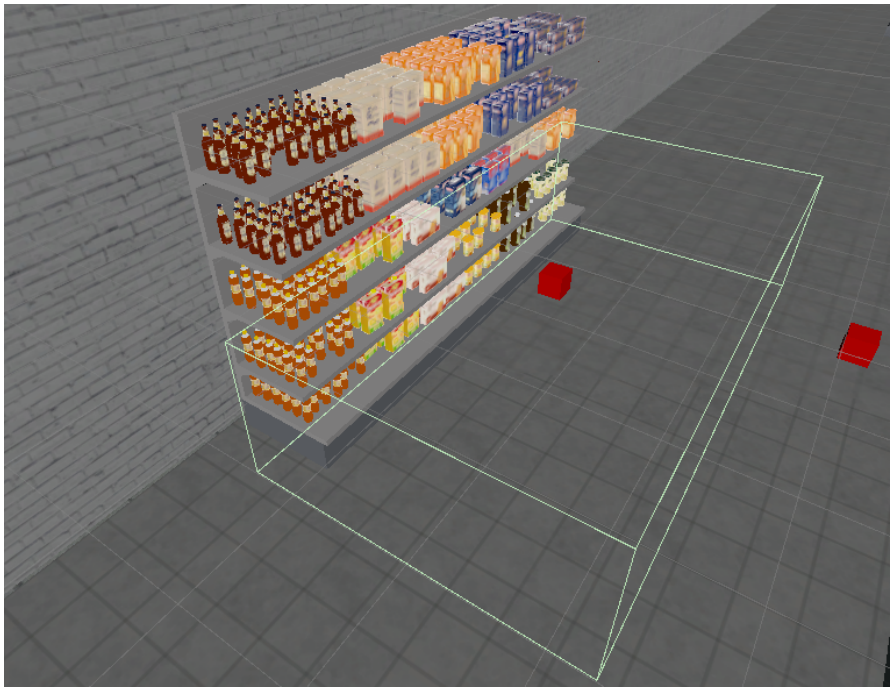
3.2.2 Practical Implementation

Before the agent goes through any of this states, it must effectively be created (or spawned), we now explain in detail how the "Client Spawner" element of the simulation behaves.
The "Client Spawner" must have a series of features to achieve maximum flexibility both in the testing phases and the actual simulation.
In detail, the implemented features are the following:

-An easy way to set upper and lower limits to the number of customers in the store. This is obviously highly dependant on the size and layout of the environment.
-A way to spawn customers manually, through a keyboard command, mainly for testing purposes
-A way to spawn customers automatically, at arbitrary intervals. This is useful both in testing and in the actual simulation. The spawning interval is customizable.
-A way to have already some clients in the store as the simulation starts, so that when the user enters the store while the simulation runs, he/she is in an already populated environment as a real store would be.
-A way to keep the client count constant, so that when an agent is destroyed, another one is created to take its place.

Many of these features, besides being requirements for the final product, were very useful in performing various tests during the making of the project.

Taking into account the modular nature of the simulation, special attention must be placed on the algorithm that defines how the shopping list (labelled as “buylist” in the rest of the chapter) is filled up. We want to avoid a situation where an agent has an item in its buylist that is not actually located in the store, or a situation where an agent needs to buy an item that was previously available but has been sold out in the meantime (however failsafe procedures to confront such occurrences have been implemented nonetheless).
When the store layout has finished loading, a global list is compiled with every type of product present in the store and their relative quantity, for each entry there is also information about which shelf and *shelfboard* the product can be found in. It's important to explain how shelves are organized.
Each product can be found in more than one shelf, and each shelf can have many types of products. Also each shelf is divided into (vertical and horizontal) sections, each of these areas is known as *shelfboard.* Only one type of product is present in a single *shelfboard*.
In the current project there are a total of 9 variants of possible shelves, most of them have products only on one side, while others can have items on up to 4 sides.



Img. 3.3) A shelf object. The red cube is a *NavMeshTarget*. While the area highlighted in green is the *CartParkArea.*

Highlighted in red in the picture is a *NavMeshTarget*, while the bigger area around it, denominated *CartParkArea*, will be used in the following phase for navigation.

Then when an instance of a client/agent is created, it selects a number of products randomly from this global list, removes them and then they are added to the agent's buylist. It's important to remove items as soon as they are added to a buylist to avoid problems afterwards.
The number of items in each shopping list is clamped between a minimum of 3 and a maximum of 10 because those seemed like reasonable values in our case, however any upper/lower bound can be chosen in the final product without any modification to the script.
The buylist contains information about the type of products but also on their location in the store.
As soon as the buylist is compiled, this info about the placement of the relative shelfboard is used now to create a series of objects called *NavTargets.*
These *NavTargets* are used in the following phase as waypoints to help the agent navigate between the shelves.
The 3 dimensional vector containing the position of each *NavTarget* is calculated with the following formula:

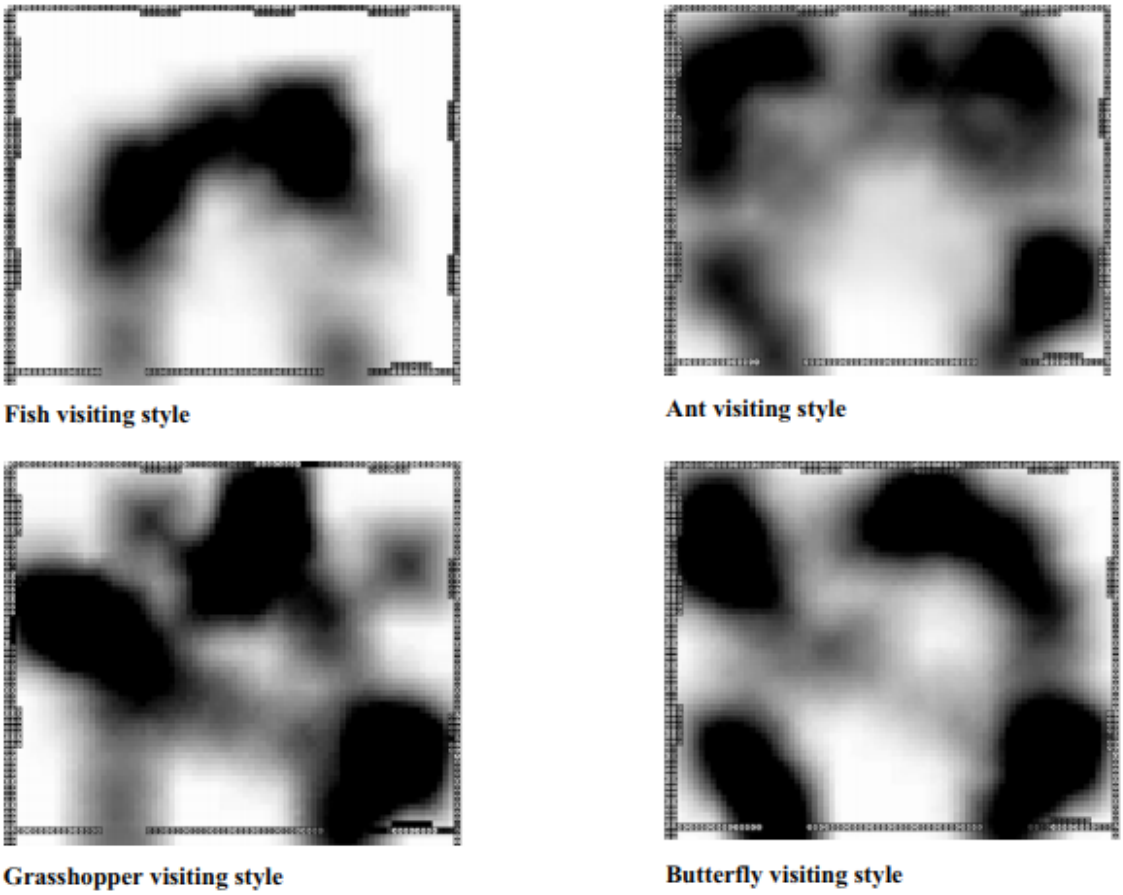
$$NavTargetposition = (x_{sb}, 0, z_{sb}) + (sb_{forward} * k)$$

where:

- x_{sb} and z_{sb} are the corresponding coordinates of the *shelfboard* itself
- $sb_{forward}$ is the forward pointing unit vector of the *shelfboard*
- k is a floating point constant dependent on the global scale of the environment

This formula basically places the *NavTarget* on the ground (because **Y**=0) in front of the *shelfboard,* in a position reachable by the agents (determined by the *k* constant).
When all the *NavTargets* have been placed, a list of coordinates corresponding to each of them is compiled. We will call this list *orderedShelfList*.
It must be specified that these *NavTargets* objects are invisible, even though in the testing phase they where shown as red cubes for debugging reasons.
When the individual buylists are set, there is no particular order to them, they are, in fact, absolutely random. It's here where come traits of “personality” can be applied to each agent.
Two papers by Vanhoof [20] and Vernon [21] have been taken into account when trying to implement these personalities. The study by Vanhoof is focused on grouping customers of grocery stores into clusters independent from socio-demographic or lifestyle characteristics, but while interesting the results are not considered meaningful in our case. The focus of this simulation is not to recreate shopping patterns of consumers on a macroscopic scale in a way that would be useful to store owners or marketing experts, interested in fighting over the “share of wallet” of customers and in organizing the layout of a store. The purpose of this project is to be immersive on a local scale from the point of view of a customer, and the behaviour-based clusters explained in the paper are not meaningful and probably not even observable at such a small scale.

The paper by Vernon is focused on museum visitors, grouping them in 4 different classes: Ants, Grasshoppers, Fish and Butterflies. A brief explanation about these behaviours will be given, in part because another study by Luca Chittaro and Lucio Ieronutti [22] focused specifically on Virtual Environments, employed Vernon's classification of visitors to show how actors in a simulation can be categorized based on their patterns of movement. Since this project focuses on a simulation of a very specific virtual environment, some elements of the technique and tool (VU-Flow) outlined in their paper could be employed in the context of a grocery store to further study the patterns of shoppers and come to a more complex categorization than the admittedly simplistic one that was implemented.



Img 3.4) Visiting Styles as visualized with the VU-Flow Tool by L. Chittaro and L. Ieronutti [22]

The *Fish*-style visitor walks predominately in empty spaces and spends little time for each item in the museum and while he/she examines most of the exhibits, few or no stops are made.
The *Ant*-style visitor on the other hand spends a quite a long time for each item on display, makes frequent stops and avoids open spaces, keeping close to the walls.
The *Grasshopper* visitors have pre-existing knowledge of the exhibits or very specific interests and only stop by the items of their liking, spending a long time on each of those while crossing open spaces to reach them.
The *Butterflies* stop frequently and usually see all the exhibits, often changing

directions and avoiding empty spaces, time dedicated to each item may vary.
Again, while interesting, the study is focused on museums, not grocery stores so a parallel between museum-goers and store shoppers may or may not exist.
However, while not implemented exactly, this user classification inspired the personality system that was implemented in the end.
Three types of personality are possible in our simulation, they are called *Randomizer*, *Optimizer* and *Anti-Optimizer*. while they are not based on any specific study on customer behaviour, they are adequate in representing broad categories of store shoppers.

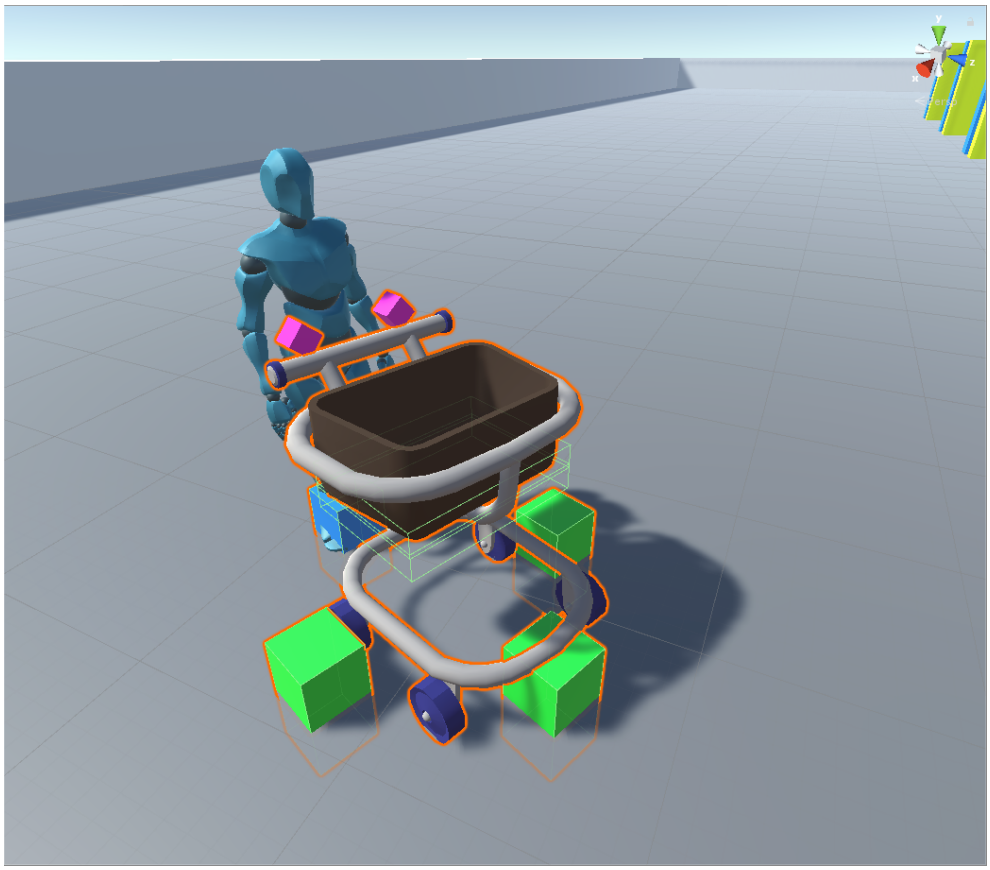
Randomizer is the easiest personality to implement, since it leaves the previously generated buylist as is. This personality represents a customer not very familiar with the layout of the store, or maybe just a person that buys stuff without an actual list and navigates in the isles as items come to his/her mind. They do not fit exactly into any specific class of Vernon's categorization (even though they often exhibit behaviours close to *Butterflies* or *Ants*) due to their random and unpredictable nature.

