UNIVERSITY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc «Advanced Informatics and Computing Systems - Software Development and Artificial Intelligence»

ΠΜΣ «Προηγμένα Συστήματα Πληροφορικής - Ανάπτυξη Λογισμικού και Τεχνητής Νοημοσύνης»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	**Optical simulation of procedure execution in an Aluminum Factory on an intelligent Virtual Environment.** Οπτική προσομοίωση εκτέλεσης διαδικασιών σε εργοστάσιο αλουμινίου σε ένα Ευφυές Εικονικό Περιβάλλον (IVA).
Student's name-surname: Ονοματεπώνυμο φοιτητή:	**Triantafyllos Galanis** Τριαντάφυλλος Γαλάνης
Father's name: Πατρώνυμο:	**Evangelos** Ευάγγελος
Student's ID No: Αριθμός Μητρώου:	ΜΠΣΠ/20007
Supervisor: Επιβλέπων:	**Themistoklis Panagiotopoulos, Professor** Θεμιστοκλής Παναγιωτόπουλος, Καθηγητής

October 2023/ Οκτώβριος 2023

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

Themistoklis Panagiotopoulos **Professor** Θεμιστοκλής Παναγιωτόπουλος Καθηγητής	**Christos Douligeris** **Professor** Χρήστος Δουληγέρης Καθηγητής	**Aggelos Pikrakis** **Assistant Professor** Άγγελος Πικράκης Επίκουρος Καθηγητής

Table of Contents /Περιεχόμενα

List of figures

1.Acknowledgments/ Ευχαριστίες

Θα ήθελα να πω ένα μεγάλο ευχαριστώ στους γονείς μου για όλη την στήριξη που μου παρέχουν στις προσπάθειες μου όλα αυτά τα χρόνια ζωής μου καθώς και στους φίλους που με βοήθησαν σε όλη την διάρκεια των σπουδών μου. Επίσης, θα ήθελα να ευχαριστήσω όλους τους καθηγητές του μεταπτυχιακού προγράμματος του Πανεπιστημίου Πειραιά "Προηγμένα Συστήματα Πληροφορικής - Ανάπτυξη Λογισμικού και Τεχνητής Νοημοσύνης" για το όμορφο ταξίδι και τις γνώσεις που μου χάρισαν και ιδιαίτερα τον Δρ. Θεμιστοκλή Παναγιωτόπουλο για την υποστήριξη και την καθοδήγηση που μου παρείχε κατά τη διάρκεια των σπουδών μου και την εκπόνηση της πτυχιακής μου.

2. Abstract:

This thesis presents the development of Optical simulation of procedure execution in an Aluminium Factory on an Intelligent Virtual Environment. It investigates the utilization of Goal-Oriented Action Planning (GOAP) within the Unity game development engine for creating intelligent agent behaviours. Furthermore, it explores the integration of LiDAR scanning technology along with a 3D computer graphics software tool, Blender to enhance the development of agents and environmental elements. The factory setting presents unique challenges that can be addressed by employing GOAP, allowing virtual characters to exhibit realistic decision-making and efficient action execution. LiDAR scanning provides a means to capture real-world objects and environments, which can be refined using Blender to create textured 3D models. By combining these technologies, this thesis aims to offer valuable insights into the implementation of GOAP, as well as the integration of LiDAR scanning and Blender, to achieve a highly immersive factory environment in the context of game development. Finally, we discuss about the behaviour of the agents take when they also follow the theories of Maslow about the hierarchy of need's.

2. Περίληψη

Η πτυχιακή εργασία εξετάζει τη χρήση του Σχεδιασμού Δράσης με Επίκεντρο τους Στόχους (Goal-Oriented Action Planning - GOAP) μέσα στο περιβάλλον ανάπτυξης παιχνιδιών Unity για τη δημιουργία έξυπνων συμπεριφορών πρακτόρων σε ένα σενάριο που αφορά την λειτουργία ενός εργοστασίου. Επιπλέον, γίνεται χρήση της τεχνολογίας σάρωσης LiDAR μαζί με ένα ανοιχτού κώδικα 3D computer graphics λογισμικό όπως είναι το Blender για τη δημιουργία πρακτόρων και στοιχείων του περιβάλλοντος. Στο περιβάλλον ενός εργοστασίου μπορούμε να εφαρμόσουμε σε αρκετά σημεία ώστε να αντιμετωπίσουμε προκλήσεις τον Σχεδιασμό Δράσης με Επίκεντρο τους Στόχους (GOAP), επιτρέποντας στους εικονικούς χαρακτήρες να εκδηλώνουν ρεαλιστική λήψη αποφάσεων και αποτελεσματική εκτέλεση δράσεων. Η τεχνολογία σάρωσης LiDAR παρέχει τη δυνατότητα απαθανάτισης αντικειμένων και περιβάλλοντος από τον πραγματικό κόσμο, και τη δημιουργία 3D μοντέλων. Η πτυχιακή αυτή αποσκοπεί στην παρουσίαση και ανάλυση αυτών των τεχνολογιών, την ανάλυση των συμπερασμάτων από την εφαρμογή του GOAP και τις ανάγκες που οι πράκτορες προσπαθούν να καλύψουν, ενώ γίνεται προσπάθεια συσχέτισης αυτής της κάλυψης των αναγκών των πρακτόρων και της ιεραρχίας που έχουν αυτές με την πυραμίδα αναγκών του Maslow.

3. Introduction

In the realm of game development, the quest for creating intelligent and immersive virtual characters (Intelligent agents) has led to remarkable advancements in Artificial Intelligence (AI) systems. These systems, responsible for controlling characters within game worlds, play a pivotal role in shaping captivating gaming experiences. Intelligent agents are software systems that can perceive their environment, make decisions, and take actions to achieve goals. They mimic human decision-making processes. Artificial Intelligence (AI) is a broader field that includes intelligent agents and other techniques to create machines that perform tasks requiring human intelligence. AI aims to develop algorithms and systems for reasoning, learning, understanding language, perceiving the environment, and solving complex problems. Intelligent agents and AI are used in various applications, from autonomous vehicles to smart assistants and recommendation systems. AI improves efficiency, automates tasks, and augments human capabilities across industries like healthcare, finance, education, gaming, and manufacturing.

As game complexity continues to increase, developers seek AI frameworks capable of managing intricate decision-making processes and dynamic behaviours. This thesis delves into one such framework, Goal-Oriented Action Planning (GOAP), and explores its application in Unity, a popular game development engine. Moreover, it investigates how LiDAR scanning technology and Blender program are utilized to generate agents and environmental elements, enhancing the overall realism and fidelity of the game world. Specifically, the scenario being considered is a factory setting, where GOAP is applied to simulate intelligent behaviours within this industrial environment.

3.1 Background:

At the first steps of game development, rule-based systems were commonly used to control character behaviours. These systems were using some predefined rules that determined how characters would react in specific conditions or events. It was a simple approach that had no ability to handle complex scenarios and adapt to changing game states.

Finite state machines

Finite state machines (FSMs) emerged as a more flexible alternative to rule-based systems and represented character's behaviour as a finite set of states that changing depending on predefined conditions. This allowed for more dynamic and responsive character behaviours but in high game complexity it was very challenging to manage and adjust the states and the conditions and often required manual adjustments, even for small changes.

Script-driven behaviours

Script-driven behaviours were another popular in which developers wrote custom scripts or behaviours to control character actions and reactions but very often a large amount of manual coding needed and there was small ability to adapt dynamically to different game situations.

All these limitations were the reason for developing more advanced AI architectures.

Behaviour Trees

Behaviour Trees (BTs) emerged as a hierarchical approach that had good control and flexibility over character behaviours. BTs have a tree-like structure, where each node has a specific

action or decision-making logic. This structure helped developers to define complex behaviour sequences and prioritize actions in a better way. BTs became widely adopted due to their scalability, reusability, and ease of customization.

Goal-Oriented Action Planning

Another important advancement in AI architectures for game development was the introduction of Goal-Oriented Action Planning (GOAP). GOAP focused in goal-driven decision-making rather than predefined behaviours and enabled characters to plan and execute actions on their own, based on the goals they have. All character actions have some preconditions, after effects and costs and GOAP gives them the ability to select the actions with the most "efficient" way to accomplished their goals, with a more dynamic and adaptive behaviour.

3.2 Research Objectives:

This thesis aims to explore the application of GOAP in Unity, focusing on its ability to create dynamic and intelligent character behaviours within a factory setting. Additionally, it investigates how LiDAR scanning, with a 3D computer graphics software tool like Blender can be employed to generate lifelike agents and environmental elements, creating a realistic factory environment. Through in-depth analysis and practical implementation examples, this study intends to provide future students and anyone concerns with valuable insights into the implementation of G.O.A.P, as well as the integration of LiDAR scanning and Blender for enhanced realism in a factory scenario.

3.3 History of the most important AI techniques:

There are many AI techniques that have been used through the years in video games development to create dynamic and intelligent behaviours for non-player characters (NPCs). Some of these techniques are:

3.3.1 Finite State Machines (FSMs):

Finite State Machines (FSMs) have been one of the earliest and simplest AI techniques used in video games. FSMs model NPCs as finite states, and transitions between these states are triggered by events or conditions. While FSMs are straightforward to implement, they lack flexibility in handling complex behaviours.

3.3.2 Behaviour Trees (BTs):

This technique introduced in the mid-2000s as a more structured approach to AI in games and use hierarchical tree structures to represent decision-making processes. Each node in the tree defines an action or condition, making it easier to create complex behaviours by combining simpler nodes.

3.3.3 Utility-Based AI:

Utility-Based AI, or simply Utility AI, considers the utility or desirability of different actions and selects the one with the highest perceived utility. This approach. allows for more sophisticated decision-making, as it considers the trade-offs between different actions based on their expected outcomes. Utility AI gained popularity in the mid-2000s for creating NPCs with adaptive and context-aware behaviours.

3.3.4 Hierarchical Task Networks (HTNs):

Hierarchical Task Networks (HTNs) are used to model AI decision-making as a hierarchy of tasks and subtasks. HTNs provide a more modular and flexible approach to representing complex behaviours, as the hierarchy allows for reusable and interchangeable components. This technique became popular in the late 2000s for its ability to handle diverse NPC behaviours efficiently.

3.3.5 Reinforcement Learning (RL):

Reinforcement Learning is a machine learning technique where agents learn to make decisions by interacting with their environment and receiving feedback in the form of rewards or penalties. RL has been applied to video game AI to create adaptive and learning NPCs, capable of improving their behaviour through trial and error. RL-based AI has gained significant attention in recent years, especially in developing autonomous characters and NPCs.

3.3.6 Machine Learning and Neural Networks:

Machine learning techniques, including neural networks, have made significant steps in AI gaming. Neural networks are used for tasks like character animation, speech recognition, and image processing. Deep Learning approaches, such as Deep Q-Networks (DQNs) and Generative Adversarial Networks (GANs), have also been employed to create more realistic and intelligent NPCs.

AI techniques in video game development are continuously evolve and adapt to the needs for more realistic gaming, from early FSMs to advanced machine learning approaches, each technique has contributed to the evolution of NPC behaviours, making virtual characters more realistic, adaptive, and engaging. GOAP has played a significant role in creating goal-oriented behaviours, a diverse set of AI methods has been employed to cater to the ever-increasing complexity and player expectations in modern video games.

3.3.7 Goal-Oriented Action Planning (GOAP)

Goal-Oriented Action Planning (GOAP) has a big history in the field of Artificial Intelligence in video game development. The roots of GOAP can be traced back to classical planning algorithms, which have been studied since the 1960s. These algorithms involve finding sequences of actions to achieve goals in a defined state space. In the late 1990s and early 2000s, video game developers recognized the limitations of traditional AI methods, such as Finite State Machines (FSMs) and scripted behaviours, in creating dynamic and adaptive NPC behaviours for complex game environments.

