Computer Science H446 Coursework Project: Pac-Man Game

Candidate Number: 1419 Centre Number: 19268

Stefan Harris

Contents

1   Introduction

This project aims to develop a program serving as a recreation of the 1980 coin-op arcade game Pac-Man, using both the household high-level programming language Python and a 3rd party library to enable game development, Pygame. The project aims at creating a rather conservative recreation of the original. The core gameplay loop will be retained as much as possible by researching the original game's mechanics, namely the mechanics of the ghosts: the algorithms that dictate where they will move, where they will move to and how to react to key game play moments. Any features outside gameplay will be mostly cosmetic and or serves as a quality-of-life improvement for the player.

2   Analysis

2.1   Identifying the Problem (An Explanation of Pac-Man)

Pac-Man is a 1980 arcade game developed and published by Bandai Namco in Japan. And published by Midway Games in North America. Despite initially receiving lukewarm reception in Japan, the game showed both critical and commercial success outside the nation and the franchise remains one of the highest-grossing video game series of all time.

Development of Pac-man begun in 1979, directed by Toru Iwatani with a nine-man team. Iwatani noticed that most arcade games of the time appealed more to a masculine audience, with conflict being quite unanimous. Discontent with this, he aimed to create a game that would appeal to woman as well as men. A colourful art style and game play loop that would appeal to all ages was the idea.

The core gameplay revolves around Pac-Man and his goal, to eat all the pellets/dots in the maze without being caught by the pursuing ghosts. Pac-man can turn the tables by tactically eating the large dots known as power pellets. This will frighten the ghosts, making them vulnerable to Pac-Man eating them for bonus points. Difficulty increases the more levels the player completes, with the ghosts becoming more aggressive among other subtle changes, like power pellets no longer fully functioning.

To add variance to the gameplay, each ghost has their own unique behaviour to catch Pac-Man, giving the illusion that they are working together and making them seem smarter than they are. They also alternate between behaviour states during a level, namely chase mode –when they are pursuing Pac-Man, and scatter mode – when they flee to the maze's corners to give the player a breather.

Figure 1: The beginning of a game of Pac-Man.

2.2 Solving the Problem using Computational Methods

One of the reasons I chose Pac-Man for my coursework is because I believe it serves as a good example on how computational methods can be applied to a large project to make it more approachable.

2.2.1 Abstraction

Abstraction helps in reducing the amount of hardware resources taken up by the program, ensuring that performance is optimal on the range of target systems. An abstract model will serve as a representation of reality, only functioning in the ways it needs to.

Pac-Man is already an abstracted form of a chasing simulation. The game takes place on a 2D plane, yet still provides a good representation of a real-life chase scenario. The omission of a 3rd dimension results in far less lines of code, storage, and processing, a necessity when acknowledging the limited hardware of the time. All the entities are sprites consisting of only a couple colours and frames of animation, a drastic simplification of how the characters are depicted in the game's promotional material and arcade cabinet. The maze does not represent any real-life location and is represented by simple outlines for walls. The entities can only move in cardinal directions.

Figure 2: An original arcade cabinet. Note how the art on the side doesn't match the game's graphics, but more-so its bright colours.

With the source material already being abstracted, it is easy to find and implement the core mechanics into my project such as the player movement, ghost behaviour and ghost states while ignoring more superfluous aspects like cut-scenes between levels.

2.2.2 Decomposition

For complex problems like this, decomposition is invaluable for breaking it down into simpler, more manageable chunks that can be prioritised, developed, and tested individually. It makes life so much easier on the developer's end.

This solution is no exception, it can be decomposed into its core components.

For example, the core gameplay loop of Pac-Man can be decomposed into a series of entities interacting with each other: Pac-Man, pellets, ghosts, bonus fruit etc. Each entity can then be further decomposed into their core behaviours. So, for Pac-man that will be his movement, his eating and dying. For ghosts, it is their movement algorithm and behaviour states. Once the game is fully decomposed, the components can then be prioritised, such as the maze being implemented before the gameplay since the maze will house all the gameplay.

