NPC-Behavior-in-Video-Games: Non-player Character (NPC) Behavior in Video Games

Mason Stott
University of Tennessee
Knoxville, TN, USA
mstott3@vols.utk.edu

Jihun Kim
University of Tennessee
Knoxville, TN, USA
jkim172@vols.utk.edu

Arthur Jur
University of Tennessee
Knoxville, TN, USA
gjur1@vols.utk.edu

Julian Halloy
University of Tennessee
Knoxville, TN, USA
jhalloy@vols.utk.edu

***Abstract*—INTRODUCTION: Non-Player Characters (NPCs) are integral parts of many video games. There are growing expectations of players for intelligent and responsive NPCs, while game developers are seeking effective methods to implement complex NPC behavior without incurring high development costs.**

OBJECTIVES: This paper aims to evaluate and compare several approaches to implementing NPC behaviors in video games. The goal is to provide game developers with practical guidance on choosing the method that best suits their game. METHODS: We implement and observe four AI techniques: finite state machines (FSMs), behavior trees (BTs), goal-oriented action planning (GOAPs), and utility AI. Each technique is implemented in the Unity game engine and evaluated based on ease of use, implementation effort, and the complexity of the behaviors it can generate.

RESULTS: FSMs are preferred when the action space is relatively small and transitions are easy to manage. BT provides a better structure for managing complex or numerous transitions. GOAP is best for complex but predictable behavior, especially when the action space is large. Finally, utility AI is ideal when the action space is large and requires unpredictable, emergent behavior.

CONCLUSIONS: There is no single approach that fits all situations. The best choice depends on the game's requirements, expected behavioral complexity, and development resources. This study provides a comparative guide to help developers make informed decisions when designing NPC behaviors.

I. Introduction

Non-player characters (NPCs) play a crucial role in creating believable and dynamic experiences in video games. Their behavior can have a tremendous impact on a player's sense of immersion and engagement, making the design of these systems a critical component of the game development process. The motivation for our project arises from the growing complexity of modern games and the growing expectations of players for intelligent and responsive NPCs that react believably to stimuli during gameplay. Achieving such nuanced behavior, however, can be a nebulous undertaking, as there are so many approaches to consider when designing these systems, and therein lies the crux of our project.

We aim to supply a one-stop resource for determining the most appropriate design approach for any given game project, weighing the strengths and weaknesses of various well-known standards for these systems. The approaches we investigate are as follows:

- Finite state machines (FSMs)
- Behavior trees (BTs)
- Goal Oriented Action Planning (GOAP)
- Utility AI

We evaluate these methods in terms of their implementation overhead, ease of use on the design side, and the potential depth of behavior they offer relative to expended effort.

While this project is not entirely novel, as comparative studies on such frameworks do exist, it is unique in terms of the scope of the project and the uniformity of the evaluation context. We seek to fill the gap in existing research by providing a comprehensive evaluation framework that considers the complexity of implementation, ease of use for designers, and depth of behavior produced across all of these approaches. In the end, we hope to have a valuable resource for developers in the design process of their next game project to make an informed decision on their approach to NPC behavior design.

II. Related Work

Since NPC behavior is important to game developers and players, there has been quite a bit of research surrounding the different strategies employed. Saia et al. explore the uses of finite state machines (FSMs) for mimicking human strategies in fighting games [4]. The authors point out that since the state transitions in a FSM are fixed, the behavior becomes predictable and players can exploit it. The authors highlight that traditional FSMs are static and rule-based. Another method commonly used is the behavior tree. Robertson and Watson explore the use of behavior trees in strategy games [3]. The authors describe behavior trees as "hierarchical goal-oriented structures." They are also "static structures used to store and execute plans." The authors highlight that the behavior tree they created was able to summarize a lot of information in a concise manner.

