NPC Behavior in Games

Mason Stott, Julian Halloy, Arthur Jur, Jihun Kim

Motivation

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

Related Work

- FSM: S. Saini, P. W. H. Chung and C. W. Dawson, "Mimicking human strategies in fighting games using a Data Driven Finite State Machine," 2011 6th IEEE Joint International Information Technology and Artificial Intelligence Conference, 2011.
- Behavior Trees: G. Robertson and I. Watson, "Building behavior trees from observations in real-time strategy games," 2015 International Symposium on Innovations in Intelligent SysTems and Applications (INISTA), Madrid, Spain, 2015.
- GOAP: J. Orkin, "Three States and a Plan: The A.I. of F.E.A.R." Game Developers Conference, 2006.
- Utility AI: M. Świechowski, "Fuzzy Utility AI for Handling Uncertainty in Video Game Bots Implementation," 2024 IEEE Congress on Evolutionary Computation (CEC), Yokohama, Japan, 2024.

Research Questions

- 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 decypher which approach would be best applied to their unique project?

Gaps in Existing Research

- While there has been research done on different algorithms, there still is not an abundance of research on comparing these algorithms in similar game environments.
- Algorithms vary in their ease of implementation, adaptability, and transparency, which can even vary based on scenario.
- It would be helpful to know which algorithms are best in which scenarios.

Methods

Approaches

Finite State Machine

Create a set of states each associated with a certain action. States are changed based on the possible transitions explicitly defined by the developer. These transitions can be based on previous state and/or variable values.

Behavior Tree

Break down the AI's tasks into smaller, reusable nodes that are executed in a tree-like structure. Manage complex behaviors of entities in a modular and manageable way.

Goal Oriented Action Planning

Create a sequence actions for the agent to achieve one of the desired goal(s). Each action has preconditions and effects, which allow the agent to plan based on the current environment.

Utility AI

Update considerations about the world every tick and use them to calculate "utility score" for every available action. Perform the action with the highest evaluated utility.

Desired Behavior

We needed all of our approaches to have to same goals in order to be able to compare the results, so defining what we wanted NPCs to be able to do was crucial.

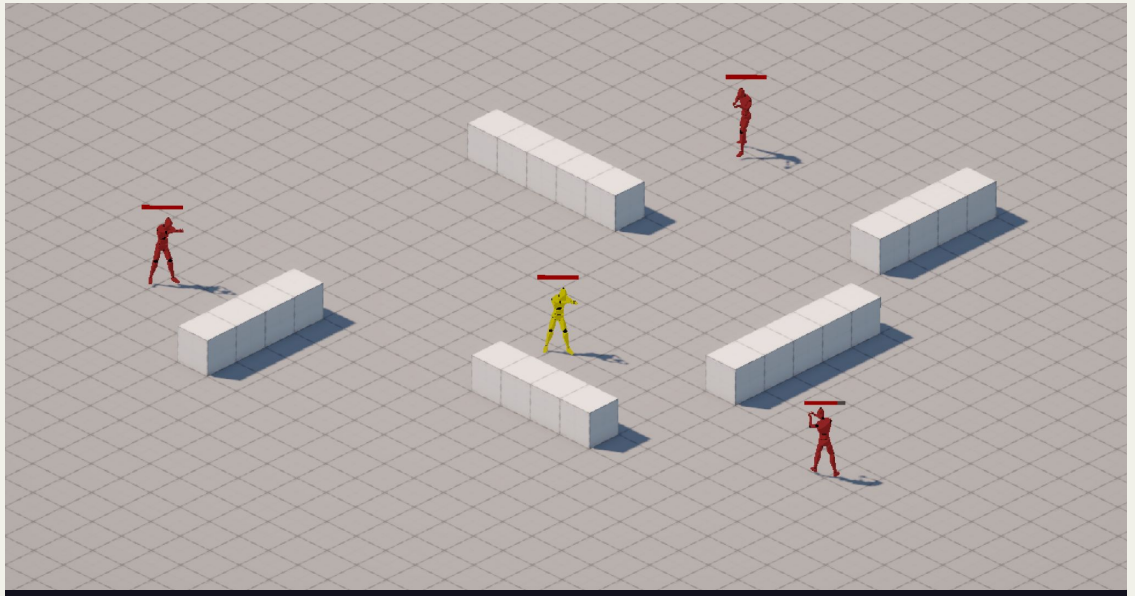
NPCs in our study are allowed the following possible actions:

- Shoot at the player.
- Throw a grenade towards the player.
- Move towards the player.
- Heal if health is low and in relative safety.