Optimizer is more complicated because, as the name suggests, this kind of customer organizes its shopping list with the objective of travelling the shortest possible distance. This personality represents a person who is very familiar with the layout of the store and doesn't need to lose time searching for stuff he/she doesn't need. They correspond to the *Grasshopper-type* visitor in Vernon's categorization.
As previously stated each agent already has a list of coordinates, corresponding to a *NavTarget* for each product in its buylist. This information is now used in an algorithm that calculates the distance between the entrance and each *NavTarget*, and selects the one which the closest.
The algorithm is then repeated for each item in the buylist, using the previously found *NavTarget* as a starting point instead of the entrance.
It must be noted that, in the first version of the algorithm, distances are calculated as straight lines without taking into account any obstacle, so some problems may arise: for example, an item in an opposing shelf of another isle is considered closer than an item in an adjacent shelf in the same isle.
To prevent this, two solutions have been considered: the first is to calculate the actual path between
shelves taking obstacles into account, but the problem is that Unity, while it offers a function to calculate a path, doesn't offer one to compute that path's length automatically. A workaround may be to calculate the length manually using that path's waypoints (which are stored in the variable *path.corners[]*).
The second solution, implemented in an early stage of prototyping, is to add an “isle” integer parameter to each shelf. When the algorithm calculates distances, it applies an arbitrary “malus” to the result if the shelf it's comparing to is part of another isle. While this solution worked fine in the mock up environment used in the early stages of development, it fails in more complicated environments because shelves cannot be clearly classified into isles, as such a restriction would greatly restrict the possibilities

of the environment editor.

The third and final personality, the *Anti-Optimizer,* uses basically the same algorithm as the *Optimizer*, but reversed since it computes the farthest distance between items. This personality may represent someone who is not in a particular hurry, or decides to buy products when he/she sees them without any specific order. They have some degree of similarity with the *Butterfly-style* visitor.

After all this setup and information gathering from the outside world, the agent can finally start its sequence of states.
In the first state the agent must go through the entrance, and head towards its assigned shopping cart and then interact with it. The seemingly simple tasks of "Going through the entrance" and "Getting the cart" hide the complex behaviour behind agent navigation, path finding and the details about the cart entity. While we will talk about the navigation and *NavMesh* components in the following chapter, we can now examine the elements in the cart component.



Img 3.5) The shopping cart object. Highlighted in green we can see the *LoadTargets* of the cart object. Also note the blue square on the back (*PushTarget*) and the purple cubes on the handle, which will come into play later when we examine the Inverse Kinematics animations.

The visible part of the cart entity, during the simulation, is only its mesh (the mesh seen in the picture was made as a temporary placeholder, switching to a better looking

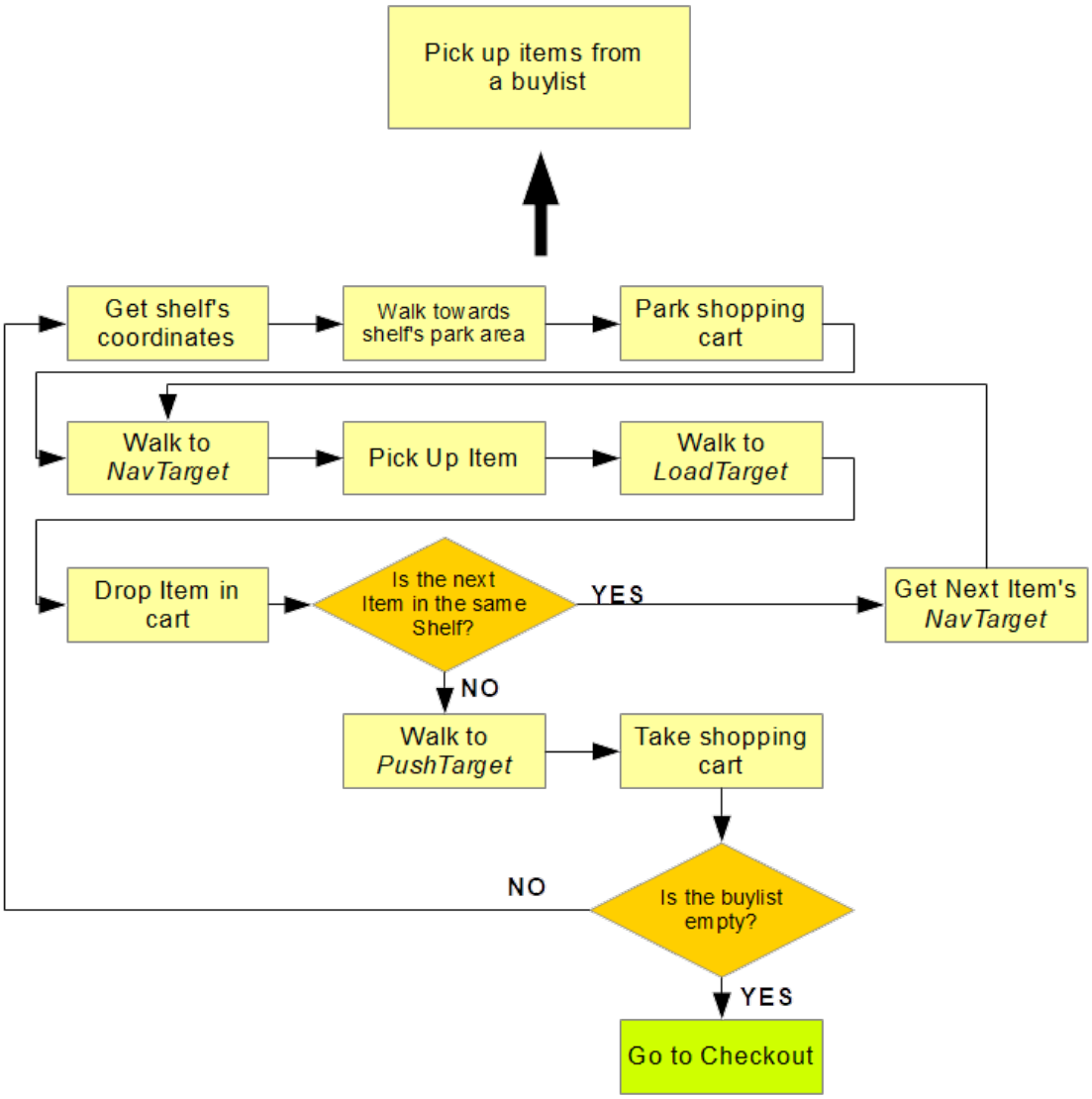
cart model is a very easy task, since almost no components are tied directly to the model's geometry), however highlighted in the picture there are several other hidden elements. The blue element on the back, where the client stands, is the so-called *PushTarget*. When the client is searching for the cart, the position of the *PushTarget* is returned, not the centre of the model.
The three green cubes around the cart are denominated *LoadTargets*, and are used in the following phases as destinations for the agent when it needs to put in or take out items in the cart.
There are three *LoadTargets* instead of just one because not all of them may be reachable at a particular moment. Take for instance a situation where the cart is parked near a shelf or a wall, int this case one of the side *LoadTargets* cannot be reached, so when the agent needs to perform an action involving taking an item to or from the cart, it queries the cart entity (through the built-in Unity function *NavMesh.CalcuatePath*, which returns a boolean result) for the nearest reachable *LoadTarget*. The purple cubes near the handle are used in the animations involving Inverse Kinematics that will be talked about in the next chapters.
Even if they cannot be seen clearly in the picture, the cart basket has five box-shaped colliders around its edges (one for each side except the top), used for physics collision with the objects. We opted for five separate colliders instead of one hollow mesh collider because it has much less polygons (making the already resource intensive physics engine work with simpler geometries) and also we want to be tied as little as possible to the actual mesh.
In this way if in a future version of the project the model of the cart must be switched for one reason or another, the only thing we need to do is to tweak the box colliders a bit instead of having to deal with a completely different mesh collider that may or may not work as intended.

Finally, the last sub-state is just to go through the entrance gate. This was achieved by placing an invisible *NavTarget* a short distance past the gate itself, avoiding the event that the agent passes through a “forbidden” area (like “Entering through the exit” which is self-explanatory as to why that would be wrong).
After entering the actual shopping area, we can proceed to the next super-state.

3.3 Picking Up Items from a Shopping List

3.3.1 Logic

This can be considered the main phase as it's the one that takes up most of the time in the agent's life and also shows off some of the most complex examples of navigation and animation techniques.
Let's see how the intermediate-level state machine for this phase is organized:



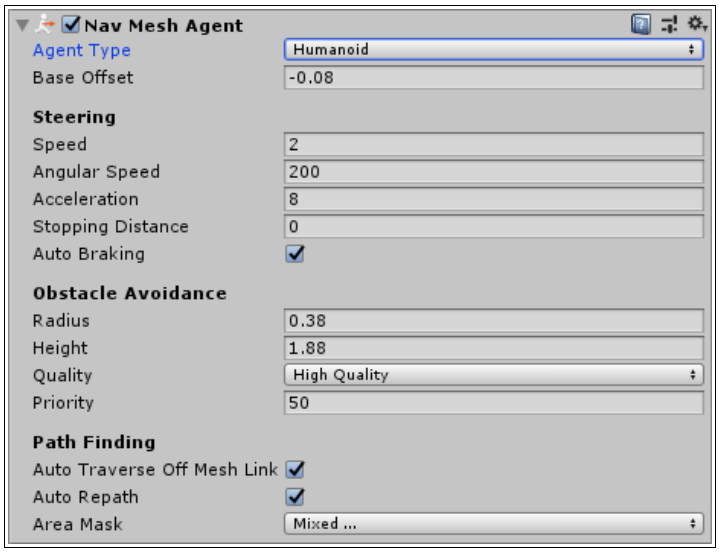
Img 3.6) Intermediate-level state machine of the “Pick up items” phase

As we can see there are many more sub-states when compared to the previous super-state, and while there are two branching paths, again the structure is predominately linear.

In this super-state we want to reproduce a realistic behaviour where the agents navigate between the shelves and products of the store, both with and without their shopping cart depending on the situation, avoid collisions with other agents and objects in the environment. Also they need to approach shelves, pick up items and put them in the cart until their shopping list is completed. Again, this seemingly simple series of tasks implies several factors related to: navigation, path finding, collision avoidance, animation ans physics. We will now discuss how we dealt with all of these problems.

