Advanced Tech Report on AI Squad Behvaiour

Alessandro Bufalino
19017120

University of the West of England

May 22, 2023

In this report, we explore an approach to create autonomous artificial intelligence (AI) agents in the Unity game engine for use in a first-person shooter (FPS) game. Finite state machines are used dictate the behaviour of the agent based on its current context.

1 Introduction

In this report, we discuss the growing importance of developing responsive and dynamic AI agents to create an immersive player experience in video games. We demonstrate the effectiveness of using finite-state machines for AI behaviour within a Unity (Technologies, 2023)based project, focusing on user-controlled squads capable of performing actions such as taking cover and defending positions to promote strategic gameplay. The report details the design and implementation of this project, showcasing its potential in enhancing the gaming experience.

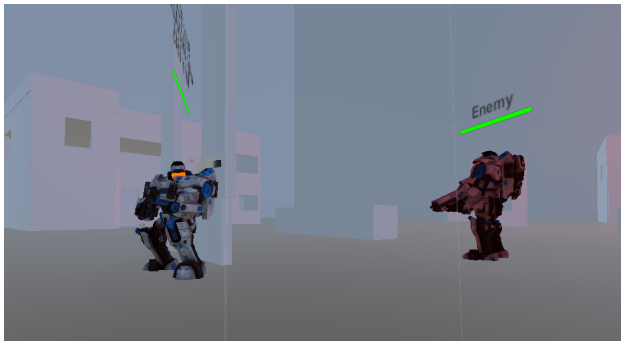

Figure 1: *Showing a friendly NPC taking cover behind a wall*

2 Related Work

As games continue to grow in size and complexity, there is an increasing need for systems of NPCs or AI agents to fill the gaps within the game world. Several AI methods that are used in games, include Finite-State Machines, Behaviour Trees, or providing the agent with private statistics about the player to aid the decision-making process. Although these examples can be integrated into various genres, they all share a common trait: deterministic AI implementation. Most games lean towards deterministic behaviours because they are "predictable, fast, and easy to implement, understand, test, and debug" (Bourg and Seemann, 2004). While nondeterministic approaches can offer a refreshing and unpredictable gaming experience, they can also introduce issues and bugs.

One example of a game that partially implements its AI using a nondeterministic approach is Hello Neighbour (Dynamic-Pixels, 2017). In this game, the AI adjusts its patrol routes and sets traps based on the player's previous actions and locations. While this concept is designed to keep the player engaged by encouraging them to find new strategies, it often results in the game becoming nearly unplayable due to conflicts between puzzle-solving and AI patrolling. As reported by Kunzelman (2017), "I had solved the puzzle, and the AI kept getting in my way, forcing me to restart". This issue could have been mitigated by employing a fully deterministic AI approach, which allows for simple checks that prevent the AI from interfering with the player's progress.

On the other hand, AAA game development aims to simulate complex environments with engaging, realistic, and believable NPC behaviour. Traditional scripted behaviours, like Finite State Machines and Behaviour

Trees, might not meet modern consumers' expectations for realism and unpredictability (Llargues Asensio et al., 2014). This emphasizes the need for advanced AI systems to enhance player immersion. Techniques like reinforcement learning, procedural content generation, and neural networks are explored in modern games. Developers should consider these methods or complex FSMs to achieve desired NPC behaviour realism and unpredictability.

For this project, a deterministic method was chosen to ensure that the player can trust the agent to follow orders consistently. The Finite State Machine method was selected for its ease of implementation and scalability.

