Donald Dunagan

811-648-053

14 December 2018

Py-World, for the Course CSCI 6550 Artificial Intelligence

1. Introduction

In the early 2000's, Dr. Donald Nute of UGA developed V-World, an environment simulator implemented in the Prolog programming language for the purpose of examining the actions of artificial agents placed into a virtual world. These environments are temporally and spatially discrete with the world's foundational block cells, actors, and environment all displayed on the screen like a video game. Actors in the environment take turns performing actions. An artificial agent has two attributes: strength and damage. If the agent's strength drops below zero or its damage rises above 100, the agent dies. In regard to the agent perceiving the world around it, an agent can perceive the 5 x 5 grid of cells surrounding it without hinderance and may be able to perceive properties of surrounding objects. Some objects of the world are animate while others are not. Examples of animate objects include dragons, hornets, and snails while examples of inanimate objects include first-aid-stations and fruit. Some animate objects will harm the agent while others a just a nuisance. Some inanimate objects will affect the agent's strength and damage attributes. While an agent is theoretically capable of epic feats such as slaying a dragon with a flower and saving a princess who is under the spell of a witch, the sample agent, Bumble, spends his time wandering around eating and collecting things and running away from other animate objects.

For my final project, I implemented Py-World, an environment simulator for artificial agents in the Python programming language using the Pygame library. The environment is fully observable in that the entire world is displayed as a grid of cells and the agents inside of it have access to the complete state of the environment and known in that the effects of actions are known and can be relied upon. The environment is multi-agent as there are two goal-based agents as well as reoccurring unintelligent agents. The existence and movement of the unintelligent agents as well as the semi-random spawning of items make the environment dynamic. Lastly, agents' attributes and locations change over time causing them to perform different actions depending on the current state, but there are a finite number of states and actions making the environment deterministic, sequential, and discrete.

I have developed three different types of agents which populate Py-World: two goal-based agents, and one unintelligent agent type. The two goal-based agents implement the Knight and Rogue classes commonly found in role playing games. The Knight prioritizes killing Orcs while the Rogue prioritizes collecting Gold. The Orcs which are found in the world are unintelligent and simply wander around.

The main game loop gives each actor a ply which includes making a priority-based decision (in the case of the goal-based agents), movement, and action. The game runs until both of the goal-based agents have died due to either their health or stamina reaching zero. The following sections will cover the environment, items, the agents and their decision making and navigation processes, and concluding remarks.

2. The Environment

The Py-World environment is composed of a 30 x 20 cell map which contains grassy fields, trees, and mud pits. Each cell is implemented as a Tile class which has a stamina cost for passing over it, a 16x16 pixel .png image, and a boolean which determines whether a tile can be traversed. While the majority of the map consists of grassy Tiles which have a stamina cost of one, there are also muddy Tiles which have a stamina cost of four. The tree Tiles which make up the perimeter of the map and are also found scattered throughout the environment have a stamina cost of 1,000 and cannot ever be traversed. The trees around the perimeter keep the unintelligent agents from leaving the map. The reason for having both a stamina cost and a traversal boolean will be explained in the section on actor navigation.

The actors and items in the world are drawn on top of the cell corresponding to their x and y coordinates. In addition, there is a message box off to the right of the map which displays messages when actors die, lose or regain health, pick up items, etc.

3. Items

Py-World has a single Item class with three child classes: Gold, Fruit, and Heart. Each Item class has a 16x16 .png and an x and y coordinate on the map. Only the two goal-based agents can pick up items. Items are picked up when either the Rogue or the Knight walk over them. The unintelligent Orcs simply pass over them. If a pile of gold is picked up, the agent's collection of gold increases by one. If a Fruit is picked up, the agent's stamina is increased by half, but will not exceed full capacity. If a Heart is picked up, the agent's health is increased by half, but will not exceed full capacity.

When the game is initialized, three Gold objects are created, three Fruit objects are created, and two Heart objects are created. Only two Gold, two Fruit, and one Heart are placed randomly on the map, however. The remaining objects are initialized at the coordinates (-1,-1) where they are out of play, not drawn, and cannot be interacted with. In order to keep the game going, a randomly selected out of play item is randomly placed back on the map every ten game loops. Additionally, if an Orc is killed and there is at least one item out of play, a randomly selected item will be dropped as loot.

4. Agents

4.1 Agents and their Priorities

The Actor class serves as the parent for the Knight, Rogue, and Orc classes. An actor has x and y coordinates on the map, a 16x16 pixel .png sprite, attack, defense, hp, stamina, gold collected, and kills stats, a name, and a priority. When the actors are drawn on the map, their hp and stamina stats are displayed as health and stamina bars drawn above them. One thing to note, though, is that the stamina stat is not used for Orc actors as they just wander around and would die too quickly. They are considered more as part of the environment.

When the game is initialized, three Orcs are created, one Rogue is created, and one Knight is created. The Knight and Rogue are randomly placed on the map, but only two of the three Orcs are placed on the map. The third Orc is initialized at the coordinates (-1,-1), is not drawn, is out of play, and cannot interact with anything nor be interacted with. To keep things interesting, a dead Orc, if there is one, is respawned every fifteen game loops.