3.3.2 Practical Implementation

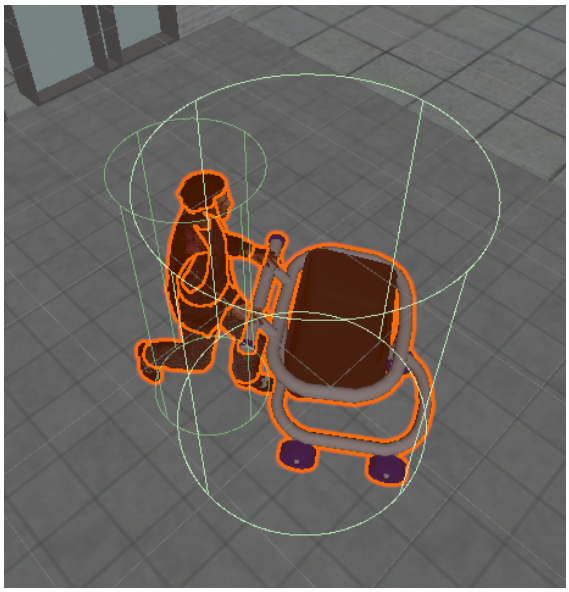
In the previous chapter we talked about how the coordinates of shelves and their respective NavTarget are calculated and stored in a list by the agent.
However we didn't say much about how the navigation is handled in practice..
To make the agents move we used the built-in Unity component *NavMeshAgent*. The component has several parameters and looks like this:

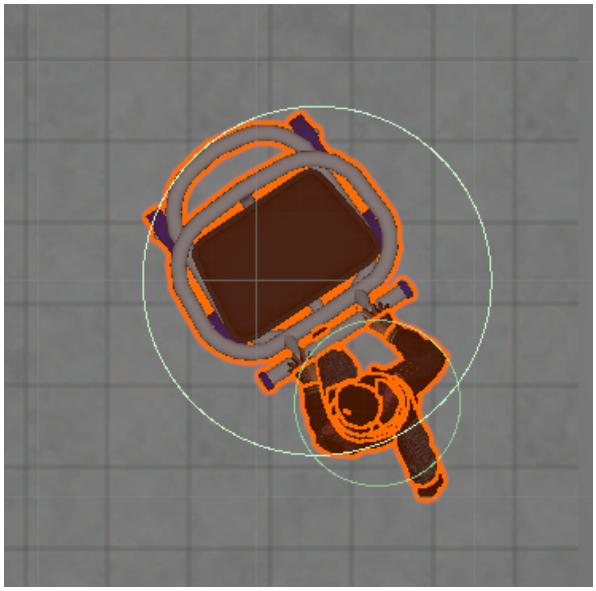


Img 3.7) The NavMeshAgent component and its parameters

To make the agent move we have a choice between two methods: *SetPath()* and *SetDestination()*.
The first one takes a Path-type object as argument and sets a fixed (pre-calculated) path which may be optimal, but has the drawback of not considering dynamic obstacles (such as the user or other agents).
The second method, *SetDestination()* takes a position (*Vector3*) as argument, computes a path at the beginning and dynamically re-calculates the route at fixed intervals. While less optimal in terms of distance travelled, this method takes into account dynamic obstacles, actively avoiding other agents. For this reason this is the method that has been implemented.
However, there is one huge draw-back in opting for the *NavMeshAgent* module: the

center and shape of the navigation mesh cannot be set. This is not a problem if we have a roughly humanoid-shaped agent by itself, but it becomes problematic if we add, as in our case, a shopping cart. The additional encumbrance of the shopping cart is not taken into account when navigating, and this results in undesirable effects, such as carts being stuck in shelves or clipping with objects in general.
This problem was solved by implementing two NavMeshAgent components: one for when the agent is alone as discussed above, and another larger NavMesh for the agent-cart system.
As soon as the agent takes or leaves the cart the two NavMeshes are switched, in the following pictures we can see the difference.

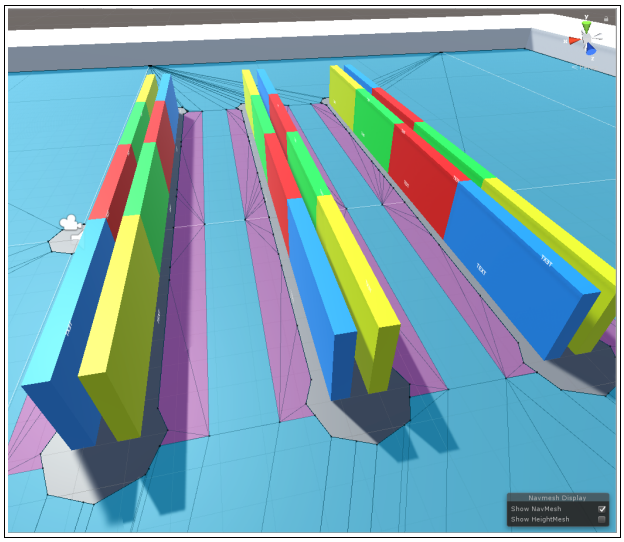


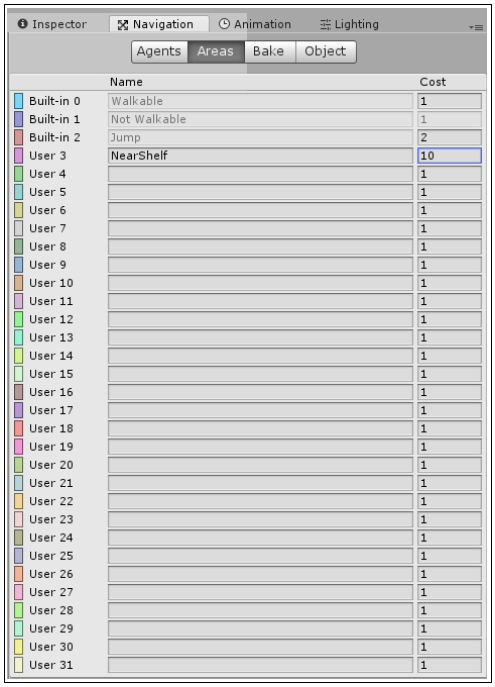


Img 3.8 & 3.9) A perspective and isometric view of the two different NavMeshes, highlighted in green. We can see how the larger NavMesh adequately covers the entirety of the agent-cart system.

By implementing this dual *NavMesh* system navigation and collision avoidance has improved considerably. When switching to the larger *NavMesh*, the pivot of the agent is moved slightly forward, but this is not a problem since in reality, when a human moves with a cart, he/she doesn't swing it wildly while planted on his/her feet but kind of rotates around an advanced pivot. Based on this fact, the change of pivot isn't noticeable and doesn't detract from the realism of the agents' behaviour.
There is however a drawback from using two different *NavMeshes*, but this is intrinsic in how Unity handles navigation.
To navigate, each **type** of *NavMeshAgent* uses data from a previously baked *NavMeshSurface*. A *NavMeshSurface* is a polygonal surface based on the layout of the environment, that tells the agent where it can or cannot pass through. It's important to stress that each **type** of *NavMeshAgent* needs its own separately baked *NavMeshSurface*, because characteristics such as the agent's radius, height, maximum slope angle etc. are dependent on the type and not the single instance of a *NavMeshAgent* component.

The radius and height parameters settable in the component are only passive and not active. For example, if the Humanoid-type agent has a radius of 0.5, we can set the radius a particular instance of that type of *NavMeshAgent* to 5000 or even more without affecting its navigation behaviour.
These parameters are considered passive because, while they don't matter for the agent itself, they matter for other agents when they are avoiding each other.
This is a drawback because two *NavMeshSurfaces* must be baked once the scene is loaded instead of only one, and this increases the overall loading time of a scene, but since we are trading significantly increased realism in navigation for a slightly longer load time, we consider this an acceptable trade-off.
Unity automatically creates the *NavMeshSurfaces* through a script with the same that must be attached to every element we want to be part of the navigable surface. Every other element (such as the shelves or other obstacle) creates “holes” in the *NavMeshSurface* which are treated as non-walkable areas. However this doesn't take into account doors or passages of various nature, so we implemented a modification to the code (in the form of a C# script called *NavMeshIgnore*) that deactivates their mesh during the baking process and re-activates them immediately after. In this way we guarantee that areas such as the main door or the internal entrance gate are walkable through.
During the prototyping phase, before implementing the dual NavMesh method, another solution was considered for avoiding agents navigating too close to the shelves.
This is based on the built in Area Cost parameter in the Navigation module. This enables to assign to a particular surface a cost for traversing that area. For example, in the prototype environment, we set normal ground with an AC of 1, and the “NearShelf” area to a greater value, like 5 or 10, so that the agents walked near the the shelves only when absolutely unavoidable.
Variations of this technique may be employed in further developments, for example, to better implement Vernon-Style personalities in the sense of avoiding/preferring walking in open spaces or near-wall areas. This is easily doable thanks to the fact that for each instance of a NavMeshAgent, personal area filters can be easily set.

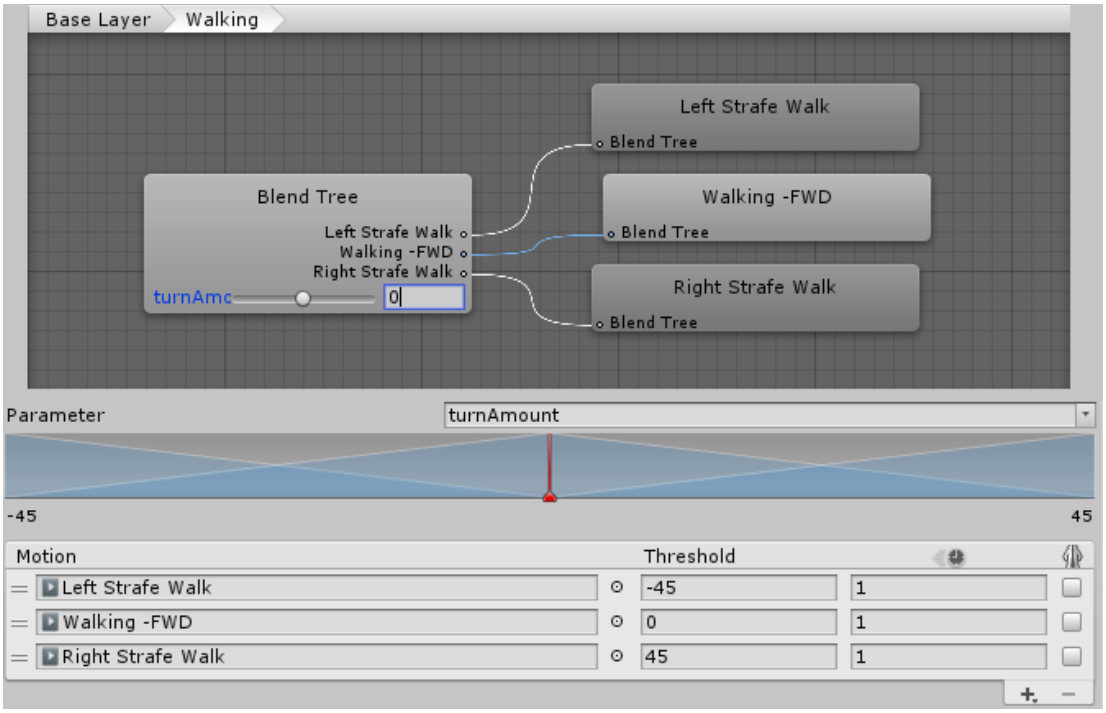




Img 3.10 & 3.11) The area cost window and its appearance in the editor. In purple we see the "NearShelf" areas with cost 10, in blue the normal "Walkable" area with cost 1. Grey areas are considered not traversable.

Another important aspect of the *NavMeshAgent* is the priority parameter. The priority is used in dynamic collision avoidance. When two agents with different priorities are about to collide with each other, the one with lower priority (by convention higher priority is the lower number, so the highest is 0 and the lowest is 99) stops or slows down to let the other agent pass by before resuming its path.
When agents are created, an algorithm assigns to each of them the highest available priority, so that no two agents have the same priority value. Note that the cap for priority is 100 agents, so if we pass this threshold, by having more than 100 agents on the scene, a random priority is assigned until some agents leave the store (i.e. they are destroyed).
Another device to improve dynamic collision avoidance between agents is to randomize their maximum speed. Even if the priority system shouldn't allow situations where two agents are colliding with each other, this may happen due to the variable nature of the environment. Setting agent to a different floating point maximum speed value helps quite a bit in avoiding agents being stuck with each other.

One a destination for the *NavMesh* has been set, the agent walks towards their destination.
This is a good place to explain how the walking animation is handled.
The animation speed is tied directly to the velocity of the *NavMesh*, so that a fast agent will have a faster walk cycle than a slow one (avoiding agents sliding or skidding on the floor). Also this animation is not a simple forward walk cycle, but instead is a blend of a normal walking animations and two lateral strafing animations. This was achieved with the built-in 1-Dimensional blending feature in Unity's animator.



Img 3.12) The animator setup for the walk animation(s).