Base Implementation

For the foundation of our project, we built a simple isometric shooter game in which the player could walk around and fire a weapon.

Enemy behavior was implemented logically but they had no "brain" controlling them, which is where the approaches we're investigating come in.

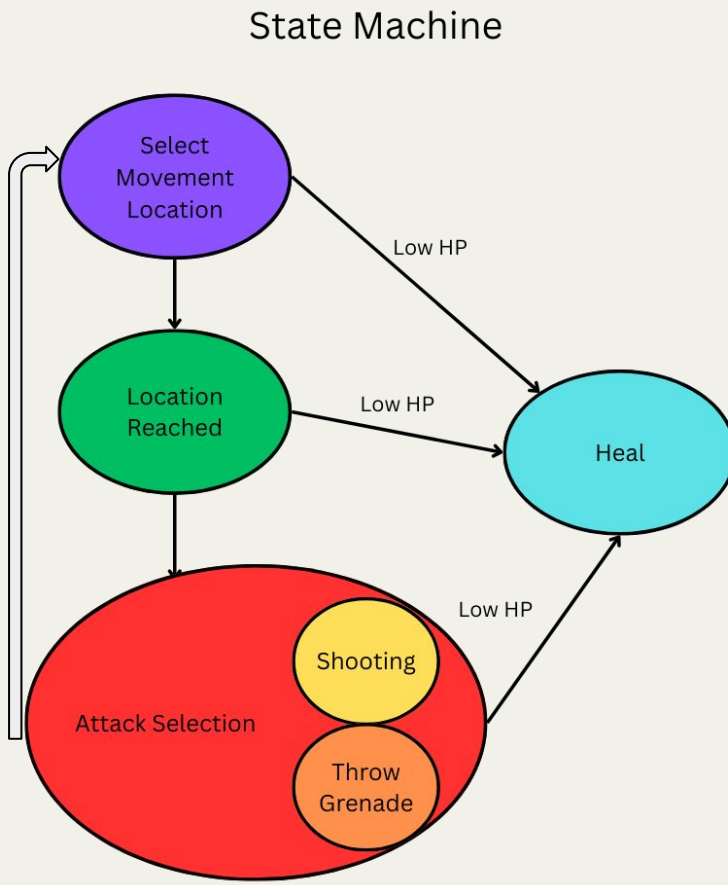


```
#region Actions

// 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
```

