Computer Science Project:

Name: Danny Cullen

Candidate Number: 1194

College: King Edward VI

Centre Name: 31190

Project Title: Pac-Man Remake

Qualification: OCR H446/03

Date: 28/03/2025

Analysis Section

Introduction:

I am going to be making a re-creation of the game Pac-Man with all of the original functions but I will also be adding many features, such as new maps, power-ups and more. To do this I am going to be using the library called pygame to create the GUI for my game. This is achievable using computational methods because I can use many different algorithms to develop the functions, such as the ghost movements, which work by finding the quickest route to Pac-Man on the map, I may possibly do this using the dikjstra or A* algorithm. I will also be using key inputs gotten from the user to then determine the direction of Pac-Man and he will then go in that direction until he can't anymore. The Aim of my project is to create an up-to-date version of the original Pac-Man game by adding new features and improving many of the existing features. Pac-Man is a game with the objective to collect all the pellets scattered all across the map while avoiding 4 ghosts that are actively following you and will lose you a live upon collision. The map is shaped as a maze with many different directions to go in to avoid the ghosts. I will be adding new sections and entire new maps to my game that will be changed as you go through the levels, increasing in difficulty. I will also make it so that every 5 levels, a new ghost will be added. I may also decide on more features as I see any issues or opportunities warranting them as I go through the development of the game.

Stakeholders:

There are going to be multiple types of stakeholders for my program. This means that I need to identify the type of person these stakeholders are and make sure my program is going to meet their needs as they are the people that are going to be using it.

Type 1:
This stakeholder would be someone who plays games as a hobby and plays a lot of different games. This means that this game needs to be as appealing as some of the more popular games. This means that the game needs to be difficult and interesting otherwise it will be to easy and won't be very fun for this type of stakeholder for my program.

These are the stakeholder characteristics:

- familiar with technology and video games
- has a good computer (for running games)
- Understands how to run and play games
- Likes varied types of games

Type 2:
This stakeholder would be someone who enjoys playing arcade and mobile (phone) games. This would be somebody less familiar with the games industry and is only really interested in playing these quick and easy to set-up games. This means that my game needs to have a simple GUI and needs to be easy to run with minimal set-up requirements

These are the stakeholder characteristics:

- unfamiliar with technical ideas and video games as a whole
- typically has a laptop/tablet computer
- doesn't really understand how to run a game on a computer
- Only likes very specific types of games

These are questions that I asked somebody who fits in the type 1 stakeholder category:

1. how long would you play a game like this in one sitting? 10-20 mins

2. are you interested in this type of game? yes

3. do you play any other games, if so which? Tetris etc.

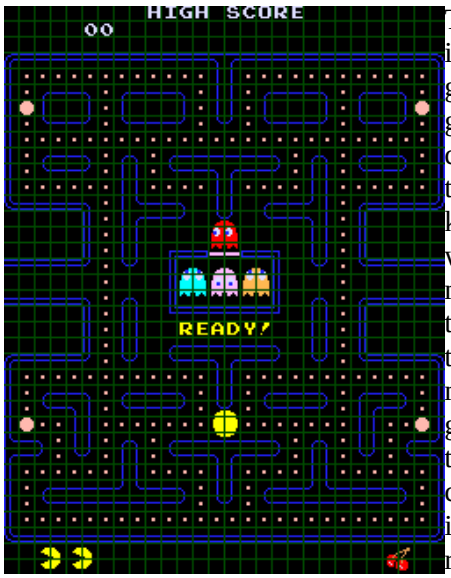
4. would you be more inclined to play if it deviated from the original and there were more features? More features

5. how much do you like games in general? A lot.

I have found that the target audience for my game are people who enjoy playing arcade games and fast, level based games. Most people would likely play a game like mine for around 10 to 20 mins in one sitting. This means that I need to make the game so that can understand and quickly play the game in a short period of time. I have found that people would be more likely to play the game if I added more features and improved the game so that it has more maps, power-ups etc. For the most part, my program will only be suitable for people who are interested in playing games. The people who would want to play my game are the type of people who already play games such as Tetris and other arcade games.