The breakthrough for GOAP came with the development of the video game "F.E.A.R." in 2005, where Jeff Orkin, the lead AI programmer, implemented a form of GOAP for the game's AI enemies. The use of GOAP in "F.E.A.R." showcased the potential of goal-driven planning in creating lifelike and challenging behaviours for NPCs. Following this success, GOAP gained popularity in the game development community and found applications in various game genres. Over the years, researchers and developers have contributed to the refinement and optimization of GOAP, proposing variations to address specific challenges and integrate it with other AI techniques. The integration of GOAP into major game engines has made it more accessible to developers, leading to its widespread adoption. Today, GOAP continues to be an

active area of research, with ongoing efforts to improve its scalability, performance, and adaptability in creating dynamic and immersive NPC behaviours. The history of GOAP showcases its evolution from classical planning algorithms to becoming a prominent AI technique in video game development, contributing to more engaging and realistic gaming experiences. As game worlds continue to grow in complexity and player expectations increase, GOAP remains a valuable tool in shaping the future of AI-driven NPC behaviours and interactive virtual environments.

By employing the concept of goals and actions, GOAP enables virtual characters to autonomously plan and execute actions to achieve desired objectives. Utilizing action preconditions, effects, and cost considerations, GOAP empowers characters to make intelligent decisions in response to complex game scenarios.

3.4 Analysis of Goal-Oriented Action Planning (GOAP):

3.4.1 Goal-Oriented Approach:

1. GOAP focuses on achieving specific goals rather than prescribing rigid behaviours or actions for agents.

- Agents define their goals and the conditions that must be satisfied to consider a goal achieved.
- The agent's decision-making is driven by the prioritization and evaluation of goals, allowing for dynamic and adaptive behaviour.

2. Action Planning:

- GOAP employs a planning process to determine the sequence of actions that an agent should execute to achieve its goals.
- Actions are defined as atomic units of behaviour that can be executed by the agent.
- The planning algorithm generates a plan by considering the current state, the desired goals, and the available actions that can be performed.

3. State Representation:

- GOAP utilizes a state representation to capture the current state of the environment and the agent.
- The state consists of a set of variables and their corresponding values that describe the attributes and conditions of the agent and the environment.
- The agent reasons about the state to decide which actions are applicable and which goals are achievable.

4. Action Preconditions and Effects:

- Actions in GOAP have preconditions and effects associated with them.
- Preconditions represent the conditions that must be met for an action to be executed.
- Effects describe the changes in the state that occur after an action is performed successfully.

5. Decision-Making and Plan Execution:

- GOAP provides a decision-making process for agents to select the most appropriate actions to execute.
- The agent evaluates the desirability of each action based on factors such as the goal priority, action costs, and potential state changes.
- Once a plan is generated, the agent executes the actions sequentially, updating the state as it progresses.

6. Flexibility and Adaptability:

- GOAP offers flexibility in designing agent behaviours as it allows for easy modification and addition of new goals, actions, and conditions.
- Agents can dynamically switch goals, adapt plans, or reconsider actions based on changes in the environment or new priorities.

3.4.2 Limitations:

- GOAP may face challenges in handling complex and dynamic environments with a large number of goals and actions.
- The planning process in GOAP can be computationally expensive and may require optimization techniques for efficiency.

GOAP provides a robust and efficient framework for creating intelligent agent behaviours that can adapt to changing circumstances and pursue specific goals. Its goal-oriented approach and action planning capabilities make it a popular choice in game development, robotics, and other domains requiring sophisticated agent decision-making and behaviour.

4. Maslow's Hierarchy of Needs

Maslow's Hierarchy of Needs is a psychological theory proposed by Abraham Maslow in 1943. It describes the hierarchical structure of human needs, representing the progression of motivation and fulfilment from basic physiological needs to higher-level psychological needs. The pyramid-shaped hierarchy consists of five levels or categories of needs, with each level building upon the previous one. Here's an overview of each level:

Figure 1: Maslow's Hierarchy of Needs

Physiological Needs:

Physiological needs are the most fundamental and essential requirements for human survival. Some of these needs are food, water, shelter, sleep, and basic bodily functions and must be met first, as they are necessary for sustaining life. When they are not fulfilled, they become the primary focus of motivation and drive human behaviour.

Safety Needs:

Safety needs include physical and emotional security, stability, and protection from harm or danger.

After physiological needs are satisfied in a good level, individuals will seek for safety and stability in their environment. This includes personal safety, financial security, a stable job, health, and a secure living environment.

Social Needs:

Social needs are the desire for social connection, companionship, affection, and a sense of belonging.

Humans are social beings that have a natural need for relationships, friendships and acceptance within family, friendships, and communities.

Esteem Needs:

Esteem needs include the desire for self-esteem, self-respect, recognition, and a sense of accomplishment.

Self-Actualization Needs:

Self-actualization needs represent the highest level of human needs in Maslow's hierarchy. They involve the realization of one's full potential, personal growth, and self-fulfilment. Self-

actualization is about pursuing one's passions, engaging in creative activities, seeking personal development, and experiencing a sense of purpose and fulfilment in life.

According to Maslow, individuals go through these levels of needs in a sequential manner. As lower-level needs are satisfied, higher-level needs become more prominent and the driving force behind human motivation. Maslow suggested that the accomplishments of these needs is necessary for someone to reach their highest potential and achieve personal well-being and satisfaction.

It is important to note that while Maslow's Hierarchy of Needs provides a valuable framework for understanding human motivation, and the theory remains influential in various fields, including psychology, human resources, and self-development, providing insights into the factors that drive human behaviour and well-being.

5. Creating the 3D Environment

5.1 LiDAR - LiDAR Scanning:

LiDAR stands for Light Detection and Ranging. It is a remote sensing method that uses light from a laser to collect measurements and measure distances. It is also known as laser scanning or 3D scanning and 3D models and maps of objects and environments can be created from this procedure.

5.1.1 LiDAR System

A complete LiDAR system is made up of several components. All of the components work together to generate, record and georeference the data. The main components are:

- Lidar Source, Detector and Scanning Mechanism
- Timing Electronics
- Global Positioning System (GPS)
- Inertia Measurement Unit (IMU)
- Computer

The way a LiDAR sensor works depending on the sensor used, is to measure distance by shooting a precise, high-powered laser at a target and closely measuring the pulse that bounce off objects and return to the LiDAR sensor. The sensor uses the time it takes for each pulse to return to calculate distance (time of flight). Each of these pulsed laser measurements, or returns, can be processed into a 3D visualization known as a ‘point cloud’.

5.1.2 LIDAR FORMULA

The entire process of bouncing a beam of light or laser off an object, receiving the returned signal, and calculating its absolute position in space can be represented mathematically using this formula:

d = c * t / 2

In the formula, each letter represents:

d is the distance

c is the speed of light

t is the time of the flight

LiDAR is actually quite similar to how radar and sonar measure distance, except instead of using radio or sound waves, LiDAR systems use light. By taking into account the direction the light was sent, the position of the LiDAR scanner, and the distance between two points, LiDAR payloads are able to derive the exact 3D positions of every point from which signals return, or bounce back.

LiDAR traces its roots back to the early 1960s, when lasers were first invented and scanners using them were mounted to planes. Back then, the word LiDAR wasn't even an acronym—it was just a quick combination of the words "light" and "radar."

At first, LiDAR was mainly used to help make maps of small rivers and streams. But in the 1980s, with the emergence of GPS, LiDAR became an integral tool in collecting large-scale geospatial data and in creating topographical maps.

LiDAR sensors of the 1980s were large, clunky, and fairly inaccurate. They were also almost exclusively mounted to large, piloted airplanes, and their operation was manual, expensive, and did not always provide a good return on investment.

Today, LiDAR technology is cheaper, smaller, and more accessible than ever before, leading to its proliferation across dozens of different industries and fields. In fact, recent iPhones have LiDAR scanners that can create 3D models from up to 4.5 meters (15 feet) away.

5.2 BLENDER

Blender is a powerful and versatile 3D modelling and animation software that we used to fix any imperfections from LIDAR scanning and adding movement to agents in a virtual environment.

Fixing Imperfections from LIDAR Scanning:

- LIDAR scanning can produce highly accurate point cloud data, but it may also contain imperfections due to various factors, such as sensor noise, occlusions, or inaccuracies in the scanning process. Blender's advanced 3D modelling capabilities allow users to import LIDAR point cloud data and use various tools and filters to clean, process, and refine the scanned geometry.

- Blender's point cloud editing tools, mesh reconstruction algorithms, and clean-up functions enable users to remove noise, fill gaps, and smoothen surfaces, resulting in a more accurate and visually appealing representation of the scanned environment.

5.3 Mixamo

Mixamo for Agent Animation:

Mixamo is a web-based service provided by Adobe that offers a vast library of pre-made 3D characters and animations that allows users to easily animate virtual agents from a wide range of animations without the need for complex manual animation work. Once the desired animations are chosen, Mixamo automatically applies them to the 3D agents and provides a downloadable animation file, which can be easily imported back into Blender for further scene integration and refinement. Then can easily export 3D agents from Blender as FBX or other compatible formats.

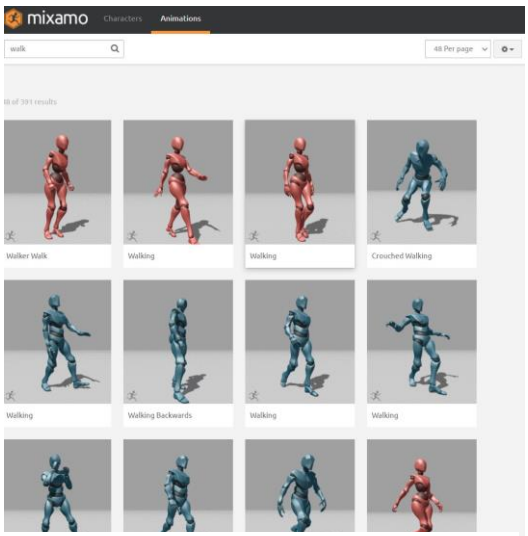


Figure 2 : Mixamo

By using Blender's capabilities to address LIDAR scanning imperfections and integrating Mixamo's animation library, developers and 3D artists can create more realistic and lifelike virtual environments with dynamic and expressive agents. Blender's open-source nature and active community support also make it a popular choice for game developers, researchers, and artists seeking to enhance the quality and realism of their projects.

In recent years, the integration of LiDAR scanning technology combined with 3D modelling and animation software like Blender has revolutionized the process of generating realistic agents, objects and environments for games and other purposes.

5.4 Factory Scenario:

The scenario considered in this thesis is a factory setting, where GOAP is applied to simulate intelligent behaviors within an industrial environment. The factory setting poses unique challenges, such as placing orders and managing them, production lines, coordinating tasks, common needs for the agents (eating, having a break, going to the bathroom) and responding to dynamic events. By applying GOAP in this context, someone can create virtual characters that exhibit realistic decision-making and efficient action execution, mirroring the complexities found in real-world factory scenarios.

5.5 Project Application

In order for the game to be created, the Unity Engine 2021.3.1f1 was used and the programming language for the scripts was C#.

5.6 Methodology used

This thesis started with learning how we can scan with a LiDAR sensor properly, what app to use, what is the ideal lighting, what surfaces are good for scanning, what angles to use, in what speed, etc.

