**3CB113 | Artificial Intelligence for Games Assessment**

**Introduction.**

For this project I aimed to create a recreation of Pacman using AI elements. Pacman was a 2d maze exploring arcade game developed and released by Namco in 1980. The game featured a system for creating new maze level for the player to try and complete, as well as four enemy ‘ghosts’ that the player had to avoid, each with different behaviour depending on the game state.

In the original game, the enemy ghosts had three behaviours, Chase shatter and frightened. The chase behaviour has the ghost move to position relative to the player. The scatter behaviour had the ghosts target as specific corner of the map and move there, to give the player some room to breathe. The frightened behaviour had the ghosts move as far away from the player as they could. To implement these behaviours in my game, I chose to use Finite state machine technique to implement them. I chose to use this method for the ghosts, because it is a method that closely resembles the behaviour of the original game, while also allowing a greater degree of control over the behaviour of specific ghosts. This method is also computationally light as the ghosts only change states every few seconds, meaning it wouldn’t impact the performance of the game much.

The original Pacman game had 256 levels; I couldn’t find any documentation on how the original game created or stored levels. By using procedural content generation and a maze generation algorithm, I could generate a randomly create maze and fill it with pellets, fruits, and other game objects that the player would need to play a game of Pacman. I chose to use this method as it would take up less file space than designing each level from scratch and because I’d be using integer math for the generation, it would be much faster than loading levels from existing data. Using randomly generating mazes would also inform the design decisions of the Finite state machines of the ghosts as they wouldn’t be able to traverse random levels properly, they could for pre-generated one.

**Implementation Report.**

**Finite State Machine.**

The ghost’s behaviours are operated by a finite state machine, implemented in the FSM class. This model for agent behaviour works by changing the state of a game object whenever a specific condition is met. For example, whenever the player eats a power pellet, all active ghosts turn blue and switch to their flee behaviour. Finite state machines are an abstract model which meant that you can create specific behaviour for specific agents and simply overwrite them in the finite state machine class and it would still function correctly. I chose to use the finite state machine because of this abstraction because it meant that I could alter the behaviours of the ghosts very easily and fine tune them to work better within the game. See the ‘GhostV2’ class and ‘FSM’ class for more details.

**Movement:**

The movement in Pacman is controlled by nodes that spawn wherever a there is an intersection in the maze. This works the same in my game. Whenever one of the ghosts collides with a node it will perform a check to confirm which direction will take it closer to its target and will either A) move in that direction or B) move away from that direction based on its state. This will be explained further when talking about specific states. The only difference between these nodes in my game and the original is that in the original the ghosts were not allowed to turn around and head back in the direction they came from in the original. I chose to allow them to turn around because I felt that it added to the challenge of the game.

Figure 1: State Diagram from Pacman

Diagram, schematic

Description automatically generated

Figure 2. State diagram from my game.

Diagram

Description automatically generated

**Chase State:**

Figure 3. The ghost’s target’s in chase mode.

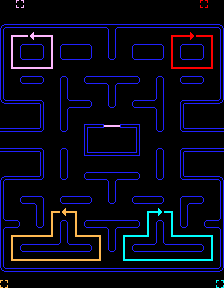
Qr code

Description automatically generated

For the chase state, the ghosts follow a target relative to the player. The pink ghost’s target is four tiles directly in front of where the player is facing. The red ghost’s target follows the player directly. The blue ghost’s target is two tiles in front of the player, then calculate how far the red ghost is from that tile and double that to determine where the blue ghost is heading. The orange ghost works differently to the others in that whenever it is more than 8 tiles away from the player, it will act exactly like the red ghost, but once it gets within 8 tiles, it would suddenly shift to using it’s scattering behaviour. This is how the chase behaviour was implemented in the original game, and I chose to implement it in the same way. From this state, the ghost can enter two other states: the flee state which is activated when the player eats a power pellet or the scatter state which triggers after the ghost has been chasing the player for seven seconds. Read the ‘ChaseBehaviour’ class and the target classes for more insight.

