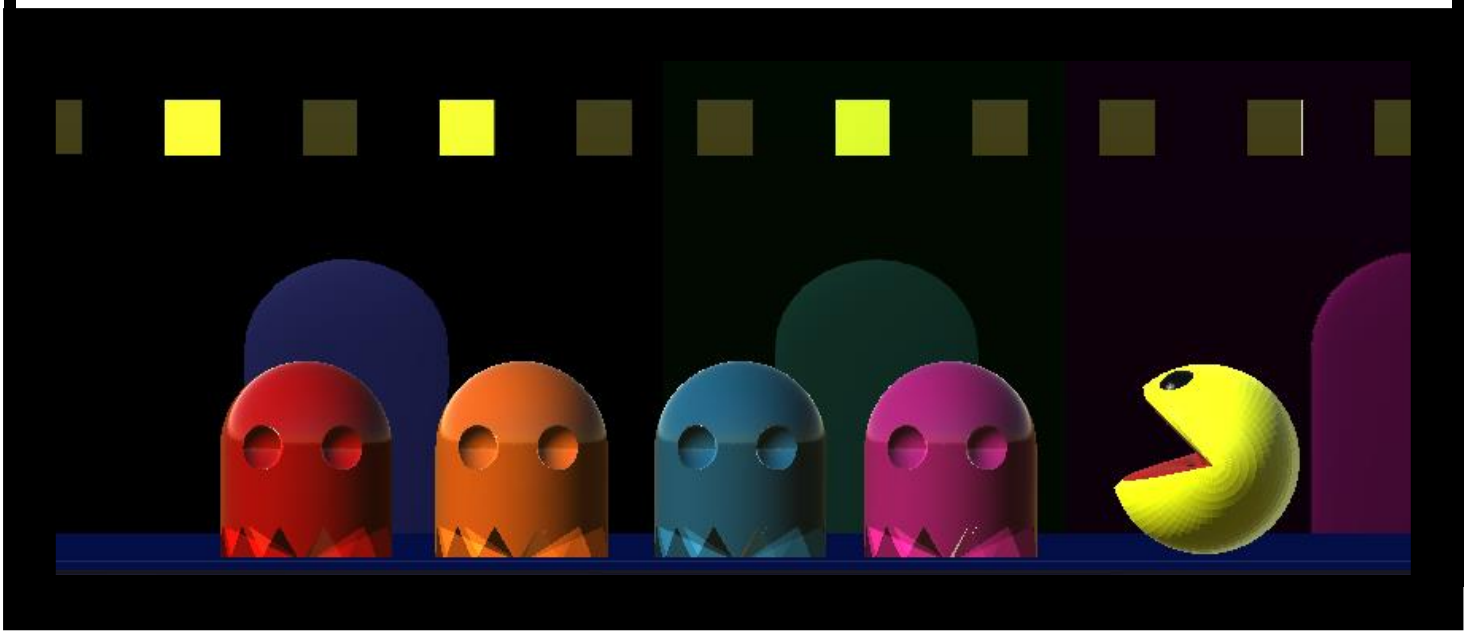
ICS2211 – Game AI Project

Pacman

Nikolai Debono

Michelle Falzon



Abstract

This project involves the creation of a virtual reality game that implements a form of artificial intelligence. A 3D version of Pacman was developed for Google Cardboard, and the ghosts' movements were programmed using the A* pathfinding algorithm.

Narrative

The original Pacman was created by Japanese video game designer Toru Iwatam in 1980. This game instantly became a huge success. Other games at the time involved the player interacting with objects such as in Pong. Pacman was innovative as it was the first game to portray a character which the user could control. Pacman's name originates from the Japanese term paku-paku which refers to the sound of munching. In fact, the idea of Pacman came to Iwatam while eating pizza as the shape of Pacman is similar to a pizza with missing slices.

The player controls Pacman through a maze, eating pellets and avoiding enemies. The level is cleared by eating all the pellets. There are 4 ghost enemies; Blinky, Inky, Pinky and Clyde (red, blue, pink, orange respectively). These ghosts each have a different movement pattern. If Pacman collides with one of them, he dies. However, if Pacman eats one the power pills, the ghosts become vulnerable and Pacman may eat them for more points. Fruit scattered around the map may also be ingested for more points.