Some more advanced strategies include Goal-Oriented Action Planning (GOAP), introduced by Jeff Orkin [2]. Orkin describes how GOAP was used in the development of the game F.E.A.R. He highlights the fact that the planning system reduced the burden on the developer to specify the agent's behavior. In GOAP, the planning system will search for the actions required to achieved certain goals. Another strategy is Utility AI, which is explored by Swiechowski [6] as well as a novel extension of it (Fuzzy Utility AI). This strategy models the utility of certain behaviors by evaluating utility curves.

Utility curves define the relation ship between a consideration and the utility for a concept (e.g., attacking). A consideration is a measurable subset of the game state. The model will select the action that yields the largest utility. This allows for more flexibility to adapt to specific game states that may not always be accounted for in other methods, such as finite state machines.

A. *Research Gaps*

While there has been research done on different algorithms, there still is not an abundance of research comparing these algorithms in similar game environments. Algorithms vary in their ease of implementation, adaptability, and transparency, which can even vary based on the scenario. It would be helpful to know which algorithms are best in which scenarios. Additionally, existing comparisons of different algorithms can quickly become obsolete as the game development landscape is ever-changing.

III. Research Value

The primary users of the results of this work would be video game developers. Video game developers have a goal of creating interesting and fun-to-play games. These developers want easier methods of development for more rapid prototyping. Additionally, they want NPCs that behave in a controllable but unique way to provide depth and interactivity to the game.

There is already research on NPC control in game development. However, we hope to provide a novel comparison between several modern approaches to help game developers evaluate which option is best for them.

IV. Research Questions

Motivated by these considerations, we formulate the following research question:

- What are the benefits and drawbacks of various video game non-player character design patterns?
- Do we see more compelling behavior as a result of more complex approaches?
- How can one decipher which approach would be best applied to their unique project?

V. Methodology

Our approach for this project can be split into 3 distinct phases.

A. *Phase 1: Base Gameplay Framework*

In order to consistently evaluate these frameworks, we need underlying gameplay logic for these NPCs to interact in. For this, we began the project by implementing a simple, small-scope game in the Unity game engine, which we use as our testbed for these behavior approaches. Within this sandbox, we set up numerous gameplay scenarios focusing on different aspects of the NPCs' decision-making processes and compare their behaviors.

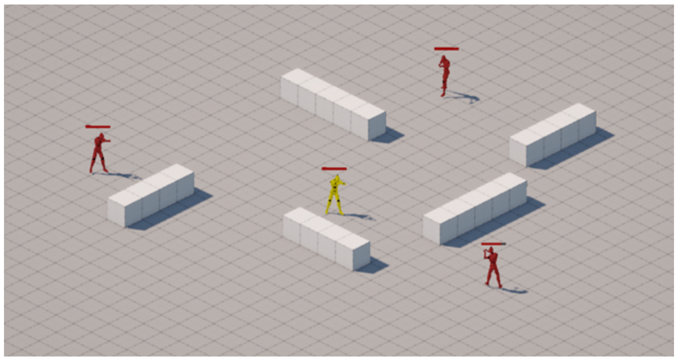
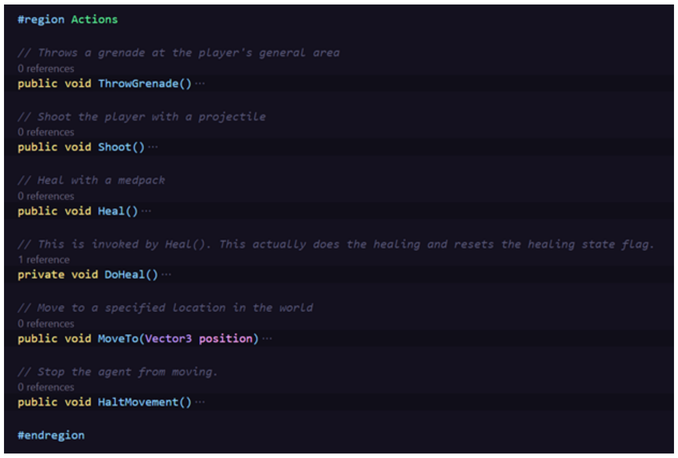