**Scatter State:**

Figure 4. Ghosts scattering behaviour, as it behaved in the original game. (Chad Birch 2010)



For the scatter state, the ghosts also follow a target that would take each ghost to their own respective corner of the maze. In the original game, these targets would sit outside the corners of the maze. As seen in figure 4(Chad Birch, 2010) the ghosts would move in loops to try and reach this target and because they couldn’t double back on themselves to try and get back to the target. However, due to the fact the ghosts in my implementation could double back on themselves, I had to take another approach to creating these loops around the maze. Initially, The implementation I chose to implement was to simply have the ghosts move randomly, but this proved to make the game too challenge as the ghosts became impossible to predict in this state. To solve this problem, I created a box collider in each corner, and used it to generate a series of points in a ‘I’ shape; this modification to the behaviour made the ghost’s separate into respective corners, which also making their movement much easier for the player to understand. This behaviour can transition into two other behaviours: after ten seconds it will transition into the chase behaviour, but if a power pellet is eaten, It will transition into the flee behaviour. Read the ‘ScatterBehaviour’ class and the target classes for more insight.

Figure 5. Example of scatter path used by ghosts in my implementation.

A picture containing text, clock

Description automatically generated

**Flee State:**

The flee state is different to all the other states in the game. Firstly, the ghost would change to using a different sprite. As for movement, rather than moving toward a target, the ghosts move away from one, specifically move away from the player’s current location. I chose to implement this in the same way as the original game, by having the ghost check which direction would lead them farthest away from the player and having them go in that direction. My implementation doesn’t behave exactly like the original game due to the fact that ghosts can double back on themselves, meaning when fleeing from the player, they will just run into a corner and stay there as long as the player doesn’t get closer. In the original game, if the ghost was eaten it would walk back to the ghost house and then respawn in the scatter state and if it wasn’t it would simply go back to the scatter state. In my implementation, if the ghost was eaten, it would teleport back to the ghost house and go into the idle state for a few seconds before going into the house state and if it wasn’t eaten it would go back to the scatter state. A ghost in the idle or home states cannot enter this state. Read the ‘FleeBehaviour’ class for more insight.

**States exclusive to my implementation:**

The states I added to my implementation of the game were the House state and the Idle state. The house state is the state that the ghost enemies start the game in. During this state, the ghost’s will move up and down in the ghost house for x number of seconds before leaving. This behaviour works exactly like it does in the start of Pacman, except in my implementation it is a unique state that the ghost can be in. The home state always transitions into the scatter state. Read the ‘HomeBehaviour’ class for more insight.

The idle state is a transitional state that the ghosts enter into when they have been eaten by the player. In this state, the ghost has no behaviour and simply teleports back to its starting position waits for 2 seconds before. This behaviour is meant to replace the ghost’s eyes running away from the player after they eaten in the original game. I chose to implement in this fashion because it was easier to implement than the ghost travelling back to its starting position. The behaviour always transitions into the Home state. Read the ‘IdleBehaviour’ class for more insight.

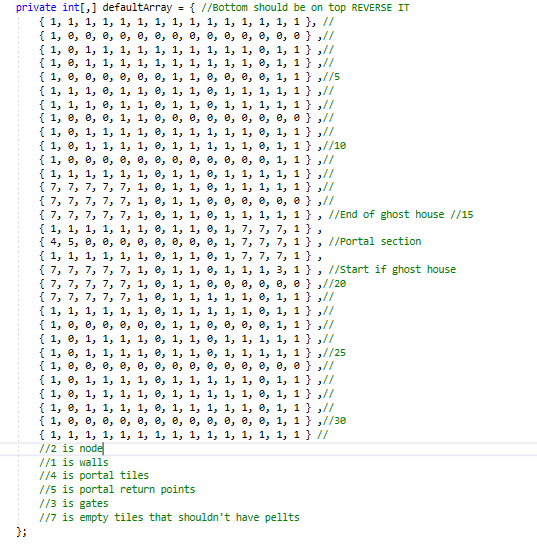
**Procedural content generation.**

In order to use procedural content generation to create new levels for the game, I had to first figure out how to represent the original maze and numerical values. The first decision I made for this was to use integer math when creating and loading levels, I chose this because integer math can be handled by single bits of information while floating point numbers are much slower, and that would cause a drastic drain on the performance of the game.

