Goal Oriented Action Planning

A smarter AI technique

Vedant Chaudhari · Follow
9 min read · Dec 12, 2017

GOAP or Goal Oriented Action Planning is a powerful STRIPS (Stanford Research Institute Problem Solver) like planning architecture designed for real time control of autonomous character behavior in games. GOAP will give your agents choice and the ability to make intelligent decisions without having to maintain a complex finite state machine (FSM).

Introduction

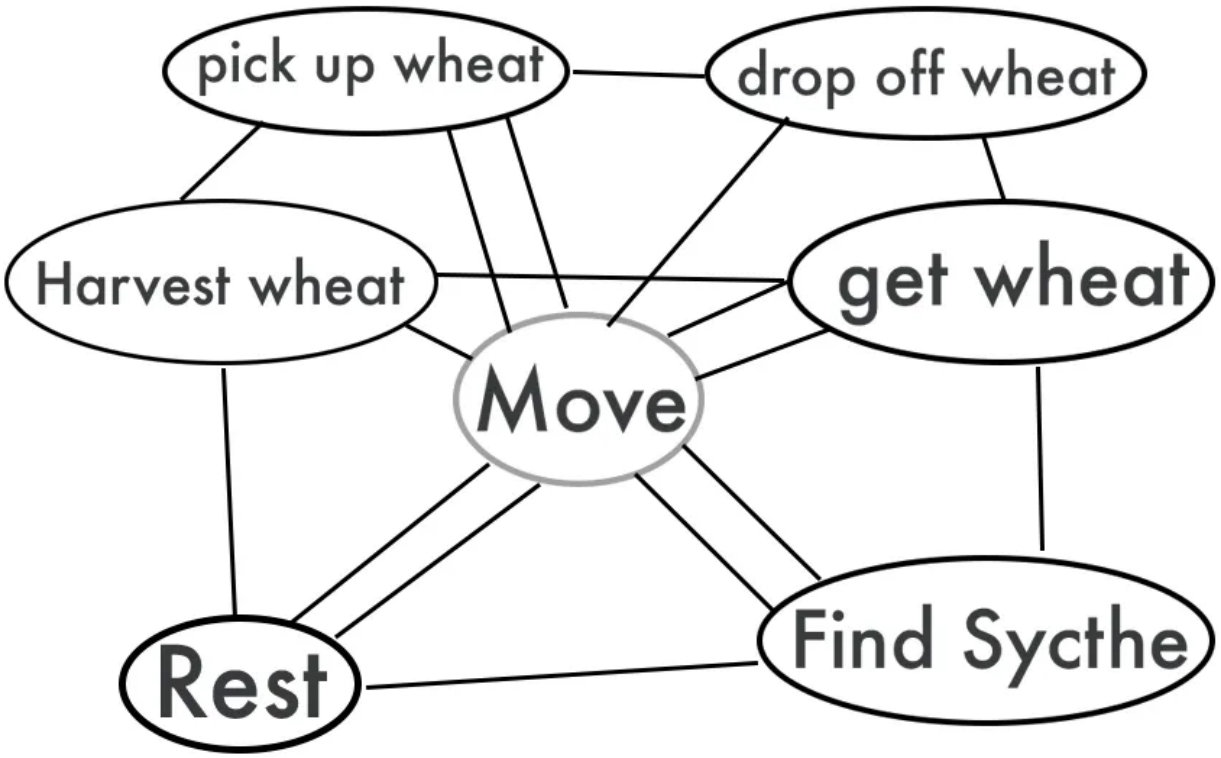
GOAP is an artificial intelligence system for autonomous agents that allows them to dynamically plan a sequence of actions to satisfy a set goal. The sequence of actions selected by the agent is contingent on both the current state of the agent and the current state of the world, hence despite two agents being assigned the same goal; both agents could select a completely different sequence of actions.

