# Goal Oriented Action Planning

**TSBK03 Advanced Game Programming - Project**

Filip Jakobsson - Filja442

December 31, 2024

# Contents

# 1 Introduction

The main goal of this project is to figure out how effective Goal-Oriented Action Planning (GOAP) is compared to simpler methods like Finite State Machines (FSM). The problem is to see which of these techniques works better for creating a non-player character (NPC) that can handle complex goals in a dynamic and unpredictable environment. This project aims to explore how well GOAP can adapt and make decisions compared to the straightforward, rule-based approach of FSM. To achieve this, the project has been implemented as a small 2D RPG, which serves as an effective simulation for testing these AI techniques in a dynamic environment.

To test this, the project involves creating an NPC in a survival game world. The world includes challenges like gathering resources, managing hunger, and fighting monsters. The NPC will need to collect wood and stone to craft arrows, which are necessary for fighting monsters. If it doesn't have arrows, it must avoid monsters instead of engaging them. Additionally, the NPC has a hunger bar that decreases over time. If the hunger bar reaches zero, the NPC takes damage, making it crucial to gather food regularly to survive.

Resources like wood, stone, and food spawn randomly on a timer, adding unpredictability to the environment. The NPC must constantly evaluate and prioritize its actions: should it focus on crafting arrows to defend itself, gather food to keep its hunger bar from depleting, or simply run away from monsters? This scenario offers a great opportunity to observe how GOAP and FSM manage these overlapping demands and adapt to the constantly changing conditions.

## 1.1 Feature plan

This section highlights the key features that the project will prioritize to ensure successful implementation. The mandatory features form the foundation of the project, defining its scope and core functionality. Completing these features is crucial to achieving meaningful results and meeting the main objectives.

In contrast, the optional features are intended to enhance the project's overall quality. While they can improve the implementation and potentially lead to better outcomes, they are not essential to fulfilling the project's primary goals. If time or resources are constrained, the optional features may be set aside to focus on completing the mandatory elements.

### 1.1.1 Mandatory Features

- A survival world, including monsters and resources.
- An NPC using Goal-Oriented Action Planning(GOAP).
- An NPC using a simpler AI method, such as a Finite State Machine (FSM).

- The GOAP-based NPC must have at least three goals, each with at least one associated action.

### 1.1.2 Optional Features

- The GOAP-based NPC should survive longer than the FSM-based NPC in the survival world.
- The GOAP-based NPC should have at least five goals, each with five associated actions.
- The graphics should be visually satisfying and provide a good visual experience.

# 2 Background information

This section explains the background needed to understand the problem and the methods used to address it. It explores the concepts of Goal-Oriented Action Planning (GOAP) and Finite State Machines (FSM), focusing on why they matter in game AI development. Additionally, it situates the project within a broader scope, demonstrating its relevance and potential impact.

## 2.1 Goal-Oriented Action Planning (GOAP)

GOAP is an AI technique designed to handle complex decision-making by breaking down goals into achievable actions. Unlike simpler methods, GOAP enables an NPC to dynamically select and prioritize actions based on the current state of the environment and its objectives. This approach makes it especially effective in situations requiring adaptability and strategic planning.

Implementing GOAP involves several key components:

- **Goals:** These define the desired outcomes the NPC aims to achieve, such as "Stay alive" or "Kill enemy."
- **Actions:** Each action represents a discrete task the NPC can perform, such as "Gather wood," "Craft arrows," or "Attack monster." Actions have preconditions (states that must be true for the action to be performed) and effects (the changes the action causes in the world state).
- **Planner:** The core of GOAP, the planner searches through available actions and determines the optimal sequence to achieve a given goal. This is often done using algorithms like A* to evaluate costs and priorities.
- **World State:** A representation of the current environment and the NPC's status. The planner uses the world state to decide which actions are viable and how they contribute to the goals.
- **Dynamic Replanning:** If an action becomes invalid (e.g., a resource is no longer available), the planner can reevaluate and create a new plan in real-time.

According to the dissertation by Edmund Long [2], GOAP provides significant advantages in terms of scalability, flexibility, and behavior diversity in NPCs. The ability to decouple goals and actions makes GOAP highly reusable and modular, which is crucial for modern game development.

In this project, GOAP is used to help an NPC evaluate changing priorities, such as deciding whether to gather resources for crafting arrows, collect food to satisfy hunger, or avoid a nearby monster. The GOAP planner carefully analyzes the available actions and identifies the best sequence to achieve the NPC's current goals. By adjusting to changes in the environment, like resource availability or new threats, the system ensures the NPC acts intelligently and adapts flexibly in real-time.

