Michael Landes
CS 8803 Game AI

Game AI Project 1: Ms Pacman

Instructions

1. Unzip “mspacman.zip”
2. Navigate into the newly created directory, you should see game/ as a subdirectory
3. Compile the source code as follows
   "javac -cp . game/Exec.java"

4. Run the game as follows
   "java -cp . game/Exec"

Introduction

The objective of this project was to implement the 4 AI Ghosts in a way that replicates the behavior of the original game. For this project, I used the *Game Internals* page and more specifically, the *Pacman Dossier* as my rule source, and attempted to implement Ghost Controllers that followed these specifications as closely as possible.

There are some key differences in the provided game engine that prevent an exact replication of the original Pacman logic. I will discuss the provided game engine and my implementation in detail from the perspective of the Pacman Dossier document, and then I will provide a commentary on the A.I. implementation.

Finally I will discuss assumptions, challenges, and answer the questions provided in the project description.

Rules

In this section I will go through the rules of Pacman as described in the *Pacman Dossier* one by one and discuss relevant issues in the provided game engine and my resulting implementation. This will suffice as the rule set I followed in implementation.

The engine implements all the mechanics of scoring, lives, level advancement, and other game details. For the most part this handles the more monotonous details of Pacman with one major and important exception. The engine is for MS Pacman and not Pacman. There are 4 different level maps and some other minor differences, however for the most part, I have stayed true to making the Ghost’s behavior match their behavior from the original game.

There are three states: chase, scatter, and frightened. This is fine, however in the engine provided, the logic for frightened is provided. Rather than being able to write a controller that handles all three of the states – I had to write a controller that was *aware* of all three states, but only governed directly chase and scatter. More specifically, the game engine implements the “reverse direction” piece of the logic when a power-pill is eaten. Unfortunately, there is no way to directly reverse the direction of a ghost along his current path of travel. The engine asks the controller for a direction when the controller reaches an intersection, so the best we can do is reverse a direction if the controller is currently in an intersection point.

The ghosts follow a pattern of timings to move from one mode to the next. The moves switch back and forth between chase and scatter and they start in scatter for each level. For each level, the exact times can be different during when this switch occurs. I have mimicked exactly the original Pacman logic for when these switches occur, including differentiating for the specific levels. During these mode switches, if a power-pill is eaten, the progression is temporarily paused while the frightened mode occurs. I have also respected this behavior.

As I said, the game engine implements ghost controls for frightened mode including the random number generator and direction choosing algorithm. When a power pill is eaten, I simply let the engine do its work, and jump back into controlling when the phase ends and scatter or chase mode is re-entered.

The original Pacman game changed speeds of ghosts and the player at different intervals, there is no support for this in the provided engine, so I ignored all speed-related features including the “Cruise Elroy” speed changes. Note that I did not ignore the “Cruise Elroy” scatter target behavior – more on that later.

In the original game, Pacman can corner – which means turn more quickly. The engine doesn’t support this feature, and so I ignored it.

The engine takes care of all home behavior, pulling ghosts back to home, and controlling their release times. I did not interfere with this.

Since the maps are different in Ms Pacman, and speed changes aren’t implemented, there are no “exploitable areas” such as those that existed in the original game. I did not try to bring these back into play in any way.

The original game divides the game into legal and dead space tiles. The provided engine does not have a concept of tiles, but does have a similar concept of nodes. Nodes are essentially every bit along the valid path traveled in the maze. This eliminates the existence of a concept of ”dead space.” In the original game, ghosts could target dead space, but in the provided engine that really isn’t possible. Instead I created a wrapper to the node interface which I called “tiles” and which represents what would have been the live-space tiles from the original game. My interface ignores the idea of dead space, but I can use it effectively to accurately mimic the correct ghost behavior.

Using my tile interface, I can determine by node which tile a current actor is in, and convert from raw x-y coordinates to tiles in the tile-space.

I made no attempt whatsoever to verify if the “pass-through” bug exists in the provided engine, although I suspect that it does.

The engine implements path choosing logic based on provided target tiles. Therefore, my controllers simply had to provide target tiles as closely as possible to the target tiles that the original game would have provided at any given moment. Path planning and intersection decisions were already all implemented.