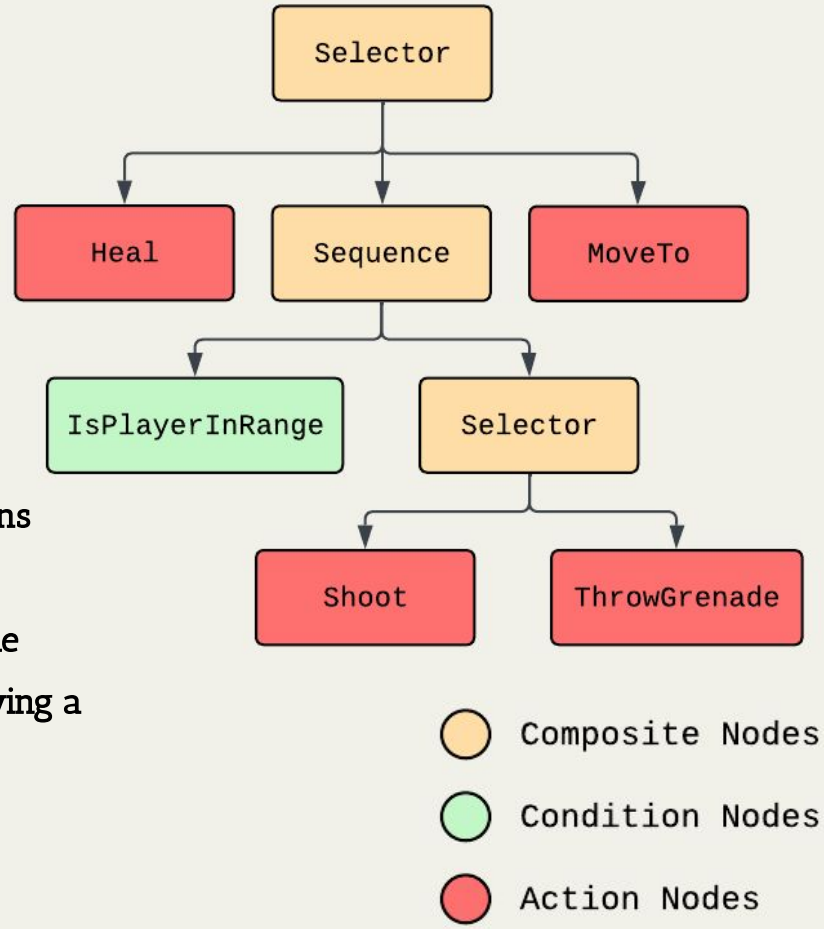
Results / Observations

Finite State Machine

- Implementation:
  - EnemyAIBrain contained several different states which covered the different actions the NPCs could take and transitioned between them based on explicitly defined transitions which could be based on the previous state or variables.
- Observed behavior:
  - Enemies have an explicitly defined routine where they select a location, move until reaching it, then stop and take and action based on environmental variables
- Benefits
  - The behavior of NPCs can be explicitly defined to occur in a specific order
- Drawbacks
  - The more complex an NPC is, the harder it can be to keep track of state transitions and decide what the "best" transitions are.
  - Complex behaviors may lead to code bloating and increased bugs