## 2.2 Finite State Machines (FSM)

Finite State Machines (FSM) are one of the most traditional and widely used AI methods in game development. They work by organizing an NPC's behavior into predefined states, such as "gather resources," "fight monster," or "run away," and specifying transitions between these states based on predefined conditions. For example, an NPC might switch from "idle" to "attack" when an enemy spawns. This simplicity makes FSMs relatively easy to implement and debug, but they can struggle with flexibility and adaptability in complex environments.

As highlighted by Devang Jagdale in the paper [1], FSMs are particularly effective in structured scenarios where predictable and manageable behaviors are desired. The system ensures that an NPC remains in one state at a time, transitioning only when specific conditions are met. Despite their limitations, FSMs are still a popular choice in modern games due to their straightforward design and ability to deliver consistent results.

## 2.3 Broader Scope and Relevance

This project is a effort to improve NPC behavior in games. By comparing GOAP and FSM, it aims to highlight the strengths and weaknesses of each method, providing useful insights for game AI development. A better understanding of these techniques can help developers design more engaging and lifelike NPCs, leading to better player experiences. Beyond gaming, the findings from this project could also have applications in areas like robotics, simulation, and other fields that rely on adaptive AI behavior.

# 3 About your implementation

This section describes how the project was implmented.

## 3.1 Tools Used

For this project, I chose Unity as the main development platform. Having prior experience with Unity, I found it to be a versatile and reliable game engine, making it a perfect fit for this type of work. It made it easy to quickly set up a survival game world with features like a hunger meter, resource spawns, enemies, and crafting systems. This efficiency made Unity an excellent choice for experimenting with and comparing various AI techniques, as it allowed for fast iterations and testing. The decision to implement this project as a small 2D RPG also influenced the selection of Unity, as its 2D development tools made it easier to create the game world.

I also used GitHub for version control and to work across multiple computers. The full project is available on GitHub for readers to explore in detail.

## 3.2 Program Structure

The program is structured around several key components that make up the survival game world:

- **NPC:** The central character controlled by the AI techniques, equipped with health, a hunger meter, and the ability to gather resources, craft arrows, and fight enemies.
- **Hunger Meter:** A mechanic that requires the NPC to gather food periodically. The hunger meter ticks down over time, and if it reaches zero, the NPC begins to take damage. The rate of hunger depletion and the damage taken are adjustable.
- **Resources (Food, Stone, and Wood):** These are necessary for survival and crafting. Food replenishes the hunger meter, while stone and wood are required to craft arrows. The NPC must decide what to gather based on its current needs.
- **Enemies:** Spawned entities that target the NPC. If an enemy gets close and the NPC has arrows, it will attack. Otherwise, the NPC will take damage.
- **Resource Spawner and Enemy Spawner:** These spawn resources and enemies randomly across the map at adjustable intervals, allowing for dynamic changes to the game's difficulty.
- **HUD:** Displays critical information such as the NPC's health, hunger level, gathered resources, current action, enemies killed, and time survived.

These components interact to create a dynamic and challenging environment for testing and comparing the AI techniques.

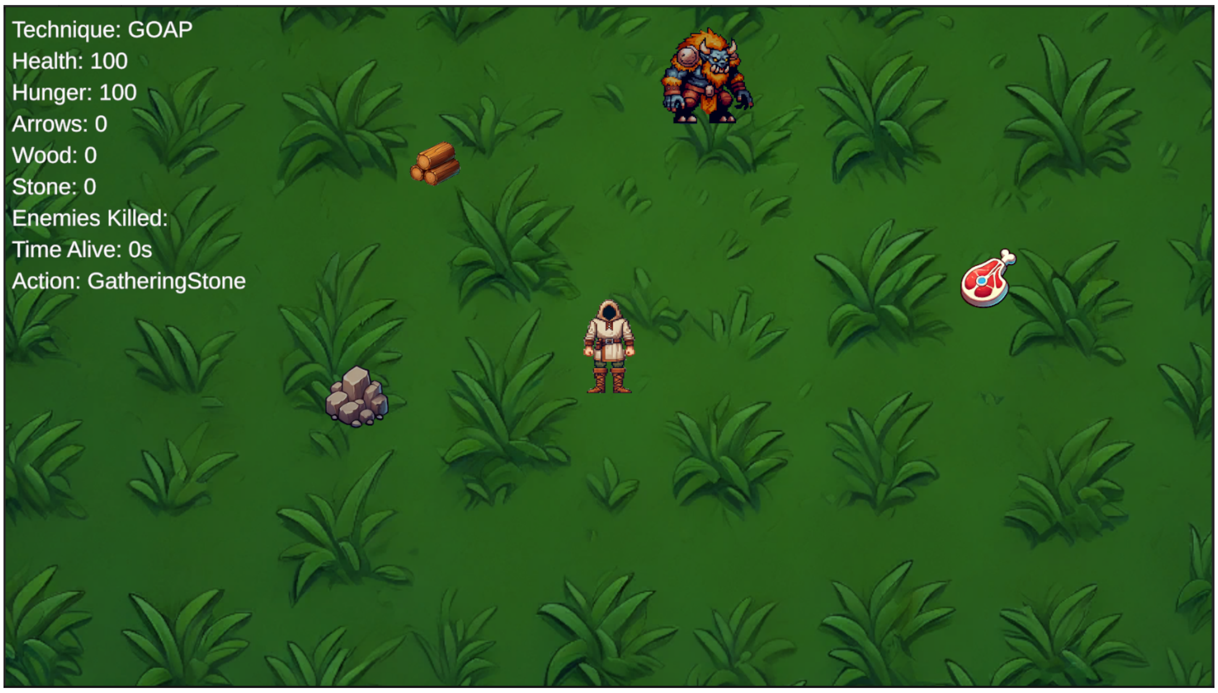
Technique: GOAP
Health: 100
Hunger: 100
Arrows: 0
Wood: 0
Stone: 0
Enemies Killed:
Time Alive: 0s
Action: GatheringStone

