PacMan Simulator

Vyom Srivastava
Department of Computer Science
vs14490@uga.edu

Sharmin Pathan
Department of Computer Science
sp59764@uga.edu

Abstract

Computer games or video games include a high degree of user control over the gaming console / hardware and software used and generally a greater capacity of input, processing and, output. Computer games are usually built around Artificial Intelligence. In this project, we have built a simulation of the PacMan game. This simulator performs a perfect play. A perfect play is the one where the PacMan achieves the maximum possible score (by eating every possible dot, power pellet) without losing a single life. We have demonstrated one of the most popular path finding algorithms i.e. A* for determining the path for PacMan and for the ghosts to chase PacMan.

1 Introduction

The first PacMan game was first released in Japan in May 1980 by a Japanese video game designer Toru Iwatani. The player plays as the PacMan moving through a maze eating pac-dots. On finishing the pac-dots, Pac-Man is taken to the next stage. Four enemies (Blinky, Pinky, Inky, and Clyde) roam around the maze, to catch the PacMan. PacMan loses his life on encountering the enemies. Player loses the game when all his lives are lost.

We have built a simulation of the PacMan game. The game environment highly resembles the Nintendo version of PacMan. No player input is taken. Our algorithm fed into the game, finds the best suitable path for the PacMan to make its way through the enemies while eating all the pac-dots. The difficulty has been divided into two levels. The walls act as obstacles.

1.2 Our Approach

Here, we build an algorithm (A*) that finds the best path for the PacMan to finish off the pac-dots and to escape its enemies. The algorithm takes care to avoid enemies and walls (obstacles) that appear randomly. The functioning is similar to the normal PacMan game. The simulation provides an ideal approach to finish off the game.

The entire game is framed as a matrix with its elements being pac-dots (path) and wall (obstacles). The enemies appear as per their behavior and the PacMan makes his way away from them. On

encountering an enemy in front of it, it flips its direction. It looks out for the remaining pac-dots and finds a suitable way to finish them.

We've used Unity game engine to develop the simulation environment.

1.3 Problem Statement

To build a simulation of the Nintendo version of the PacMan game and demonstrate a perfect play using A* path finding algorithm.

2 Preliminaries

The system was up with the following to implement the artwork and required gameplay setup.

Unity was used for game development. The graphics folder contains the images and artwork required for the PacMan, ghosts, pellets, power pellets and tilesets. The tileset was then broken to identify the edges and corners and design a maze accordingly. These tiles act as obstacles (walls of the maze) beyond which the PacMan and ghost cannot move.

The animations folder contains the animations involved in the movement of the PacMan and ghosts. The usual opening and closing of PacMan's mouth and sliding throughout the maze for both PacMan and ghosts is implemented here. The pellets disappear as soon as the PacMan consumes them.

3 Gameplay

PacMan can move freely throughout the maze. It cannot move through the walls. Thus, the walls act as obstacles. On collision with a wall, it flips its direction and finds the best possible path towards the remaining pac-dots. Whenever PacMan occupies the same tile as an enemy, it is considered to have collided with the ghost and PacMan loses a life. When all pac-dots are eaten, the game advances to the next level. The power pellets provide the PacMan with a temporary ability to eat the enemies.

The game has four enemies, namely Blinky, Pinky, Inky, and Clyde. These enemies (ghosts) roam the maze, trying to catch PacMan. The enemies appear to move about randomly to the player, but their behavior is strictly deterministic and they try to chase PacMan. This can be used by the player as an advantage.

3.1 Perfect Play

The artwork is in the Graphics folder under Assets. Scripts include the algorithm for Path Finding and behaviors of the PacMan and enemies. We have used A* algorithm that finds the shortest

distance for the PacMan to finish the pac-dots. However, this path is continuously updated since PacMan might collide with a wall or is about to have collisions with an enemy.

On colliding with a wall, the direction of PacMan is flipped and the path finding algorithm now updates the path to the next available shortest path in order to finish the remaining pac-dots. If the PacMan is about to make a collision with any of the enemies, it again flips its direction and the path finding algorithm computes a new path towards the remaining pac-dots.

4 Game Design

Since all of the enemies chase PacMan and have a similar behavior, we have demonstrated this behavior using a single enemy. Since the main idea was to implement a perfect play which involves finishing the pacdots, we have focused more on the path finding for the PacMan to finish the pacdots whilst escaping from the enemies.

4.1 Work Distribution

The artwork of maze and arrangement of the pacdots and implementation of A* algorithm is implemented by Vyom & Sharmin. Behaviors are handled as follows: Vyom managed the working of PacMan, and Sharmin worked with the enemy behavior.

5 Path Finding

A* is the heart of path finding algorithms. It is basically used in game development and is a combination of Dijkstra's algorithm and best-first search. Although, it's quite simple. In a nutshell, A* generates possibilities of neighboring nodes towards the destination and selects the one with the least projected cost. It has a set of open nodes and closed nodes. The lowest cost is computed amongst the set of open nodes. The associated cost function is,

$$f(x) = h(x) + g(x)$$

where, g(x) is the cost from the current node to its neighbor node and h(x) is the distance from the neighbor node to the destination node. The path that returns the lowest value for the cost function is selected.

The neighbor nodes that cannot be traversed and the ones which are already traversed are put into the closed set. The current node is saved as the parent node after it progresses on the selected path and is removed from the open set. This serves the purpose of backtracking.

For the PacMan, we have every corner as the node. We compute the cost associated with all the neighboring nodes to the PacMan's current node and select the one with the least cost. If there happens a situation such that pacDots from all the neighboring nodes have been eaten, we look for the entire maze and select a node which has pacDots left to be eaten and from amongst these nodes, we select the one with the least distance.

For the ghosts, we have two modes, namely scatter and chase. For the initial 7 seconds, the ghosts are in scatter mode i.e. they move randomly throughout the maze and for the next 20 seconds they switch to the chase mode where they follow PacMan. They again switch to the scatter mode. This switching happens four times and in the fifth iteration, the ghosts chase PacMan.

5.1 Astar Algorithm

```
Add the start node to OPEN                    //OPEN is the set of nodes to be evaluated
loop
    current = node in OPEN with the lowest f_cost
    remove current from OPEN
    add current to CLOSED
    if current is the target node             //path has been found
        return
    foreach neighbor of the current node
        if neighbor is not traversable or neighbor is in CLOSED
            skip to the next neighbor
        if new path to neighbor is shorter OR neighbor is not in OPEN
            set f_cost of neighbor
            set parent of neighbor to current
            if neighbor is not in OPEN
                add neighbor to OPEN
```

6 Concluding Remarks

A* works well for a static design. Since it has a deterministic set of nodes. In our game, we tagged every corner as a node. By adding additional walls, there were more nodes that had to be considered and A* wouldn't count these new corners as nodes. This failure happened since A* was unaware of these newly added nodes as they remain to be untagged. It's not possible to tag new corners as nodes during the gameplay. So, we had issues working with A* in a dynamic way.

References

[1] https://www.youtube.com/watch?v=tjxKxZsofdk
How to make a game like PacMan in Unity 5 – The Weekly Coder

[2] http://homepages.abdn.ac.uk/f.guerin/pages/teaching/CS1013/practicals/aStarTutorial.htm
A* pathfinding for Beginners – Patrick Lester

[3] http://web.mit.edu/eranki/www/tutorials/search/
Pathfinding using A* - Rajiv Eranki