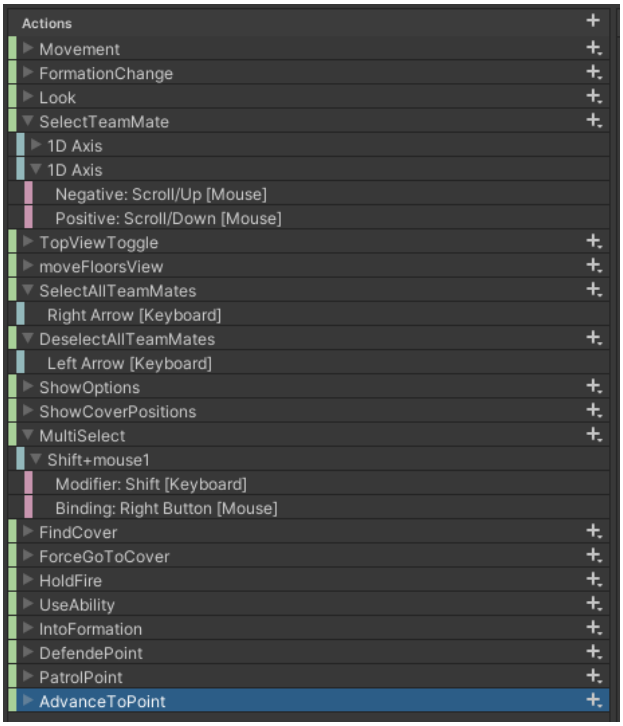

Figure 2: *Unity input system, showing the setup for each of the agent selection methods.*

3 Method

3.1 User Interface

The goal of the UI implementation was to ensure that it did not interfere with the gameplay, meaning it would have minimal impact on the player's objectives at any given point. For example, if the game were to use a radial menu, it would not only obstruct the player's view of the target but also require the player to stop firing to select an option, which could result in frustrating scenarios (Nelson, 2021).

In this implementation, the player can bring up a side action menu (Figure 3) when at least one of the AI agents is selected. The agents can be selected using the mouse scroll wheel (Figure 4) or by clicking on them with the mouse. The implementation of this feature is made possible by using the new input system (Figure 2) introduced in Unity, which simplifies the addition of modifiers (such as left Shift + right mouse button) and keeps the code compartmentalized thanks to its individual function calls.

To interact with the AI, the player will need to use the number keys to decide on the action that the selected AI should take while the action menu is open. Some choices in the menu are dynamic, depending on the state of the currently selected agents.

- The "Hold Fire" action is a toggle, which means that when selecting multiple different AI agents, they might have different values simultaneously. In such cases, the majority value will be shown in the Action Menu.
- The ability button will disappear when more than one AI agent is selected, as different abilities necessitate different scenarios, and activating an ability at the wrong time could lead to unexpected outcomes in terms of gameplay.

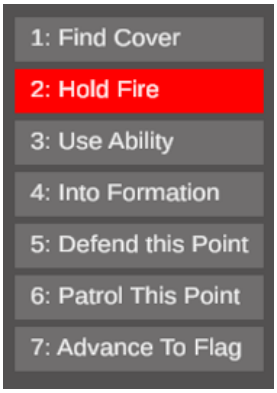

Figure 3: *Action Menu being deployed, and showing the currently selected agent to be holding fire and not shooting.*

Another addition to the UI is the Status Menu, as shown in Figure 4. This menu was implemented not only to display the currently selected agent, but also to provide information about their health and distance, giving the player context about the selected agent's situation. The menu also includes an icon that changes based on the agent's current state.

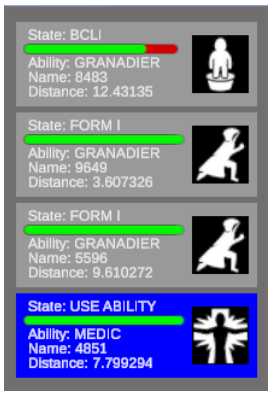

Figure 4: *Status Menu showing the 4th agent being selected and the different icons indicating different types of states.*

There are times when the Status Menu is insufficient, and additional context is necessary. In such cases, a quick pop-up has been added to alert the player about certain events involving the agents. This system works with a queue, ensuring that even if multiple messages come in, they are displayed neatly. Messages can range from the death of an agent to the resurrection of one, and for quicker recognizability, the colour changes depending on the message, as shown in Figure 5.