The Knight has an attack power of four (which kills an Orc in two hits), a defense stat of one (the Knight requires eight hits to be killed), an initial priority of "killOrc," and his name is "Knight." The Rogue has an attack power of two (which kills an Orc in four hits), a defense stat of zero (the Rogue requires only two hits to be killed), an initial priority of "gold," and her name is "Rogue." The Orcs have an attack power of two, a defense stat of zero, no priority (they just wander around), and are all named "Orc." All agents have an hp stat of eight and the Knight and Rogue have a stamina stat of 128.

If an actor tries to move into a cell which is occupied by another actor, they will attack that actor, exceptions being that Orcs do not attack Orcs and the Knight and Rogue do not attack each other. The actor that is attacked will have their hp decremented by the attacker's attack power minus their own defense stat. For the goal-based agents, attacking an Orc costs four stamina. If an actor’s health reaches zero, they die and are placed at the coordinates (-1,-1) where they are not drawn, are out of play, and cannot interact with anything nor be interacted with. If the Rogue or the Knight die, they cannot be revived.

The Knight and the Rogue decide what to do based on a priority system. If their stamina and health are above fifty percent, their priority is to kill Orcs and collect Gold, respectively. If their stamina or health drop below fifty percent, they will reprioritize and seek out Fruit or Heart, with priority in that order. If there are no Gold in play, the Rogue will wander around randomly and if there are no Orcs in play, the Knight will wander around randomly.

4.2 Agent Navigation

The Orcs wander around randomly all the time. They have a one in five chance of moving up, down, left, right, or standing still every time they receive a ply. They are kept inside of the game world by the perimeter of trees because of the traversal boolean. For this reason, they also do not walk through the trees spread around the inside of the map. They are unaffected by the muddy tiles.

Depending on their set priority, the two goal-based agents, the Knight and the Rogue, will move towards either the nearest Orc, Gold, Heart, or Fruit. During the process by which they decide how to do so, they have access to the entire world. A limited “field of view” was not implemented. Path finding is done with the A* algorithm. The Knight and the Rogue will avoid the muddy tiles (stamina cost = 4) in preference of the grassy tiles (stamina cost = 1) and will never walk through a tree (stamina cost = 1000) on their way to their goal. They will walk through the mud, however, if it is the optimal route.

Different Python implementations for A* can be found on the internet, but I wanted to implement one myself. Mine is context specific, functional, and original, if a bit of a bodge. My implementation for A* utilizes a search Node class which roughly corresponds to a cell in the map grid and has x and y coordinates, a parent Node, a cost value (the stamina cost of the cell to be traversed plus the cost value of the Node's parent), a heuristic value (Manhattan distance from the goal cell), and an F value (cost value plus heuristic value). The Node class has a custom comparison function that the priority queue uses to give priority to Nodes with smaller F values.

I implemented A* following the pseudo code in the Chapter 3 lecture notes. Because of my Node implementation, I ended up having to use two separate frontiers. Part of this was when I would check if children Nodes were already in the frontier or the explored set, I was checking for object memory location equality and not state equality. The other part was that in the Python queue.PriorityQueue, queued objects cannot be removed specifically. They must be popped in order to be removed from the queue. I ended up using a queue.PriorityQueue frontier for storing Nodes, a list frontier for storing x and y coordinate pairs, and an explored set which stores x and y coordinate pairs. This is not an optimal implementation by an stretch, but it works as intended.

In Pygame, coordinates begin in the top left corner (0,0) with x increasing as you move to the right and y increasing as you move down. The bottom right cell of the map is at coordinate (29,19), for example. My A* implementation takes the current x and y coordinates of the actor and the target x and y coordinates, finds the optimal path to the target coordinates, and then traces the path back through the Nodes' parent attributes in order to return the relative coordinates of the move to make: up (0,-1), down (0,1), left (-1,0), or right (1,0) as those are the moves an actor can make and an actor can only move one cell at a time.

5. Concluding Remarks

Overall, this was a rewarding project and I think that the deliverable is pretty neat. I was able to take some of the information learned in class and apply it, essentially, to the domain of game development, which I knew nothing about beforehand. Certainly, the biggest challenge was learning the aspects of Pygame that I needed to implement the project, but significant thought and effort also went into the design and implementation of the world, actors, items, and their interaction. Implementing A* in this specific context and without taking something off of the shelf was also challenge.

Ideas that I have in regard to expanding this project include, first and foremost, refactoring and cleaning up the codebase, limiting the field of view of the goal-based agents, and implementing another goal-based agent which adheres to the Healer class common in role playing games whose main priority would be to identify other goal-based agents who are low on health and make contact with them in order to heal them. Additionally, as things are now, the environment is not harsh enough. The Fruit and Heart spawns are too frequent, the Orcs do not deal enough damage, and the Orcs have no intelligence. Since getting close to the final implementation, I have yet to see both the Rogue and the Knight be killed. The game may be more interesting if it were a bit better balanced.

* Accreditation for the game assets that I did not create myself can be found near the top of the source code.