Fig. 1. Screen Capture of Base Game Implementation

Our base implementation is a simple isometric shooter game in which the player can walk around and fire a weapon. The player character aims towards the location of the mouse. The user can click to shoot and use W, A, S, and D keys to move. The actions available to the enemies were defined in the `EnemyBehavior` class. In figure 1, we can see the yellow player character along with the three red enemy characters. The characters also have a health bar above them. For a bit more complexity, there are some obstacles in the scene that the player and enemies cannot shoot through. However, grenades can be thrown over these obstacles. We define the following possible actions for the NPCs in our study:

- Shoot at the player.
- Throw a grenade towards the player.
- Move towards the player.
- Heal to restore health.
- Stop moving.



```
#region Actions

// Throws a grenade at the player's general area
0 references
public void ThrowGrenade() ···

// Shoot the player with a projectile
0 references
public void Shoot() ···

// Heal with a medpack
0 references
public void Heal() ···

// This is invoked by Heal(). This actually does the healing and resets the healing state flag.
1 reference
private void DoHeal() ···

// Move to a specified location in the world
0 references
public void MoveTo(Vector3 position) ···

// Stop the agent from moving.
0 references
public void HaltMovement() ···

#endregion
```

Fig. 2. Excerpt of `EnemyBehavior` class methods

B. *Phase 2: Behavior Implementation*

Once the base-level gameplay logic was complete, we moved on to implementing the different behavior frameworks. We worked on each of these implementations on a separate git branch. Each of these frameworks is completely removed from one another, and is implemented in such a way that they have the same objectives in the scope of gameplay so that we can effectively evaluate their performances free of any situational bias.

C. Phase 3: Comparison and Assessment

Next, we test characters controlled by each decision framework in the same gameplay scenarios and record our observations about similarities and differences between them. We examine which ones are doing better/worse and which are displaying more nuanced behavior.

VI. Results

A. Finite State Machine

A state machine is a model used in programming and system design to represent the behavior of a system as it transitions between different conditions or ”states.” Each state defines a specific situation or mode the system can be in, and transitions define how and when the system changes from one state to another based on inputs or events. In simple terms, a state machine works like a flowchart: the system starts in one state, and depending on what happens (like a change in a variable’s value), it moves to another state. This makes state machines very useful for controlling logic in systems where different actions need to happen based on changing conditions. State machines can even be stacked upon each other so that there are multiple states that can occur at one time, but in that scenario, it becomes closer to something else called a pushdown automata, which is a more complex algorithm and was not one of the algorithms we had chosen to explore in this project.

The benefits of the state machine algorithm are that it is easy to set up if the user wants simple and well-defined behavior with explicit control over the action order that occurs. Fewer states are easier to keep track of and less likely to result in coding or state transition errors, since, as the number of states increases for more complex behaviors, the number of states and their transitions can become harder to keep track of. This means that state machines are best for simple behaviors rather than complex ones.