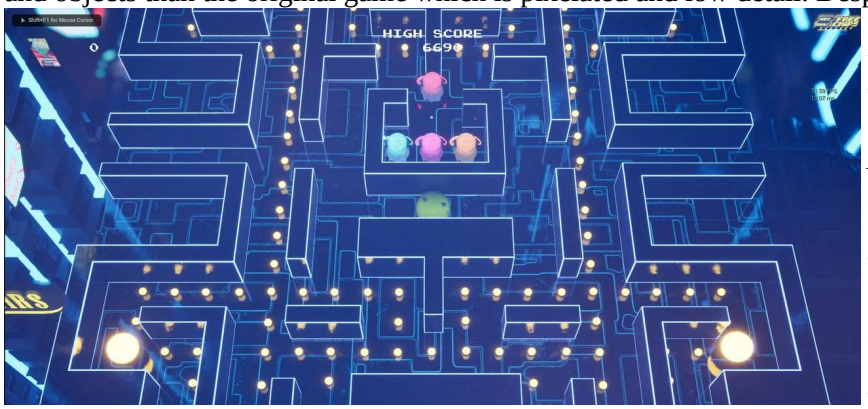
Research:

Pac-Man is a maze action game developed by Midway and released in 1980. You have control of the main character (Pac-Man) and move around the maze in order to collect all the dots. The ghosts (enemies) are always chasing you, if you collide with one then you lose a life and get transported back to the spawn area. You can collect power ups that can give you extra points or the ability to kill the ghosts. The game has infinite levels and as you go up the levels everything gets faster and more difficult by changing the way the ghosts move and reducing the time you get with the power-up to kill the ghosts. The game has infinite levels and will just keep going until you have lost all 3 lives. You may gain an extra life for reaching 10,000 points. The game will keep track of high scores and was originally meant to be played on arcade machines. My version will retain most of these attributes, but also add a few more as mentioned previously.



This is what the original game looks like. The resolution for the original is not standard, and since I am going to have my game run on PC, I am going to have to change the aspect ratio of the game. I also want my game to have a resizeable window so that it can be used on any computer no matter what screen resolution it has. The board as seen in the image is sectioned off into tiles, This is not seen by the player but is key to understanding how the original game works. The tiles are used as what are called “target tiles”, every pellet is in the centre of its own tile, meaning when Pac-Man occupies a tile, the pellet for that tile will be taken away and the score will go up. A sprite is considered to occupy a tile when its centre is in the tile. For ghost movement, they have 3 modes. These are Chase, Frightened, or Scatter. Chase is when the ghost is going after Pac-Man, the ghosts are in this mode most of the time. Scatter is when the ghost will go to their specific designated corner, it is trying to get to a tile off the board so it will then just circle its corner. Frightened is where the ghost will change colour to blue and run away from Pac-Man, in this mode that take a random turn at every corner. For my game I will be keeping theses three modes and they will work in a similar way. I will use dikjstra and A* algorithms to map the ghosts to a tile on the screen where Pac-Man is. This will continually refresh as the ghosts and Pac-Man move.

Some people who try to make modern remakes of old games will put in many modern features to completely changes how the game looks and feels to play. For example, in my research I have seen some people make modern versions of Pac-Man and added 3D graphics meaning that you look at the maze from a diagonal view so that you can see the depth, this means it will be less clear and more difficult to see where the ghosts are. This definitely adds something to the gameplay that wasn’t there before in the original game. They have also added a 3D background giving more immersion and depth to the game to make the player feel more enticed by it. These graphics improvements can be seen in the example below. In this example, there are also many lighting improvements, very smooth animations, lots of shadow effects and vastly more detailed sprites and objects than the original game which is pixelated and low detail. Despite this, I feel that I want



my game to stay as a 2D game like the original version so as to keep the same feel and gameplay as the original to make a true remake. Also 3D would be difficult to implement because pygame is the only game engine that I know how to use and it does not support 3D. Something I also have to consider is the games menus, on the original game there is a very basic menu that is show before you play, I would like to improve on this by making a more full-fledged, more aesthetically pleasing menu that shows the scoreboard and game mechanics is a much more modern look with a background and transparency etc.

On the left is the original Pac-Man menu screen which only features a black background with very pixelated text, only showing the high score instead of a full scoreboard. I would like to add a background showing a full working game blurred behind the menu elements. As well as this I will obviously not be having credits in my game as it will be for PC and will not require coins to play. To make my scoreboard I will be saving the scores and names to a text file and they will be retrieved from there to be displayed in-game so that the high scores will be saved between sessions.

---------| **Next Page** |---------

Computational Methods / Problem Identification:

There are many different features that I am going to be implementing into my project and this means that I need to make sure that they can be solved using computational methods.