The LiDAR scanner we used in this thesis was from an iPhone 14 Pro Max

Figure 3: IPhone - LiDAR sensor

Figure 4 : IPhone

and after scanning a lot of different places, objects and persons in the aluminium factory of Lamda Leventis ABEE that is placed in Kalyvia Attica we managed to have some results.

We used Scaniverse application, a free application from App Store for the creation of 3D models through LiDAR scanning.

Figure 5: Scaniverse

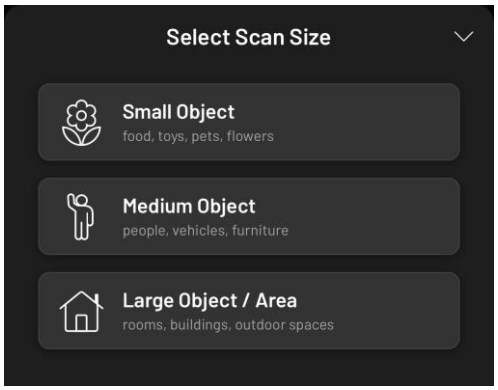


Figure 6: Size Selection

Depending on the size of the object or the area we wanted to scan we adjust the mode to small – medium or large.

When scanning an object, we have to slowly move around the object we want to scan targeting with the lens of the LiDAR sensor and try not to leave many gaps. Most of the apps show in the screen if there are any areas that need filling so you scan them more carefully. Depending of the size of the object we do this procedure a few times to have a better result and until we have no gaps in our scans.

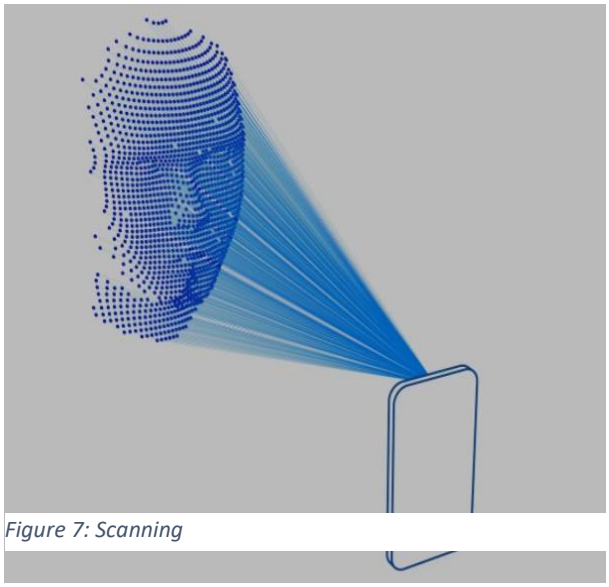


Figure 7: Scanning

Another thing that someone should take into account during this kind of scanning is lighting, it shouldn't be very bright or very dark and the surfaces shouldn't be very shiny in order not to deflect light. If there is any shiny surfaces one way to solve this problem is to apply chalk on them and fix it later on Blender. Also the person who does this process needs to have a steady hand and move smoothly through all the sides and the angles of the object or the area he scans. Being careful in this procedure can reduce the time we will spend later to fix any bad or missing surfaces.

At the pictures that follow we can see the screen of the phone during the scan of a CNC machine:

Figure 8: CNC scanning 1

Figure 9: CNC scanning 2

We can see the red and white area that inform us that the specific areas have not be scanned or need to be scanned more carefully in order to have a better result in the specific area. After we scan our object in every side and at the top so we don't have any blanks (red and white areas) in our object we can stop scanning.

And here we can see the final result of the previous scan:

Figure 10: CNC scanning result

And the same from one of our characters:

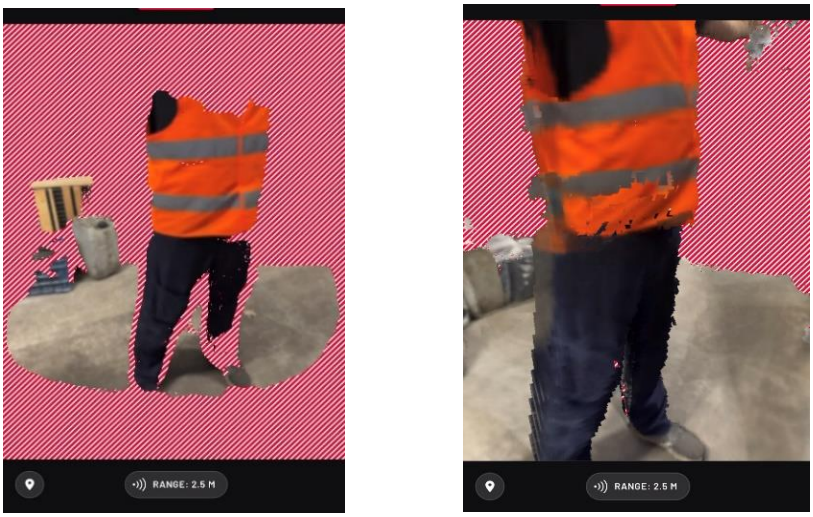


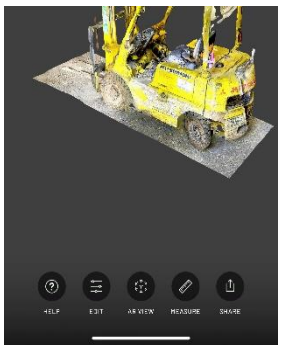
Figure 11: Screen Caption during scanning

And the final result:

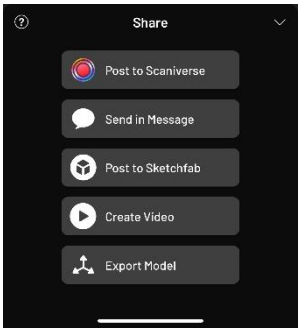
Figure 12: Human scanning final result

After finishing scanning we came up with a library of scans that we transferred from our phone to a pc for further processing where it was needed. We export the scans from the app as FBX file's witch is friendlier to BLENDER and we can manage them better.

The steps for this procedure was:



Step1 (SHARE)



Step2 (EXPORT MODEL)

Step3 (FBX)

Figure 13: Exporting scans

And then we can import the scans to blender from where we save them simply by following the procedure that is shown below:

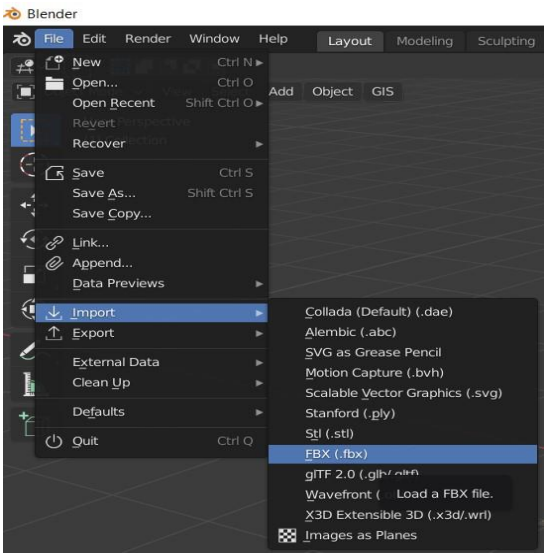
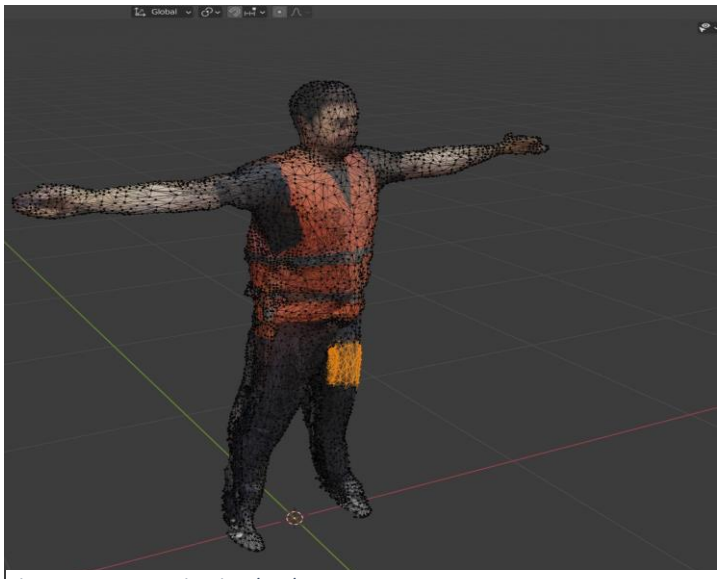


Figure 14: Importing Scans to Blender

File ⇨ Import ⇨ FBX

After importing our scans there are plenty of techniques that we can apply to fix any imperfections. Most of the times we used Clean Up tools (Decimate geometry, Fill holes, Make Planar Faces, Degenerate Dissolve (http://docs.blender.org/manual/en/2.81/modeling/meshes/editing/cleanup.html)), Shrinkwrap Modifier (https://www.youtube.com/watch?v=8vLX8e1zbY8), (https://www.youtube.com/shorts/gh7pN1OjYzQ) and Sculpt mode features.



In order to be able to use Sculpt mode in a scanned model one of the best options is to select our model in Edit mode and click Mesh at the Bar at the top, then Clean Up and Merge by distance.

In this way all the vertices of our mesh will stay connected when we use our brushes in Sculpt mode.

Figure 15: Processing in Blender

Another choice is to use instant – meshes program and then import the fixed mesh in to blender as an FBX file, sometimes might be easier this way. (https://github.com/wjakob/instant-meshes)

But anyone can choose what feel is best for him.

When we are satisfied with the result we export the scan as an FBX file so we can import it in to unity to make our scene.
In case of a Human scan we also perform the same procedure but now we have to apply character 3D Rigging to our humanoid and also give him some movement.
3D Rigging is the process of creating a skeleton for a 3D model so it can move. Most commonly, characters are rigged before they are animated because if a character model doesn't have a rig it can't be deformed and moved around.

In Blender we can apply the Humanoid rigging option and then adjust the rigs to our character properly.

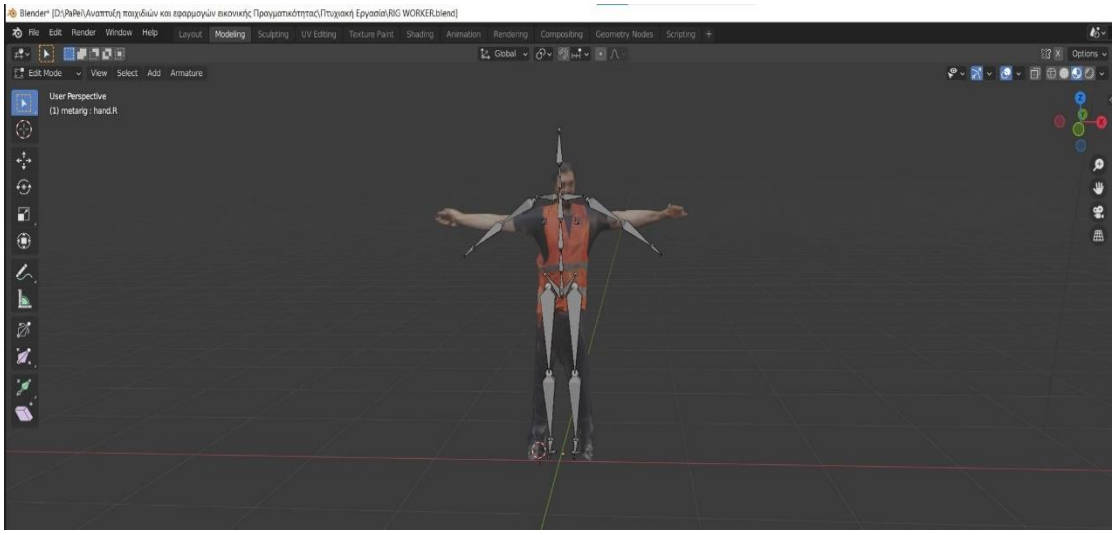


Figure 16:Humanoid rigging

Then we can download from Mixamo the animation we desire and attach it to our character.