The parameter called *turnAmount* that governs the animation blend expresses the angle between the forward-pointing unit vector of the agent's mesh and the direction of the *NavMesh*'s velocity. This makes it so that when the agent is turning, it progressively blends into a strafing animation, making movement much more natural.
Experiments have been done both with actual turning animations and these strafing animations, but the effect obtained with the latter turned out much better in terms of realism of movement. Also important to note is that (this is valid for both rotation ad strafing animations) the two animations must be different, and it is not sufficient to use the same animation twice but reversed. This is because the leading foot of the walk cycle must be consistent. If we blend a walking animation that leads with the right foot to another that leads with the left foot, the resulting blended animation will have no foot movement at all or turn out as something that resembles more a bunny hopping than a walk cycle.
Even though Unity offers functions to automatically calculate angles between two 3-dimensional vectors (even though here they are treated as 2-dimensional, since there is no vertical movement), we implemented a custom formula (because those functions weren't working very well) to calculate the *turnAmount* parameter:

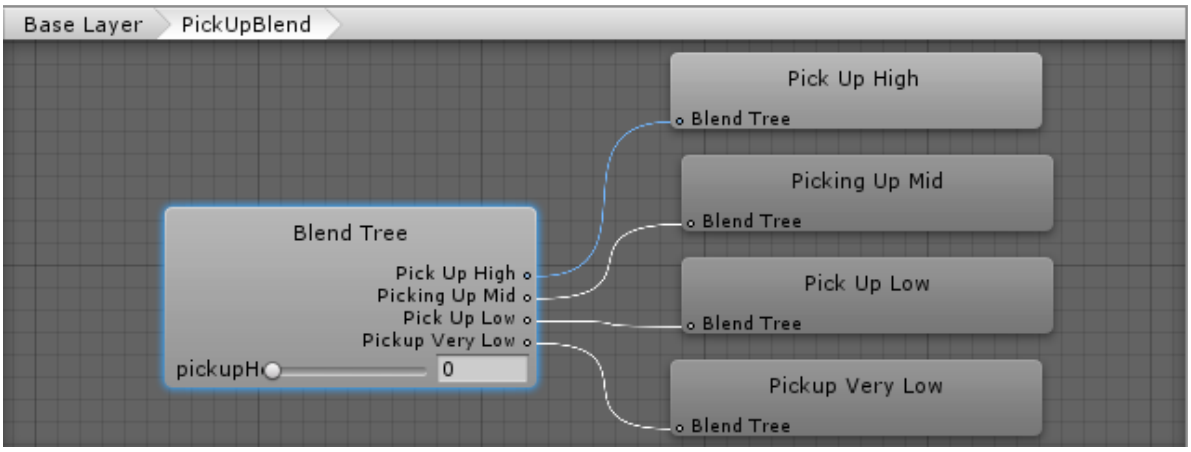
$$turnAmount = (arccos(fwd \cdot vel) * \frac{180}{\pi}) * (Sign(vel \cdot rgt))$$

where:

-*fwd* is the normalized forward-pointing vector of the agent's model
-*vel* is the normalized vector of the *NavMesh*'s velocity

-*(180/π) ≈ 57.3* is used to covert radians into degrees (57.3 is an adequate approximation in most situations and also spares a division).
-*rgt* is the normalized right-pointing vector of the model

With these necessary consideration about the agent's walk animations out of the way, we can proceed to the next sub-phases.
When the agents arrive in proximity of a shelf, they collide with the so called *CartParkArea* of that particular shelf. As soon as this happens the *CartParkArea* instead tells the customer agent to leave its cart a short distance from the actual destination, to avoid clipping with the shelf itself and also because when shopping in real life one is more likely to park their cart *a short distance* from the item they need to pick up, and not push the cart *against* the shelf. Also note that both the *NavMeshTarget* and the *CartParkArea* objects are not solid and only have a collider set as a trigger.
Going back to our agent: after getting the coordinates of the *NavMeshTarget* of the shelf they need
As the agent enters the *CartParkArea's* trigger collider, the cart is unparented from the agent, the NavMeshes are switched (so that now only the smaller one centered on the character is active) and also the Inverse Kinematics script is deactivated (more on this later). The cart is now set as a *NavMeshObstacle* so that its parent and/or other agents will actively avoid it. The now cart-less agent proceeds to the actual *NavMeshTarget*, and as soon as its collider is triggered, the agent adjusts its rotation to face the actual object it wants to pick up.
The actual object is given now, thanks to a function called *GetCloser()*, that returns the closest available product from a group.
The number of *CartParkAreas* for each shelf may vary depending on how many sides that shelf can have products on.
These areas are quite big relatively to the size of the shelf (they usually occupy an area bigger than the shelf itself), this creates situations where the agents park their cart in positions that are always a little different depending on their angle of approach. This improves the realism of the simulation, because if the agents always parked their cart in the same, predictable, spot, that would have seemed too robotic. A similar semi-randomized approach has been set also for queueing at the cash counter (more on that later).
When the agent has finished rotating towards the item, it plays the pickup animation. Here is where we need to talk about the animation techniques used for pickup.
As a base we have another 1-dimensional blend of four different animations. Each of these is a different pickup animation at a different height, which will serve as a base for implementing our custom Inverse Kinematics based script.

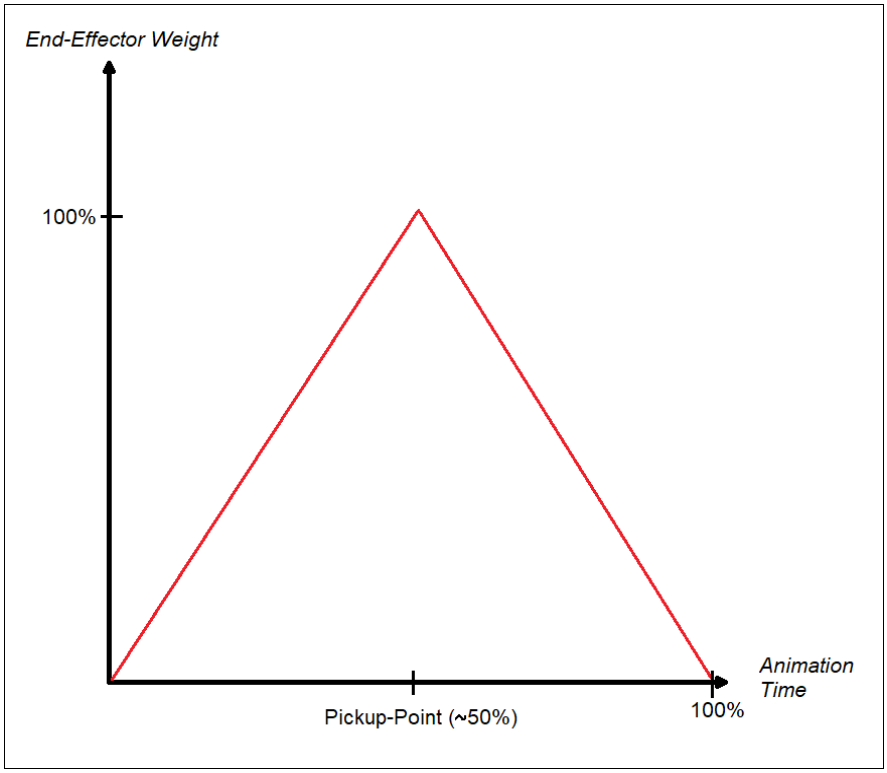


Img 3.13) The animator setup for the pickup animation(s).

The blending of these animations is governed by a the *pickupHeight* parameter, clamped between 0 and 1. This parameter is a constant calculated through an admittedly trial-and-error process, since it must be based on the specific features of the employed animations, which cannot be generalized.
The problem in using a limited number of pre recorded animations is that we cannot take into account every possible height for the pickup, since the layout of the shelves and the products placed on them may vary. Also they have no flexibility on the horizontal axis at all.
To fix these issues we implemented a custom IK script.

For this project we used several open-source animations taken from the database offered by the Adobe site Mixamo.com. The animations found on this site are generally of very high quality, and also they are highly compatible with the Unity game engine.
While the quality of these animations is very high, their variety is certainly limited.
A particular issue were the animations involving picking up an object or pushing the shopping cart.
Since the placement of various objects or the height or angle of the cart's handle could vary, a pre-recorded approach wasn't optimal to go with. Especially for the pickup animations we wanted the maximum possible freedom in item placement, and not be limited by the number of pre-made animations we had in setting up the shelves and items they contained.
The obvious solution was to go the procedural way. At first a Forward Kinematics approach was considered, but the complex calculations involved seemed a bit too daunting. Then, after a bit of experimentation, a solution was found in an hybrid method between pre-recorded animations and an Inverse Kinematics approach.
Unity offers a (somewhat limited, to be honest) couple of simple functions to work with IK, in the form of *SetIKPosition(effector, position), SetIKRotation(effector, rotation)* and *SetIKWeight(effector, weight)*.
The basic examples shown to use these functions, while interesting, didn't produce great results. While the IK actually worked, the effect was not realistic in the slightest, because it set the end effector to the final position instantaneously and in an awkward

way.
So the solution was to blend this IK-generated animations with a pre-recorded pick-up animation from Mixamo.com. The key to achieve a smooth, semi-procedural animation was to play with the *SetIKWeight()* parameter.
Instead of setting it to a fixed value, we made it start from 0 and then linearly increased the weight to 100% until a particular point in the animation (for the pickup animation, the spot where the character grabs the object) and then decreased the value back to 0 while the animation was finishing.

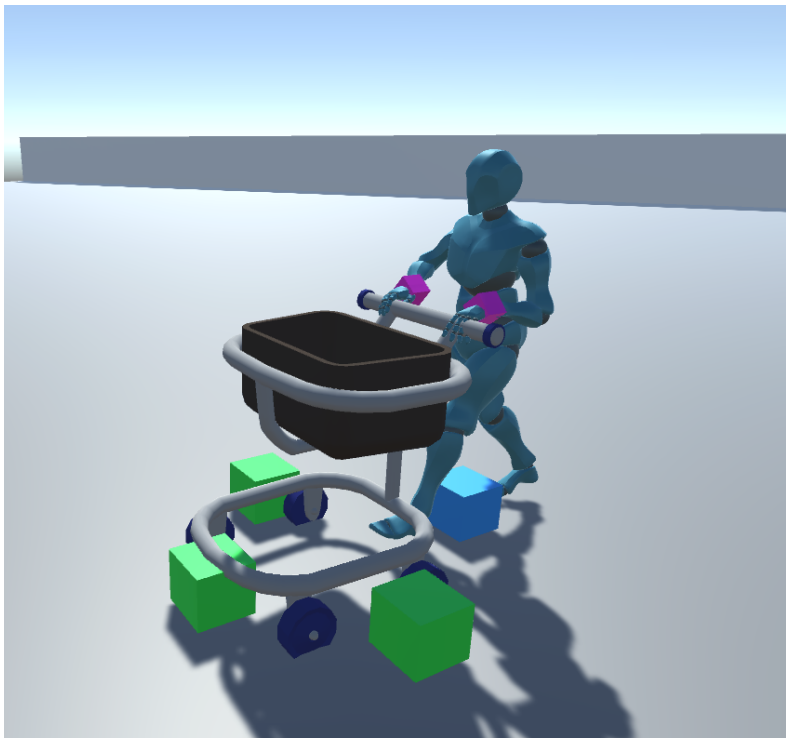


Img 3.14) A graph showing the logic behind the custom IK script.

While one may argue that a better result could be obtained with a non-linear function, (parabolic, exponential, a composite function with slices of sinusoids etc.), the end result was extremely satisfactory as is, with the added bonus that this approach is optimal in the sense that it keeps the calculations at runtime at a minimum.
The same approach was applied to the basic character walk animation, which had the problem of not being suited for pushing a shopping cart.
To increase even more the realism of the pickup animation, an additional animation layer with an avatar mask that includes only the hand (or even only the right hand, since all pickup animation used until now are one-handed) with different “grab” hand poses can be implemented.
Also a problem in our case is that agents reach for the center of mass of the product, instead of one of its sides. While this works well with round and small objects such as fruits, it's not very realistic when dealing with boxed items such as pizza or pasta.
To improve this, a calculation that finds the edge of a boxed product based on its collider can be made as soon as an agent starts the animation to pick it up.

Determining *which* side of the box the animation should aim for however is a different story and a simple calculation is not necessary, because if the side the “grab-target” is placed is against some kind of wall or other object, that would mean the agent's hand clipping with those objects.
This box/collider approach also doesn't fit for bottled products, where the “grab-target” usually is not on the edge.
Another approach could be to place these “grab-targets” by hand for each product (both via local coordinates, or actual invisible “grab-target” objects parented to the product). However this is an even worse problem because, even in a relatively small environment such as the one that was used for testing, there are thousands of products already, and placing these “grab-targets” would mean to double or even triple the amount of objects in the scene, and since performance is a major concern in the simulation, this surely can have a negative impact.
Also this would mean to attach custom scripts or objects to each product based on its shape, an operation that while easily doable requires some time to be implemented and also will be required if any new type of product is added to the grocery store database.

When the cart is being parked or pushed again, the custom IK script is called so that the agent-customer's arm move in a natural way going from one state to another. Also, by using this procedural approach, we are not tied to a specific cart 3D model, and it will be very easy to implement in future iterations of the project where more high-quality models may be used.



Img 3.15) The IK script in action. Note that the original animation was a simple walk cycle, the hands on the cart's handle are placed procedurally. Also note the purple cubes, used as targets for the IK functions.