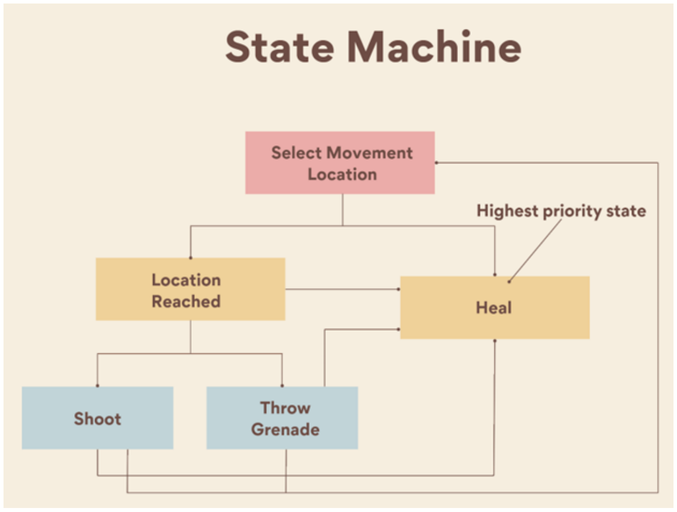


Fig. 3. Finite State Machine

B. Behavior Tree

Behavior Tree consists of four nodes: selector, sequence, action, and condition nodes. Selector nodes evaluate their child nodes in order and execute the one whose condition is met. Sequence nodes execute all their child nodes sequentially, making them suitable for tasks requiring multiple steps, such as checking whether the player is within range, verifying if an ability cooldown has elapsed, and executing an attack (e.g., shooting or throwing a grenade). Action nodes represent the enemy’s executable behaviors, such as healing, moving, shooting, or throwing grenades. Condition nodes evaluate specific criteria, such as determining whether the player is within attack range. Using the nodes in the behavior tree, the enemy decides whether to attack, move, or throw grenades based on the player’s position, their health, weapon cooldowns, and other conditions.

Behavior trees provide a modular and hierarchical framework in which each node represents a self-contained task or decision. This structure allows for flexible and scalable AI design, making it easy to add or modify behaviors without modifying a lot of code. Therefore, the implementation of this method was fairly simple. However, the transition between behaviors can feel a bit too rigid if not carefully designed. In our observations, this results in AI that feels unnatural/predictable.

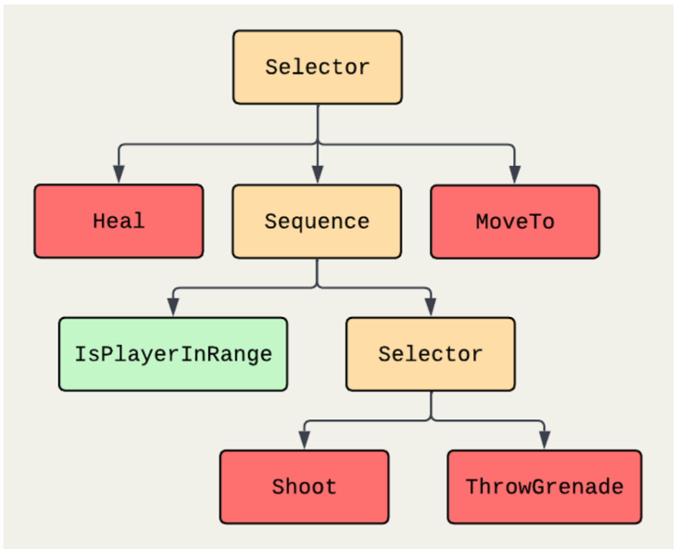


Fig. 4. Behavior Tree: the red nodes are action nodes, the green node is a condition node, and the orange nodes are composite nodes

C. Goal-Oriented Action Planning

Goal-Oriented Action Planning (GOAP) is a strategy in which the agent has some beliefs, actions, and goals. This can be seen in figure 5. The beliefs are beliefs that the agent holds about the environment. These could be taken from some sensors of the agent. In our implementation, we include an attack sensor and a chase sensor. These sensors are sphere colliders that check when the player is within a certain radius of the enemy. Other beliefs can be information about the state of the agent, such as their health.

Actions are the actions available for the agent to take. These actions have preconditions and effects. Preconditions are the conditions that must be met in order to take the action. A precondition could be something like the chase sensor detecting the player in order to be able to chase the player. The effects of an action are simply the expected effects after taking the action.

Goals are simply desired effects for the agent to achieve. These goals have priorities that will determine how the planner chooses the goal. In our implementation, we have the goal "SeekAndDestroy" which has the desired effect of attacking the player.