Ghost Movement:

This is where I want the ghosts to be able to navigate the maze and chase after Pac-Man so that they are constantly trying to go from their current location to the location of Pac-Man's sprite, while only going through the corridors of the maze. For this I will be able to use the A* algorithm which will be able to find the fastest way to get from the Ghost's current location to Pac-Man's current location. I believe this can be done any time the ghost reaches an intersection and it will decide which way the ghost goes.

The A* algorithm uses heuristics and path values to find the shortest route through a given set of paths to the destination. I can use this to my advantage with the ghost movement as A* is computational and can be done using python. My game will have many different paths, the ghosts current location and Pac-Man's (destination) current location. I can calculate everything using this, meaning the A* algorithm will be a good fit for my program and will allow my ghost movement to be computational using Python.

Save-Games:

This can be done through the use of a text file as the state of the game cannot be saved in RAM, but needs to be on the permanent storage of the device so that it can be retrieved and used even when the computer has been rebooted and the memory area for the program may have been over-written.

Player Movement:

The player movement can be done by taking inputs from the keyboard that the user can press so that the movement of Pac-Man's sprite can be done by the program based on what the user wants. This is essential as the user needs to have control over the game to make it at all playable and useful.

Collisions:

Collisions are definitely possible to add to my game using computational methods. This is because I plan to have each map stored as a 2D linked list in my program. This means that when Pac-Man is moving the linked list can be traversed using x and y values as both of the indexes to check if the next grid square along is available to move into. Therefore meaning that this problem should be easy to create a computational solution for in my program.

Overall conclusion:

Program Inputs:

- The inputs to my program are going to be very simple. There will be a total of 8 keys on the keyboard that my program will respond to, W,A,S,D,Up,Down,Left,Right. This incorporates 2 of the major game control schemes used by other games. Pygame will make it easy to detect theses and cause action to happen when they are pressed

Program outputs:

- My output is the display of the games window. This will be generated using the pygame library and with asset images being placed in the correct locations based on the given inputs from the user about which direction they want the sprite to move in.

Features and Limitations:

Features:

Below is a list of the essential features that I want my game to have:

Feature	Justification
User can control the game using both, W,A,S,D and arrow key control schemes	The user must be able to control the game to make it playable, and both of these are very standard control schemes.
Extra Maps/Mazes for the user to play after winning the first level.	This adds extra content to the game, meaning that my game would be improving on the original, appealing to my stakeholders
Ghost need to move around maze and user needs to avoid them	For the game to actually be fun, the user needs something to avoid (something that can make them loose) to add suspense.
Pac-Man sprite needs to collide with walls and Ghosts	Colliding with walls is essential as the user can easily cheat (even accidentally) and ruin the game, colliding with ghosts allows the player to loose.
The player needs to have 3 lives (be able to loose the game)	This is essential as the original game struck a good balance with 3 lives and it means the user can loose, which adds purpose to the gameplay
Power-ups and items need to be in the maze and the user needs to be able to collect them	This is a feature from the original game that adds extra content and needs to be repeated in my version to be able to compete
Pac-Man sprite needs to be animated like in the original game	This adds extra life and movement to the game making it much more immersive as well as aesthetically pleasing.

Limitations:

The following list details some of the limitations I run into with my ideas:

- 3D: I cannot add this effect as I am using pygame which (practically) is a 2D support only game engine that won't allow me to make 3D effects. (also ruins the art style I want to keep from the original)
- Compatibility: I am using pygame which means that the compatibility of my game will only work with laptop or desktop computers as it doesn't support the major mobile operating systems (iOS and Android)
- Ghost Movement: I am going to be using the A* algorithm to create the ghost movement code as the Pac-Man specific movement logic would be too difficult to implement, this means my gameplay will differ from the original despite me trying to make it as similar as possible

If I were to do future releases the some of these limitations may be fixed.

Overall Features/Limitations:

For my project, based upon my previous research and ideas I am going to create a 2D remake of Pac-Man with many more features than the original, such as new maps / mazes, extra power-ups and more upgraded graphics, all while keeping the original pixelated style. I want to add these features because the original game is very basic due to its age, but with modern languages and resources I can improve the game to make it more interesting to play. I have some limitations such as not being able to use 3D because of pygame not supporting it and it not fitting with the original style of the game that I want to keep. If I were to make the game 3D I would need to learn a new game engine library and the game would not be in its original art style anymore, which is something I want to keep, meaning I will definitely not be doing 3D. One other feature I would like to add is the ability to save the current state of the game so that you can stop, then continue on with the same game another time. I will also be adding sound effects to my game to make it more immersive for the player and more true to the original game as that also had sound effects. I want to keep many gameplay features similar to the original, but unfortunately will not be able to with the ghost movement as the originals logic for it is too complex therefore I will switch it for the A* algorithm instead.