For example, let's say an agent's intended goal is to collect wheat. A human could reach this goal in many different ways such as:

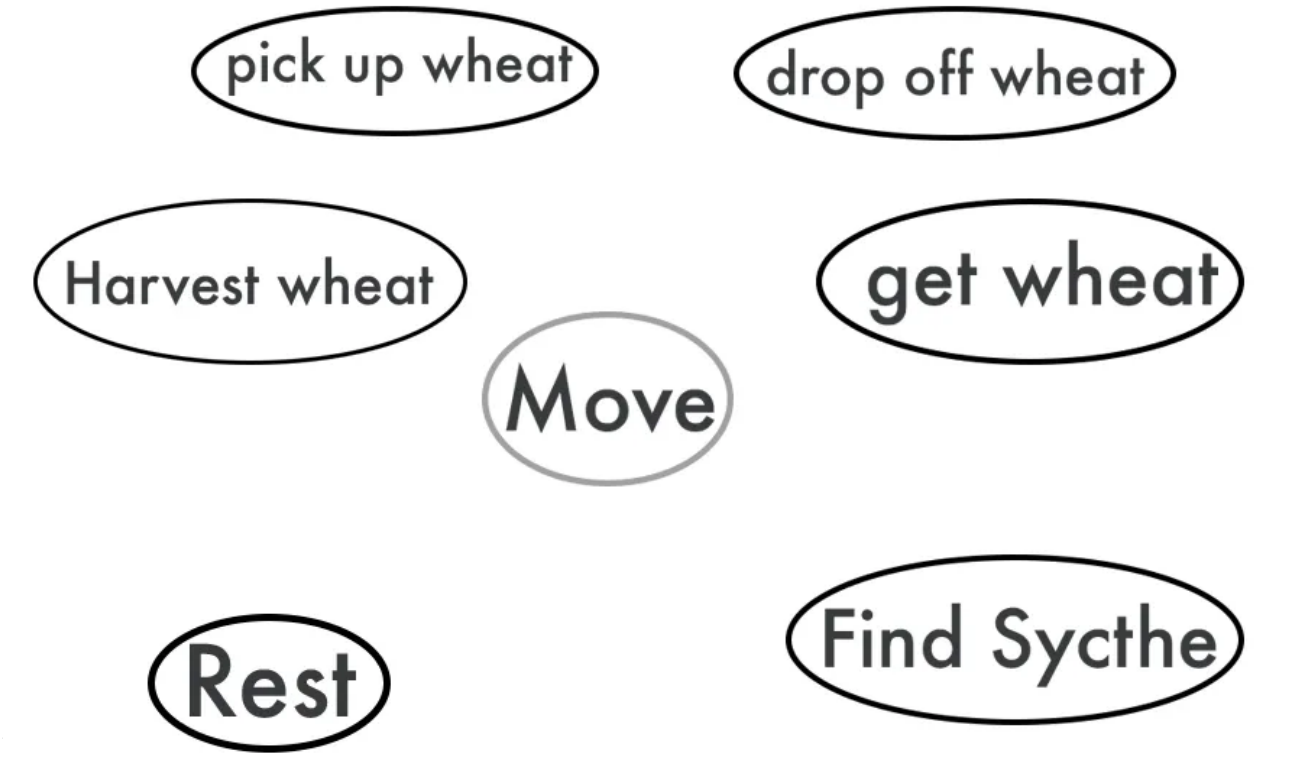
- Find scythe >> harvest wheat >> get wheat
- Find wheat silage >> harvest wheat >> get wheat
- Use hands to harvest wheat >> get wheat

GOAP will intelligently choose the most intuitive sequence of events based on the provided preconditions. i.e. if the agent does not have access to a scythe then they must harvest manually.

The behavior defined above could be implemented using a FSM, which would look like this:



or like this using GOAP:



A *goal* is any condition that an agent wants to satisfy. In GOAP, goals simply define what conditions need to be met to satisfy the goal, the steps required to reach these satisfactory conditions are determined in real time by the GOAP planner. A goal is able to determine its current relevance and when it is satisfied.

