Dynamic Planning in Games: the GOAP Approach

Jordan Ellis, and Joshua Shlemmer

1 Overview of GOAP

GOAP (Goal-Oriented Action Planning), was originally implemented by Jeff Orkin for the game F.E.A.R. during his time at Monolith Productions. GOAP derives itself from STRIPS (Stanford Research Institute Problem Solver), a more academic solution to dynamic planning. GOAP is known for the emergent behavior that spawns from its simple components. At the root of the system is a Planner. Each planner has knowledge of the current worldstate in relation to its agent. Additionally, each planner can be customized by the user in two specific ways: setting a desired worldstate, and specifying a set of actions the planner can use to obtain this desired worldstate. All of these elements collectively are used during the pathing stage to obtain a set of actions the planner will enact to achieve its desired worldstate.

2 The End Goal

In our game, Genetic Drift (a couch based twin-stick shooter where players customize characters by selecting abilities and mutations), we had three basic AI types that we wanted to model: Blocky, Roundy, and Spikey. Each behavior mimics a specific play style the game's abilities lend themselves to.

3.1 Blocky

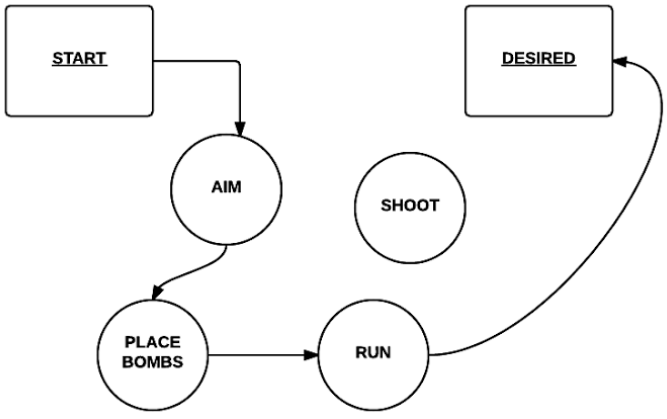

Figure 1 Blocky Behavior Diagram

Blocky is all about laying down abilities and surviving. His job is to charge the target, drop bombs, and run away, all while shooting at the target. As you can see from the diagram above, Blocky has a very simple set of actions. Even with this limited scope of actions, the planner is able to create a multitude of plans based on the situation

3.2 Roundy

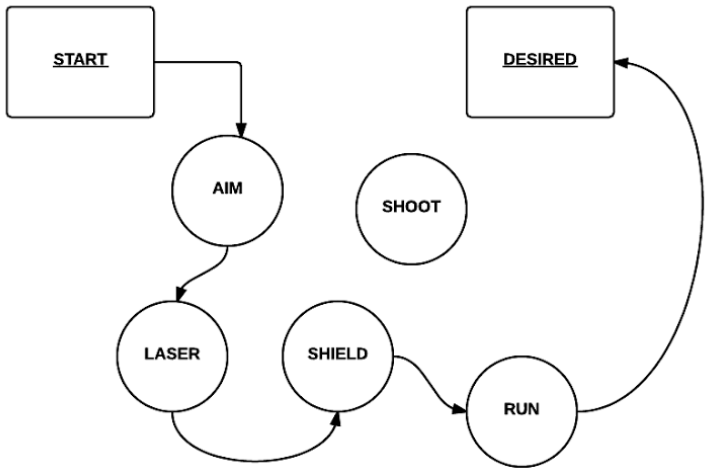

Figure 2 Roundy Behavior Diagram

Roundy's job is to stay as far away from his enemies as possible and attack them using the long-range laser ability, while also using his shield to increase his life expectancy.

3.3 Spikey

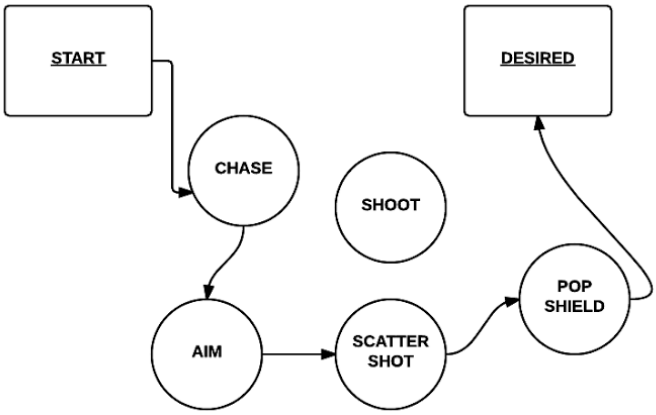

Figure 3 Spikey Behavior Diagram

Finally Spikey, the most aggressive of the bunch, uses multiple offensive abilities all geared towards dealing as much damage to the target as possible.