2.2.3 Logic, Branching and Iteration

These are pivotal for adding interactivity and dynamism to the solution. Without these the code is completely linear and with little to no meaningful interactivity,

which is not optimal for a game, a program that is supposed to react to user input.

Iteration is essential because it allows the game to repeat operations indefinitely, significantly reducing the number of lines in the code base. It also allows the game to run indefinitely via a game loop.

Logic is an integral part of the gameplay; the ghosts will use logical comparisons to decide what will be the next best direction to get themselves closer to Pac-Man or to decide if a direction is eligible to move in or to decide if it is time to switch to a different behaviour state.

Branching in the form of if/else statements must be used for the game to respond to user input and produce an action. The game can check for user inputs each frame using a series of if statements. For example, when the player presses the left movement key, Pac-Man will move left, otherwise he will not.

Iteration in the form of for/while loops is integral for the game to run in real-time. A main game loop will be used to clear the screen, update the entities, and refresh the display for each frame. Iteration will also be used in entity behaviour; the ghosts will iterate through each cardinal direction to decide what direction to move in.

2.2.4   Pipelining and Concurrent Thinking

Pipelining allows for one process to start while another is finishing. Concurrency is when multiple processes are split into time slices and ordered into a schedule to be worked on.

These methods allow for an increased program throughput, which is important for a game like this. The game should run at a consistent frame rate, preferably 60 frames per second; all computation between frames needs to be done within a ~16.67ms window. Major drops in framerate will mean choppy movement, stuttering and less responsive controls, the latter being the most unbearable since the game can demand quick reactions. The throughput pipelining and concurrency offers will ensure that all computation between frames will be done within the window, keeping performance at an optimum.

Concurrent processing via scheduling algorithms is present in almost every modern operating system. The same can be said for pipelining, which is a mainstay in most contemporary CPU architectures. These two methods will not need to be explicitly implemented in my code because these are implicit optimizations that are autonomous.

2.2.5 Performance Modelling

Performance modelling means analysing the performance of algorithms to determine what processes are taking up the most time and addressing them accordingly. It is vital for the game to run smoothly so that there is minimal delay between the player's input and what appears on screen. Excessive frame drops and delayed input can make the game cumbersome to play. Knowledge of the scalability of algorithms will help in predicting any future performance issues and optimisations. Big notation is suitable for this task. For example, if I know that an algorithm has a time complexity of , I can seek an algorithm that scales better, stopping the poor scalability form becoming an issue later in development.

Python's built-in libraries such as 'time' will be used to measure the elapsed time between an algorithm starting and ending, with this information, it is possible to find the percentage of time an algorithm takes in a single frame or even the approximate number clock cycles it uses. Optimisations can then be achieved based on this data. However, knowing the Big O is arguably more beneficial as the time for an algorithm to compute will very across hardware configurations.

2.2.6 Thinking Ahead

This refers to the careful planning to make sure the project is completed to a good standard and within time. Thinking ahead is useful as it allows you to determine how much time should be spent on each key part of the solution.

I can achieve this by dictating a programming paradigm that the solution will revolve around, which will be Object Oriented Programming (OOP). By sticking to this, I can plan out how each entity of the game will translate to objects using Unified Modelling Language (UML) diagrams. OOP lends itself well to reusability and modularity through inheritance and polymorphism. Which is a big plus for saving time and for keeping the code base clean and simple.

Thinking ahead also refers to caching, the process where frequently accessed data can be temporarily stored somewhere for quick access when needed. In software this can be a web cache in local storage, in hardware this can be a small primary storage inside a processor. All modern consumer grade CPUs have a cache that is used implicitly, so I do not need to implement caching in my code.