GOAP makes the code more modular and easy to maintain, by decoupling the states from each other, each action can be worked on incognizant of the others. Games suffering from complex FSM implementations for agents should utilize this method. Furthermore, GOAP allows adding and removing actions spontaneously; An agent must simply have a defined list of actions which are automatically processed by the GOAP planner.

Actions

Every agent is assigned *actions* which are a single, atomic step within a *plan* that makes an agent do something. Examples of an action are playing an animation, playing a sound, altering the state, picking up flowers, etc.

Every action is encapsulated and ignorant of the others.

Each defined action is aware of when it is valid to be executed and what its effects will be on the game world. Each action has both a *preconditions* and *effects* attributes which are used to chain actions into a valid *plan*. A precondition is the state required for an action to run and effects are the changes to the state after an action has executed. For example if the agent wants to harvest wheat using a scythe, it must first acquire the tool and make sure the precondition is met. Otherwise the agent harvests using its hands.

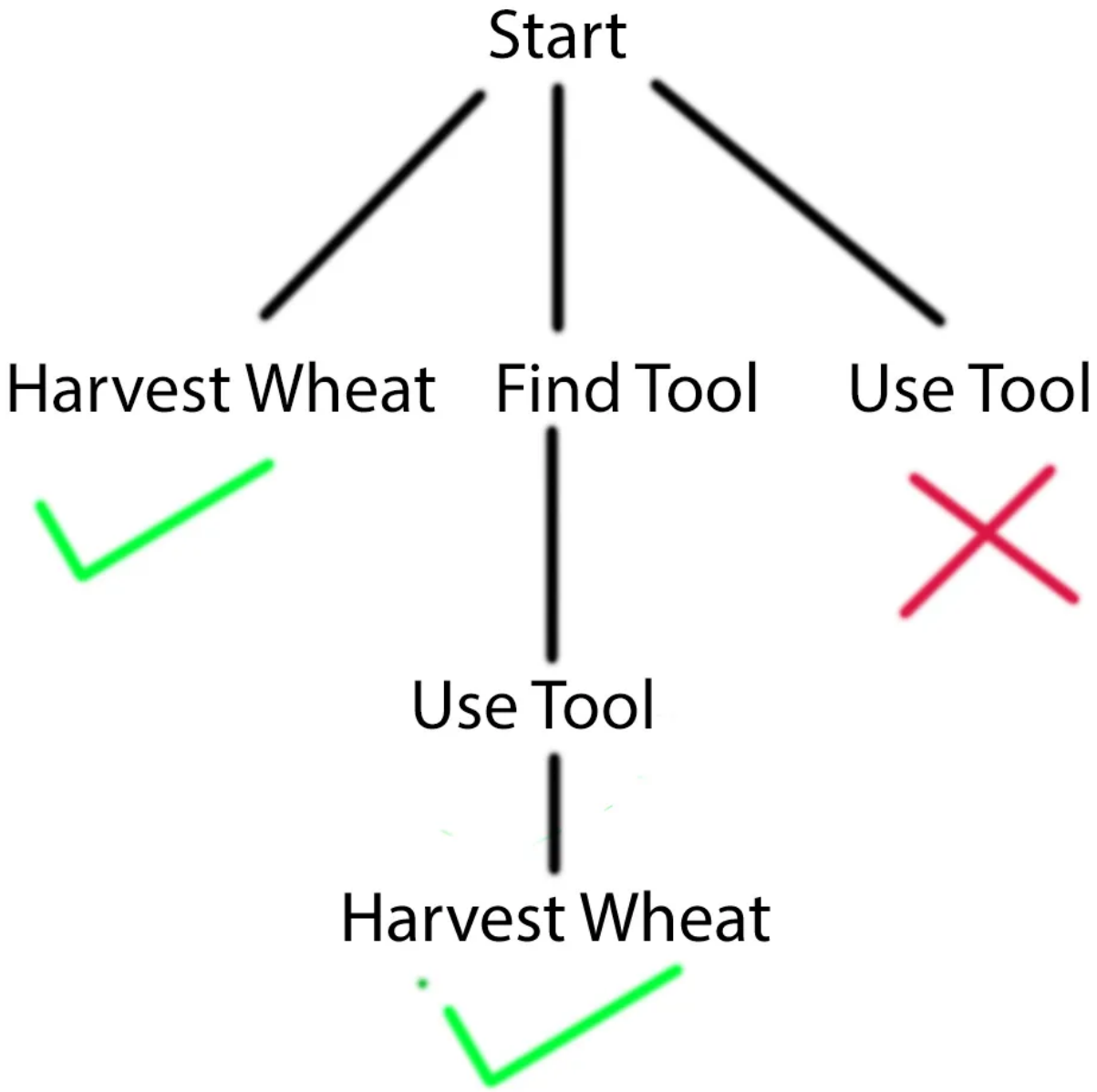
GOAP determines which action to execute by evaluating each action's cost. The GOAP planner evaluates which sequence of actions to use by adding up the cumulative cost and selecting the sequence with lowest cost. Actions determine when to transition into and out of a state as well as what occurs in the game world due to the transition.