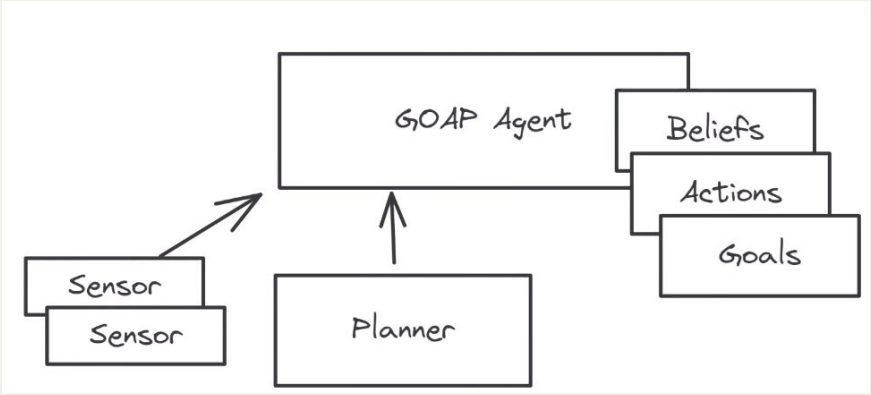


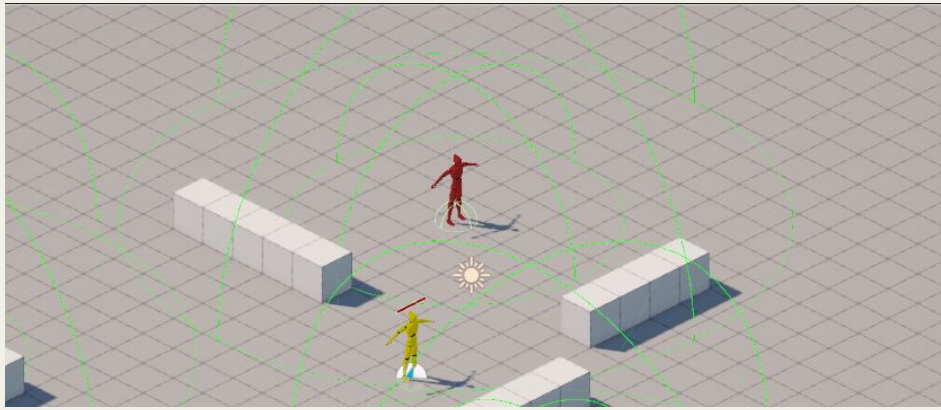
Behavior Tree



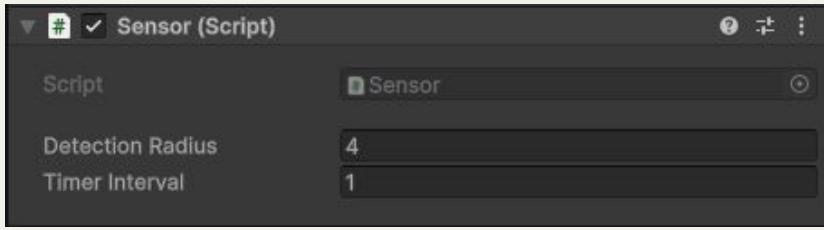
- Implementation:
  - Selector Nodes : Tries each child node (behaviors) in order based on conditions
  - Sequence Nodes : Run all child nodes sequentially.
    - ex. Check Player in range -> Check Cooldown -> Shoot/Throw Grenade
  - Action Nodes : Actions to be executed, like healing, moving, shooting, or throwing a grenade.
  - Condition Nodes : Check whether the player is in range for a shoot.
- Observed behavior:
  - The enemy decides whether to attack, move, or use grenades based on the player's position, health, cooldowns, and other conditions.
- Benefits
  - Each node represents a self-contained task or condition, so it is easy to add new behaviors without modifying a lot of code. (Flexible and Scalable)
- Drawbacks
  - The transition between behaviors can feels a bit too rigid. If the AI is not designed carefully, it could become unnatural and predictable.