The red ghost, Blinky, uses the simplest movement algorithm; it simply chases Pacman. Pinky, the pink ghost, tries to intercept Pacman by predicting his future position. Inky, the blue ghost, relies on a point in front of Pacman's position and the vector from Blinky to this point. This vector is doubled to reveal the location to which the blue ghost will travel. Lastly, Clyde (orange ghost) has two modes. When Pacman is in far he chases Pacman as Blinky does, otherwise if Pacman is in close proximity he moves to his scatter position (the position he takes when pacman eats a pill).

Furthermore, Pacman made such a hit as it was one of the first games aimed at both males and female audiences. Toru Iwatam ensured that the game was non-violent and that the characters were cute in order to attract more females. As a result, the game won the Guinness World Record for the most successful coin operated game in 1999. All sorts of Pacman related merchandise began to be made and sold. Till today, various versions of Pacman are being made with bigger mazes, better graphics and different objectives. One well known version of the game is Ms. Pacman which was released only a year after Pacman. In this version, the player controlled Ms.Pacman and had more mazes, faster gameplay and moving fruit. Lastly, Pac-Man is also credited for laying the foundations for the stealth game genre, as it emphasized avoiding enemies rather than fighting them.

We wanted to bring back this classic and retro game but in a new dimension and thus we decided to develop Pacman 3D. Through this project, we wanted to rekindle the interest and nostalgia in the older generation of gamers while introducing a more modern version to the younger generations.

Implementation of Artificial Intelligence

The ghost movement was implemented using the A* Pathfinding algorithm.

The A* system mainly uses the following three scripts:

- Node.cs
- Grid.cs
- Pathfinding.cs

Pathfinding requires the unity world to be logically divided into a grid of nodes. The path returned by the A* algorithm will be an ordered list of these nodes.

Node.cs

A node is made up of:

- gridX,gridY
- worldPosition
- walkable
- seekerdistance
- targetdistance
- parent

A node is represented using the Node.cs class. Every node is made up of a *gridX* and a *gridY* variable representing the x and y position relative to the logical grid. The centre of the node represents the node's world position. The in-game world can have obstacles. This is represented in the Node class as a Boolean variable that denotes whether or not a node can be walked on, i.e. whether or not that node can be part of a path. When a path is being calculated, a node stores the distance from the node to the seeker's starting position as *seekerdistance,* and the distance from the node to target as *targetdistance. Targetdistance* is used as a heuristic value in the A* path finder. When building the path, the parent attribute holds the node that last linked to this node.

Grid.cs

The Grid.cs script creates and maintains the logical grid of nodes. The grid is formed based on the inputted world size and node radius. The function *CreateGrid*() creates the game's logical grid. It first finds the world's bottom left point vector. Then every grid node is initialized using the current grid coordinates. The world position vector of the node is found using the world's bottom left point and it is also used as one of the constructor inputs. The function also checks whether each node is walkable by creating a sphere collider at the specified world point. If the collider touches an object in the *unwalkable* layer, the node is marked as unwalkable.

The function *GetNeighbours()* returns the neighbouring 8 nodes of a specific node.

The function *NodeFromWorldPoint(worldPosition)* converts a position vector worldPosition into one of the nodes in the grid. The general idea is to find the node's coordinates in terms of a percentage of the grid's world size. The world coordinates can be negative (since grid starts at (0,0,0)). However negative values can't be translated directly into an array's index since arrays indices start at 0. Therefore half the grid's world size is added to the dimensions to make sure they get translated into positive percentage values. The percentage values are then converted to values between 0 and 1. The x and y indices are calculated by multiplying the percentage and the grid's logical size for each dimension.

Pathfinding.cs

The pathfinding.cs script is used to find the shortest path between two grid nodes by implementing the A* algorithm. This is done through the *FindPath* function which accepts two parameters of type vector3, a starting position (*startPos*), and a target position (*targetPos*). The latter two represent world positions and are first converted to the relevant grid nodes (startNode, targetNode). Furthermore, the A* algorithm requires a list of nodes called *openSet* and a set of nodes called the *closedSet*. The openSet holds all the nodes that are waiting to be evaluated. This evaluation consists of calculating the distance between the current node and the target point (heuristic distance) and the distance between the node and the starting point. The two values are then added to find the total cost. Once evaluated, the node with the least cost is chosen. The closedSet holds all nodes that have already been checked.