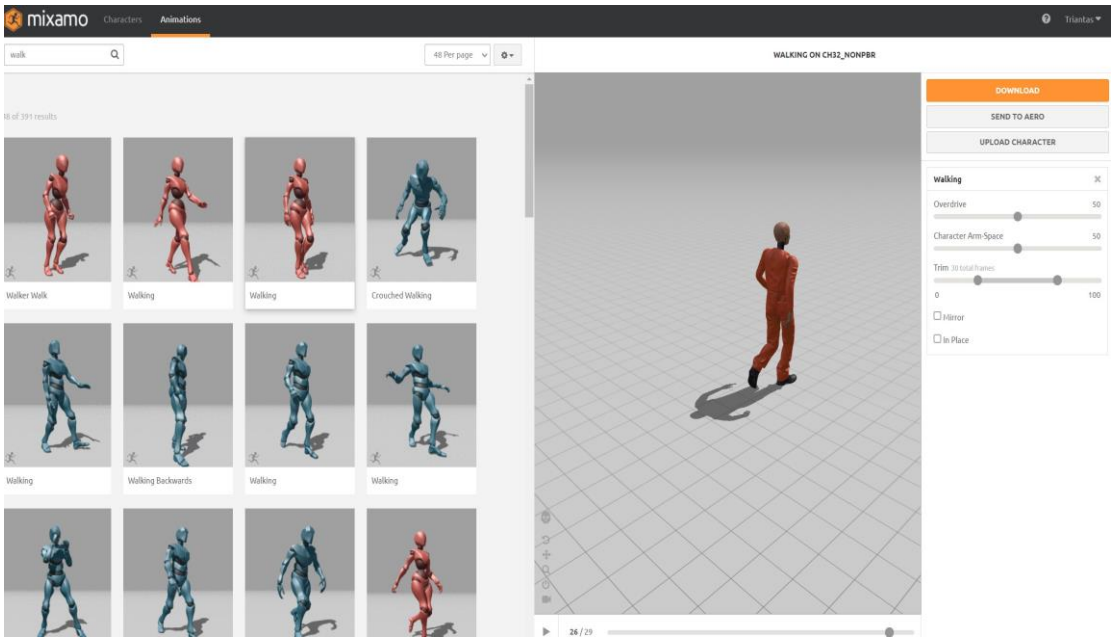
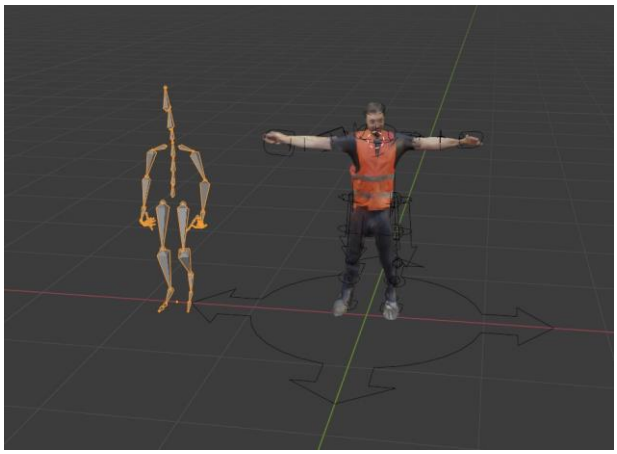


Figure 17: Choosing animation from Mixam



After downloading the animation, we desire we pair the rigs of the animation figure to the ones we created for our character or we can do it later in Unity.

Figure 18: Pairing Rigs

After making some adjustments to the bones of the animation and the ones of our character we can pair them and the desired animation is now set to him.

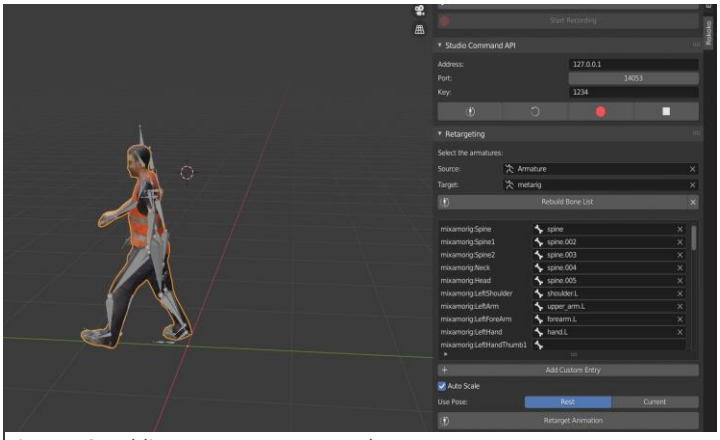


Figure 19: Adding movement to our character

When everything is ready we start a new project in Unity and make a Library of our scans. We also make a typical scene to add them.
In this thesis we ended up with a scene looking like the picture below:

Figure 20: Final Scene

The scans that added in the final scene are:

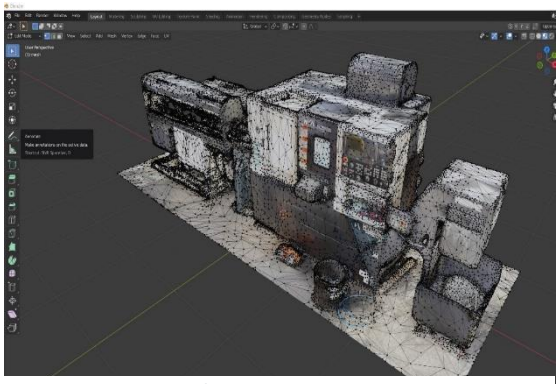


Figure 26 : CNC Machines

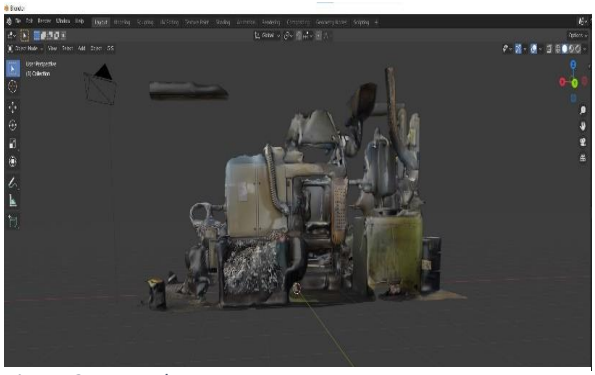


Figure 25: Foundry press

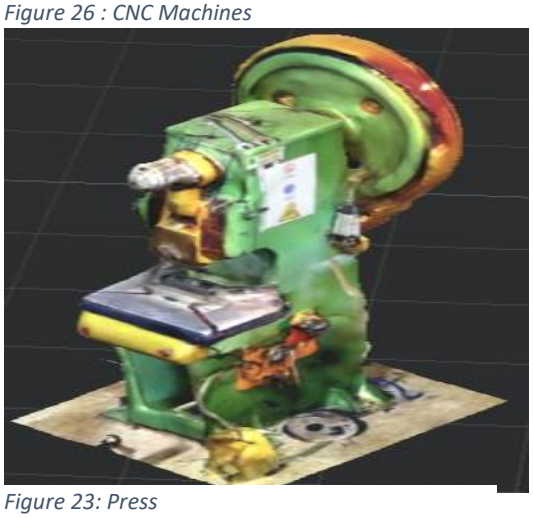


Figure 23: Press

Figure 24:Clark

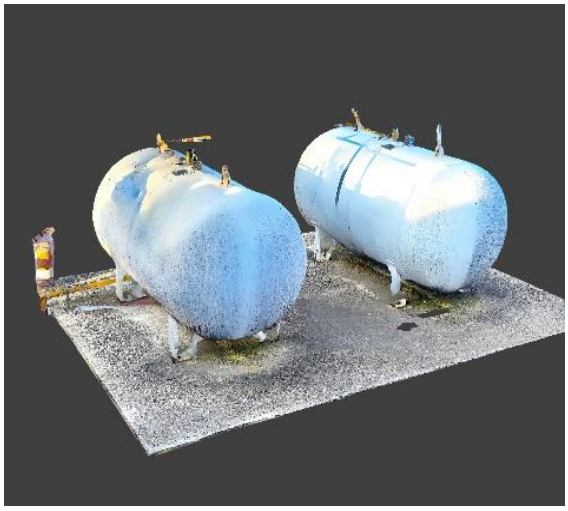


Figure 22: Gas tanks

Figure 21: Materials for use

Figure 28: Welding Department

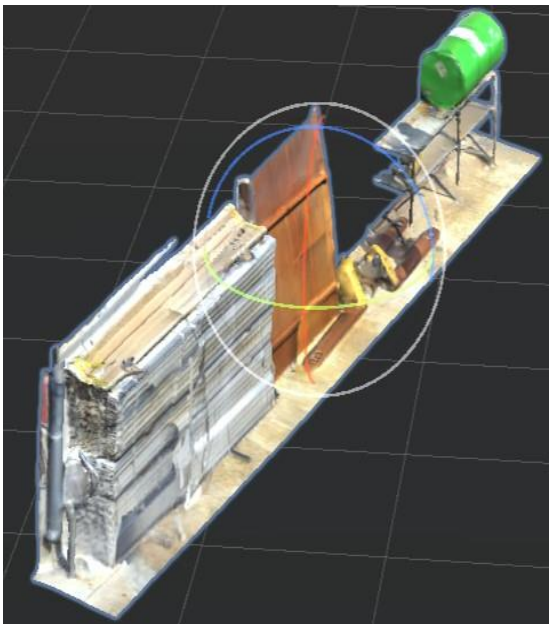


Figure 27: Aluminum Storage

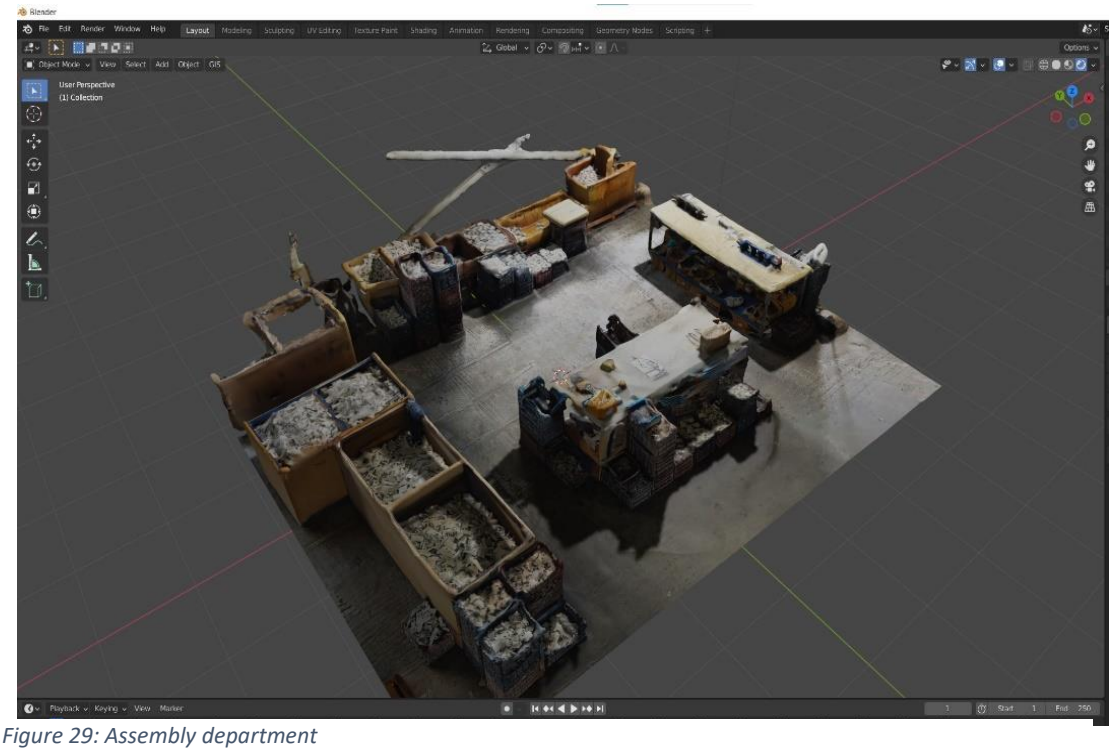


Figure 29: Assembly department

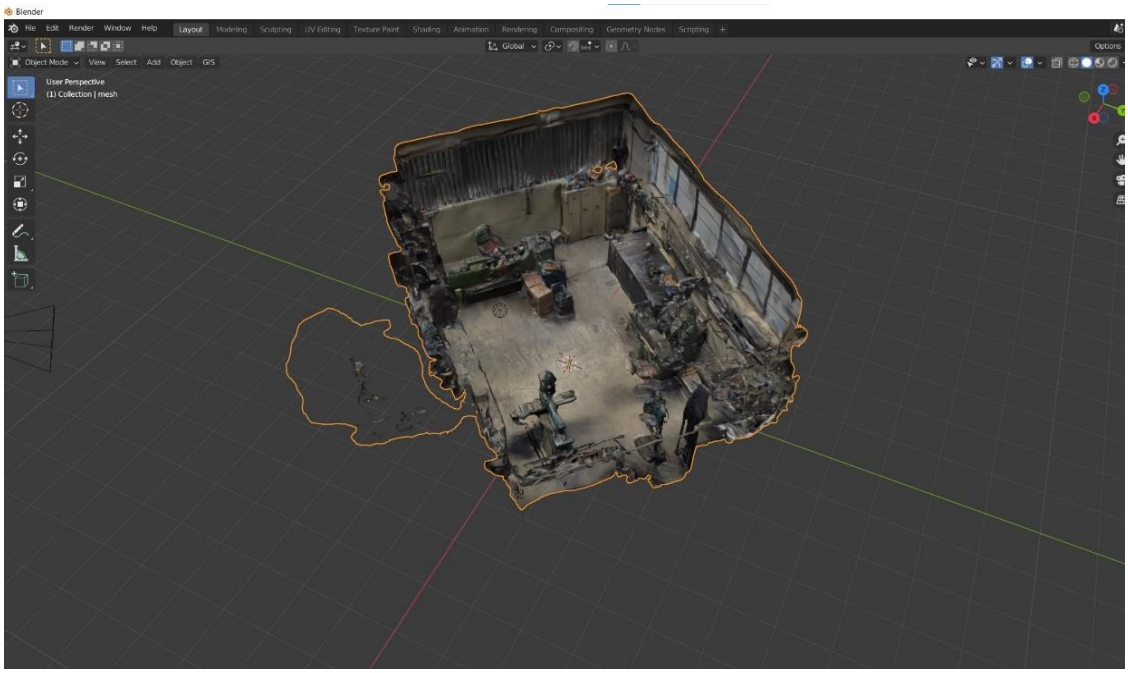


Figure 30: Mechanical workshop

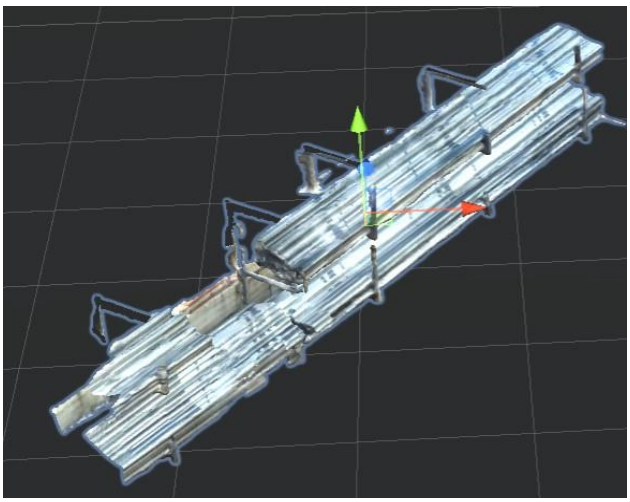


Figure 31: Aluminum Storage2

Figure 32: Car

Figure 35:Worker 1

Figure 33: Worker 1V2

Figure 34:: Worker 2

6. Factory scenario - Code

The scenario of this thesis is to try to mimic real life events that happen a random day in a factory. This involves in one side the customers that enter the facilities, register at the reception desk, waiting at the waiting area, placing their orders, seeking for an advice etc. And from the other side there is the stuff that works there, the secretaries, the advisors, the designers, the supervisors and different type of workers.
Every type of character that is in the scene has some beliefs, some goals and some needs and what we did was to try to mimic the way someone prioritizes them in the simplest way.

As an example we can use the Supervisor (one of our main characters).
He has some goals in this scenario.
First of all, he has to arrange the orders and then to supervise the workers, the machines, to check the stock of the materials and the areas.
But when a need that is bigger than his goals, let's say he has to use the toilet, or he is very tired and need to have a rest and something to eat then he stops following his goals and does what is more important and crucial for him.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

Unity Script (5 asset references) | 0 references
public class Supervisor : GAgent
{
    // Start is called before the first frame update
    Unity Message | 0 references
    new void Start()
    {
        base.Start();
        SubGoal s1 = new SubGoal("manageOrders", 1, false);
        goals.Add(s1, 4);

        SubGoal s2 = new SubGoal("rested", 1, false);
        goals.Add(s2, 5);

        SubGoal s3 = new SubGoal("supervised", 1, false);
        goals.Add(s3, 3);


        SubGoal s4 = new SubGoal("checked", 1, false);
        goals.Add(s4, 3);

        SubGoal s5 = new SubGoal("relief", 1, false);
        goals.Add(s5, 4);

        Invoke("GetTired", Random.Range(15, 20));
        Invoke("NeedRelief", Random.Range(2, 5));

    }

    0 references
    void GetTired()
    {
        beliefs.ModifyState("exhausted", 0);
        Invoke("GetTired", Random.Range(2, 5));

    }

    0 references
    void NeedRelief()
    {
        beliefs.ModifyState("busting", 0);
        Invoke("NeedRelief", Random.Range(2, 5));