Requirements:

Software:

I am going to be creating my program using the Python programming language using the pygame library to create the GUI. This means that my program will work with all the major operating systems. Theses include, Windows, macOS, and any Linux distribution. However this list does not include phone operating systems and unfortunately pygame is not supported for phone operating systems. My program will be made using Python 3.12 (latest version at the time of writing) which means that you should have at least this version to be able to run my program and have it all work as intended. This means that as long as the operating system and platform supports Python 3.12+ and pygame, then my program can be run on it.

Hardware:

In terms of hardware, my program should be able to run on pretty much any modern computer system that supports the user of the software I detailed above. However, to run my program you will need to have a keyboard and monitor connected to the computer as my program takes inputs from the keyboard and outputs the GUI to the devices monitor. Without these my program will not be usable in any functional capacity. An internet connection should not be needed as long as the user is able to get the program onto the computer in another means rather than through the internet. Although absolutely no connection is required once the program is on the computer as it will run perfectly fine without it.

Storage:

My software is going to be very small in size on its own (100KB at the most) however this need to be added to the storage requirements of Python 3.12 and pygame. This will be about an extra 110MB depending on the specific installation. Almost every modern computer has an extremely significant amount more storage than this and should be negligible in comparison to the total it has. This means that extra storage or any storage concerns should not be warranted for trying to run my program on any somewhat modern hardware.

The storage my program requires may change as it is used. This is because it may store the save-games as text file (.txt), depending on how big these are (expected to be around 1KB at most) the storage taken up by the game may fluctuate slightly.

Success Criteria:

No.	Requirement	Criteria	Justification
1	Moveable Pac-Man sprite	User can control Pac-Man sprite using WASD and arrow keys Sprite will turn in corresponding direction Sprite will move in new direction	This is a key feature as the rest of the game cannot work without and is how the user inputs are used by the game. Without this, user inputs would do nothing
2	Sprite Collides with map/maze	Sprite can move freely back and forth through corridors Sprite won't turn toward or move into walls Sprite can only change direction left/right when directly on grid square of map	This is also a key feature. This is something I definitely need in my game as the user would be able to cheat and go through walls otherwise, making the game far less playable.
3	Proper Ghost Movement	Ghosts need to go towards Pac-Man, taking the shortest route through the maze Ghosts cannot turn around 180 degrees at any point (mechanic from the original game)	Ghosts need to be able to move around the maze so that the user has something to avoid, if they chase Pac-Man this means the user will also need to actively predict where they need to go.
4	Pac-Man and Ghost Collision	In normal mode (without power-up) Pac-Man colliding with Ghost causes life lost and all sprites re-spawn. In Power-Up mode, collision will cause ghost to re-spawn and extra points to the player	This is definitely needed as Pac-Man can't lose otherwise. This is essential for the game to have any risk or fun.
5	Ghosts re-spawn	When Ghost or Pac-Man dies, or the game is starting, Ghosts need to spawn in centre of maze.	This is needed as the game would break after losing a life otherwise.
6	Dots disappear when colliding with Pac-Man	When Pac-Man sprite collides with any dot in the maze it needs to disappear. At this point, the player needs to gain points.	This will allow the user to know which ones they have got and see how many points they have. This will vastly improve the user experience.
7	Sprite maze roll-over	When the Pac-Man or Ghost Sprite moves past edge of map it should come back at the other side	This is a feature in the original game that I think makes the game a bit more interesting as it provides an unconventional route of escape from the ghosts, therefore I want to re-create it in my version.
8	power-up/items correct spawn locations	When the game starts, the power-ups need to be slightly larger versions of the dots and need to be in the 4 corners of the map items need to appear at intervals during the game in the middle of the map	The power-ups need to spawn in the correct locations as they need to be roughly an equal distance from each other the force the player to traverse the whole maze.
9	Win detection, new level	When the play has gotten all the pellets, including the power-up ones then the game stops and it moves onto the next level	The player needs to have an indication that they have won so that they know that they have won or else they will be confused.
10	Sound effects	Make sure that the Pac-Man animation sound effects are always playing, and that the power-up, item and ghost kill sound effects all play correctly with no delay.	This is not an essential feature but I think it's one that will make the game a lot more immersive to play and make the user a lot more invested.
11	Main menu before game starts	Menu with games name, instructions on how to play, and play button. Game should start when the user clicks on the play button	This allows the player to get a good idea of how to play with instructions before they play. This

