Hareesh Ganesan
April 27, 2012
CS108-Hewner

Independent Design Document

Basic Use Case

A basic use case of the AI system would deal with making an AI Agent that could follow a user agent. In order to do this, you simply extend the BasicAIAgent class, and add the following update code to the calculateLocation method.

```
List<FighterSprite> fs = myLevel.getFighters();
    if(getCurrentLocation().distance(fs.get(0).getCurrentLocation())>40){
        MotionAction follow = new MotionAction(this, fs.get(0).getCurrentLocation());
        follow.performAction(elapsedTime);
    }
```

After this, the AI Agent need only be instantiated in the Level Factory and added to the list of FighterSprites.

Complex Use Case

I think my design can handle some very interesting parts of the fighting genre that make game designers lives a lot easier. One feature that it allows is giving the game designer an easy way to instantiate AIAgents that prioritize and select from various strategies, each of which returns a grouped-action pattern designed to accomplish a certain goal.

Because the design is flexible, it is simple for a game designer to define sets of actions that occur in a time period using the ActionSeries or Goal objects. ActionSeries objects allow fixed-time action sets to be specified. Goals repeat actions until a failure or success state is reached. Goal objects can be used together with Strategy classes that define a macro-strategy for an AIAgent. Once Strategies are defined, they can be loaded with weights into AIStrategyAgent. After loading the strategies, the Agent will select a strategy and execute that set.

Let's say the goal is to select a strategy that moves toward an AI agent for however long it takes, and then hit that AI agent.

The AI Strategy agent will call its update method. The update method will perform an Action from the current ActionSeries. If that ActionSeries is done, we will select a new ActionSeries. In order to select a new ActionSeries, we will use the Strategy currently being followed. If no Strategy is currently in use, we select a strategy based on a weighted random selection. If a Strategy is currently in use but it has completed, we select a new Strategy. If a Strategy has not been completed, we ask it to give us the next ActionSeries we must complete. Each Strategy object contains a list of GoalSeries objects it must accomplish for success. Upon queries for a new ActionSeries, the next GoalSeries object in the list is returned. If there are none left to return, we set the strategy to complete. If there are further actions, we continue. If the Action failed for some reason, upon exit from the ActionSeries, the Strategy object should be set to complete and a new strategy should be selected.

Design

Actions

My solution for this design is subclassing the FighterSprite class and implementing an AIAgent. This AIAgent contains an calculateLocation method that would need to be passed in relevant information, and update the AI Agent appropriately. The calculateLocation method typically executes an Action related to the AI. It would be the game designer's discretion to code how the next action would be determined.

There must be some central repository that allows the AI agent to properly learn what the game state is. This repository would be the CombatInstance object that contains within it references to all of the sprites and NPOs. An AI agent would be created with a reference to this instance so it can use any information in the game state to generate its next action.

The AI agent also must share an interface with the PhysicsEngine that is identical to what is used by the human agent. In order to design this system, an InputHandler object is declared that maps keys to Action objects. The InputHandler is an instance variable of a given game screen. On update, the handler would check if a key was being triggered and call the performAction method of the object mapped to it. This would allow generic key mapping for users. The AI taps into this process by directly declaring the Actions in their calculateLocation method and then calling the performAction method.

Actions are generic objects that allow for actions to be executed whenever their perform action method is called. Actions are used primarily in pair with the InputHandler but also as a way of effecting change for an AI agent. Actions can also be paired with InputHandlers on non-combat screens to enable easy option selection.

ActionTimers are the final part of the Action set up. ActionTimers control how much actions can be repeated. They can be set to available or unavailable. When they are available, they can be activated, triggering the timer. Once the timer triggers, the *action* method will return true while there is time, allowing a MotionAction to be completed. When the time completes, the ActionTimer becomes unavailable, and *action* returns false. For example, a character's jump timer could be off until it jumped, at which point it would trigger. After some time moving up, the jump could be turned off with the timer. The character would not jump again because the ActionTimer is unavailable, and then Timer would be made available when the fighter landed.

Strategy

The goal for the design of the AI Strategy system is to allow users to easily create novel AI Agents without programming in depth strategy. This means the game designer could easily specify a new AI agent that selects different strategies (offense, defense, npo) based on preference weightings that are given. The strategies can be added dynamically to an AI agent to keep the design open. On each update, the AI agent would attempt to complete the action set specified by a given strategy. Once complete, the next Strategy would be selected.