    }


}
```

Figure 36: Supervisor Script

In the script above we can see the Supervisors script. One of the scripts that are attached to the character "Supervisor". In this script we can set the needs of the character and its importance to him.
Also we can program some needs to appear randomly, (Invoke method), between a specific period of time.

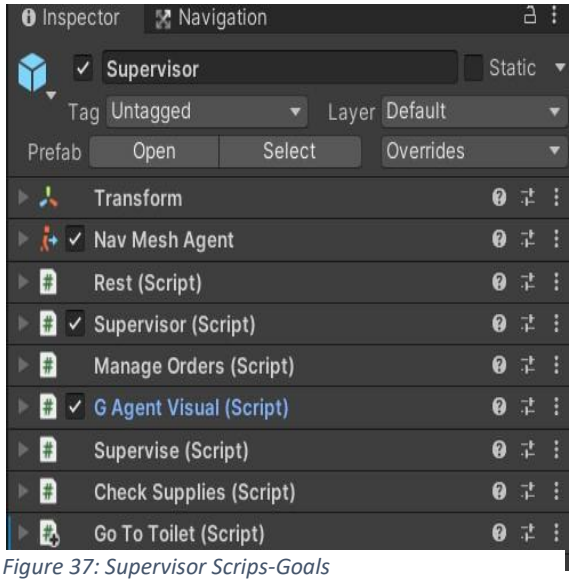


Figure 37: Supervisor Scrips-Goals

Each character has a script like this in which we organize and prioritize his needs (beliefs) and how his actions will affect the world (Worldstates) and the other characters.
In this scenario the physiological needs we attached to the characters except of their main goals are simple needs like going to the bathroom, take a break when tired or go to the restaurant to eat something, and we can add whatever we want, anytime and give it the priority we think is the best between others and all of these by using G.O.A.P.

So, the Supervisor (Green hat) wants to:

- manage his orders at his office:

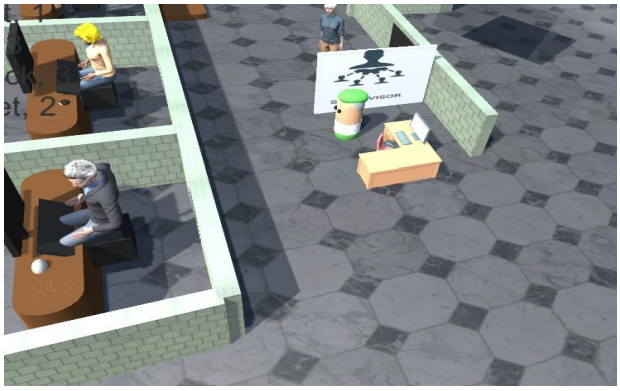


Figure 38: Supervisor manage orders

- Check the different Departments of the factory:
  (Green Squares)

Figure 39: Supervisor Checks Departments 1

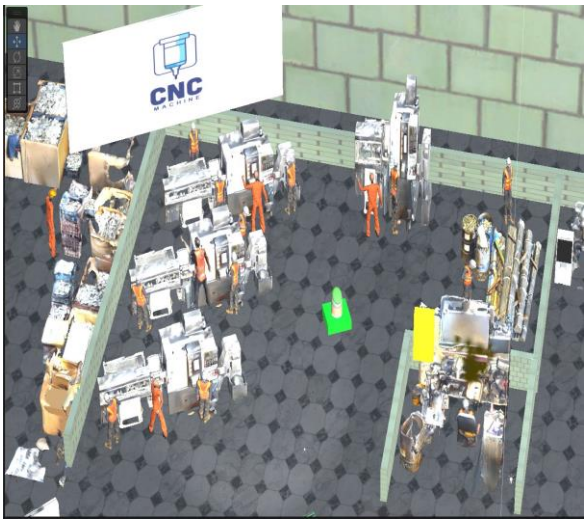


Figure 40: Supervisor Checks Departments 2

- Check the stock of the materials:

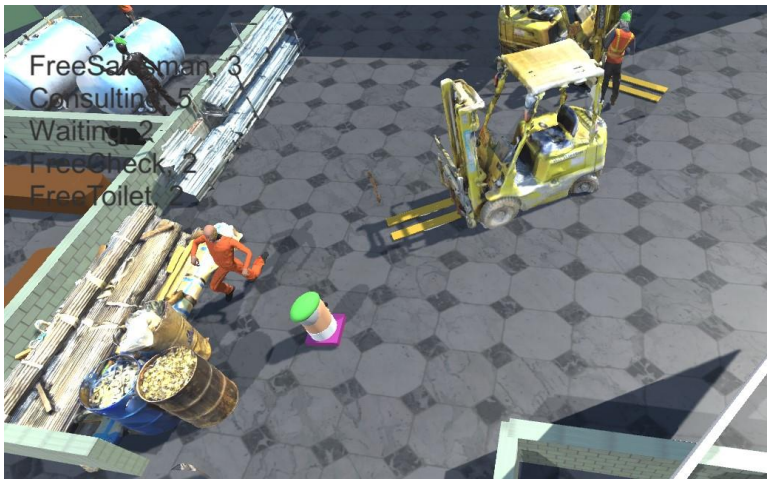


Figure 41: Supervisor checks the stock of the materials (purple square)

But when he needs to visit the toilet or the restaurant he will give priority to this needs and go.

- Supervisor visiting the toilet:

Figure 42: Supervisor visits the bathroom

- Supervisor visiting the restaurant:

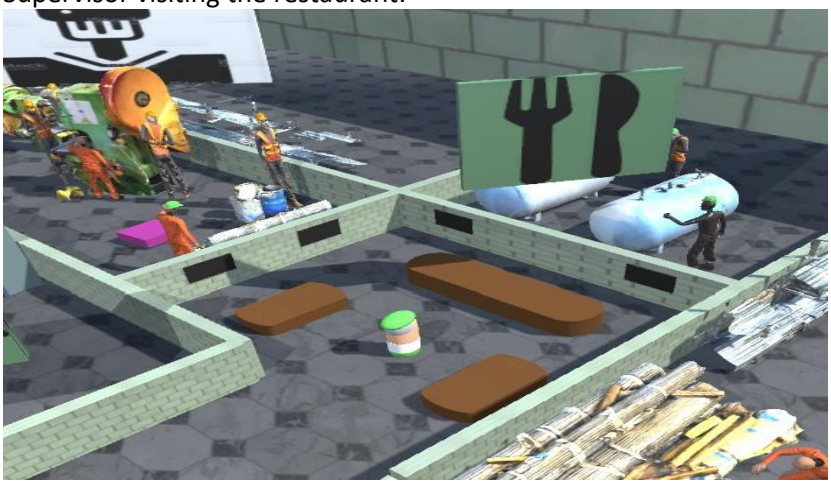


Figure 43: Supervisor visiting the restaurant

Every time an agent uses one of the resources, or when a resource or another NPC is added, the rest of the world gets the information and acts accordingly.

The secretary for example knows when a client will enter the facilities and go to the waiting room, and her goal is to take him and when a salesman is available to guide him there and stay with him to help with the order.

In the picture above we can see the blonde secretary approaching the customers (black suit figure) waiting area.

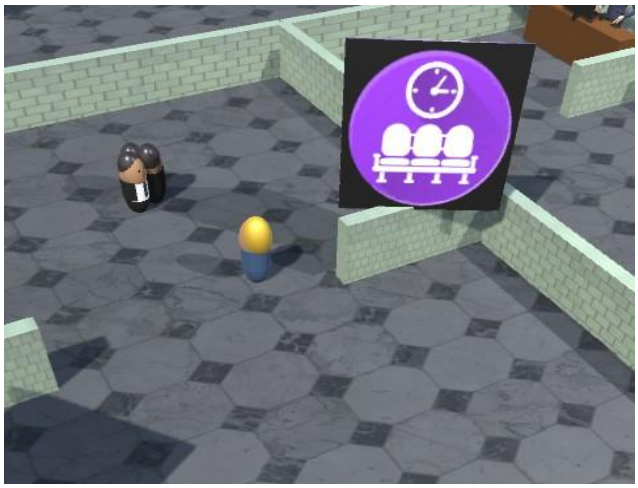


Figure 44: Secretary approaching the customers in customers waiting area

And guide him to one of the free salesmen to place an order.
After finishing this task, she will keep doing her job until she feels the need to take a rest or visit the bathroom.

Figure 45: Secretary guides customer to the salesman

The customer has a main goal to place an order. The steps to succeed his goal are:

- Go to the factory
- Get registered
- Go to the Consultants
- Go to the customers waiting room
- Wait for the secretary to guide him
- Follow the secretary to the salesman
- Leave the facilities

We can see some screenshots of his actions:

Figure 46: Customer enters the factory

Figure 47: Client registration

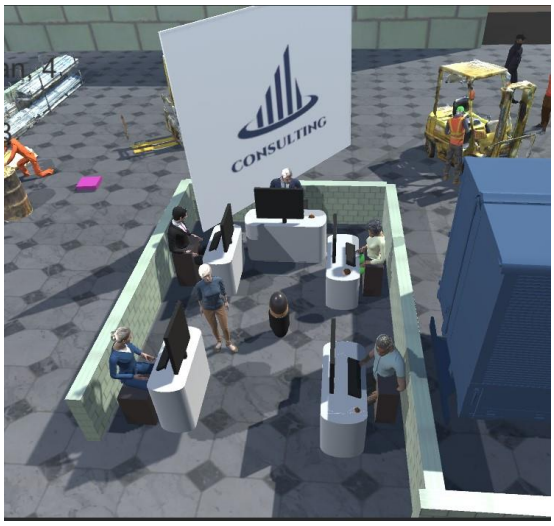


Figure 49: Client goes for consulting

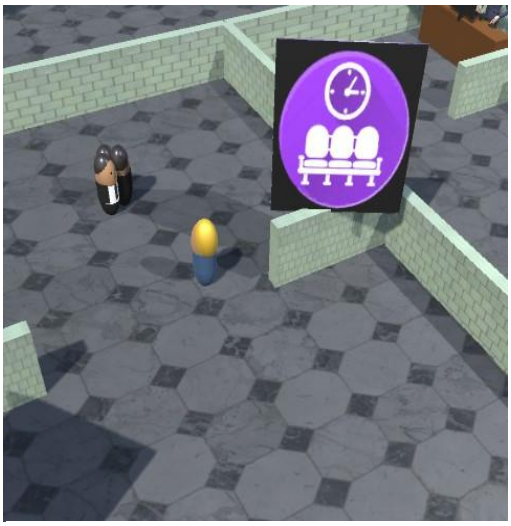


Figure 48:Client waits for the secretary to guide him

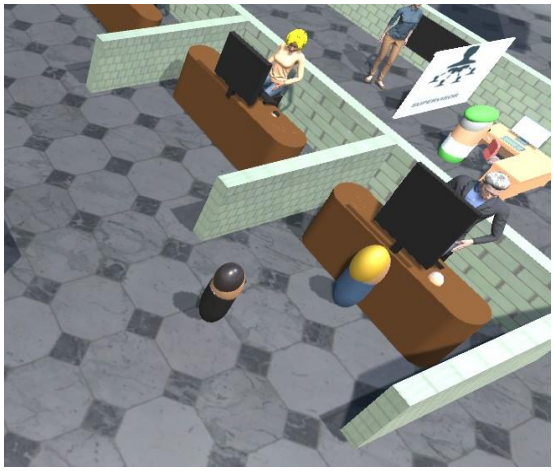


Figure 50: Follows her to the salesman

Figure 51: Client leaves

Every action of every agent have some preconditions that need to be met and some after effects that can affect the word and other agents and we can set them in a way that would feel right. This way we can give a more realistic sense in the scene.

At the picture below we can see the preconditions and after effects of the:
Supervisor:

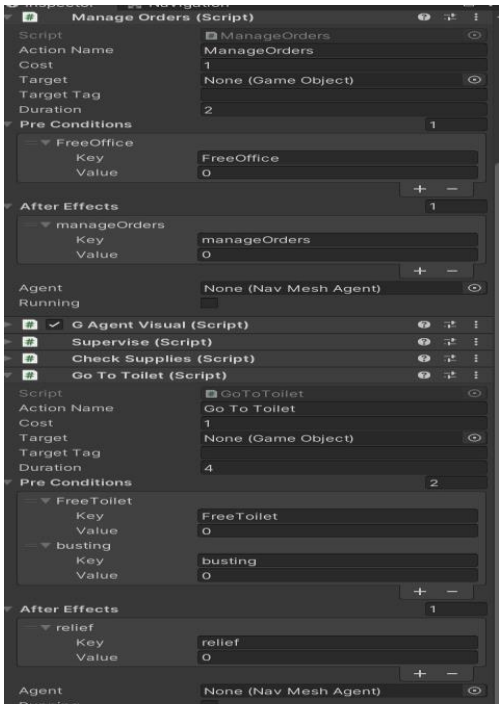


Figure 52: Supervisors pre conditions and after effects

In our scenario we assume that there are more than one supervisors, so in order for one of them to go to an office to view and manage the orders we set a precondition that it has to be a Free office, (the information to the rest of the world comes from GWorld script as we will see later), and we set the duration to two (2) seconds. The after effect of this act is that he managed the orders.
The same we did for his need to go to the toilet, only this time we set two preconditions, the first is to be a free toilet (the information to the rest of the world comes from GWorld script as we will see later) and the second is that he feels he has to go. The aftereffects is that he will feel relief after this action.

Working this way, we can manage every action of every agent and make him act autonomously even in more complicated scenarios. The only difficulty is to set his goals carefully.
In the picture below we can see the same thing for the client:

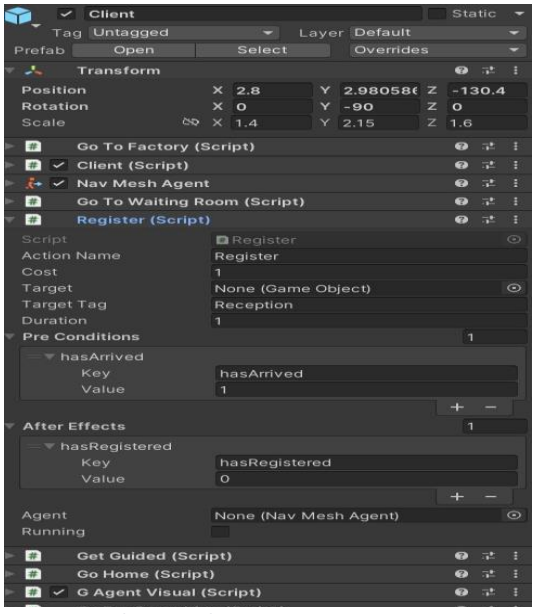


Figure 53: Cliens goals

His final Goal is to place his order. In order to do that he has to arrive to the Factory, go to the reception to get register, go to the advisors if he wants some advice, then in waiting room, wait for the secretary to guide him to a free salesman, place his order and return to his home. This is what we see in this picture with every act having its own preconditions and after effects and in the end of all this procedures to accomplish his Goal.

To understand better how, we accomplished that, we have to get into the code behind all this. In this project the main classes are the following:

GAction Class:

-Represents actions that game agents can perform.

-Stores action details like name, cost, target, duration, preconditions, and effects.

-Allows checking action achievability and pre/post action execution.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

Unity Script | 45 references
public abstract class GAction : MonoBehaviour
{
    public string actionName = "Action";
    public float cost = 1.0f;
    public GameObject target;
    public string targetTag;
    public float duration = 0;
    public WorldState[] preConditions;
    public WorldState[] afterEffects;
    public NavMeshAgent agent;

    public Dictionary<string, int> preconditions;
    public Dictionary<string, int> effects;

    public WorldStates agentBeliefs;

    public GInventory inventory;
    public WorldStates beliefs;

    public bool running = false;
    0 references
    public GAction()
    {
        preconditions= new Dictionary<string, int>();
        effects= new Dictionary<string, int>();
    }

    Unity Message | 0 references
    public void Awake()
    {
        agent = this.gameObject.GetComponent<NavMeshAgent>();
        if (preConditions!=null)
            foreach (WorldState w in preConditions)
            {
                preconditions.Add(w.key, w.value);
            }
        if (afterEffects!= null)
            foreach (WorldState w in afterEffects)
            {
                effects.Add(w.key, w.value);
            }
        inventory = this.GetComponent<GAgent>().inventory;
        beliefs = this.GetComponent<GAgent>().beliefs;
    }

    1 reference
    public bool IsAchievable()
    {
        return true;
    }

    1 reference
    public bool IsAchievableGiven(Dictionary<string, int>conditions)
    {
        foreach(KeyValuePair<string, int>p in preconditions)
        {
            if (!conditions.ContainsKey(p.Key))
                return false;
        }
        return true;
    }

    14 references
    public abstract bool PrePerform();
    14 references
    public abstract bool PostPerform();
}
```

Figure 54: GAction Class

GInventory Class:

-Manages an inventory of game objects (items).