For performance reasons, all products on the shelves a set as static objects, that is not only they have no physics applied to them but also they can't move at all.
For this reason a method was implemented that literally clones the object and returns and identical non-static copy to which physics can be applied.
A problem with our IK script is that it always needs a target, so when a static product is destroyed and the non-static clone is parented to the agent's hand, also the target for the script disappears, and even then, the target would be already in the agent's hand, so the script wouldn't work properly.
To avoid this, when a static product is destroyed, an empty, invisible placeholder object is created in its place, while the information about its transform are passed to the IK script.
Note that the script requires a reference to a transform and not a simple position or rotation, otherwise it wouldn't work with dynamic targets (such as the cart's handles. We could also send a new position to the script every frame, but that wouldn't be very efficient from a computational standpoint).
With this placeholder object, the “returning” part of the pickup animation works much more smoothly.
Note that the placeholder are destroyed as soon as they have fulfilled their purpose, since they are practically useless after the pickup animation is completed.
Another issue with the pickup and drop animations is that they have events tied to them. Unity grants the possibility to set specific events to happen at specific points in the animations. These events call actual methods in the code, with the added feature of passing values to them.
For example, the method that handles the destruction of a product on a shelf and its replacement by a non-static clone is tied to a specific moment in the animation (when the hand of the agent closes to actually grab something). A glaring issue with this practice is that it makes so that these events are completely tied to the animation itself, and if we want to replace the animation, we must also make new events for that animation.
The alternative to this procedure would be to implement a collision system between the character's hand the bounding boxes of the products. While this would solve the problem on the animation side, it would create a new problem on the character's side, where a collider would be tied to the geometry of the model. Also note that collision systems require quite complex calculations, so it's best to find alternatives to them when possible.
Another issue the agents can incur in is that the *NavMeshTarget* relative to a specific product could be in an unreachable area (i.e. an area that is outside the *NavMeshSurface* or a spot that is occupied by an obstacle or another agent or cart). This may happen for a variety of reasons. The most common we encountered during testing was due to the topology of some of the shelves, but that was fixed easily by disabling some problematic *shelfBoards* (note that those *shelfboards* were in places hard to reach not only for the agents, but also for the user).
NavMeshTargets can still be in unreachable places due to the layout of the environment, since there is nothing preventing the designer in placing shelves in front

of each other or in hidden/unreachable spots. Instead of trying to limit the creative possibilities of the environment editor, a failsafe code on the agent's side was implemented. As the agent enters the CartParkArea and leaves the cart, it enters in a “Reach for Item” state while activating a timer (a setting of 6 seconds for the timer was used in the final implementation, but it can be changed to anything). If in this time gap the actual *NavMeshTarget* cannot be reached, the agent returns to the cart and proceeds to the next item in the shopping list instead of remaining stuck there.
This doesn't detract too much from the realism of the simulation, as such a behaviour can be interpreted as someone just looking around and not finding what he/she wants or changing their mind at the last moment, a kind of behaviour that is not uncommon to witness in an actual grocery store.

As we already explained how the *LoadTarget* and *PushTarget* elements of the cart work, let's examine how the drop animation is handled.
Since we couldn't find any drop animation that fitted our needs, one of the pickup animations was used, only in reverse.
For the item drop itself, several solutions were experimented with. A simple “teleportation” from the agent's hand to the cart's basket proved to be the simplest solution, while this is certainly an easy and efficient, it's not particularly pretty to look at.
Much more convincing for an external observer is an actual implementation of Unity's physics engine. While this works reasonably well with static objects, the colliders of the products and the cart's don't cooperate too well when moving.
To avoid strange behaviours (such as the object clipping through the cart as soon as it moves), an intermediate solution was implemented. The products' physics is activated when the agent drops the item into the cart, but after as short time (more or less 2.0 seconds) the object's rigidbody is set to kinematic so that physics don't apply to it any more. On top of this, the product is parented to the cart itself, preventing it from moving in any way possible.
While this solution worked reasonably well when only one or few agents are present in the scene, it failed catastrophically when the scene was crowded.
At first we attributed this problem to the fact that if an agent is ever so slightly pushed away (by another agent) it would miss the cart entirely.
To avoid this, we set the priority of the agent to 0 (the highest possible priority) while performing the dropping animation, so that every other agent would avoid it while passing nearby. Even after this implementation, the problem still presented itself, so another element must have been causing these annoying malfunctions.
As it turned out, Unity performs an operation of “animation culling” while a character is not on screen. This means that its animation is not played at all, and since when we unparented the product we assumed that it was in the correct position corresponding to the character's hand, the object was dropped from a seemingly random location.
Fortunately to fix this problem Unity allows to turn off the animation culling, so that animations are played even when offscreen. While this solution avoids the problem, we are using more resources to play unneeded animations.

The other solution, the one that was implemented, is to calculate the exact position the hand is in while the drop event occurs, and then teleport the object to that position as soon as it is unparented from the agent and its rigidbody is activated.
So now, if the agent is offscreen, the drop starts from the correct position anyway, and if the character is on screen nothing changes from before since the product is teleported to the position it already is in.
The drawback of this solution is that it is closely tied with the animation clip employed, and the character scale also (even though the latter is an easily fixable problem, the former not so much).
Another alternative solution would be to implement a “pseudo-physics” system where items travel to their assigned place in the cart's basket even if it wouldn't be physically possible for them to reach that position. This would avoid both the ugly “teleportation” effect and the computational resources required by the physics engine, at the cost of some realism in the animation.

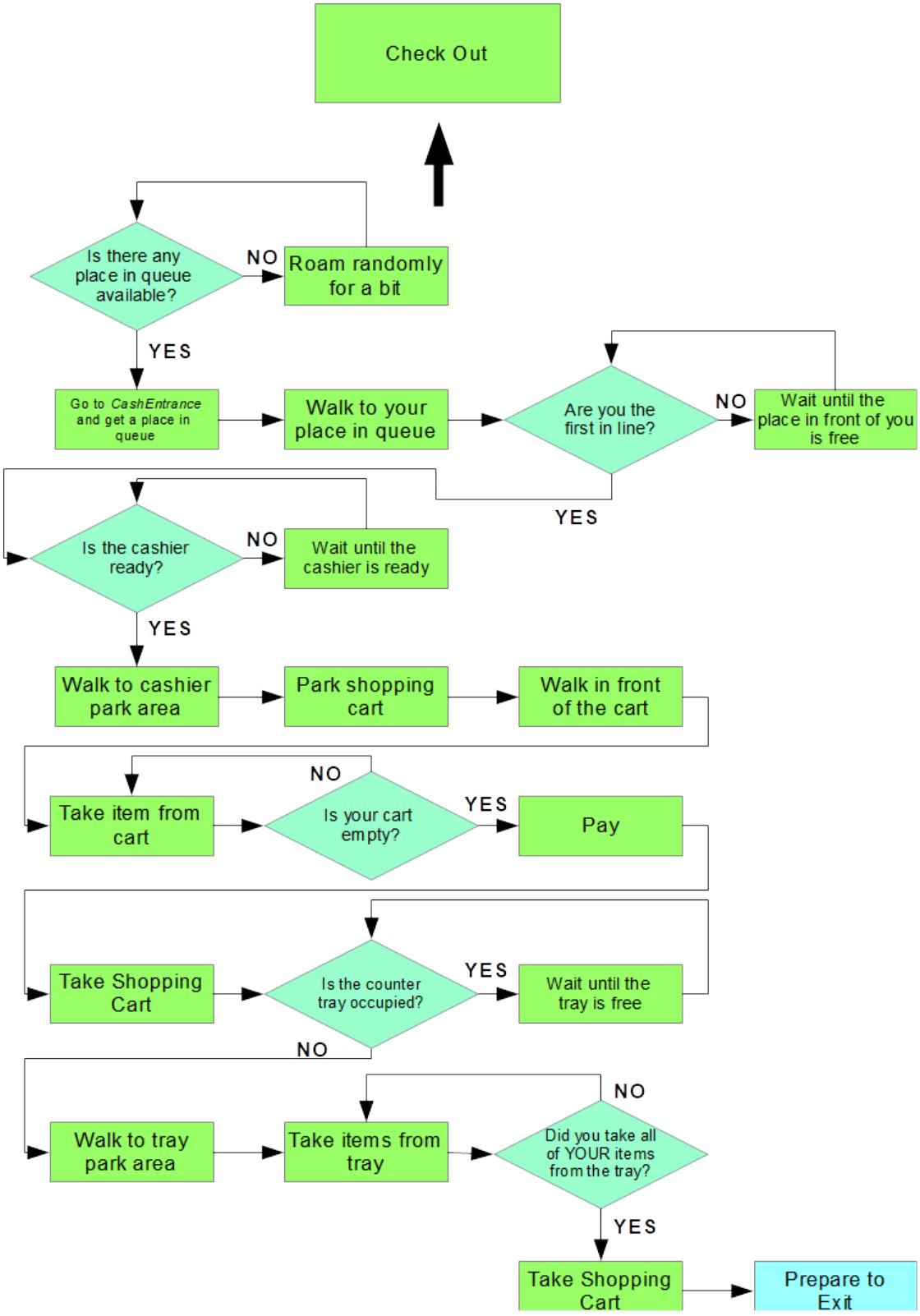
After dropping the product in the cart, the agent consults its buylist to check if the next item on the list is on the same shelf as the previous one, if it is, it goes directly to the *NavTarget* for that item, if not, it goes back to the cart's *PushTarget*, re-parents the cart to itself, switches *NavMeshes* again and then proceeds to the next *CartParkArea*.

Then the process is repeated until the agent has taken every item in its list. After that, it proceeds to the next “super-state”, Check Out.

3.4 Check Out

3.4.1 Logic

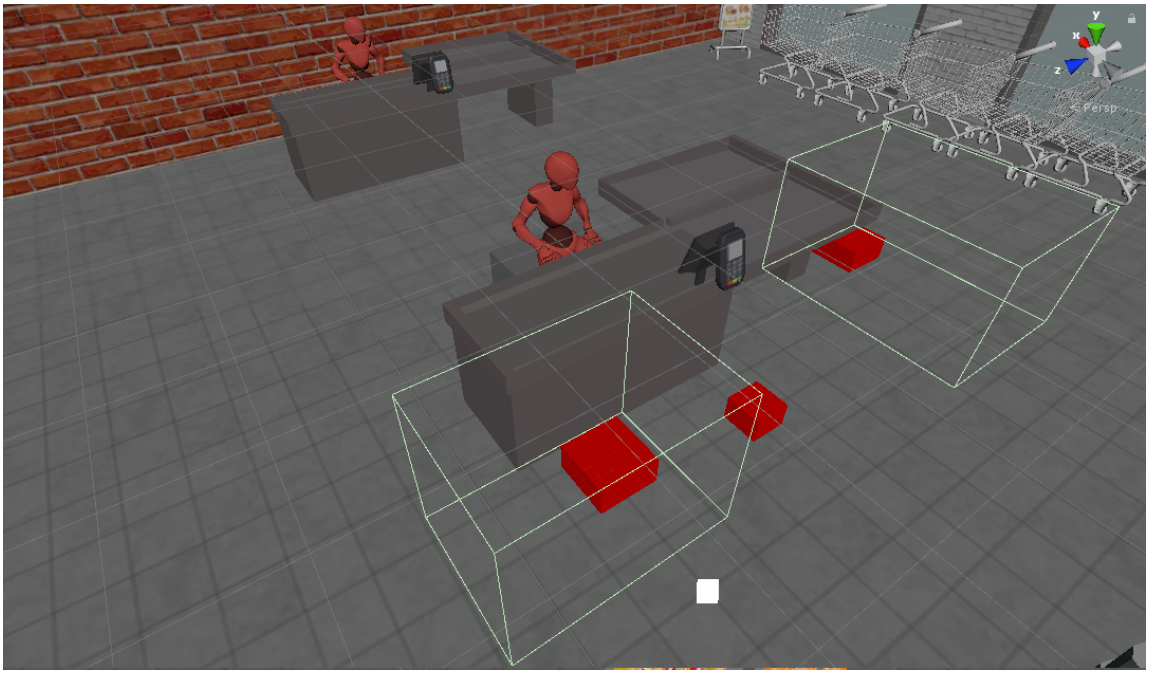
While checking out at a grocery store is considered quite a simple task in real life, it is quite complicated to implement in a program when we think in detail about all the small micro-tasks that it involves.
The agents must first of all chose which counter they want to queue at, actually stand in line until their turn, go to the cashier, unload their cart, pay, re-collect all their items and then finally exit.
While it may not be as impressive for an external viewer when compared to the previous super-state, it is evident from the state machine that follows that this phase is the most complicated from a logical standpoint, as it has many sub-states and more branching paths than the previous phases.



Img 3.16) Intermediate-level state machine of the “Check out” phase

3.4.2 Practical Implementation

Before going into details about how all of this is handled, we must explain how the cash-counter / cashier entities are organized, with the visual aid of the following pictures:



Img 3.17) The cash counter object. The red cubes are the three NavTargets, the highlighted volumes in green are the two CartParkAreas

Img 3.18) The cash counters and the targets (highlighted in green) used for organizing the queue.

Each Cashier counter has three different *NavTargets*, and two CartParkAreas, as illustrated in the picture above.
The first park area, on the left, is used when the agents approach the counter to unload the cart and put items in their cart on the conveyor belt.
The second one behind the counter, near the tray, is used in a following phase, when the agent re-loads the cart after paying.
These *CartParkAreas* work the same as the one associated to the shelves, basically as soon as the agent's collider triggers them, the cart is unparented and they proceed alone to perform the necessary operations. However, as soon as the agent approaches the first *CartParkArea*, only the smaller *NavMesh* is used until the agent is destroyed.
Even though this may lead to some (very minor) clipping, it is necessary to enable the agents to move smoothly in this relatively cramped space. The paths travelled in this phase are very linear and have little room for variance, so that clipping between the cart and other meshes is unlikely.
Also, the sliding door asset as it is in the current simulation is too small for the larger agent+cart *NavMesh*, so that they are actually trapped in the store if they are navigating with it.
The three *NavTargets* are used for navigation, but only the one in the middle, in front of the POS terminal has a trigger collider.