2.2.7 Stakeholders

This game can attract a variety of stakeholders, like how the original arcade release aimed to satisfy a wide demographic. These stakeholders include:

- **Children** – Aged 5 to 12, male and female. Children that have an interest in video games and or play video games as a pastime. These children may not be very versed in technology but will be able to pick the game up easily due to the simple controls and easy to grasp winning and losing conditions. They may also be attracted to the colourful appearance of Pac-Man and the ghosts. The game was designed for an arcade setting, allowing it to be played in short bursts. Suitable for children with short attention spans. The score can serve as a gateway to competition amongst a group of children. While the controls and graphics may be the most important to them, they may ask for a high score/leader board system to further facilitate competition. While possible, implementing this is not on top of my priority list. Implementing the gameplay first will not only satisfy all stakeholders, but also guarantees that the game will serve as a strong showcase of solving a problem using computational methods and algorithmic thinking within the time that I have. If I have enough time, I can add these extra features.
- **Adults** – Aged 20 to 50, male and female. Adults that remember playing the original arcade realise during its peak or remember playing later versions or realises. They may be a fan of Pac-Man, looking for a simple way of playing a recreation of Pac-Man without having to toy with emulation or other methods. Akin to the children, some adults may not be versed with technology, so they may seek a simple way of playing Pac-Man, which this solution will serve. The simple controls and arcade-centric design allows adults to use the game to get some entertainment during their free time. While the simplicity of the game can be appealing for young children, it can become boring for adults, they may ask for variety seen in other realises, like multiple maze layouts. #latercrossref Again, I want to focus on implementing the core gameplay, with extra features being added if I have enough time.
- **Programmers** – Likely hobbyists that have an interest in how retro games work. These people are likely to have less of an interest in the game and more so on its code. They may find use in an implementation of Pac-Man's underlying algorithms in a modern high-level language, they can fork the code for use in their own projects, without having to do research or use an emulator's disassembler and debugger to try and make sense of the algorithms. These stakeholders may ask for the source code to be easy to read and understand. An aspect of my solution that will be a priority.

3   Researching Existing Solutions

To replicate the game play of the original Pac-Man, I needed to do some research on existing solutions. This involved a dive into the original algorithm that dictates

how the ghosts traverse the maze, their unique behaviour, and their various states. With this knowledge I will be able to confidently translate these algorithms into Python and Pygame.

The commercial success of the original of Pac-Man naturally led to many other realises over the years. Some offer simple quality-of-life improvements, to adding whole new mechanics. I will assess these features and decide whether to implement them or not.

3.1 Pac-Man (A Look into Ghost Movement and Behaviour)

One of the reasons I decided to do this project was because of this comprehensive video I watched a while ago from a YouTube channel called Retro Game Mechanics Explained. The video goes over every facet of the ghosts' behaviours in detail.
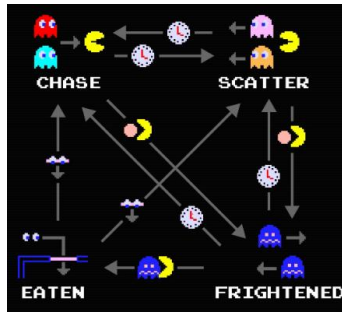

Figure 3: Video graphic showing the four states and how they interact [1].

The video begins with an explanation of the four states and how each ghost moves between them. There are 4 states: chase, scatter, frightened and eaten. The ghosts will alternate between chase and scatter depending on a timer. When Pac-Man eats a power pellet, they will temporarily enter frightened mode and then enter eaten mode when Pac-Man touches them. Upon entering the ghost house, they will regenerate and either enter scatter or chase depending on the timer. This is something that I will definitely add.

The video then explains the timings for scatter and chase mode, with the quirk of Blinky constantly being in chase mode when there are only a few dots left in the maze, something that I will also add.