These three components are fed to a planner to form a plan for the agent. A plan is a series of actions the agent should take to achieve a goal. The planner will look at the desired effect of a goal (starting with the highest priority goal) and stitch together actions that achieve that effect. The final action should achieve the desired effect, and prior actions should be taken if needed to satisfy the preconditions of the final action.

In our testing, we observed that the enemies exhibited expected behavior such as chasing the player, shooting the player when they were in range and had line-of-sight, and throwing grenades at the player. However, the behavior of the agent was predictable. We believe that with the addition of more actions and goals to the agent, we would see more complex behavior. Luckily, this framework makes the addition of new actions and goals fairly simple. You just need to define the action's preconditions and effects, and then add the action's `IActionStrategy` interface, which defines what happens when the action starts and stops. However, the initial implementation of GOAP was difficult, as it required definitions of the AgentAction, AgentGoal, and AgentBelief classes. Additionally, GoapPlanner had to be implemented with an algorithm to determine the plan of action to take. Overall, the initial development cost for this approach was high, but further expansion is relatively low.

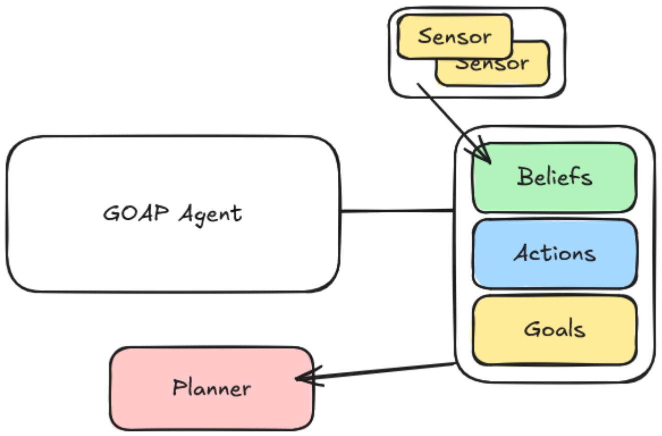


Fig. 5. Goal-Oriented Action Planning

D. Utility AI

Utility AI involves the agent assessing the usefulness (utility) of all of its available actions by assessing the relevant considerations for those actions. Considerations are defined by the designer, typically mapping some gameplay metric to a utility score based on a hand-made curve asset. Actions can have multiple considerations, and then the final utility score of the action is computed by aggregating the scores of each of its considerations in some way (most commonly multiplicatively). Every update, the AI Brain computes the utility scores of all available actions and chooses to perform the best possible (most useful) action amongst its options, as shown in the above figure.

Utility AI is a relatively simple design pattern to implement, as the complexity primarily arises from the process of defining and tuning the different considerations for the agent to assess. Since this approach lacks any explicitly defined transitions like FSMs and BTs and every consideration is updated continuously, a Utility AI agent has a more complete picture of its situation and can thus make more informed decisions. This also has the added benefit of leading to emergent behavior, as, again, there are no hard-coded sequences the agent must follow to change its current behavior. An example of emergent behavior we saw in our implementation was that the agents realized that they were safer throwing a grenade from behind cover rather than shooting at the player out in the open, and so they tended to do that more often.

The main notable drawback of this approach is that tuning the considerations and weighting them appropriately, such that the agents behave intelligently, can be time-consuming. That said, we would argue that this is more of an issue for gameplay designers than the actual programmers, and being so highly configurable could be seen as another benefit of this approach. The only other main drawback is that debugging strange behavior can be somewhat difficult, due to the aforementioned presence of emergent behavior, though this is hardly specific to our implementation.