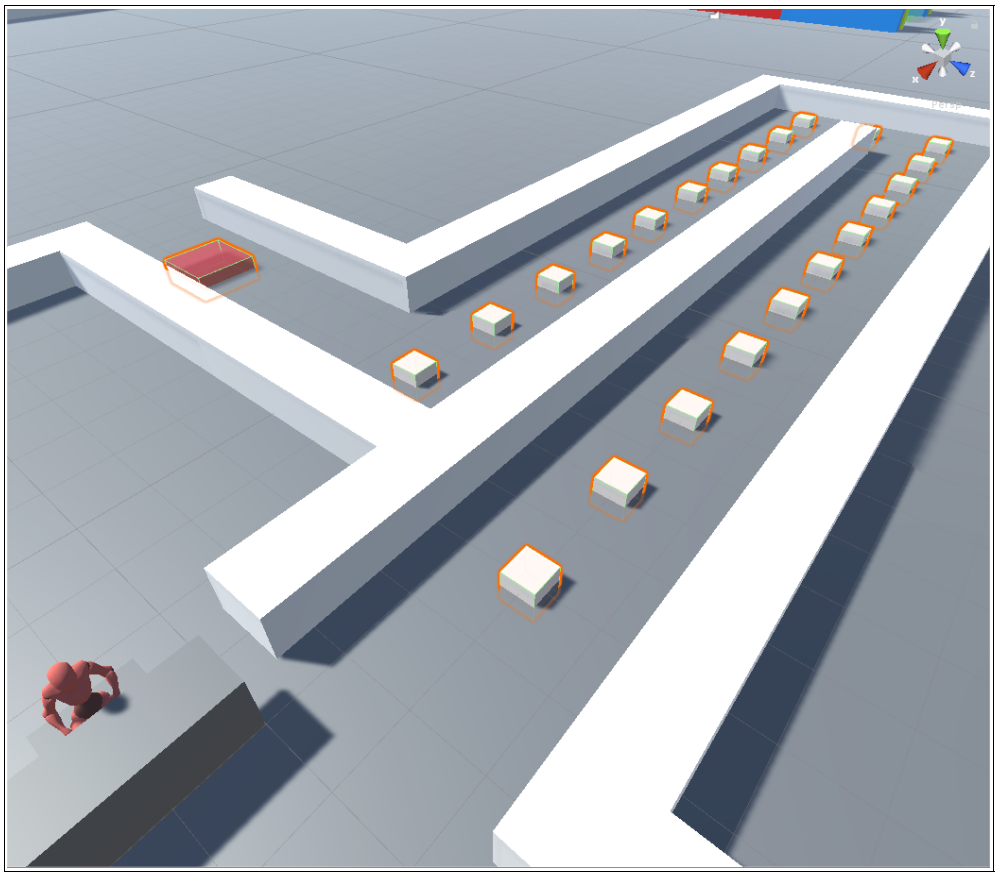
In the second picture we see a series of invisible *NavTargets* (6 for counter plus a larger one at the end, called *CashEntrance*). These are used by the agents as targets when they are queueing at the counter.
The position of these queue targets is automatically set by an algorithm that runs after the scene is loaded. The algorithm sets an fixed number of queue targets starting from its NavTarget associated with the conveyor belt, slightly randomizing their position both on the X and the Z axis (in fact we can observe how the two rows of queue targets, while containing the same number of elements, have a different overall length).
This randomization is applied with the intent to increase the realism. If the agents queued based on targets that are always in the same position, it would seem a very robotic behaviour, unlikely to happen in a real scenario.
Note that the overall number of places available in each queue is fixed. This obviously a simplification that can be improved.
An pathfinding style algorithm could be used to place an arbitrary number of queue targets (neatly placed around obstacles) that fits the number of customers in the store, but a trade off between flexibility and the amount of computational resources used at startup must be considered. This solution however proved to work reasonably well if the environment is not extremely crowded.
Any number of cash counters may be present, the script is capable of handling any amount without modifications.

Now that we have a grasp on how the cashier counter entity is organized, we can go through with the agents' sequence of states.

As soon as they completed their shopping list, the agents check if there are some free places in any of the queues. If there aren't any, the basically roam around randomly for a bit until a queue place is freed.
Practically this is achieved by selecting random *NavTargets*, chosen among any currently present in the store and walking towards them until they receive the information that a place in queue is free.
Every agent as a method that is called by other agents if they are in this “Pre-queue” or “queue” state to inform them that a place in queue is not occupied at the moment.
If there are places free in a queue, then the agent decides in which one of the possible queues he will go towards. They always opt for the queue with less people in it at the moment of decision, concordantly with what a real person would do.
However their place in queue is not assigned immediately, before that they navigate towards the area called *CashEntrance*, placed at one end of the queue line (represented as the larger volume at the end of the line in the picture), and only when they collide with this volume they are assigned a place in the queue.
While this may seem unnecessary and overly complicated, it more closely resembles how a real queue would work. If a place is automatically assigned as soon as an agent completes its shopping list, this could create a situation where it has an assigned place in queue that is in front of someone else that may have finished shopping at a later point in time but is spatially closer to the check out area.
In a grocery store (or any store, bank, post-office, etc.) the queue order is dictated by who *arrives* first, not by who *is finished* shopping first.
When the agents are actually in queue, they navigate towards their corresponding queue target.
If they are the first in line, they periodically check every 3 seconds or so (the timer is slightly randomized, with the intent of reducing their robotic-ness) if the cashier in front of them is in a *free* or *occupied* state.
A cashier is considered *occupied* if an agent is currently unloading their cart on the conveyor belt and has not yet proceeded to pay, at that point they are set as *free*.
If the cashier is free they proceed to the first *CartParkArea* in front of the conveyor belt and send a message to every other client (in the “queue” or “pre-queue” state) that a place has been freed.
As soon as an eligible agent receives this message, they wait a randomized amount of time (about 1-1,5 seconds) and then they advance one place in the queue. The randomization of waiting time, again, is introduced to make the agents more believable.
When in the *CartParkArea*, agents walk in front of their cart and start taking items from their cart and drop them on the conveyor belt.
Picking items up and dropping them is handled exactly in the same way as discussed in the previous chapter, with the aid of IK and physics to make their actions more realistic.
These actions are repeated until the shopping cart is empty. The items are checked from an internal list, the agents do not actually “see” what's in their cart. This is a failsafe measure implemented to avoid the agents being confused if for any reason an

item is lost or removed from their cart (this happened quite a lot during testing, mainly due to the physics engine not working exactly as intended). So if an item that should be in the cart's basket is not physically present (for whatever reason), it is teleported to the agent's hand at this moment from whatever position it may be in.
While this obviously it's not realistic and quite immersion-breaking, it's much better than having the agent stop and the whole system fail altogether as a consequence.
Again, trade-offs between consistence and realism at all costs must be considered.
When items are dropped on the conveyor belt, a script simulates its behaviour and drops them into the tray at the end of the counter.
Here the simulation is completely physics based (even the “sliding” of the product down the slope of the tray) and did not cause issues of any kind.
When this unloading operation is completed, the agent proceeds to the POS system to pay, and a simple “push button” animation is played.
As a self-imposed limitation, it was established that in the grocery store environments of this simulation the only possible payment system is via digital means, both for the autonomous agents and for the end-user. Cash payment can be implemented in a future version.
When the “pay” phase is completed, the agent checks if someone else is in the pickup tray area, collecting his/her paid items. If there is, the agent waits a little bit before checking again. As soon as the tray area is found free, the agent moves, along with the cart, towards the second *CartParkArea* in front of the item tray of the counter. Now it positions itself on the back of the cart and starts picking up items from the tray and drop them one last time in the shopping cart. Again nothing new to say here about how the animations are handled.
One improvement that could also increase the immersion would be to implement shopping bags, but this adds the complexity of making them believable, as shopping bags (plastic or paper ones) are not rigid and the physics to describe their behaviour would not be very simple.
As it was for the previous sub-state, the agents do not “see” what items are in the tray instead they check their original shopping list. This is important because, if they took items based on what they saw in tray, they would take the products of the next client that is currently unloading the cart, practically “stealing” items from them. This of course is not a desirable behaviour (while thieves and shop-lifters exist in real life, implementing such agents is beyond the scope of this project).
When this sub-state is complete, the can proceed to exit the store.

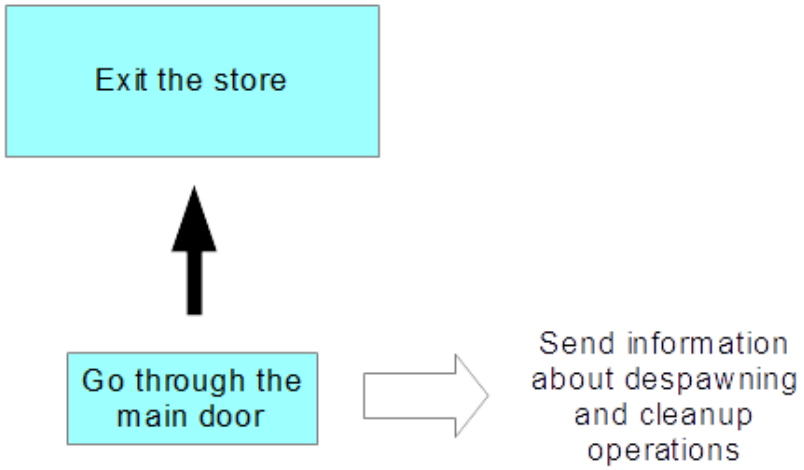
In the prototyping phase of the agent system, a different approach to the check out area was taken, where we assumed a store with a single line (following the example of the *Carrefour* store chain, to give an example) and multiple cashiers. In this instance everything works exactly the same, with the exception that only one queue line is present and the agent that is first in line checks not only one cashier but every one present, and the part of the script that decides in which line to queue in isn't present.
Since this code is readily available, both styles of check out area can be implemented, giving more freedom to the designer of the virtual environment.



Img 3.19 The single-line version of the check-out area as laid out in the prototype of the agent system. The small white cubes are the queue targets, the larger red one is the *CashEntrance* element.

3.5 Exit the store

By far the simplest of the high-level phases, only one sub-state is present here:



Img 3.20) Intermediate-level state machine of the “Exit the store” phase

With their cart now full of paid products, the agents can proceed to exit the store through the main door.
To simplify the procedure, we decided on agents leaving the store with their cart. This avoids the need of more complicated sub states such as putting items in shopping bags or such.
However, if similar features need to be implemented in future versions of the projects, a script for leaving the cart in the deposit area, and making newly-spawned agents check for carts abandoned in this zone is already ready to be used.
Even then, clients leaving the store while still having a shopping cart is not entirely unrealistic, because many actual grocery stores allow this to facilitate transport of the acquired goods to one's car (or any other mean of personal transportation).

Agents outside the store walk for a little while, towards an element called *Despawn Cube*. As soon as they collide with this (invisible) entity, they are destroyed.
While they are being destroyed, the send data to the environment to perform some “cleanup”. This cleanup phase takes care of destroying every other item related to that agent, that is: now useless *NavMeshTargets*, empty placeholders used for IK animations, their shopping cart and every product purchased. Also upon destruction the client counter present in the *Client Spawner* object is decreased.
If the feature is enabled, the Client Spawner also takes care of spawning an new client, to keep the agent count constant.

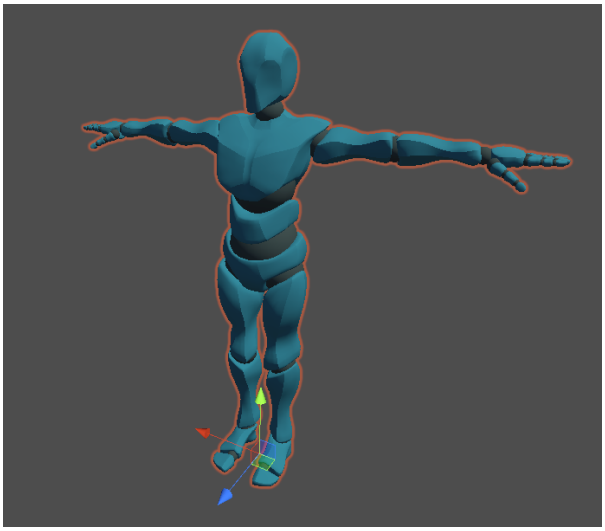
3.6 3D Models, Skeletons, Animations

For the prototyping phase we used the standard “Y-bot” 3D model provided by the site Mixamo.com.
This model is commonly seen in simulation prototypes made with engines such as Unity or Unreal Engine due to its native compatibility with these development environments and also because of its relatively high polygon count (about 55.300) it provides a good benchmark for testing.
The main reason for using this model however is the highly standardized skeleton hierarchy, which facilitates a lot the procedure of *animation retargeting* when the need arises.
Animation retargeting, or *skeleton retargeting*, is the name of the practice that makes it possible to carry over an animation made for a specific skeleton hierarchy to a different one, provided they are reasonably similar.
This process is essential in managing animations, especially but not only while using motion capture practices. Without *skeleton retargeting* it would mean to have the same animation repeated for each different skeleton hierarchy, greatly discouraging changing and or improving character models over the course of development of an application.
Both Unity and Unreal Engine provide a simple way of dealing with skeleton retargeting for human figures. Note that this process is not limited to humanoid figures, and custom transitional hierarchies can be defined for any type of skeleton.
Both engines offer the possibility of defining a “transition skeleton” where bones of a hierarchy must be mapped on a standardized table containing the most important bones and joints of the human body.
When both the skeleton we are retargeting from and the skeleton we are retargeting to are mapped on the transitional hierarchy, the actual retargeting can happen.
Also fundamental in this process is assuring that both skeleton start from the same pose. While the most common poses used are the *T-pose* or the *A-pose* (and it is considered “good practice” in the industry to use one OR the other exclusively), any pose can be used for retargeting, provided the two skeletons have the same one.
Both Unity and UE provide a way to tweak the starting pose of the models that need to undergo a retargeting procedure.
However the transitional skeleton is, by design, very simple and standardized. Because of this, with the added issue that bone proportions and weight-painting are different for each model, *skeleton retargeting* is not to be considered the be-all end-all solution to any animation problem.
Take for example a slim human figure and a huge humanoid monster with gigantic shoulders and arms. While retargeting animations between these figures would be technically possible, the end result wouldn't look very good. Very common is the occurrence where arms and hands clip with the model's torso (to ease this phenomenon, Mixamo.com offers for the vast majority of their animations a customizable “Arm space” parameter).