The next decision I had to make was how I would represent things in the data structure, I chose to represent empty space as the number 0. Then I hit a problem, how would I represent each different wall tile there were a total of nine distinct tiles that made up the walls of the maze – to simplify this problem I chose to simply represent all of the walls as the number 1 and combe back to the problem later. The number 2 was used for node tiles, but those are generated differently in the final game so they’re not in the array anymore. I chose to add a black tile that wouldn’t have any pellets as the number 7. The ghost house gate tile was unique tile and had to be represented differently so it was number 3. The portal tiles were number 4, and the portal return point was 5. I would use these numbers when generating new levels as well. Number 6 was reserved for tile that could be replaced by the level generator.

I stored half of the first level of the game as the variable ‘defaultArray’ in the class TileData. I only used half of the level as Pacman levels are mirrored vertically so I could save on disk space by only saving half in memory. The data was stored in the reverse order as the load level method generates from the bottom to the top, so that it would generate in the correct direction.

Figure 6. The data representation of the first level of the game.



**Maze Generation:**

In order to use the generate new levels for the game, I implemented a recursive algorithm. The way this algorithm works is simple but effective for the game. The first step of this generation was to create an array of integers that was half the size of the full maze. Then the algorithm would pick a random starting point from the maze and start generating a maze from there.

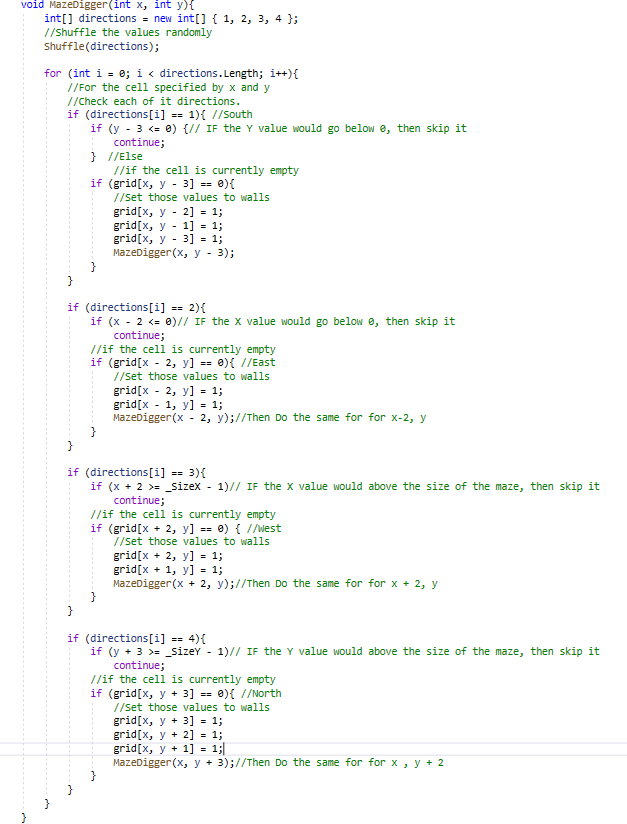
From the random starting position, the algorithm would pick a random direction, and check if going anywhere from that direction would be out of the bounds of the array; if it wasn’t out of bounds the algorithm would move x spaces and mark all of the cells between its old position and new position as walls, then would do the same for the other 3 directions. If it was travelling vertically, it would move three spaces and if it was moving horizontally, it would only move two. This vertical bias improved the visual appearance of the mazes substantially which is why it was implemented. The maze would start generating from each the new position which is what makes the algorithm recursive. See figure 8 for details.

Figure 7. A simple visual representation of the maze algorithm.

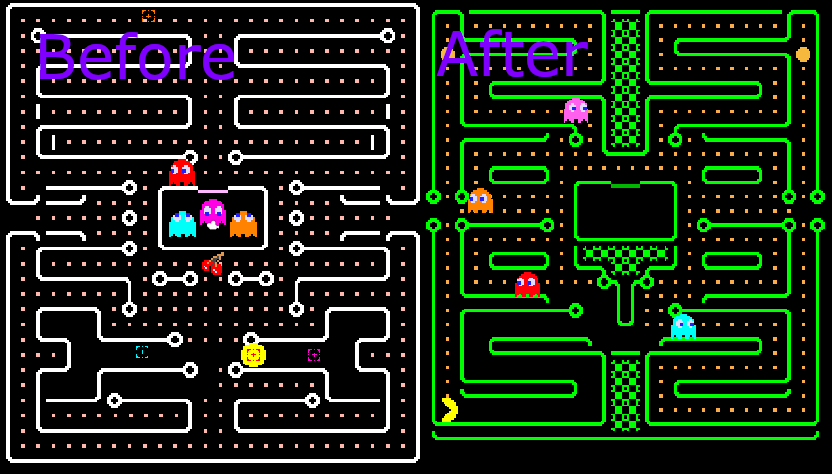
A picture containing text, crossword puzzle, shoji

