Hareesh Ganesan

April 13, 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 macro-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 object. These ActionSeries objects can be used together with Strategy classes that define a macro-strategy for an AIAgent. Once Strategies are defined, they can be dynamically loaded with weights into the AIStrategyAgent. After loading the strategies, the Agent will select a strategy and execute that action set.

The user could, in a complex use case, define an Offensive strategy, a Defensive strategy, a Critical Health strategy, and a Powerup strategy. Based on weights, the agent would seek out moves optimizing that strategy. The ActionSeries objects and the actual selection could be made non-random, and could easily implement a Q-learning system or use adversarial move-based search trees to make more or less optimal characters. By changing the weightings, one could even introduce methods of making the AI more or less difficult.

Design

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.

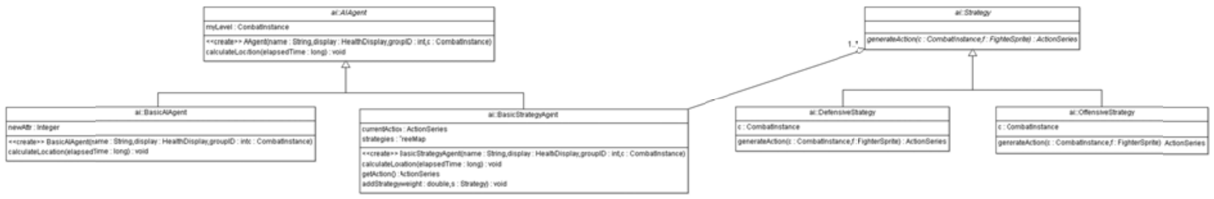
The goal for the design of the AI Strategy system is to allow users to easily create novel AI Agents without programming in depth strategy. This means the game designer could easily specify a new AI agent that selects different strategies (offense, defense, npo) based on preference weightings that are 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. Once complete, the next action 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 ActionSeries 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.

In order to enable multi-action groups, an ActionSeries object that extends 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.

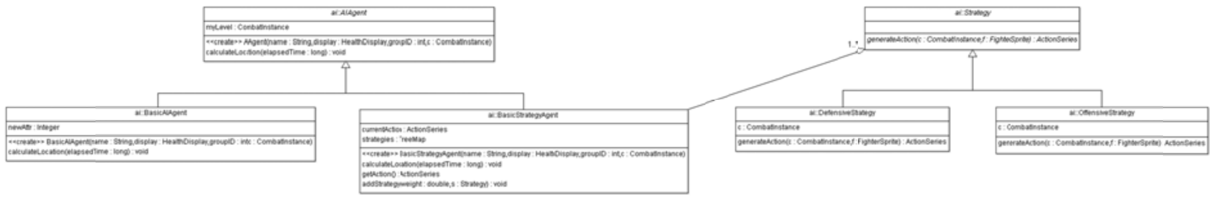
UML Diagrams

AI Agents



ai::AIAgent
myLevel : CombatInstance
<<create>> AAgent(name : String,display : HealthDisplay,groupID : int,c : CombatInstance)
calculateLoction(elapsedTime : long) : void
ai::BasicAIAgent
newAttr : Integer
<<create>> BasicAIAgent(name : String,display : HealthDisplay,groupID : intc : CombatInstance)
calculateLocation(elapsedTime : long) : void
ai::BasicStrategyAgent
currentActio : ActionSeries
strategies : 'reeMap
<<create>> BasicStrategyAgent(name : String,display : HealthDisplay,groupID : int,c : CombatInstance)
calculateLocation(elapsedTime : long) : void
getAction() : ActionSeries
addStrategyweight : double,s : Strategy) : void
1..*
ai::Strategy
generateAction(c : CombatInstance,f : FighteSprite) : ActionSeries
ai::DefensiveStrategy
c : CombatInstance
generateAction(c : CombatInstance,f : FighterSprite) : ActionSeries
ai::OffensiveStrategy
c : CombatInstance
generateAction(c : CombatInstance,f : FighterSprite) : ActionSeries

Action Objects



ai::AIAgent
myLevel : CombatInstance
<<create>> AAgent(name : String,display : HealthDisplay,groupID : int,c : CombatInstance)
calculateLoction(elapsedTime : long) : void
ai::BasicAIAgent
newAttr : Integer
<<create>> BasicAIAgent(name : String,display : HealthDisplay,groupID : intc : CombatInstance)
calculateLocation(elapsedTime : long) : void
ai::BasicStrategyAgent
currentActio : ActionSeries
strategies : 'reeMap
<<create>> BasicStrategyAgent(name : String,display : HealthDisplay,groupID : int,c : CombatInstance)
calculateLocation(elapsedTime : long) : void
getAction() : ActionSeries
addStrategyweight : double,s : Strategy) : void
1..*
ai::Strategy
generateAction(c : CombatInstance,f : FighteSprite) : ActionSeries
ai::DefensiveStrategy
c : CombatInstance
generateAction(c : CombatInstance,f : FighterSprite) : ActionSeries
ai::OffensiveStrategy
c : CombatInstance
generateAction(c : CombatInstance,f : FighterSprite) : ActionSeries