So, the occurrence of making animations from scratch for each different skeleton hierarchy is not to be ruled out in every situation.
This is especially true for facial animations, since nor Unity nor UE offer a way to map facial bones natively. Also facial bones are much less standardized than bones in the rest of the body, and some models don't use them at all in favor of blend shapes/morph targets.
After setting up the agent system in the actual project, we exchanged the Mixamo "robot" models with realistic looking human figures using retargeting techniques. Since we abided by all the required good practices, switching models is a relatively smooth procedure. Any type and or number of client models can be implemented in future versions of the project, provided they have an appropriate skeleton structure. These new models, while being much more realistic, have a drastically lower polygon count. Until now, three models were adopted:

-Joe, a middle-aged man with moustache: 8257 polygons
-Vito, a college-aged caucasian male: 7795 polygons.
-Calogero, an old man: 6516 polygons.

Their average polygon count is about 7.500 polygons.
Note that besides the number of polygons, well realized textures, normal maps and specular maps contribute equally if not more to the realism of a 3D model.




Img 3.21 & 3.22) The Mixamo model used in the prototype and the old man model (our personal favourite) side-by-side in T-pose.

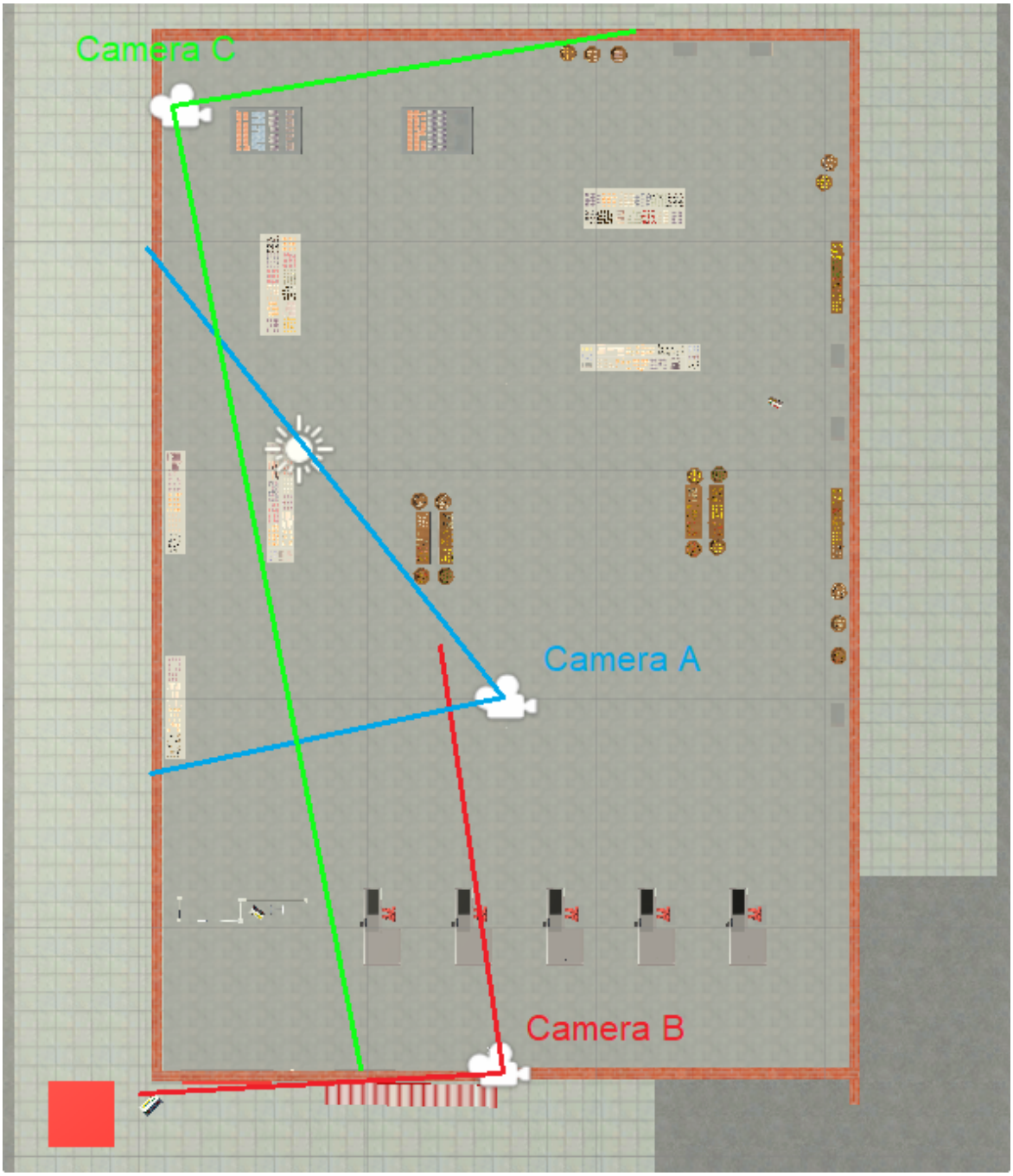
4. EXPERIMENTAL TESTS

4.1 Objective

To test the performance of the agent system, several performance tests have been performed.
The objective of the tests was to asses the efficiency of the agent system and how it performs from a resource usage point of view. The metric used for the test are the Frames Per Second average values. To better evaluate how the system performs, four different cameras have been setup each framing a strategically chosen area of the store.
The performance test involved spawning agents in batches of 5 and periodically checking the frame rate for each of the four cameras. Since the frame rate was quite variable (even without agents), their average value was recorded, after a period of 20 to 30 seconds as soon as the FPS were stable (short spikes in memory usage happened when agents were spawned).
A cap was established at 60 agents, because at that point the store was incredibly crowded and busy, at the limit or even beyond what a real store of that size can realistically accommodate (keep in mind that each cashier can have a maximum of 6 agents queued, so the total is 30).

4.2 Test Setup

The environment for the test is a medium-small grocery store template presenting all the required characteristics to be meaningful for evaluating the performance from a quantitative and qualitative standpoint.
This is the floor plan of the testing environment:



Img 4.1) The virtual environment used for testing. An indication about the position and field of view of three cameras is also present. The entrance is located in the bottom left.

All the possible elements are present: every type of shelf, an entrance sliding door and gate, five different cash counters plus several obstacles of different nature.
Four different cameras have been set up. Three, called **A**, **B** and **C**, are located inside the store as CCTV cameras would. The cameras were placed in strategic places, to frame different portions of the store with different contents and amount of agent traffic.

-**Camera A:** the camera with the most restricted point of view, it focuses on three particular shelves and an important transit zone for agents going back and forth from the entrance to the back sections. The traffic of agents here is highly variable. The number of rendered polygons from this point of view (without any agents) is about 580 thousand.

-**Camera B:** this camera focuses on the entrance and the two nearest cash counters. This is a mid to high traffic zone for the agents, as they all pass through here for entering and exiting. The polygon count is around 1 million.

-**Camera C:** this camera, placed in the back of the store, frames almost everything except the entrance. Through its field of view almost all the agents present in the store are visible at any time. The rendered polygons are around 1.9 million.

A fourth camera is placed outside the store, pointing at the sky and used as a baseline for control. The polygon count, while technically not being zero, is negligible.

As we previously stated, the average polygon count for our agent models is about 7.500 per agent, plus about 4.000 for the cart.
So if 10 agents are rendered, about 110.500 polygon must be added to the viewport.
The specifications of the machine used for testing are the following:

Processor:	Intel Core i7-6700 @3400GHz
RAM:	16GB
System:	Windows 10 64bit
GPU:	Nvidia GeForce GTX 970
VRAM:	4043 MB
Resolution*:	1280x720

**(The resolution indicated here is relative to the window running the simulation, not the resolution of the screen.)*

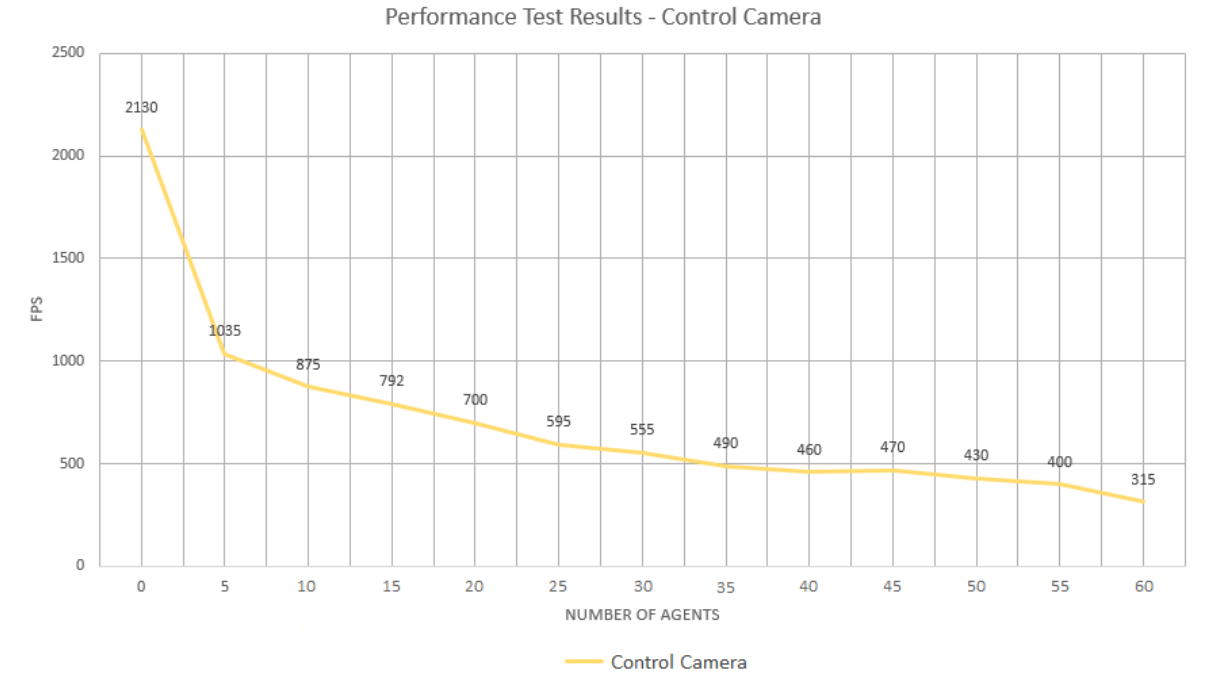
Also a series of debug-only functions are available in this testing phase, such as:

-Modify the *Client Spawner* parameters, specifically the maximum number of clients, a toggleable auto-spawn function, the gap in seconds between any

spawn, and a function that re-spawns clients as soon as one is destroyed, to keep the count constant during testing.
-A command to toggle cameras.
-An FPS viewer, both instantaneous and averaged (we employed the Unity plugin *Graphy*).