An explanation of the movement algorithm is then given. The ghosts use a targeting system to determine the tile to move towards. Each cardinal direction is checked to see if it's a valid direction to move in. A direction is considered invalid if it leads into a wall or causes them move back the way they came. Out of the valid directions, the distance between the tile the direction will lead to and the target

is calculated. The direction that yields the smallest number will be the direction the ghost will move in. If all directions yield the same distance a priority system is used to pick one. Up is given the highest priority, then left, down and right. All of this will be included.

Figure 4: Video graphic showing Blinky choosing a direction [1].

The various modes are then explained. During scatter mode, the ghosts' target tile is set to a corner of the maze. When chasing, Blinky's target tile is directly on Pac-Man, Pinky's is four tiles ahead, Inky's is dependant on Blinky's and Pac-Mans postion, and Clyde is on Pac-Man, until he gets within eight tiles of him, in which case he will scatter away for several seconds. When frightened the ghosts will u-turn and begin to move in random, eligible directions according to a random number generator (RNG). All this will be added.

The video addresses a bug that occurs when multiplying the 'up' unit vector. The code that handles applying magnitude to unit vectors treats the x and y components as one 16-bit value rather than two 8-bit ones, leading to the multiplication of the y component overflowing into the x. For Pinky this means that her target tile is three tiles up and four left when Pac-man is facing upwards. This is obviously a bug, therefore I won't replicate it in my solution.
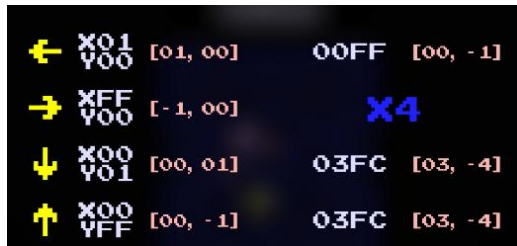

Figure 5: Video Graphic showing the overflow error. Multiplying `$FF` by 4 makes `$3FC`. The left most nibble overflows into the x component. [1].

3.2   Ms. Pac-Man

Released in 1982 by publisher Midway, Ms Pac-Man is the questionably licensed sequel to Namco's original arcade release.

Figure 6: Start of a game of Ms. Pac-Man. Note how the walls are now solid pink and that they're now two sets of warp tunnels.

The gameplay does not stray far away from the original, instead it adds new quality-of-life improvements and new content that still retains the timelessness of the original's gameplay:

- The maze walls are now filled with a solid colour that contrasts better with the black rather than being simple outlines that can blend into the background, especially so considering the blur CRT displays have, which were in ubiquitous during the era Pac-Man came out in. This offers a nice usability feature for the visually impaired, who may struggle to distinguish the walls from the background. This is something that I will certainly implement into my solution.

- One gameplay enhancement comes from the ghost behaviour. At the start of the level, Blinky and Pinky ghosts will move randomly for the first couple of seconds before falling into their typical behaviour. This not only adds a small ripple to the gameplay for experienced players but can also serve a good challenge to overcome and enhance the algorithmic complexity of in my solution. If I have enough time, I will add this.

- Ms. Pac-Man adds three new mazes that appear back-to-back for each level. These new mazes have two pairs of warp tunnels. This allows the player more options to evade the ghosts yet also gives the ghosts more options to intercept the player. While a nice addition, I will not prioritise it since I

would like to get the core gameplay loop down pat before adventuring into other aspects of the solution.

- The bonus fruit is that appears in the original now moves through the maze. This makes the eating of it more frantic since you need to pay attention to the fruit is along with the ghosts. While a neat addition that can add extra challenge for experienced players, I am afraid that this may confuse novices into believing that it is just another threat. They may subconsciously associate anything that moves besides them to be one, so will put unneeded effort into avoiding it, making the game harder for themselves. While I will not consider adding this for the meantime. I may add it if I can use it to showcase a programming technique. Maybe the bonus fruit can inherit the movement of the ghosts using OOP?

References

[1] Retro Game Mechanics Explained. *Pac-Man Ghost AI Explained.* 2019. URL: https://www.youtube.com/watch?v=ataGotQ7ir8 (visited on 03/07/2021).