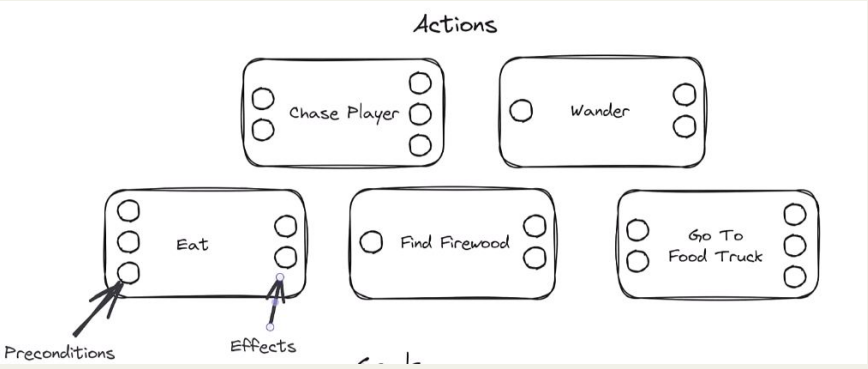
GOAP

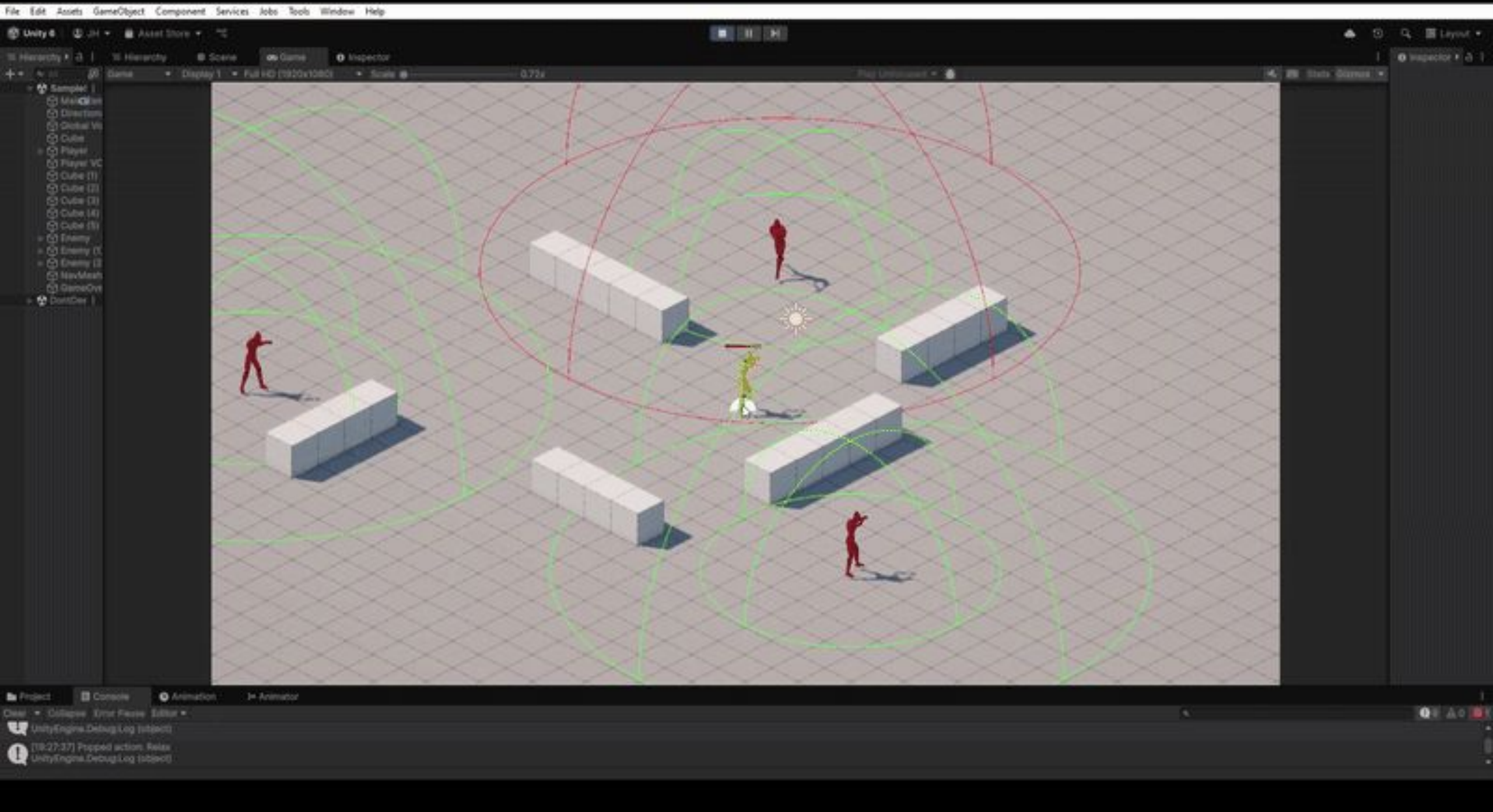




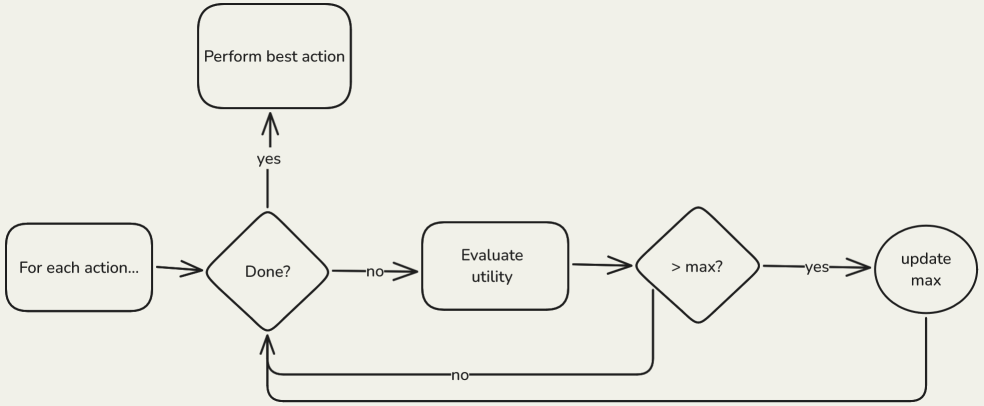
- Implementation:
  - EnemyAIBrain contains the actions, goals, and beliefs held by the agent.
  - AgentAction class: contains preconditions and effects of the action. Also contains the IActionStrategy interface used to perform the action.
  - AgentGoal class: contains priority and desired effects.
  - AgentBelief class: contains beliefs of the agent, such as if a condition is true or an observed location.
- Observed behavior:
  - 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.
- Benefits
  - The addition of more actions and goals is fairly simple, and can greatly increase the complexity of the agent behavior.
- Drawbacks
  - Complex initial implementation.