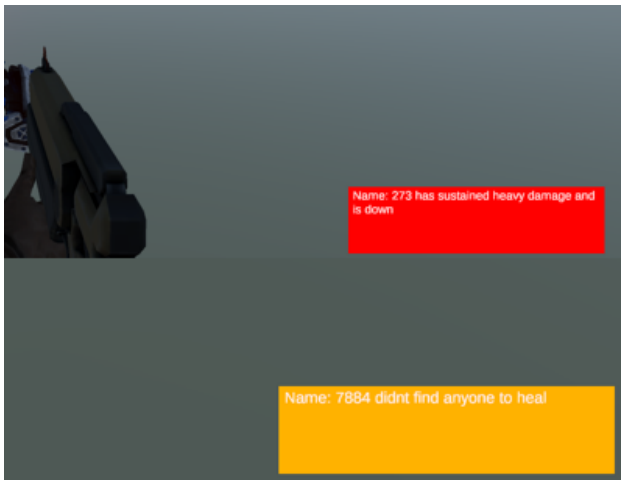

Figure 5: *Pop-up showing a message on the current state of the friendly agents of the player.*

3.2 Squad Behaviour

3.2.1 Agent's States

Each squad member has their own StateManager script, which manages the state of the object and the variables required for interaction with the world. This script also holds references to each of the possible states the agent can be in, with each state having its own specific behaviours and actions.

```
1    private agentBaseState[]
     statesList = new agentBaseState
     [17]
2     {
3     new TmDead(),
4     new TmFindCover(),
5     new TmGoToCover(),
6     new TmIdleWaiting(),
7     new TmbehindCoverFrontFight(),
8     new TmbehindCoverFrontIdle(),
9     new TmInFormationFight(),
10    new TmInFormationIdle(),
11    new TmPatrollingAroundPoint(),
12    new TmUseAbility(),
13    new TmbehindCoverLateralActive()
     ,
14    new TmbehindCoverLateralIdle(),
15    new TmGoToForcedCover(),
16    new TmAdvance(),
17    new TmDefendPoint(),
18    new TmMedic(),
19    new TmGranedier()
20    };
```

Each state inherits from an abstract class, this is done as the base class won't be instantiated it will only be used to implement the different versions of the states. All states contain 3 main functions.

```
1
2 public class TmbehindCoverFrontFight
      : agentBaseState
3 {
4   public override void OnUpdate(
     agentStateManager agent) {} //
     called every update
5   public override void OnExit(
     agentStateManager agent){}  //
     called when the state is exited
6   public override void EnterState(
     agentStateManager agent) {}  //
     called  when the state is first
     entered
7 }
```

On top of the 3 main functions, in the BaseState class, there are Utility functions, these are functions that are used from multiple states so instead of clogging up each BaseState variation, they can easily be called from the override class script. One of the main examples for this implementation was the need to calculate the Euclidean distance ($\sqrt{(x1-x2)^2+(y1-y2)^2}=d$) of the agent object to the closest enemy to set the right alert value.

```
1     public bool RayCasterPoint()
2     public bool RayCasterPlayer()
```

```
3     public bool RayCasterEnemyList()
4     public List<GameObject>
      CheckForEnemiesAround()
5     public bool ReachedDestination()
6     public void LookAt()
7     public void ShootAt()
8     public Quaternion GenRandomRot()
9     public void GoToPoint()
10    public float TimerCheck()
11    public void SmoothRotateTowards
      ()
```

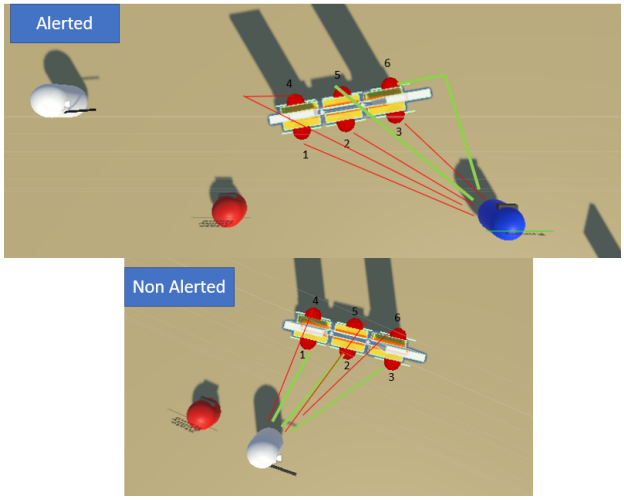

Figure 6: *Simple Overview example using the Ucover of the different possible positions in cover, valid points of cover shown in green. Blue – Enemy // White – Player // Red - agent.*