			will reduce the guess work for the user in terms of how to play.
12	Save game functionality	When the user closes the game while in the middle of a game, the state should be saved and an option to continue that specific game should appear when they start up the game the next time	This will really help with the user experience of playing the game as the game will not be lost if the user needs to something else. They could easily come back and continue
13	New maps / mazes	The game should have new mazes for the player to play in rather than just the one from the original. This will give the game a bit more variation and improve on the original	I think this will make my game stand out from the original as it would be a huge addition that gives it a lot of extra content than the original game.

Design Section

To design the parts of my game I am going to break down the discrete parts of the game that govern how each mechanic and portion of the Pac-Man game works. The code for the game as a whole can be broken down into the following sections:

- Pac-Man Sprite Movement
- Map Generation
- Input Detection
- Ghost Path-Finding
- Collision Detection
- Pac-Man Sprite Animations
- Ghost Sprite Animations
- Power-up Modes
- Score-Count

This is a comprehensive list of all the components that will make up the overall function of my game once it has been completed. Each of these sections will be their own method/function and can all be developed separately and then brought together in the main program to make them all work to create the finished game that I am looking to create.

Also, to make the development of the game less cluttered and complex I can break theses sections down again to make more focused sub-sections that will make the development far more simple and easy to do. This breaking-down of problems is very useful as it reduces the risk of mistakes and errors in the code by making the development and integration much more simple and streamlined. This reduction in mistakes and errors will also help decrease overall development time, creating extra time for any unforeseen issues.

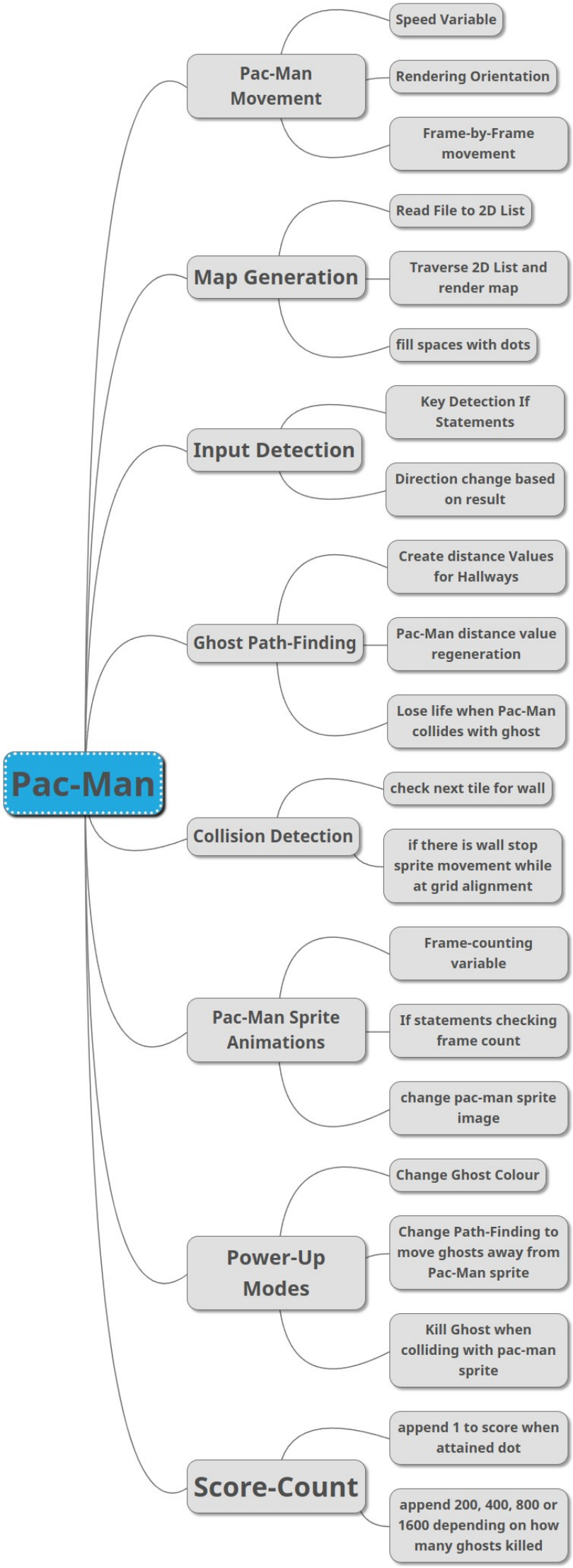
The following table will break-down all the sections and sub-sections of my game.