The original game had certain fixed-target tiles that were in dead space. Since no concept of dead space exists in the provided engine, I used fixed target tiles that would be as close as possible to where the original dead-space tiles would have been. For each ghost, a specific tile is assigned in scatter mode that brings them to their own respective corner of the board.

Blinky – the original behavior for the red ghost is to target the location of Pacman during chase mode. When a certain number of dots have been eaten, Blinky adopts a characteristic called “Cruise Elroy” where he continues to target Pacman even in scatter mode. I have mimicked this behavior exactly, including the point at which Cruise Elroy comes into play for each given level.

Pinky – the original behavior of the pink ghost is to target four tiles ahead of Pacman in the direction he is facing. This behavior had no concept of the maze and did not respect walls. There was a bug when Pacman faced upwards that the target moved an additional four tiles to the left after having moved four tiles up. My implementation is again exactly the same. However, since I can’t reference dead-space tiles, I use x-y coordinates, and then search the maze for the closest (smallest Euclidean distance) live-space tile in my tile interface to that location. This creates a behavior where if Pacman is facing the edge of the maze or a wall that is too long, the target tile will move either to the closest end of the wall, or the inside edge of the maze. Since the direction doesn’t change, this pretty much gives an accurate Pinky behavior.

Inky – the blue ghost uses the target two tiles ahead of Pacman and the location of Blinky to extract a direction vector from Blinky to the tile ahead of Pacman. Inky then doubles this vector and uses the result as his target. I calculate the two-tile-ahead target with the same rules as above, and use the x-y coordinates to double the vector. Again, I use a closest Euclidean distance to find the closest valid live-space tile to the desired location.

Clyde – the yellow ghost targets Pacman when he is further than 8 tiles away and his scatter corner when he is closer that 8 tiles away from Pacman. I again used the x-y coordinates to determine Euclidean distance, and then divided it by 8 times 4 which is the distance between two tiles to determine if Clyde fell within or outside of that range.

Finally – I made no attempt to see the split screen at level 255, although I suspect that it doesn’t exist in fact, I know that the engine only goes up to level 16 or so, so I am sure it doesn’t exist.

A.I. Implementation

Here I will discuss exactly how I implemented the A.I. Most of what I did was pretty brute force. This is because the controller logic is pretty simple. I used a mix of state machines, and decision trees.

The state machine is simple, it has three states for each mode of play Chase, Scatter, and Frightened. Frightened is more of a pseudo-state since the engine handles all of its logic anyways. I use timers to track progress through a state, and switch state as each mode finishes according to the original Pacman rules. I then expose the current mode to my controller, so that all it has to do is query the mode when it is trying to decide what target tile to use.

The controller itself is more of a decision tree than anything else. The first question it asks is *what mode am I in?* The answer is provided by the state machine as described above. If the mode is scatter, then the controller asks my internal game map (the tile interface) for the fixed target tile according to ghost. The map keeps track of this. If the mode is frightened, there is no target tile, since the engine is handling the logic, so we don’t assign target tiles. If the mode is chase, then again I query the game internal tile map for the appropriate target tile according to which character is asking. The game map provides in its implementation methods to calculate all of the relevant target tiles based on the current game structure and locations of current actors. The important tiles that must be computed are the four fixed location

corner tiles, the current Pacman tile, the current Pacman + 4 tile, the current Pacman + 2 tile, and the current Blinky tile. From this, we can extrapolate all the necessary target tiles.

A state machine made the most sense to track mode, since the mode transition logic is so simple. A decision tree made the most sense for the ghost controller, because there was so much repeated or similar behavior among the ghosts.

Assumptions

Most of my assumptions are laid out in the section describing the rules of the game. I tried to build my internal tile-map so that it could accurately create itself from any provided node interface. In this way, I don't really have to make many assumptions about the game since I am abstracting its interface away.

I assume though, that the engine correctly handles mechanics that it is responsible for – namely, the ghost house timer, the frightened timer, the collision logic, and the path-planning logic.

In terms of ghost behaviors, I made assumptions that assigning target tiles on a "closest valid tile to actual x-y target coordinate" basis would be good enough for the logic to appear tight.