Pseudocode of the A* algorithm in FindPath()

```
Add the starting node to the open set
while(openSet has nodes)
{
        Find the node with the shortest total cost by looping through the openSet. If during the comparisons the total costs are the same,
                the node with the shortest total path is chosen based on the shortest heuristic (target distance)

        Remove the chosen node from the openSet and add it to the closedSet

        if( chosen node == targetNode)
        {
                return findPath(startNode,targetNode)
        }

        for each neighbour of the current node
        {
                if(neighbour is not walkable OR neighbour was visited (meaning neighbour is in closedSet))
                {
                        skip to next neighbour
                }

                calculate the new seeker distance by adding the node's old seeker distance to the distance
                        between the currentNode and the current neighbour node by calling method getDistance()

                if the node has not been visited yet or the newly calculated seeker distance is smaller than the current seeker distance
                {
                        set neighbour's seeker distance to the calculated seeker distance
                        set neighbour's target distance to distance between neighbourNode and targetNode)
                        set neighbour's parent to the currentNode.

                        if(the openSet doesnt ontain neighbour) add neighbour to openSet
                }
        }
}
```

The function *GetDistance* is used to find the distance between two nodes in the grid. The vertical or horizontal distance of a node is 1 whereas the diagonal distance is 1.4 (For easier manipulation these values are multiplied by 10 and thus are 10 and 14 respectively). The distance is found by calculating the total diagonal distance required + the remaining horizontal distance.

The function *returnPath*() is used to generate the final path list. Starting from the end node, the current node is added to a list and its parent is now the current node. IF the current node is equal to the seeker node, the path list is reversed and returned.

Movement Scripts

RedGhostMovement.cs

The aim of this script is to make the red ghost follow Pacman by moving to Pacman's current position. Throughout the game, the red ghost's targetNode will be set to Pacman's current position. Once the targetNode is set, the PathFinding script is used to return a path as a list of nodes. For each node in the path, the corresponding world point of the node is obtained and the red ghost moves to that point using a defined step size. During the time when the ghosts are vulnerable (meaning Pacman has eaten a pill), the red ghost's target position will alternate between his home corner and the centre of the map (0,0,0).

BlueGhostMovement.cs

The blue ghost's movement depends on pacman's and the red ghost's position. The blue ghost's targetPosition is a fixed position in front of pacman's. The vector between the red ghost's position and this target position is calculated. This vector is then doubled and used as the new target position. The movement is handled similarly to the red ghost's movement.

PinkGhostMovement.cs

The pink ghost predicts pacman's future path based on his current direction and tries to intercept him. The value of the predicted position is calculated through the vector addition of Pacman's position and his movement direction. The predicted position is used as the target node in the pathfinding algorithm. The ghost movement is then handled similarly to the other ghost's movement.

OrangeGhostMovement.cs

The orange ghost's constantly switches between two modes based on Pacman's proximity to the orange ghost. The first mode is active when the ghost's distance from Pacman is larger than 8 grid nodes. In this mode the ghost follows Pacman directly, much like the red ghost's movement. The second mode is triggered if the orange ghost is less than 8 nodes way from Pacman. In this mode the ghost will move towards his home position until he is more than 8 nodes away from Pacman.

Conclusion and Future Enhancements

The game may be further improved by adjusting the A* path finding algorithm to work with a Heap data structure. Since a path is currently calculated every frame, even a relatively small performance improvement would be noticeable in the gameplay. The game could be further improved by diminishing the amount of paths generated per second.

In order to provide better gameplay, a proximity marker for the ghosts can also be added in order to warn the player when the ghosts are close behind Pacman and mark their position. The magnetic button may be used to reverse rather than forcing the user to turn around 360°. Ideally, more levels will be added with varying difficulties.

Lastly, a short introduction and menu could be displayed before the game starts. The top scores may be logged and displayed to the user to create a sense of competitiveness.

References

http://pacman.com/en/pac-man-history

https://developers.google.com/cardboard/unity/guide

http://www.redblobgames.com/pathfinding/a-star/introduction.html

https://www.yobi3d.com/v/ZjubtKj59y/Ghost.stl

https://www.cgtrader.com/free-3d-models/food/fruit/green-apples

https://github.com/trishume/CharlesGameScripts/blob/master/Player/CharacterControls.cs