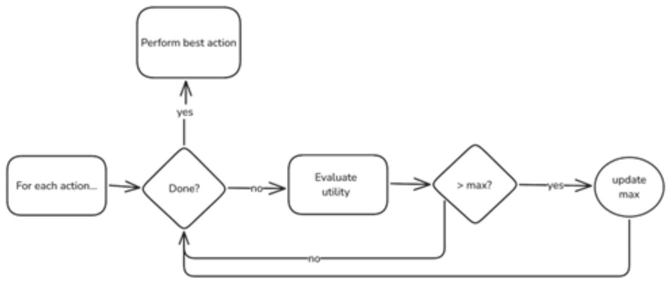


Fig. 6. Utility AI

VII. Limitations

A. Internal Validity

Feldt and Magazinius [1] find that software engineering research commonly has threats to validity. One such type of threat is an internal threat to validity. These threats are part of the study methodology and raise concerns about whether the study's conclusions are valid. An internal threat to validity in our study is that the implementation tasks were distributed to group members with varying levels of experience in game development. These differences in expertise may have influenced individuals' perceptions of the difficulty of implementation. To address this potential bias, we conducted peer reviews of each other's work and decided on the difficulty of each implementation.

B. External Validity

Feldt and Magazinius [1] also discuss external threats to validity. These threats are concerned with whether a study's results can be generalized to other situations. In our project, the development was conducted only in the Unity game engine. Therefore, the implementation difficulty and results may differ in other game engines, which may affect the generalization of the results of this study. We were not able to mitigate this threat since it would require implementing the four strategies in multiple game engines, which is beyond the time we have available to complete the project.

C. Construct Validity

Sjøberg and Bergersen [5] discuss construct threats to validity, in which unclear or vague definitions of concepts or measures are used in a study. Our evaluation of the implementation is limited to qualitative analysis. Quantitative evaluation would require extensive playtesting and data collection involving a large survey of numerous playtesters, that are beyond the scope of this study.

VIII. Future Work

Future research could explore different gameplay scenarios to provide a more comprehensive evaluation of the strengths and weaknesses of each AI approach. Additionally, comparing traditional behavior architectures to machine learning agents. Finally, exploring hybrid approaches, particularly those that incorporate principles of utility AI, can uncover opportunities to enhance the flexibility, responsiveness, and scalability of existing behavioral models.

IX. Conclusion

Choosing the right approach for modeling NPC behavior largely depends on the type of game and the design goals. Importantly, increased implementation complexity does not always lead to significantly improved or more realistic behavior. Each AI paradigm offers unique advantages and is best suited for certain situations.

- Finite State Machines (FSMs): Best when the action space is relatively small and transitions are easy to manage.
- Behavior Trees (BTs): Preferred when complex or numerous transitions make FSMs unwieldy, and allow for more explicit decision logic.
- Goal-Oriented Action Planning (GOAPs): Best for scenarios that require complex but predictable behavior, especially when the action space is large.
- Utility AI: Ideal for games that require unpredictable, emergent behavior and delicate decision making in a large action space.

References

[1] Robert Feldt and Ana Magazinius. Validity Threats in Empirical Software Engineering Research - An Initial Survey. pages 374–379, 01 2010.

[2] Jeff Orkin. Three states and a plan: the AI of FEAR. In *Game developers conference*, volume 2006, page 4. Citeseer, 2006.

[3] Glen Robertson and Ian Watson. Building behavior trees from observations in real-time strategy games. In *2015 International symposium on innovations in intelligent systems and applications (INISTA)*, pages 1–7. IEEE, 2015.

[4] Simardeep Saini, Paul Wai Hing Chung, and Christian W Dawson. Mimicking human strategies in fighting games using a data driven finite state machine. In *2011 6th IEEE Joint International Information Technology and Artificial Intelligence Conference*, volume 2, pages 389–393. IEEE, 2011.

[5] Dag I.K. Sjøberg and Gunnar R. Bergersen. Improving the Reporting of Threats to Construct Validity. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, page 205–209, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Maciej Świechowski. Fuzzy Utility AI for Handling Uncertainty in Video Game Bots Implementation. In *2024 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2024.