-Provides methods to add, find, and remove items by their tags.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

3 references
public class GInventory
{
    public List<GameObject> items = new List<GameObject>();
    6 references
    public void AddItem(GameObject i)
    {
        items.Add(i);
    }

    2 references
    public GameObject FindItemWithTag(string tag)
    {
        foreach(GameObject i in items)
        {
            if (i.tag == tag)
            {
                return i;
            }
        }
        return null;
    }
    6 references
    public void RemoveItem(GameObject i)
    {
        int indexToRemove=-1;
        foreach (GameObject g in items)
        {
            indexToRemove++;
            if (g == i)
                break;
        }
        if(indexToRemove>=-1)
            items.RemoveAt(indexToRemove);
    }
}
```

Figure 55: GInventory Class

GPlanner Class:

-Implements a goal-based planning system for generating action plans.

-Plans actions to achieve specific goals based on the current world state.

-Utilizes a search graph and recursive methods for planning.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;


16 references
public class Node
{
    public Node parent;
    public float cost;
    public Dictionary<string, int> state;
    public GAction action;

    1 reference
    public Node(Node parent, float cost,Dictionary<string,int> allstates, GAction action)
    {
        this.parent = parent;
        this.cost = cost;
        this.state= new Dictionary<string, int>(allstates);
        this.action = action;
    }
    1 reference
    public Node(Node parent, float cost, Dictionary<string, int> allstates, Dictionary<string, int> beliefstates,  GAction action)
    {
        this.parent = parent;
        this.cost = cost;
        this.state = new Dictionary<string, int>(allstates);
        foreach(KeyValuePair<string, int> b in beliefstates)
            if(!this.state.ContainsKey(b.Key))
                this.state.Add(b.Key, b.Value);
        this.action = action;
    }
}


2 references
public class GPlanner
{
    1 reference
    public Queue<GAction> plan(List<GAction> actions, Dictionary<string, int> goal, WorldStates beliefstates)
    {
        List<GAction> usableActions = new List<GAction>();
        foreach (GAction a in actions)
        {
            if (a.IsAchievable())
                usableActions.Add(a);
        }

        List<Node> leaves = new List<Node>();
        Node start = new Node(null, 0, GWorld.Instance.GetWorld().GetStates(), beliefstates.GetStates(), null);

        bool success = BuildGraph(start, leaves, usableActions, goal);

        if (!success)
        {
            Debug.Log("NO PLAN");
            return null;
        }

        Node cheapest = null;
        foreach (Node leaf in leaves)
        {

        List<GAction> result = new List<GAction>();
        Node n = cheapest;
        while (n != null)
        {
            if (n.action != null)
            {
                result.Insert(0, n.action);
            }
            n = n.parent;
        }

        Queue<GAction> queue = new Queue<GAction>();
        foreach (GAction a in result)
        {
            queue.Enqueue(a);
        }

        Debug.Log("The Plan is: ");
        foreach (GAction a in queue)
        {
            Debug.Log("Q: " + a.actionName);
        }

        return queue;
    }
    2 references
    private bool BuildGraph(Node parent, List<Node> leaves, List<GAction>usuableActions, Dictionary<string, int>goal)
    {
        bool foundPath = false;
        foreach(GAction action in usuableActions)
        {
            if(action.IsAchievableGiven(parent.state))
            {
                Dictionary<string, int> currentState = new Dictionary<string, int>(parent.state);
                foreach(KeyValuePair<string, int> eff in action.effects)
                {
                    if(!currentState.ContainsKey(eff.Key))
                        currentState.Add(eff.Key, eff.Value);
                }
                Node node= new Node(parent, parent.cost + action.cost,currentState,action);
                if (GoalAchieved(goal, currentState))
                {
                    leaves.Add(node);
                    foundPath = true;
                }
                else
                {
                    List<GAction> subset = ActionSubset(usuableActions, action);
                    bool found = BuildGraph(node, leaves, subset, goal);
                    if (found)
                        foundPath = true;
                }
            }
        }
        return foundPath;
    }

    1 reference
    private bool GoalAchieved(Dictionary<string, int> goal, Dictionary<string, int> state)
    {
        foreach (KeyValuePair<string, int> g in goal)
        {
            if (!state.ContainsKey(g.Key))
                return false;
        }
        return true;
    }
```

Figure 56: GPlanner Class 1

```
133
        1 reference
134     private List<GAction> ActionSubset(List<GAction> actions, GAction removeMe)
135     {
136         List<GAction> subset = new List<GAction>();
137         foreach (GAction a in actions)
138         {
139             if (!a.Equals(removeMe))
140                 subset.Add(a);
141         }
142         return subset;
143     }
144
145 }
```

Figure 57: GPlanner Class 2

GWorld Class:

-Manages global game world state and resource availability.

-Uses queues for clients, salesmen office, spots, checks.

-Initializes queues based on tags in the scene.

-Provides methods for adding and removing entities.

-Follows the singleton pattern.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using System.Linq;
4   using UnityEngine;
5
        32 references
6   public sealed class GWorld
7   {
8       private static readonly GWorld instance = new GWorld();
9       private static WorldStates world;
10      private static Queue<GameObject> clients;
11      private static Queue<GameObject> salesmen;
12      private static Queue<GameObject> offices;
13      private static Queue<GameObject> spots;
14      private static Queue<GameObject> checks;
15
            0 references
16      static GWorld()
17      {
18          world = new WorldStates();
19          clients = new Queue<GameObject>();
20          salesmen = new Queue<GameObject>();
21          offices = new Queue<GameObject>();
22          spots = new Queue<GameObject>();
23          checks = new Queue<GameObject>();
24
25          GameObject[] sales = GameObject.FindGameObjectsWithTag("Sales");
26          foreach(GameObject s in sales)
27              salesmen.Enqueue(s);
28          if (sales.Length > 0)
29              world.ModifyState("FreeSalesman", sales.Length);
30
31          GameObject[] offs = GameObject.FindGameObjectsWithTag("Office");
32          foreach (GameObject o in offs)
33              offices.Enqueue(o);
34          if (offs.Length > 0)
35              world.ModifyState("FreeOffice", offs.Length);
36
37          GameObject[] spt = GameObject.FindGameObjectsWithTag("spot");
38          foreach (GameObject sp in spt)
39              offices.Enqueue(sp);
40          if (spt.Length > 0)
41              world.ModifyState("FreeSpot", spt.Length);
42
43          GameObject[] chck = GameObject.FindGameObjectsWithTag("check");
44          foreach (GameObject c in chck)
```

Figure 58: GWorld Class 1

```
        foreach (GameObject sp in spt)
            offices.Enqueue(sp);
        if (spt.Length > 0)
            world.ModifyState("FreeSpot", spt.Length);

        GameObject[] chck = GameObject.FindGameObjectsWithTag("check");
        foreach (GameObject c in chck)
            offices.Enqueue(c);
        if (spt.Length > 0)
            world.ModifyState("FreeCheck", chck.Length);









        Time.timeScale = 5;

    }


    1 reference
    private GWorld()
    {

    }
    2 references
    public void AddClient(GameObject p)
    {
        clients.Enqueue(p);
    }
    1 reference
    public GameObject RemoveClient()
    {
        if (clients.Count == 0) return null;
        return clients.Dequeue();
    }

    2 references
    public void AddSalesman(GameObject s)
    {
        salesmen.Enqueue(s);
    }

    2 references
    public void AddSalesman(GameObject s)
    {
        salesmen.Enqueue(s);
    }
    1 reference
    public GameObject RemoveSalesman()
    {
        if (salesmen.Count == 0) return null;
        return salesmen.Dequeue();
    }
    1 reference
    public void AddOffice(GameObject o)
    {
        offices.Enqueue(o);
    }
    1 reference
    public GameObject RemoveOffice()
    {
        if (offices.Count == 0) return null;
        return offices.Dequeue();
    }

    2 references
    public void AddSpot(GameObject sp)
    {
        spots.Enqueue(sp);
    }
    2 references
    public GameObject RemoveSpot()
    {
        if (spots.Count == 0) return null;
        return spots.Dequeue();
    }


    0 references
    public void AddCheck(GameObject c)
    {
        checks.Enqueue(c);
    }
```

Figure 59: GWorld Class 2

```
104             checks.Enqueue(c);
105         }
            0 references
106         public GameObject RemoveCheck()
107         {
108             if (checks.Count == 0) return null;
109             return checks.Dequeue();
110         }
111
            27 references
112         public static GWorld Instance
113         {
114             get { return instance; }
115         }
116
            15 references
117         public WorldStates GetWorld()
118         {
119             return world;
120         }
121     }
122
```

Figure 60: GWorld Class 3

WorldStates Class:

-Manages a collection of states representing the global world state.

-Uses a dictionary to associate state keys with values.

-Provides methods for checking, adding, modifying, removing, and retrieving states.

-It allows for flexible representation and manipulation of the game world's state, which can influence agent decisions.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEditor;
4   using UnityEngine;
5
6   [System.Serializable]
    4 references
7   public class WorldState
8   {
9       public string key;
10      public int value;
11  }
12
    9 references
13  public class WorldStates
14  {
15      public Dictionary<string,int> states;
16
        2 references
17      public WorldStates()
18      {
19          states = new Dictionary<string,int>();
20      }
        0 references
21      public bool HasState(string key)
22      { return states.ContainsKey(key); }
23
        0 references
24      void AddState(string key, int value)
25      {
26          states.Add(key, value);
27      }
28
        22 references
29      public void ModifyState(string key, int value)
30      {
31          if(states.ContainsKey(key))
32          {
33              states[key] += value;
34              if (states[key] <= 0)
35                  RemoveState(key);
36          }
37          else
38              states.Add(key, value);
39      }
        2 references
40      public void RemoveState(string key)
41      {
42          if(states.ContainsKey(key))
43              states.Remove(key);
44      }
        0 references
45      public void SetState(string key, int value)
46      {
47          if (states.ContainsKey(key))
48              states[key] = value;
49          else
50              states.Add(key, value);
51      }
        3 references
52      public Dictionary<string, int>GetStates()
53      { return states; }
```

Figure 61: WorldStates Class

GAgent Class

Consists of two classes: SubGoal and GAgent.

SubGoal class

This class represents a sub-goal within the agent's behavior. It is used to specify what the agent should aim to achieve in its environment. Here are the key elements of the SubGoal class:

- sgoals: This is a dictionary that maps a goal name (a string) to an integer value that represents the priority of the goal.

remove: A Boolean value that indicates whether the sub-goal should be removed once it's achieved.