Description automatically generated

Figure 8. The MazeDigger method from the MazeGenerator class.



There were a few issues with this method for generating mazes. The first issue I encountered was that maze would randomly generate with sections that were closed off but still had pellets generating inside of them, which meant that the game was unwinnable for the player. In order to solve this issue, I had the shuffle method used to choose random direction was changed to use a different randomization method, I also a made sure that the algorithm would only overwrite cells that were walls. The final change inverting the output of the maze generator because in it’s representation, empty space was 1 and walls were 0 and by inverting it, I could bring it in line with the representation of the rest of the maze and maze sure there were fewer block off areas.

Figure 9. Before and after of maze generation.

The second generation issue I encountered didn’t have some of the required components to be a pacman level, such as the ghost house and portals. The simple fix I implemented was to have permanent generation features in the game which included portals on the left and right. Another permanent feature was the ghost house in the middle because there would be nowhere for the ghosts to spawn otherwise.

A pathway that led pacman directly to where the ghosts spawn was a required feature. This generation feature were required because without them levels could generate with entire halves cut off from each other, and the ghosts wouldn’t be able to reach the player. These generation features can be seen in the ‘TileData’ class in the variable blankArray. Once the level had finished generating it would be mirrored to create a vertically symmetrical level like those in the original game.

**Visuals:**

The visual appearance of the maze was handled in a very simple way, in that a tile would check what its neighbours were and use that information to determine how it should look using the method GetTileVisual from the TileData class. For example, if the tile above and the tile to the left of a given tile were both walls, it would be replaced with a northwest corner tile.

As for the colours of the maze they were simply handled by changing the colour of the tile map using the ShiftColour method in the same class.

**Special tile Generation:**

Node tiles were generated by checking around a tile to make sure there were two empty tiles around it, and that those two empty tile did not form a straight line. Cross sections were allowed, but if the tile connected only by the north and south or east and west directions then the tile was node given a node. Nodes were generated on their own tile map to make sure they functioned separately of the maze. See the CheckIfNode method from the TileData class. While this implementation was fairly simple, it also did not impact performance due to the fact that the nodes are not rendered by the game at all.

The pellets, power pellets and fruit tiles were all generated on a separate tile map to prevent them from overwriting the maze. They were generated through the use of Perlin noise. This works pseudorandomly generating numbers between 0 and 1. When a tile is generated one of these random values is generated and if the value produced is greater than 0.1 it would be regular pellet tile, this is the most common tile that the player can interact with as it has a roughly 90% chance of spawning. If the value produced was under 0.1 but more than 0.05, then the tile would become a power pellet tile, which has a 5% chance of spawning for each tile. If the value is lower than 0.5, it will be a fruit tile which also has a 5% chance of spawning. You can see roughly how this works through the getPerlinValue method in the TileData class.

**Discussion.**

**Finite state machine.**

The goal of the finite state machine was to recreate the behaviour of ghosts as close to the original game as possible. I felt that I did this quite well, even if my recreation was not exact as I didn’t really understand how the original game used transitional behaviour like the eyes running back to the ghost house after a ghost has been eaten.

During development, I made several design decisions for the purpose of improving the player’s experience. Initially I chose to implement the scatter behaviour as random movement, but that felt too unpredictable and made the game almost unplayable, but with the current scatter behaviour. So, the approach of having a set path in a corner of the maze was developed to make the ghosts more predictable. One change to their behaviour I should definitely have implemented was preventing the ghosts from back-tracking as it made for complex behaviour in the original game and would have made the scatter behaviour much easier to implement as it could have been handled by a single transform rather than multiple like it was in my implementation.