3 Specification and Implementation

Before we show how to define the actual GOAP planner, it is helpful to define each of the specific pieces that it is comprised of:

3.1 States

States are used to define a single attribute about the game world surrounding the agent, as well as attributes about the agent itself. While these states are simply just flags representing something such as *LOW_HEALTH*, we decided to take this a step further. In our implementation each state is defined as a predicate. This predicate contains logic for determining if its flag is true or false, given the agent it is pertaining to. This eliminates the need for outside logic having to manage or even know about the state or its value. This comes with an additional feature of being able to specify how often the states are reevaluated.

```
REGISTER_STATE(LOW_HEALTH, [](Entity *entity)
{
    auto health = entity->GetComponent<HealthComponent>();
    if (health)
    {
        auto cur = health->GetHealth();
        auto max = health->GetMaxHealth();
        if (cur / max <= 0.5f)
            return true;
    }
    return false;
})
```

3.2 Worldstates

Worldstates are a collection of flags (a bit-field), where each flag represents a state. As you might recall, each state represents an aspect of the current world. There are two different use cases for worldstates. The first is for defining a target worldstate. This is explicitly done by the user. Secondly it can define the current state of the world. This is created dynamically utilizing the state predicates. Worldstates can also be used inside of actions to define preconditions and postconditions. Since the user specifically chooses which states to set to true or false, we actually need to represent three different bit states: true, false, and inactive. To do this, we construct a mask that has a bit set for every flag that we do not care about. This is used anytime we compare two worldstates through bit comparisons since we only want to compare the bits that are relevant. An example of this is when we check to see if an actions preconditions are met.

3.3 Actions

As mentioned, actions have preconditions and postconditions. Both the pre and post conditions are defined as worldstates. Actions also have an associative cost and an action handler containing the action's logic. Action handlers were created using the same mentality as the states, focusing on self-containing logic without any outside systems having to even know that it exists.

```
bool ExampleActionHandler(Entity *entity)
{
    // return false: called next frame
    // return true: move onto the plan's next action handler
}
```

3.4 Planner

The planner is just an aggregation of all of these objects listed above. More specifically it contains one current worldstate, one desired worldstate, an update interval, list of possible actions, and the current plan in the form of *ActionHandlers*.

4 Building a Planner

For our implementation of GOAP, we wanted to make defining a planner as painless as possible. First, the planner is constructed as a component attached to the entity. Once a planner is constructed, defining it is as simple as: giving it actions, setting its desired worldstate, and setting its update interval. To make this cleaner in code, we used the chaining technique to make action and worldstate definitions fit cleanly into a single statement even though it takes multiple function calls to define. Here's an example:

```
GOAPlanner *planner;

planner->AddAction("Shoot", ShootActionHandler)
        .AddPre(STATE_TARGET_IN_SIGHT, true)
        .AddPost(STATE_TARGET_DEAD, true)
        .SetCost(2);

planner->SetDesiredWorldState(
        WorldState()
        .SetFlag(STATE_TARGET_IN_SIGHT, true)
        .SetFlag(STATE_TARGET_DEAD, true)
    );

planner->SetUpdateInterval(0.5f);
```

5 Creating Plans with A*

While we are not going to specifically explain how to implement A*, we decided to explain some of the differences in using A* for GOAP. One major change is, when checking for neighbor nodes like you would in normal A*, you instead get all actions that have all their preconditions met. For each action we create a future worldstate by applying that action's postconditions onto the current worldstate. These future worldstates are then used for calculating the heuristic by counting how many flags this new worldstate has that oppose the target worldstate.

6 Pros, Cons, and Takeaways: Where to Go From Here

Overall, we found that GOAP allowed for quick definitions of behaviors since defining goals and actions was so simple and self-explanatory. When creating three different behaviors in the game, starting each off with a very basic set of actions (i.e. look and shoot) proved to be the most efficient. After that, actions were added gradually, helping shape the agent's behavior into the desired behavior. This iterative workflow was extremely fast and effective. After just a few hours, we were able to produce three very different behaviors.

Moving forward, taking a more data driven approach to creating a planner would be even better. This would allow for the creation of visual editors, making it easy for designers to create new planners without having to write any code. This would further increase iteration time by removing any need to recompile and making it more user friendly.

GOAP isn't without its downsides though. For this project, most of the behaviors could have been created in FSMs as fast or if not faster than they were created with our planner. In fact, for one behavior in particular, it was harder in GOAP then it would have been using FSMs because we needed things to happen in a very specific order for it to look right. This problem was mitigated by combining actions that needed to happen together into a single action. In this case we were lucky it was so simple, often times all you can do is change the costs of actions and hope for the path to be chosen correctly.

That being said, we still feel that the ease of defining actions outweighs a lot of these problems, and we both agree that GOAP is a much more flexible system than simply using FSMs. When it comes to creating dynamic behaviors in our next projects, GOAP will undoubtedly be a useful tool in our toolbox for doing so.

7 References

7.1 Applying Goal-Oriented Action Planning to Games
[Orkin] Orkin, Jeff 2003. AI Game Programming Wisdom 2