The GOAP Planner

An agent develops a plan in real time by supplying a goal to satisfy a *planner*. The GOAP planner looks at an actions preconditions and effects in order to determine a queue of actions to satisfy the goal. The target goal is supplied by the agent along with the world state and a list of valid actions; This process is referred to as "formulating a plan"

If the planner is successful it returns a plan for the agent to follow. The agent executes the plan until it is completed, invalidated, or a more relevant goal is found. If at any point the goal is completed or another goal is more relevant then the character aborts the current plan and the planner formulates a new one.

The planner finds the solution by building a tree. Every time an action is applied it is removed from the list of available actions.



Visualization of planning tree

Furthermore, you can see that the planner will run through all available actions to find the most optimal solution for the target goal. Remember, different actions have different costs and the planner is always looking for the solution with the cheapest cost.

On the right you can see that the planner attempts to run the *Use Tool* action but it validates as an unsuccessful solution, this is because of the precondition and effect attributes assigned to all actions.

The Use Tool action is unable to run because it's precondition is that the agent has

a tool which it finds by utilizing the find tool action. But, the agent can harvest wheat regardless of whether they have a tool or not, but it has a much lower cost without the tool, so the agent will prioritize finding a tool if it is available.

In Unity, using c# preconditions can be implemented using a Hash Set containing a Key Value Pair wherein the string is the precondition identifier and the object is a Boolean.

```
HashSet<KeyValuePair<string, object>> preconditions;
HashSet<KeyValuePair<string, object>> effects;
```

These are only used for planning and do not effect the Agent until the actions are actually run.

Moreover, some actions need to use data from the world state to determine if they are able to run. These preconditions are called *procedural preconditions*. In the diagram, the Find Tool action will only validate itself to the planner if it is able to find some container containing a tool for it to use. If the procedural precondition is unable to be met for Find Tool then the planner tells the agent to harvest wheat manually.

For example the procedural condition for harvest wheat would look like this in c#

```
// This function checks for the nearest supply pile to deposit
wheat at
// Harvesters in different areas of the world will go to their
respective chests, but if they are destroyed they can
intelligently find another closer one

public bool CheckProceduralCondition(GameObject agent)
{
    Chest[] chests = GameWorld.chests;
    Chest closestChest = null;

    float distChest;

    foreach (Chest chest in chests)
    {
        if (closestChest == null)
        {
            closestChest = chest;
            distChest = Distance(closestChest.position -
agent.position);
```

```
        }
        else
        {
        // Check if this chest is closer
           float distance = Distance(chest.position -
agent.position);

        if (distance < distChest)
        {
            closestChest = chest;
            distChest = distance;
        }

    // If no chest is found, terminate the action plan
    if (cloestChest == null)
        return false;

    SetTarget(closestChest);

    return closestChest != null;
}
```

These procedural conditions can get taxing when a game is utilizing a large amount of agents, in that scenario GOAP planning should run on a thread separately than the render thread so that the planner can continuously plan minimizing effect on the game play experience.