Figure 1: A snapshot of the 2D RPG game world showing the NPC, resources, enemies, and HUD.

### 3.2.1 GOAP

This section provides an overview of the GOAP implementation in the project. At its core, the system relies on the Planner script, which serves as the head of the algorithm. This script is responsible for the planning process, determining what action the NPC should execute based on its current goals and the game state. The planner maintains a list of all available goals and the actions that can achieve them. By dynamically evaluating the priority of goals depending on the game state, the planner ensures that the NPC always works towards the most critical objectives. For example, if the NPC's hunger level drops below 50, the ”increase hunger” goal is assigned a high priority value of 10. But, if the hunger level is above 80, the priority value is just 1.

The dynamic nature of the planner allows the NPC to switch actions as the game state changes. For example, if the NPC is gathering food and has a priority 1 and a stone resource with priority 7 appears, the planner will redirect the NPC to gather stone instead.

The Goal script supports the planner by defining the NPC's goals. Each goal includes details like its name, priority, and relevance. The relevance parameter is particularly useful for ignoring goals that don't apply to the current situation. For example, if the NPC has no arrows, the “kill enemies” goal is automatically marked as irrelevant.

Actions are defined using the Action script, a parent class for all NPC actions. This parent class provides essential functionalities, such as determining whether the action can execute (canExecute), checking completion (isDone), and resetting the action (resetAction). Specific actions, like gathering food or crafting arrows, inherit from this base class and implement their unique logic. This setup keeps the system consistent while allowing flexibility for different NPC behaviors.

### 3.2.2 FSM

This section provides an overview of the FSM implementation used in the project. To simplify development and enable quicker implementation, I opted for a task-based FSM. This approach organizes the NPC's behaviors into discrete tasks, each represented as a state in the FSM.

At the core of the implementation is the FSMController script, which manages the NPC's tasks using a state-based structure. The script cycles through tasks, such as gathering wood, gathering stone, eating, and killing enemies. Each task is tackled one at a time, and the NPC moves on to the next task when the current one is either finished or can't be completed.. For example, if no resources are available for a specific task, the NPC skips to the next task in the sequence.

Transitions between states are hardcoded, making the FSM straightforward but less adaptable than GOAP. For example, the NPC first searches for the closest resource related to the current task. If a resource is found, the NPC moves toward it. Once the resource is reached, the task is marked as complete, and the state transitions to the next task.

# 4 Discussion

This section goes over the testing results and highlights the key findings. It also discusses some of the challenges encountered during the project and the limitations of the implemented solutions.

## 4.1 Test Results

The testing was done by tweaking one specific parameter at a time while keeping everything else the same. This way, it was easier to see how the AI techniques compared against each other without other factors messing things up.

The parameter I focused on was the enemy spawn rate. This seemed like the most important one to test since it really impacts how the NPC plans its actions. When there are more enemies, the NPC has to figure out whether it's better to craft arrows, fight enemies, or gather food to stay alive. By changing the spawn rate step by step, I could see how each AI technique handled the extra pressure and adjusted to the tougher conditions.