Challenges

The game engine does something really stupid – it controls the reverse outside of the ghost controller. As if that weren't bad enough, it provides a random probability of the ghosts changing directions at any given time. I suspect that this behavior is corrected in newer versions of the engine, but this is not the case in the implementation version of the engine that we used. Essentially this makes it impossible to control reverses at mode changes and impossible to prevent random reverses from happening. If you use only my ghost controller in the original engine code, there will every now and then be random reversal of ghost direction. I made one modification of the game engine in the code I submitted, and that was to set the probability of this event to 0 so that at least theses random reversals would no longer take place. However, regardless of which engine is used (original or modified) my ghost controllers cannot change the direction of the ghosts during a mode change. The only exception is if a ghost is at the exact moment in an intersection making a decision about which path to take, he will take the opposite direction of the path he picks. This isn't really a solution and isn't true to the original game, but it is my attempt to show that my controller does track reversal events and would implement them if it had the power to do so.

The second challenge, although not debilitating like the first one, is the way that the game provides an interface to the map. The game uses nodes, which ultimately are just the path that is travelable. Since each new pixel is its own node, the nodes all overlap, and aren't a good way to think of a maze. The best way to think of the maze is in terms of a graph of unique positions that don't overlap, essentially a subset of the nodes. This is the "tile" that is referred to in the original Pacman Dossier. Also, nodes provide a poor way to calculate vectors and offsets from Pacman since they are inconsistent. When I say nodes are inconsistent, I mean that node 4 could be right next to node 3, but across a wall from node 5. I get around this whole mess at the initialization of each new level by iterating through the entire node set, and extracting only the unique nodes and converting them into tiles, which have a reference back to the original node, and more importantly an x and y location on a grid. Since my tiles use x-y coordinates, It become really simple to determine if there are tiles next to any other tile, and in which direction. I can do vector calculations using normal coordinate space math, and easily get back an index that lets me interact with the game engine. I added methods to this structure to completely hide the game's node

interface, and allow me to use the map in a simpler way. Ultimately, addressing this challenge made all of the target computing logic much easier, which goes to show that some challenges, when properly dealt with, can really be good opportunities.

Finally, there were minor challenges in how the engine controlled certain parts of the logic (specifically frightened mode) for you. Getting around this wasn't hard, in fact, it made things easier. My state machine just had to be aware of when frightened mode was occurring, and pause until frightened mode was no longer occurring. This effectively turned my state machine into a multi-level state machine. If Pacman's logic were more complex, this would have quickly gotten out of hand and been a much worse problem. A state machine would in such a case be a much less effective solution.

Questions

Why is the ghost AI effective? What is the trade-off between fun vs effective given computing power at the time?

The ghost AI is very effective because it appears much more complex than it really is. As I was experimenting with the ghosts, it was clear that there were configurations of ghost targets that would make Pacman very impossible to play. Even having every ghost simply target Pacman directly gets overwhelming for the human player. The slight variation in targeting behaviors allows the perfect amount of room for escape and the perfect amount of pressure to keep a player on his toes. By making the ghost path-planning non-optimal, we allow Pacman the ability to find clear paths to his goals and avoid collisions by taking advantage of ghost behaviors. Having implemented the ghost AI, I understand how simple it is under the covers. Two ghosts target Pacman, one of them removes this target if he gets too close, the other two target spaces ahead of Pacman. At the time, there wasn't the computing power necessary to do brute force path-planning that would get accurate and valid paths ahead of Pacman, but the brilliant thing is that it is ok not to have accurate path-planning. Since everything is constantly in motion, and since the ghosts frequently make mistakes in guessing Pacman's future path, the game becomes playable and fun for the player.

Given the processing power of modern computers, would you re-design the ghosts? Why?

I have two answers to that question. The first is no. Pacman hit the sweet spot with perfectly balancing fun vs. effectiveness in AI. Any modification at all would make the game entirely a different experience. If the goal was to have an effective Pacman game that is fun, it would take a large amount of trial and error to create a new fun experience with different behavior from the current controllers, and none of that trial and error would really depend on the computing power of modern machines. Much of the fun that exists in Pacman is dependent on really quick and simple path-planning.

The second answer is yes, but not with a goal of making a fun Pacman game. I would re-design the ghosts, change the maze to make it much bigger, make the ghosts more effective by doing more powerful and accurate path-planning (at least Djikstra or similar), and create a much more modern clone of Pacman. The game experience would be entirely different, feel bigger, more intense, but still involve navigating a maze while avoiding enemies who are hunting you.