How GOAP and FSM work together

In my implementation, I used a FSM with 3 states which were

1. Idle
2. MoveTo
3. PerformAction

The idle state is the default state an agent starts at. During this state, the agent passes its defined goal (which can be anything you choose, and there can me multiple different goals) to the planner along with the world and agent states.

It looks something like this:

```
// Remember the hashsets we used to store preconditions and
effects?
// We also use those to hold the worldstate (which is the agents
initial precondition) and the effect (which is the goal for the
```

```
plan)
// This makes the implementation more straightforward

HashSet<KeyValuePair<string, object>> worldState =
getWorldState();
HashSet<KeyValuePair<string, object>> goal = createGoalState();

Queue<Action> plan = planner.plan(actions, worldState, goal);
```

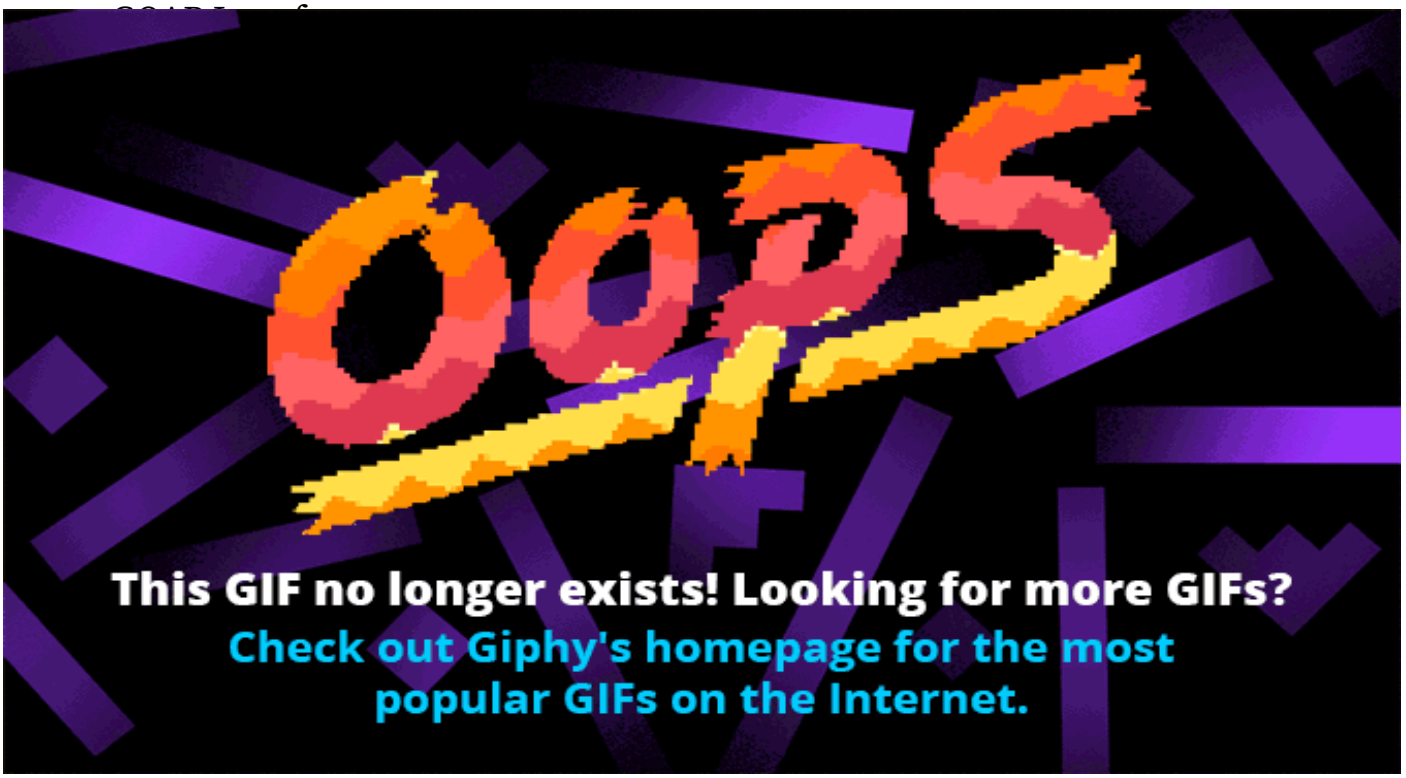
After the agent has found a plan it checks if it is within range to execute the command or else transitions to the MoveTo state. When within range of the target, the agent transitions to the PerformAction state in which the agent executes the action the planner specified next.