- The constructor SubGoal (string s, int i, bool r) initializes a sub-goal with the specified goal name s, priority i, and removal status r. It creates a new dictionary entry with the goal name and priority.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using System.Linq;
5
6
7
        26 references
8   public class  SubGoal
9   {
10      public Dictionary<string, int> sgoals;
11      public bool remove;
            10 references
12      public SubGoal(string s,int i,bool r)
13      {
14          sgoals = new Dictionary<string, int>();
15          sgoals.Add(s, i);
16          remove = r;
17      }
18  }
19
        Unity Script | 11 references
```

Figure 62: GAgent Class - SGoals

GAgent

GAgent Class:

The GAgent class represents an agent and manages his actions, goals, inventory, beliefs, and planning. If we look at the key components of this class:

- actions: A list that holds instances of GAction (actions) that the agent can perform.
- goals: A dictionary that associates sub-goals (instances of SubGoal) with their priorities (integers).
- inventory: An instance of the GInventory class, which likely manages the agent's inventory.
- beliefs: An instance of the WorldStates class, which may represent the agent's beliefs about the state of the world.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;



26 references
public class  SubGoal
{
    public Dictionary<string, int> sgoals;
    public bool remove;
    10 references
    public SubGoal(string s,int i,bool r)
    {
        sgoals = new Dictionary<string, int>();
        sgoals.Add(s, i);
        remove = r;
    }
}

Unity Script | 11 references
public class GAgent : MonoBehaviour
{
    public List<GAction> actions = new List<GAction>();
    public Dictionary<SubGoal, int> goals= new Dictionary<SubGoal, int>();
    public GInventory inventory= new GInventory();
    public WorldStates beliefs= new WorldStates();

    GPlanner planner;
    Queue<GAction> actionQueue;
    public GAction currentAction;
    SubGoal currentGoal;

    Vector3 destination = Vector3.zero;

    // Start is called before the first frame update
    Unity Message | 3 references
    public void Start()
    {
        GAction[] acts= this.GetComponents<GAction>();
        foreach (GAction a in acts)
            actions.Add(a);

    }

    bool invoked = false;
    0 references
    void CompleteAction()
    {
        currentAction.running = false;
        currentAction.PostPerform();
        invoked = false;
    }

    // Update is called once per frame
    Unity Message | 0 references
    void LateUpdate()
    {
        if (currentAction != null && currentAction.running)
        {
            float distanceToTarget = Vector3.Distance(destination, this.transform.position);
            if( distanceToTarget < 3f)// currentAction.agent.remainingDistance<1f)
            {
                //Debug.Log("Distance to Goal: " + currentAction.agent.remainingDistance);
                if (!invoked)
                {
                    Invoke("CompleteAction", currentAction.duration);
                    invoked = true;
                }
            }
            return;
        }
```

Figure 63: GAgent Class –Gagent 1

```
        if (planner == null || actionQueue == null)
        {
            planner = new GPlanner();

            var sortedGoals = from entry in goals orderby entry.Value descending select entry;

            foreach (KeyValuePair<SubGoal, int> sg in sortedGoals)
            {
                actionQueue = planner.plan(actions, sg.Key.sgoals, beliefs) ;
                if (actionQueue != null)
                {
                    currentGoal = sg.Key;
                    break;
                }
            }
        }

        if (actionQueue != null && actionQueue.Count == 0)
        {
            if (currentGoal.remove)
            {
                goals.Remove(currentGoal);
            }
            planner = null;
        }

        if (actionQueue != null && actionQueue.Count > 0)
        {
            currentAction = actionQueue.Dequeue();
            if (currentAction.PrePerform())
            {
                if (currentAction.target == null && currentAction.targetTag != "")
                    currentAction.target = GameObject.FindWithTag(currentAction.targetTag);

                if (currentAction.target != null)
                {
                    currentAction.running = true;
                    //look for a Destination and use that
                    Transform dest = currentAction.target.transform.Find("Destination");
                    if (dest != null)
                        destination = dest.position;
                    else
                        destination = currentAction.target.transform.position;

                    currentAction.agent.SetDestination(destination);
                }
            }
            else
            {
                actionQueue = null;
            }

        }
    }
}
```

Figure 64: GAgent Class -GAgent 2

These are the main scripts that form our G.O.A.P. model and create a framework for modeling and managing actions, inventories, planning, and world states within a Unity-based game or simulation. This system facilitates AI-driven decision-making, resource management, and goal achievement by game agents or characters. It provides a foundation for implementing intelligent behavior and interactions in the game world.

7. Maslow's Hierarchy of Needs and GOAP

Maslow's Hierarchy of Needs and Goal-Oriented Action Planning (GOAP) share a significant connection, as they both revolve around human behavior and motivation. Maslow's theory proposes a hierarchical structure of human needs, while GOAP provides a framework for intelligent decision-making and goal achievement in virtual agents within video games and simulations. By analyzing how GOAP-driven agents in video games fulfill these needs, we can gain valuable insights into how virtual environments can cater to fundamental human requirements and enhance player engagement and immersion.
Maslow's Hierarchy of Needs encompasses five levels: physiological, safety, love and belonging, esteem, and self-actualization.
GOAP-driven agents in video games can address each of these levels or similar in various ways, contributing to a more compelling and meaningful gaming experience for players.

At the base of the pyramid are physiological needs, such as food, water, and shelter. In virtual environments, GOAP can be utilized to program agents to prioritize their actions and do thinks that are more crucial for them, like self-preservation, seek resources, avoid hazards, providing a sense of realism and immersion for players.
Ensuring that virtual agents address their physiological needs enhances the player's connection to the game world and adds depth to the overall experience.

Moving up the hierarchy, safety needs encompass security and stability. GOAP-driven agents can exhibit behaviors that promote a sense of safety, both for themselves and the player adhering to safety protocols, avoiding danger, and cooperating with other agents, virtual characters giving a sense of security within the virtual environment, contributing to the player's emotional investment in the game.

The love and belonging needs are related to social interactions and the desire to be part of a community.
GOAP-driven agents can be programmed to display cooperative behaviors, help others, and engage in social interactions with each other and the player. This can add a sense of companionship and belonging within the virtual world, heightening the player's sense of connection and emotional involvement.

Esteem needs involve accomplishment, recognition, and self-worth.
GOAP-driven agents can exhibit competence and goal achievement, providing players with a sense of accomplishment and recognition for their actions. NPCs that react dynamically to player actions and achieve objectives reinforce the player's importance and contribution to the virtual world.

At the top of the pyramid are self-actualization needs, encompassing personal growth and fulfillment.
GOAP can contribute to self-actualization by providing NPCs with diverse and meaningful goals that align with their unique traits and attributes. Intelligent virtual agents that exhibit autonomy, creativity, and adaptability in their actions create a rich and immersive virtual environment, encouraging players to explore and engage more deeply with the game world.
In conclusion, the relative connection between Maslow's Hierarchy of Needs and GOAP lies in their shared focus on human behavior, motivation, and fulfillment. By incorporating GOAP-driven agents that address physiological, safety, social, esteem, and self-actualization needs, video game developers can

create more immersive and emotionally resonant gaming experiences. Understanding this connection between AI-driven behaviors and human needs fulfillment can lead to the design of games that offer players a deeper sense of engagement, emotional investment, and enjoyment.

In conclusion, this thesis has delved into the exploration of Goal-Oriented Action Planning (GOAP) within the Unity game engine, coupled with the integration of LiDAR scanning and Blender, to create an immersive factory scenario. The research has demonstrated how the adoption of GOAP-driven agents in the virtual environment effectively addresses fundamental human needs, as proposed by Maslow's Hierarchy of Needs, bridging the gap between AI-driven behaviors and the fulfillment of human motivations.
Maslow's Hierarchy of Needs provides a framework for understanding human motivations, encompassing physiological, safety, love and belonging, esteem, and self-actualization needs. Through the lens of GOAP, virtual agents in the factory scenario have been equipped to fulfill some needs in a nuanced manner in a way that it can create a more emotionally resonant and engaging gaming experience for players.

At the base of the pyramid, GOAP-driven agents prioritize self-preservation and safety, seeking resources and avoiding hazards, thereby addressing physiological needs and enhancing player immersion. Moving up, safety needs are met through agent behaviors that promote a sense of security and cooperation, both within the virtual environment and for the player.
Such agents create a rich and immersive virtual environment, encouraging players to explore and engage more deeply with the game world, fulfilling their desire for personal growth and fulfillment.
Moreover, the integration of LiDAR scanning and Blender has significantly enhanced the fidelity of the factory scenario, providing a visually authentic and dynamic virtual environment. LiDAR scanning accurately represents the physical space, while Blender's 3D modeling capabilities refine and enhance the scanned data.

8. Conclusion

The findings of this research contribute to AI-driven virtual environments. While the thesis demonstrates the potential of GOAP, LiDAR scanning, and Blender integration, it also acknowledges the challenges and limitations encountered during the development of the simulation. Future work could explore the integration of additional AI techniques, such as emotion modeling or reinforcement learning, to further enhance the depth and complexity of NPC behaviors.

In conclusion, this exploration of GOAP, LiDAR scanning, and Blender integration showcases the potential for creating engaging and meaningful virtual environments.
By understanding the connection between intelligent agents and fundamental human requirements, we pave the way for more emotionally resonant and immersive experiences in virtual worlds, ultimately contributing to the advancement of AI-driven simulations and game development. As the boundaries between virtual and real-world experiences continue to blur, the significance of considering human needs in AI-driven applications becomes increasingly vital for shaping more compassionate and enriching interactions in the digital realm.

9. Bibliography & References

1. Structure of Intelligent Agents and Environments, Alan Bundy (2003) (https://www.inf.ed.ac.uk/teaching/courses/ai2/module4/small_slides/small-agents.pdf)
2. AI for Game Developers, David M. Bourg, Glenn Seeman, 2004 (Chapter 1, Chapter 7, Chapter 9, Chapter 11, Chapter 14) (https://theswissbay.ch/pdf/Gentoomen%20Library/Game%20Development/Programming/AI%20for%20Game%20Developers%20-%20%20David%20M.%20Bourg%2C%20Glenn%20Seeman.pdf)
3. Artificial Intelligence For Games, 2nd Edition , Ian Millington ,2009 (https://theswissbay.ch/pdf/Gentoomen%20Library/Game%20Development/Programming/Artificial%20Intelligence%20for%20Games.pdf)
4. Behavior Trees in Robotics and AI: An Introduction, Michele Colledanchise, Petter Ögren, 2017-08-31 (https://arxiv.org/pdf/1709.00084.pdf) (https://www.sciencedirect.com/science/article/pii/S0921889022000513)
5. Finite State Machines , Karleigh Moore and Dishant Gupta (https://brilliant.org/wiki/finite-state-machines/)
6. Goal-Oriented Action Planning: Ten Years Old and No Fear! , Peter Higley, 2015 (https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc2015/presentations/Higley_Peter_Goal-Oriented_Action_Planning.pdf)
7. GOAP Analysis (https://medium.com/@stannotes/design-unpredictable-ai-in-games-part-1-architecture-3752a618db6) (https://alumni.media.mit.edu/~jorkin/goap.html)
8. Maslow's Hierarchy of Needs (https://canadacollege.edu/dreamers/docs/Maslows-Hierarchy-of-Needs.pdf)
9. MOTIVATION AND PERSONALITY, ABRAHAM H. MASLOW, 1954 (https://www.holybooks.com/wp-content/uploads/Motivation-and-Personality-Maslow.pdf)
10. LIDAR (https://www.mappedin.com/blog/product/indoor-mapping/what-is-lidar-scanning/) (https://support.apple.com/kb/SP876?locale=en_US)
11. SCANIVERSE (https://apps.apple.com/us/app/scaniverse-3d-scanner/id1541433223) (https://scaniverse.com/) (https://opentopography.org/blog/iphone-lidar-applications-geosciences)
12. UNITY (https://unity.com/) (https://learn.unity.com/tutorial/an-introduction-to-goap#) (https://assetstore.unity.com/packages/tools/behavior-ai/goal-oriented-action-planning-artificial-intelligence-72912) (https://assetstore.unity.com/)
13. MIXAMO (https://www.mixamo.com/#/), (https://www.mixamo.com/#/?page=1&type=Character) , (https://www.mixamo.com/#/?page=1&query=walk&type=Motion%2CMotionPack)
14. Class Presentations by my professor **Themistocles Panayiotopoulos**
15. A survey of Behaviour Trees in robotics and AI, Elsevier, Matteo Iovino, 8/22