As for the ghost’s movement, implementing A\* pathfinding would have made the ghosts more intelligent and make their navigation around the maze more efficient than it is in my build. This could be implemented by simply creating a list of nodes that would lead to the player’s current location and having the ghost’s following that. However, this pathfinding method could have made the AI too intelligent and made it almost impossible for the player to win.

Overall, I felt that my implementation was successful because it recreated the general behaviour of the ghosts. Making the ghost AI in this fashion was not unfeasible and took a fairly short amount of time to implement, which made it suitable for this project, though an investigation into other approaches may be necessary if I attempt this project again.

Another approach I could have taken for developing the ghost’s AI was goal-oriented action planning or GOAP. I could have used this approach to give the ghost’s specific goal like going to certain tiles or going to the player’s location, depending on current state of the game. For example, if the player has collected over 30 pellets, the red ghost could permanently switch to chasing them as its primary behaviour which could drastically increase the challenge of the game.

**Procedural generation.**

The goal of the procedural generation system was to create mazes for the game of pacman that could be played to completion. I felt that I did this quite well because the system I implemented produces levels that can be played but also has enough hardcoded safeguards to mean that almost all levels produced by this algorithm are completely beatable.

During development, I made several design decisions for the purpose of improving the player’s experience. For example, I implemented a direct path between the ghost house, the player’s starting position and the tunnels on the left and right. These direct paths allowed the player to always be in a situation where the ghosts can reach them, and they can reach the other side of the level.

Another generation decision I made for the player’s benefit was having the power pellets and fruit items randomly generate, this meant there was always a way for the player to increase their score and fight back against the ghosts. Another thing I did for the player’s experience was going out of my way to remove the enclose sections of the maze, it meant that the player would always be able to win the game which drastically improved the player’s overall experience.

One major tweak I would make to the PCG would be to modify the generation for the power pellets and the fruit objects so that they would spawn more consistently, and they would be in areas further away from the player. As it is currently implemented, the player can spawn on power pellets which uses them automatically so the player will have wasted one on accident.

Overall, I felt that my implementation was successful because it create playable mazes that were not impossible to beat. Making the PCG with the method shown was feasible in the timeframe given for implementation, though I would like to investigate another approach with the hopes of making mazes that are more like those in the original game.

Another approach I could have taken to level generation would be to take the approach described by Shaun Lebron(2012) who used a series of ‘Tetris’ blocks to represent the walls of the maze, adjusted their width and height to create more unique shapes and then used a buffer to generate the paths in-between them. This method would generate mazes that were styled perfectly to match the original game, however due to its more complicated nature it may have been intensive for the game to handle.

Figure 10. A finalized maze generated by the PCG system.