When there are no actions left to perform or the agent has fulfilled its goal, the agent returns to the Idle state and waits for a new plan.

Some Implementation Details

A standard GOAP implementation requires at least 6 base classes for it to work. These are:

- FSM
- FSM State



Demo of agents using GOAP

3. An Action implementation

As you can see, each agent fulfills their goal using the best sequence of available actions to them. In the GIF, the blacksmith has to wait until the miner has put enough ore in the chest for him to forge tools, furthermore when the agent's tools break but the miner hasn't forged enough yet; They either manually harvest or wait for it to be replenished.

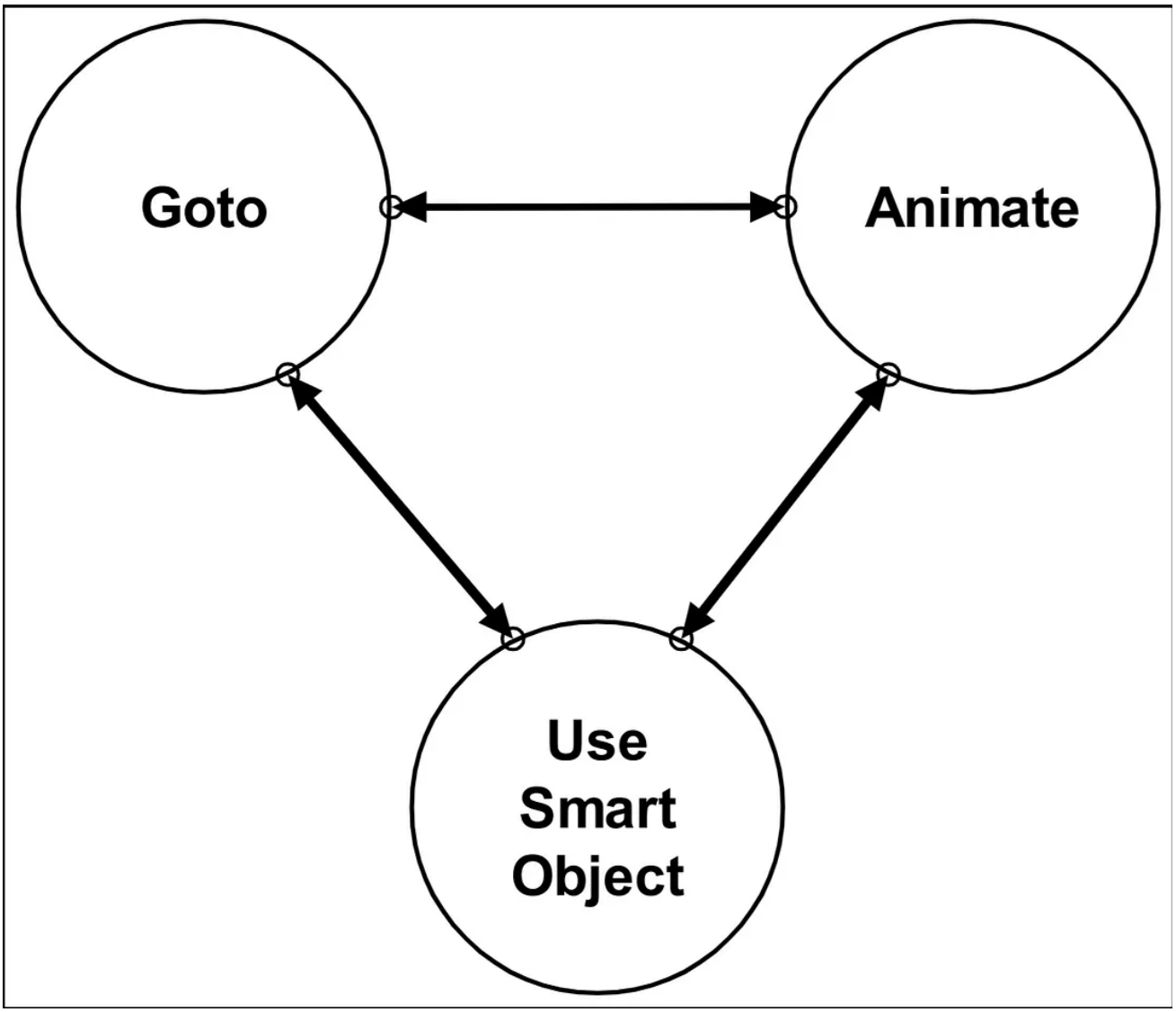
Tech Demo:

> *I'll be uploading a link to the demo at simmer.io when I can get Unity's web builds to actually work for me. I'll also upload the source code with the demo.*

Games that use GOAP

Some of the greatest games in the past decade have used GOAP at the heart of their AI architecture. **F.E.A.R** pioneered GOAP using it first, making one of the most memorable AI experiences ever.

The developers wanted to create a "cinematic" experience wherein the combat was as intense as a real human. Fear's AI agents could take cover, blind fire, dive through windows, flush out the player with grenades, communicate with teammates, etc. using GOAP. The developers themselves said their FSM only contained 3 states.



F.E.A.R's Finite State Machine

GOAP is ideal for a game that needs an incredibly varied AI that can perform multiple different actions that fit the context of the situation. A similar effect can be achieved using a state machine, but the result will often be much more complex.

> "For F.E.A.R. We decided to move that logic into a planning system, rather than embedding it into the FSM as games typically have in the past" — Monolith Productions

Furthermore, since the AI can use the planning system to make their own

decisions about state transition, it reduces a large amount of overhead for programmers and designers who can work on actions independently of each other.

GOAP is performance flexible and can work on a wide variety of systems and architectures. Naturally, it's complexity rises with both the amount of agents that are active and the amount of actions that are available to them. Despite this, as mentioned earlier, the planner can be offloaded to a co routine or alternative thread to drastically increase performance.

Conclusion

In most cases, GOAP is simply a more effective solution over a cumbersome FSM. Using GOAP, an AI can be very dynamic and have a large range of actions, without having to manually implement the interconnected states.

Sources:

ni.media.mit.edu/~jorkin/goap.html.

Orkin, Jeff. "Applying Goal-Oriented Action Planning to Games." Http://Www.jorkin.com *, doi:http://alumni.media.mit.edu/~jorkin /GOAP_draft_AIWisdom2_2003.pdf.*

Owens, Brent. "Goal Oriented Action Planning for a Smarter AI." Game Development*, 23 Apr. 2015, gamedevelopment.tutsplus.com/goal-oriented-action-planning-for-a-smarter-ai — cms-20793.*

ns, Monolith. "Three States and a Plan: The A.I. of F.E.A.R." MIT Media La gnitive Machines Group. MIIT Media Lab*, alumni.media.mit. /gdc2006_orkin_jeff_fear.pdf.*

TheHappieCat. "Combat AI for Action-Adventure Games Tutorial [Unity/C#] [GOAP]." YouTube*, YouTube, 23 July 2016, www.youtube.com/watch?v=n6vn7d5R_2c.*

TheHappieCat. "How 'Smart' AI (Basically) Works in Games (Goal Oriented Action Planning)." YouTube*, YouTube, 15 July 2016, www.youtube.com/watch?v=nEnNtiumgII.*

Artificial Intelligence | State Machine | Unity

Follow

Written by Vedant Chaudhari

2 Followers

Recommended from Medium