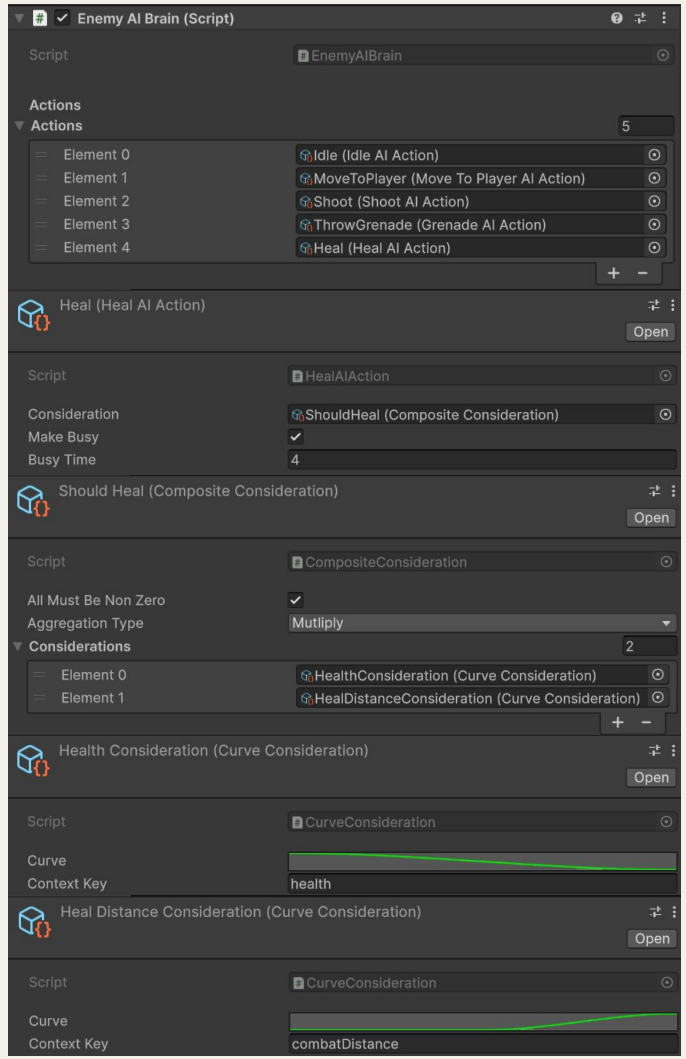





File Edit Assets GameObject Component Services Jobs Tools Window Help
Hierarchy
Scene
Game
Inspector
Project
Console
Animation
Animator
UnityEngine.DebugLog (object)
[19:27:37] Popped action: Relax
UnityEngine.DebugLog (object)