A screenshot of a computer screen

Description automatically generated with low confidence

**Evaluation.**

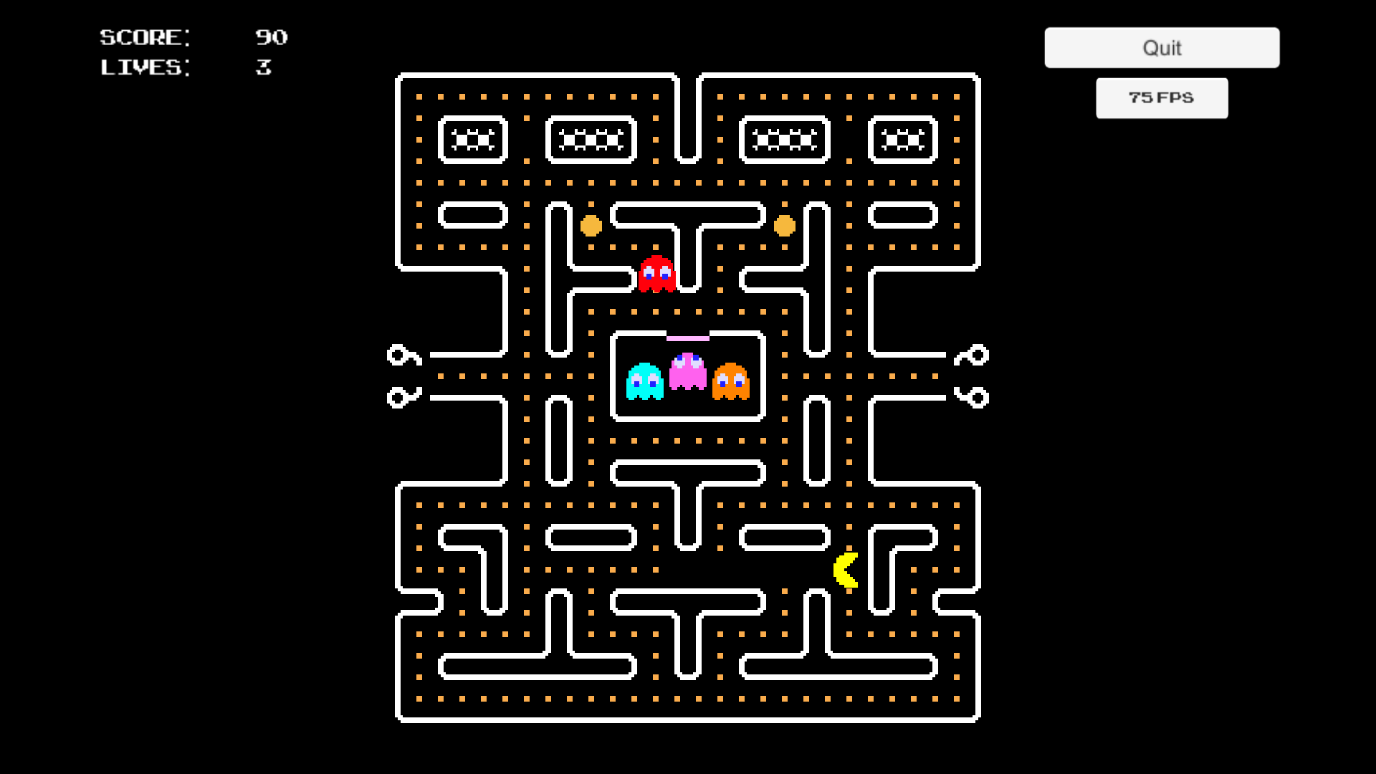
**Performance.**

The game objects used in the game only trigger when they collide with other objects. The ghost’s and the player only change their behaviour whenever they encounter a node until then they move in a set direction until they hit something. The objects act independent of other game objects like the game manager, and this means that they have no significant impact on the performance of the game. An example of my implementation doing this would be the ‘FruitManager’ class which was originally implemented to handle the updates for all of the fruit objects however this caused a significant impact on performance as they were all calling to the same object whenever one of them was eaten. To solve this problem, I made the fruit objects act independently of a manager class.

Another reason that this game performs very well was that the game was made in a 2D style meaning the game engine did not have to render nearly as much content, coupled with the fact that the levels were being generated through integer math, being handled by a modern computer and several levels could be rendered per second if necessary. To confirm the performance of the game, I wrote an FPS counter script, which calculated the value of 1 divided by Unity’s inbuilt unscaled delta time value. This produced an FPS value that was clamped between 0 and 99, then displayed it to a panel on the screen. The fps counter constantly showed that the windows build(I unfortunately cannot test the Linux build for the game but it’s performance should be similar) was running at 75 frames per second, after generating a new level this value would drop to 57 fps for a few milliseconds but would soon recover.

I could further optimize the game by introducing object pooling. This would mean that all of the game objects were pre-rendered and could be rendered by the game engine and then moved into position once they were needed. This would improve the performance of the game even further as the game objects could simply be loaded from memory and returned to it when needed rather than having to generate large amount of game objects whenever a new level was loaded.

Figure 11. The unity stats panel showing the FPS of the game.



//400

* Add an evaluation of the method used based on David’s lecture
* Performance.
* When do Ai’s need to trigger.
* Optimizations that could be made.

**Conclusion.**

//200

Sum up everything.

**References:**

Chad Birch(2010) Understanding Pac-Man Ghost Behavior. Available from <https://gameinternals.com/understanding-pac-man-ghost-behavior>. [Accessed 23rd March 2022]

Shaun Lebron(2012) Pac-Man Maze Generation. Available from: <https://shaunlebron.github.io/pacman-mazegen/> [Accessed 25th May 2022]