3.2.2 Finding Cover

Finding the right cover for the agent to take was a key part for the squad overall to work as efficiently as possible. There are 3 ways that the agent can decide its cover position (Figure 6):

- Force Cover: This allows the player to manually set the cover for each agent. After selecting an agent, the player toggles Cover mode on, which enables them to see all available cover spots and direct the agent where to go.
- Find Cover (Alerted): When an agent is instructed to find cover, it performs an Enemy check. If more than one enemy is detected, the agent searches for cover relative to the current enemy's position. Each potential cover position within range conducts a RayCast from its location towards each enemy. If the RayCast is blocked before reaching the enemy, the cover position is considered valid. The second check determines if the agent can shoot at the enemy from outside the cover. If both RayCasts are successful, the agent is directed to a valid position behind the cover.
- Find Cover (Not Alerted): This process is similar to the one above, but the RayCast is directed towards the player instead of the enemies. This results in cover positions that surround the player.

There are two types of cover positions available:

- Lateral Cover: The agent to shoot at the enemy has to leave cover from the side, this only happens in the UCover.
- Front Cover: The agent has to crouch and stand up to shoot the Enemy. This is found in both the UCover and the Basic Cover.

When the "Find Cover" state is activated, the agent attempts to spread out as much as possible. In the current project, each coverObject has six cover positions. Once a position is detected, the script checks if there is already an agent stationed in any of the other positions of that coverObject. If so, it attempts to find a new coverObject. If none are available, it iterates again without the spreading out feature. If no cover is found in any case, the agent returns to the player in the formation state.

4 Evaluation

4.1 Finite-State Machine Implementation

The finite-state machine proved to be a suitable approach for implementing AI behaviour. It allowed for relatively simple design and implementation while providing a solid foundation for creating responsive and adaptive AI agents.

4.2 Scalability and Adaptability

In terms of scalability, the finite-state machine (FSM) offers the ability to add states seamlessly, with the primary challenge being the tracking and management of each state transition. As the number of states and

transitions increases, maintaining a clear and efficient FSM can become more complex and time-consuming.

4.3 Future Work

In the future, work could focus on implementing more intricate decision capabilities such as behaviour trees, this would give the AI more autonomous control over its own state. Furthermore, improved decision-making can help mitigate any potential issues of overly deterministic behaviours.

5 Conclusion

In conclusion, this report presents a method for implementing a responsive AI system in Unity using finite state machines. The system empowers players to command troops while trusting agents to make decisions autonomously. Combined with a tailored UI system, this approach can be applied to various games to improve the player experience.

Bibliography

Bourg, David M and Glenn Seemann (2004). *AI for game developers*. "O'Reilly Media, Inc."

Dynamic-Pixels (2017). *Hello Neighbor*. URL: https://store.steampowered.com/app/521890/Hello_Neighbor/.

Kunzelman, Cameron (2017). *The Awkward Hello Neighbor Is in Conflict with Itself*. URL: https://www.pastemagazine.com/games/hello-neighbor/hello-neighbor-review/.

Llargues Asensio, Joan Marc et al. (2014). "Artificial Intelligence approaches for the generation and assessment of believable human-like behaviour in virtual characters". In: *Expert Systems with Applications* 41.16, pp. 7281–7290. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2014.05.004. URL: https://www.sciencedirect.com/science/article/pii/S0957417414002759.

Nelson, William (2021). *Best practices for designing an effective user interface*. URL: https://www.gamesindustry.biz/best-practices-for-designing-an-effective-video-game-ui.

Technologies, Unity (2023). *Unity Game Engine*. URL: https://unity.com/.