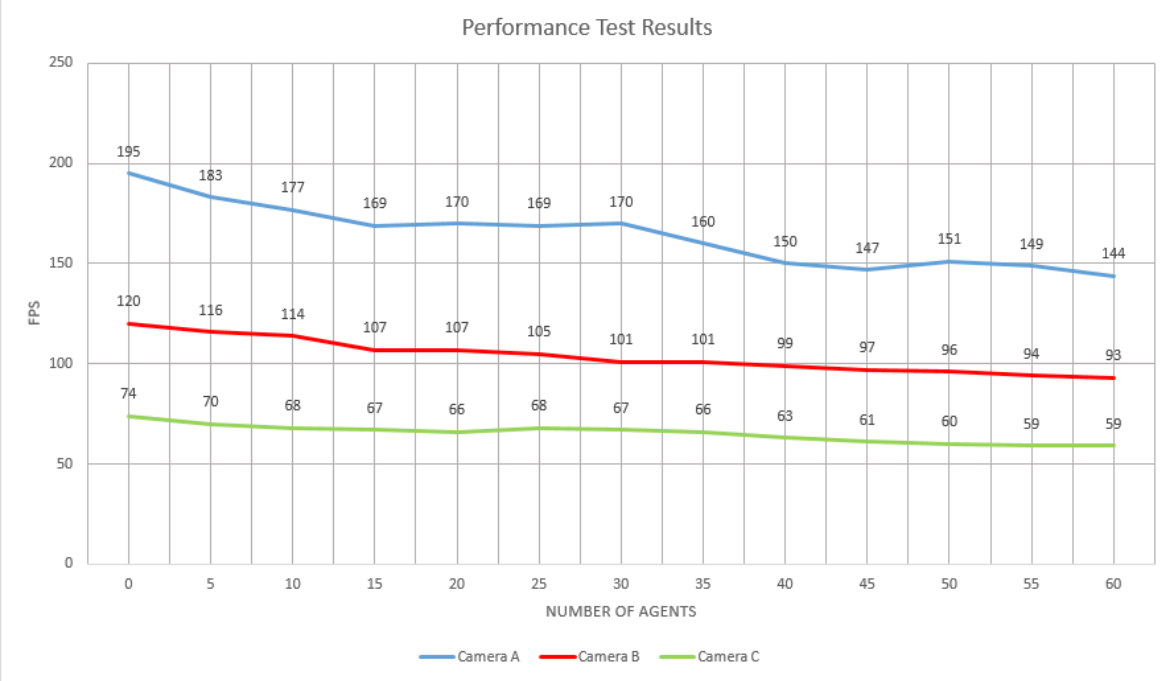
4.3 Results and Discussion

This is the graph with the results recorded from the control camera, placed outside the store:



Img 4.2) The results recorded from the control camera.

Of all the recorded values, the ones relative to this camera were the most variable of all, with frame rates even varying by the hundreds at lower agent counts.
The most notable result is the huge drop in frames from 0 to 5 agents, and after that an almost linear decrease until about 35, where the values are very stable until surpassing 55 (the increment in frames from 40 to 45 agents can be ignored because, as already stated, there was extreme variance in values). The last drop can be attributed at the fact that the environment was extremely crowded at that point, so that the dynamic paths for the agents were almost constantly re-calculated.
This results are very significant in that they represent only the computational resources employed by the scripts and not by the rendering.
The results relative to the other three cameras are presented in the following graph:



Img 4.3) The results recorded from the three main cameras.

The respective camera performance is ordered, as expected, by the number of polygons framed at 0 agents.
For all three of them we see an initial drop that stabilizes at around 15 clients until about 30-35 where we see another decrement in frames per second.
Of the three, the most unstable was Camera A, due to the fact that the agents framed at any moment is highly variable, as this is an area of transit.
However we consider these promising results, as the curves are increasingly stable as the initial polygon count increases.
And most importantly, we go under the critic threshold of 60 FPS only when there are 55 to 60 agents (and only by 1 frame), in the most comprehensive point of view where almost all the store and agents are rendered.
Note that besides the base line polygon count, at 60 agents we must add another 690.000 polygons to the rendering pipeline.
This means that the vast majority of the computational resources are not invested in managing the agents, which is a good result in our opinion, even though further optimizations surely are possible and can lead without doubt to even better results.

5. CONCLUSIONS

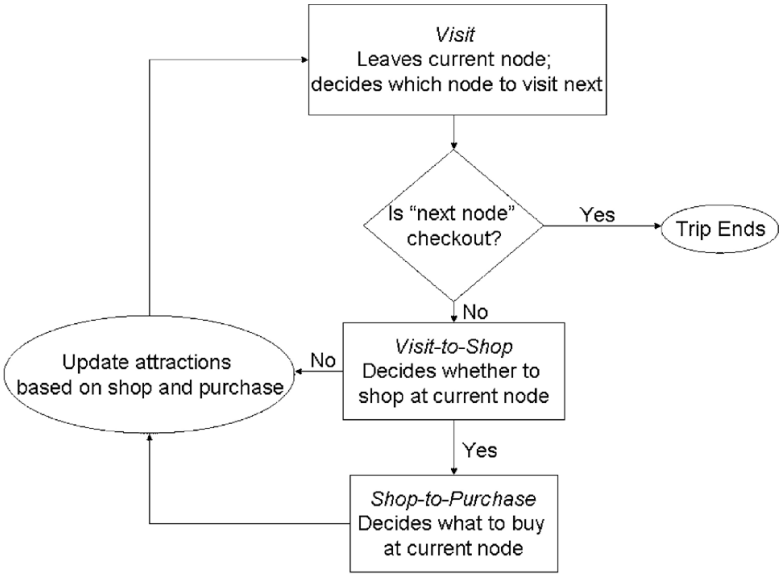
The agent-based model developed in this project had the objective of being believable from the point of view of the user as a customer of a grocery store.
In our opinion the achieved results represent a good groundwork to improve upon.
Especially due to the fact that maximum modularity in both the environmental and the behavioural aspect were of the utmost importance in our project, new or improved features can easily be implemented, or even the entire agent system can be shifted in scope.
This paper by S.K. Hui et al. [23] verifies empirically three different behavioural hypotheses about consumer's in-store behaviour that seem like perfect candidates to be implemented on top of our system.
These three theories are:

-Shopping and Buying patterns based on *Perceived Time Pressure* (concept based on the theory of Monroe [24] and Thaler [25]).

-Shopping and Buying based in *Licensing* (concept excerpted from Khan and Dhar [26] which divides products into *virtuous* and *vicious* categories).

-Shopping and Buying based on *Social Influence of Other Shoppers* (topic studied by numerous experts, mostly in the '80s and '90s, such as Latane [27], Argo et al. [28], Harrel [29] and Becker [30]).

They even model the customers' behaviour with a pseudo-state machine that would be very easy to adapt to our already-existing agent model.



Img 5.1) The three-state customer model by S.K. Hui et al. [23]

The three main states identified in the study are “visit”, “shop”, and “buy”, states that can be naturally observed in our agents.

The main difference between the model presented in this study and our model is that we focus much more on a “planning ahead” (which is, by their own admission, only marginally taken into account) aspect of shopping instead of the “impulsive” or “situational” shopping, that is admittedly absent in our agents.
However we still want to specify that the focus of our work was to create agents that are believable from a customer's point of view, not to develop an agent-based model that exactly represents the psychology and behaviour patterns of grocery store customers on a large scale that may be useful for sociology or marketing researchers, even though the basis for building such kind of simulation certainly is present.
For example, for the average customer would be very difficult to understand if another client of the store is buying something impulsively or if he/she planned ahead to buy that specific product. Such complex behaviour models are largely beyond the scope of this project, but it is very important to point out their existence nevertheless.

Another area of improvement is the interaction between the agents and the user. As of now, their only interactive feature is the agents avoiding the user (which was achieved by setting the avatar of the user as a *NavMeshObstacle*) while navigating.
One development tool that was considered during development is the *Virtual Agent Interaction Framework* (known as VAIF), that is a freely downloadable Unity package, available on Github (https://github.com/isguser/VAIF).
This frameworks allows developers of easily implementing conversation and interaction systems, providing even tools such as lip-syncing to audio files, as explained in the article by I. Gris et al. [31]
However, despite the fact that the attached documentation is quite extensive, during our testing we encountered some problems with this framework, which is still in development and is lacking some important features as of now. We also encountered some difficulties in compatibility with the most recent Unity versions, so its implementation was abandoned early.
The interaction system is entirely based on audio files, which requires actually recording audio samples and it lacks native support with vocal synthesizers, greatly reducing its usability.
Also, in contrast to what is marketed as, the conversation trees only allow simple responses from the user, basically in the form of “Yes”, “No”, “I don't know” and “I don't understand”, which in our opinion is a little too restrictive to be considered a realistic conversation system.
All these issues suggest that maybe an *ad hoc* interaction system built from the ground up and tailored to our needs is a better option, since adding “interruption” states to the cycle of our agents is entirely doable due to our previously stated focus on modularity.

Bibliography

[1] Parunak, H. Van Dyke, Robert Savit, and Rick L. Riolo. "Agent-based modeling vs. equation-based modeling: A case study and users' guide."*International Workshop on Multi-Agent Systems and Agent- Based Simulation*. Springer, Berlin, Heidelberg, 1998.

[2] K. Singh "Introduction to Agent-Based Modeling" @ https://dimensionless.in/introduction- to-agent-based-modelling/, 2019

[3] Bonabeau, Eric. "Agent-based modeling: Methods and techniques for simulating human systems." *Proceedings of the national academy of sciences* 99.suppl 3 (2002): 7280-7287.

[4] Russell, S. J. "Norvig (2003)." *Artificial intelligence: a modern approach* (2003): 25-26.

[5] Harel, David. "Statecharts: A visual formalism for complex systems."*Science of computer programming* 8.3 (1987): 231-274.

[6] Karpathy, Andrej. "The unreasonable effectiveness of recurrent neural networks."*Andrej Karpathy's blog* 21 (2015).

[7] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." *Neural computation* 9.8 (1997): 1735-1780.

[8] Xu, Kelvin, et al. "Show, attend and tell: Neural image caption generation with visual attention."*International conference on machine learning*. 2015.

[9] Juliani, Arthur, et al. "Unity: A general platform for intelligent agents."*arXiv preprint arXiv:1809.02627* (2018).

[10] Luis Bermudez "Overview of Jacobian IK" @ https://medium.com/unity3danimation/overview-of-jacobian-ik-a33939639ab2, 2017

[11] Luenberger, David G., and Yinyu Ye. *Linear and nonlinear programming*. Vol. 2. Reading, MA: Addison-wesley, 1984.

[12] Aristidou, Andreas, and Joan Lasenby. "FABRIK: A fast, iterative solver for the Inverse Kinematics problem."*Graphical Models*73.5 (2011): 243-260.

[13] Willaert, Willem IM, et al. "Recent advancements in medical simulation:

patient-specific virtual reality simulation." *World journal of surgery* 36.7 (2012): 1703-1712.

[14] Hoffman, Hunter G. "Virtual-reality therapy."*SCIENTIFIC AMERICAN-AMERICAN EDITION*- 291 (2004): 58-65.

[15] Powers, Mark B., and Paul MG Emmelkamp. "Virtual reality exposure therapy for anxiety disorders: A meta-analysis."*Journal of anxiety disorders* 22.3 (2008): 561-569.

[16] Wiederhold, Brenda K., and Mark D. Wiederhold. *Virtual reality therapy for anxiety disorders: Advances in evaluation and treatment*. American Psychological Association, 2005.

[17] Garcia-Palacios, Azucena, et al. "Virtual reality in the treatment of spider phobia: a controlled study."*Behaviour research and therapy* 40.9 (2002): 983-993.

[18] Carlin, Albert S., Hunter G. Hoffman, and Suzanne Weghorst. "Virtual reality and tactile augmentation in the treatment of spider phobia: a case report."*Behaviour research and therapy* 35.2 (1997): 153-158.

[19] Rothbaum, Barbara O., et al. "Virtual reality exposure therapy for Vietnam veterans with posttraumatic stress disorder."*The Journal of clinical psychiatry* (2001).

[20] Vanhoof, Tom Brijs Gilbert Swinnen Koen, and Geert Wets. "Using Shopping Baskets to Cluster Supermarket Shoppers."

[21] Levasseur, Martine, and Eliseo Veron. "Ethnographie d'une exposition." (1983): 29-32.

[22] Chittaro, Luca, and Lucio Ieronutti. "A visual tool for tracing users' behavior in Virtual Environments."*Proceedings of the working conference on Advanced visual interfaces*. ACM, 2004.

[23] Hui, Sam K., Eric T. Bradlow, and Peter S. Fader. "Testing behavioral hypotheses using an integrated model of grocery store shopping path and purchase behavior."*Journal of consumer research* 36.3 (2009): 478-493.

[24] Suri, Rajneesh, and Kent B. Monroe. "The effects of time constraints on consumers' judgments of prices and products."*Journal of consumer research* 30.1 (2003): 92-104.

[25] Thaler, Richard H. "Mental accounting matters."*Journal of Behavioral*

*decision making*12.3 (1999): 183-206.

[26] Khan, Uzma, and Ravi Dhar. "Licensing effect in consumer choice." *Journal of marketing research* 43.2 (2006): 259-266.

[27] Latané, Bibb. "The psychology of social impact."*American psychologist* 36.4 (1981): 343.

[28] Argo, Jennifer J., Darren W. Dahl, and Rajesh V. Manchanda. "The influence of a mere social presence in a retail context."*Journal of consumer research*32.2 (2005): 207-212.

[29] Harrell, Gilbert D., Michael D. Hutt, and James C. Anderson. "Path analysis of buyer behavior under conditions of crowding." *Journal of Marketing Research* 17.1 (1980): 45-51.

[30] Becker, Gary S. "A note on restaurant pricing and other examples of social influences on price."*Journal of political economy* 99.5 (1991): 1109-1116.

[31] Gris, Ivan, and David Novick. "Virtual agent interaction framework (VAIF): A tool for rapid development of social agents."*Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

Picture Index