To implement this strategy system, a new Strategy class is defined. The Strategy class is an abstract class that can be extended to implement specific strategies, and get new Goal objects based on the strategy selected. These strategies can be used in the selection of the next action for a StrategyAgent. To make a new strategy, the Strategy class must be subclassed. The Strategy agent must declare the specific list of goals it must accomplish in the constructor. Specific goals were defined as inner classes to better organize the goals associated with a Strategy.

To implement variable time actions, a Goal object is used. The GoalSeries object ultimately determines the action performed. The GoalSeries knows whether it has completed successfully or unsuccessfully. As long as success states continue without the goal being achieved, the GoalSeries should continuously perform an Action. Once the goal is achieved or the cutoff time is reached, the goal ends.

In order to enable multi-action groups, an ActionSeries object that implements Action is used. The ActionSeries object tree maps an action to a Long key. On each update, the time in the action series will increment. The action with the key that is closest to the current time being tracked by the action series is selected. As the key finding is done with a TreeMap, we maintain efficiency. The ActionSeries can also be reset and reused, so new objects need not be instantiated for each frame. ActionSeries require that a TreeMap with times mapped to Actions are entered in order to construct the object.

The final section of the Strategy system is the selection of which Strategy to use. The current system is dependent on weighting Strategies, and then selecting one using weighted random selections. However, in certain situations the weights must change. In order to implement situational awareness, a Situation object was developed. The Situation object knows whether a situation is occurring, and contains a selection method for Strategies it contains. The default situation is always true and has a 50-50 Offensive-Defensive weighting. When the SituationalStrategyAgent must get a new Action to perform, it iterates through an ordered list of situations, stopping on the first situation it sees that is occurring. It then obtains the strategy from this situation.

Alternative Designs

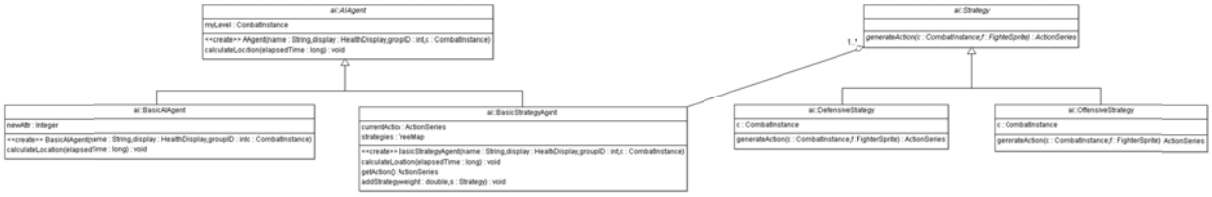
I based my designs on the principle that AI agents executed action-based events, checked their new location, and updated accordingly. I think that design makes the most sense, but there are ways of changing the implementations and details behind the AI.

I chose to use the CombatInstance as the instance of a Level that would provide information about the level's current state. While this has led to easily extensible and usable code, I think I could have gotten more efficiency out of the system by employing a Singleton that kept track of AI-related information. The state of the Singleton could have been saved to try to implement Machine Learning Features. This Singleton object could have been an AI engine that used Observers to observe state changes and relay changes in the system to the AI agent. This would make the AI agent more of a *reaction*-based agent as opposed to a proactive agent. The AI agent could then implement strategies in a State pattern that occasionally used random probabilities to dictate state transitions for variability.

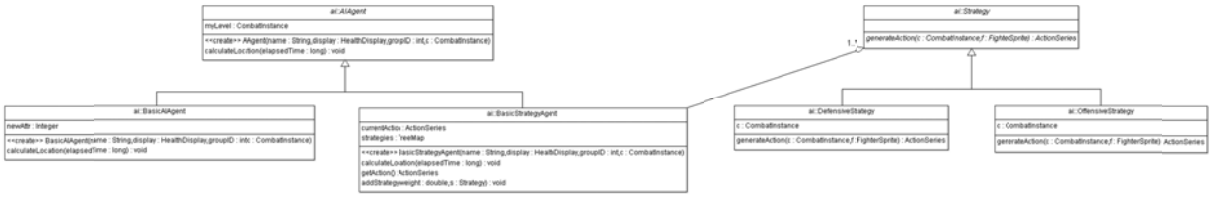
Another design I could have used would be to use Singleton ActionEngine that controls processing of Actions for both AI and humans. I could have made separate Actions for AI that the ActionEngine would select between Actions to optimize. The ActionEngine would be what was allowed to perform the Actions, so both AI and human agents would route their actions through this engine. This would be useful to organize Collisions before any changes were made to the agents. Moreover, the current system for employing Actions that are not constantly repeatable (i.e. jumping or shooting) is clunky.

UML Diagrams

AI Agents



Action Objects



Strategy