For each iteration of testing, I ran three tests for each AI technique. I then took the average of the time alive and the number of enemies killed for both techniques. This process was repeated for all iterations to ensure consistent results.

| Test | Technique | Enemy Spawn Rate | Enemies Killed | Time Alive |
|---|---|---|---|---|
| Test 1 | GOAP | 6s | 54 | 334s |
| Test 1 | FSM | 6s | 41 | 260s |
| Test 2 | GOAP | 4s | 63 | 259s |
| Test 2 | FSM | 4s | 53 | 225s |
| Test 3 | GOAP | 3s | 51 | 202s |
| Test 3 | FSM | 3s | 42 | 178s |
| Test 4 | GOAP | 2s | 34 | 194s |
| Test 4 | FSM | 2s | 15 | 101s |

Table 1: Comparison of GOAP and FSM performance under varying enemy spawn rates. The enemy spawn rate refers to the interval (in seconds) between the spawning of new enemies.

## 4.2 Result discussion

The results of the tests clearly show that GOAP outperforms FSM in every scenario, which is an excellent outcome and aligns with the goals of this project. Across all tests, GOAP demonstrated better decision-making and resource management compared to FSM, maintaining longer survival times and defeating more enemies.

Interestingly, the difference in performance between the two techniques remained relatively consistent as the enemy spawn rate increased. However, when the enemy spawn interval was reduced to just 2 seconds, as in Test 4, the FSM performed significantly worse than GOAP. This big difference highlights the FSM's inability to adapt to high-pressure situations where proper prioritization is critical.

In Test 4, the GOAP-based NPC managed to prioritize crafting arrows when supplies were low, allowing it to continue fighting enemies effectively. On the other hand, the FSM-based NPC struggled to adjust its priorities and continued to focus on actions like attacking enemies, even when it was running out of arrows. This failure to prioritize led to the FSM surviving only 101 seconds, compared to GOAP's 194 seconds.

These results suggest that in more complex game states, where the environment changes rapidly, GOAP is superior to FSM. The ability to dynamically adapt and re-prioritize actions gives GOAP a significant advantage in handling challenging scenarios.

## 4.3 Interesting Problems

This section highlights some of the problems and limitations encountered during the project.

### 4.3.1 Simple worldstate

While working on this project, I ran into some limitations because the world state was kept pretty simple. To keep things manageable with the time I had, I stuck to straightforward goals, with each goal usually tied to a single action. This approach worked well enough for the project, but it didn't fully capture the complexity needed to show the clear differences between GOAP and FSM.

A more complex setup would probably have shown those differences more clearly. For example, the goal "make arrows" could involve several more steps like: building a workbench, gathering feathers from chickens, collecting wood and stone. These mini-goals within a larger goal would lead to a more layered decision-making process, where GOAP's ability to handle interconnected actions would really stand out compared to FSM.

Even with these limitations, the results still showed that GOAP outperformed FSM, even in a relatively simple environment. This suggests that GOAP's advantages in adaptability and prioritization would become even more obvious in a more complex world.

### 4.3.2 Time limitations

One of the biggest challenges in this project was dealing with the time constraints, which meant keeping the scope fairly simple. I had to focus on creating goals that were different enough to produce meaningful results but still manageable within the limited timeframe. It was important to pick goals that would force the AI to prioritize in different ways and produce clear differences depending on the technique used.

Another key consideration was making sure the game state was challenging enough to make these goals matter. A more complex decision-making process should ideally lead to better performance. For example, the hunger meter turned out to be a great addition to the mechanics. Since it constantly ticked down in the background, the NPC had to gather food when the hunger level got too low. Ignoring this need usually ended badly, as the NPC would take damage and wouldn't survive as long.

I believe I managed the planning phase well given the circumstances. The results were meaningful, as they clearly demonstrated the advantages of GOAP, aligning with the supporting theory. That said, I may have invested more time into this project than the university credits would suggest.

## 5 Conclusions

This project clearly demonstrated that GOAP outperforms FSM in both simple and more complex scenarios. While the performance gap between the two wasn't huge in simpler situations, the difference became much more noticeable as the game state grew more challenging. When enemy spawns were more frequent, and prioritization became critical, GOAP's ability to adapt and re-prioritize actions gave it a clear edge. This was especially evident in Test 4, where FSM struggled to keep up and ended up surviving for much less time compared to GOAP.

Working on this project gave me a deeper understanding of how these AI techniques function and the kinds of situations where they perform best. Implementing both GOAP and FSM helped me see their strengths and weaknesses firsthand, and I now have a much better idea of how to use them effectively in game development. This project was also a great opportunity to improve my Unity skills, especially in building dynamic game states and testing AI behaviors. Overall I am happy with how the project turned out, it achieved its goals and provided some really valuable insights into game AI development.

## References

[1] Devang Jagdale. “Finite State Machine in Game Development”. In: *International Journal of Advanced Research in Science, Communication and Technology* 10.1 (2021), pp. 384–390. DOI: `10.48175/IJARSCT-2062`.

[2] Edmund Long. “Enhanced NPC Behaviour using Goal Oriented Action Planning”. PhD thesis. University of Abertay Dundee, 2007.