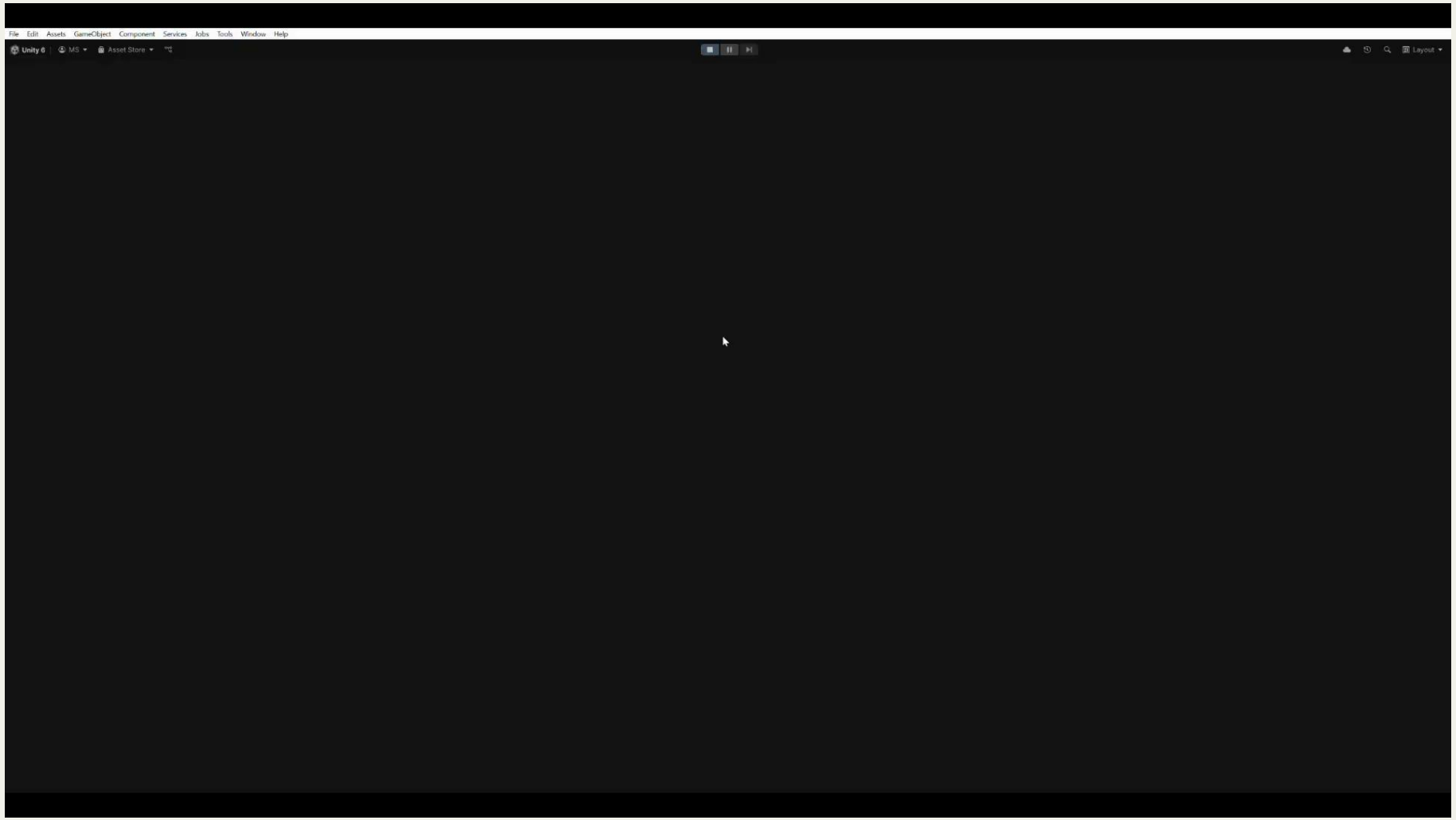
Utility AI





- Implementation:
  - AIBrain maintains a list of available actions, each action has utility evaluated by aggregating the scores of its relevant considerations.
- Observed behavior:
  - More prone to emergent behavior (most notably, the NPCs would tend to stay behind walls and throw grenades rather than shoot in the open).
- Benefits
  - Relatively simple implementation.
  - Can produce emergent behavior which feels more natural.
- Drawbacks
  - Tuning considerations/curves can be time consuming.
  - With emergent behavior, debugging can become more difficult.

Final Behavior (Utility AI)



Threats to Validity

- Internal: We divided the implementations between group members. However, we do not all have the same amount of experience. This may influence our opinion of ease of implementation. To mitigate this, we reviewed each others implementations and decided on implementation difficulty as a group.
- External: We specifically use Unity for implementation. Other game engines may increase or decrease the difficulty of implementation.
- Construct: We can only assess the results of these implementations qualitatively, as doing so quantitatively would likely involve a large survey of numerous playtesters, which we of course do not have the means to perform adequately.

Conclusions

- The right approach for NPC behavior is heavily dependent on the type of game you are trying to make.
- As complexity of your implementation increases, you don't necessarily see a benefit in resultant behaviors.
- When to use...
  - FSM - When action space is relatively small and transitions are easy to manage.
  - BT - When state machine transitions become too hard to manage and considerations become more complex.
  - GOAP - When complex but predictable behavior is desired and action space is relatively large.
  - Utility AI - When action space is large and/or emergent behavior is beneficial (less predictable).

Future Work

- Investigate different gameplay scenarios to paint a better picture of the strengths and weaknesses of each approach.
- Compare ML Agents to these traditional paradigms.
- Investigate hybrid approaches (Utility AI in particular has many principals that may enhance other approaches).

Questions?

Thank You!

Editable Icons