Sections	**Description/Subsections**
Pac-Man Sprite Movement	This section is about the movement speed/mechanics of the Pac-Man sprite. Such as the sprite only stopping when it hits a wall (unusual with most games). • Variable Definition of sprite speed (allows the speed to be changed) • Rendering orientation of sprite (render at correct rotation based on the direction it is going) • move every frame in current direction based on the movement speed variable
Map Generation	This will work by the program reading a text file that has an array of 0s and 1s in a 16x16 grid. 1 represents a wall, 0 represents empty space. • Code to read file into program (open the file and create a 2D linked list) • Code to go through the whole 2D List

	(based on x and y axis) to find which spaces are 1 and which are 0 and then print either a wall or background onto the GUI • Code to fill all empty spaces with dots (represented by 2 in the map grid)
Input detection	This is checking if the user is pressing any of the control buttons on the keyboard for the game (e.g. w, a, s, d, up, down, left or right) and based on that change the direction variable of the Pac-Man sprite so that it starts moving in the other direction. • Use pygame to detect key inputs in if statements to test if the user is pressing a key • if the if statement returns true then it will run the code to change the Pac-Man sprite to the corresponding direction.
Ghost Path-Finding	This is going to use the A* algorithm for the ghosts to find the fastest way through the maze to get the the Pac-Man sprite • create all the distance/time values for each possible hallway. • As the ghost moves keep regenerating the distance value of how far it is from the Pac-Man sprite at the end of each hallway/intersection to make sure it goes the right direction and changes according to where Pac-Man has moved during the time it has tried to go to it. • Code for Pac-Man to lose a life when colliding with a ghost
Collision detection	This is going to work by checking the next block that Pac-Man is going to go to when aligned to the grid, this way if it is a wall, Pac-Man will stop while still being aligned to the grid (exactly how it works in the original game) • code for checking if the next tile the sprite is going to move to is a wall or a space (1 or 0) • code for making sure the sprite cannot move while that is the next tile.
Pac-Man Sprite Animations	This is recreating the constant repeating animation of Pac-Man just like in the original game. It will work by counting the frames and changing the image that gets rendered for the sprite every few frames.

	• Variable for counting each frame as it goes by • code to check if a certain amount of frames has gone by since the animation last changed state • code to change the image variable to a different imported image so that a different one gets rendered depending on the state of the animation.
Ghost Sprite Animations	This will be very similar to the Pac-Man sprite animations but with different images and possibly a different amount of stages in the animations. Repeated for each ghost in the game.
Power-Up modes	The original game completely changes its state when Pac-Man attains a power-up. It means that Pac-Man can kill the Ghosts instead of the other way round as it usually is. The graphics also change for the ghosts as they all turn a dark blue to show the player that the game is in this alternate state. • Code to change the colour of the ghosts • Ghost path-finding needs to change so that the ghosts run away from Pac-Man instead of chasing Pac-Man • Code to kill the ghosts when colliding with Pac-Man
Score-Count	Keeping the players score and displaying it at the top of the screen • code to append 1 to the score for every dot Pac-Man gets • code to append 200, 400, 800 and 1600 depending on how many ghosts Pac-Man kills when having got a power-up.

The Following Diagram shows the layout/structure of my program


Pac-Man
Pac-Man Movement
Speed Variable
Rendering Orientation
Frame-by-Frame movement
Map Generation
Read File to 2D List
Traverse 2D List and render map
fill spaces with dots
Input Detection
Key Detection If Statements
Direction change based on result
Ghost Path-Finding
Create distance Values for Hallways
Pac-Man distance value regeneration
Lose life when Pac-Man collides with ghost
Collision Detection
check next tile for wall
if there is wall stop sprite movement while at grid alignment
Pac-Man Sprite Animations
Frame-counting variable
If statements checking frame count
change pac-man sprite image
Power-Up Modes
Change Ghost Colour
Change Path-Finding to move ghosts away from Pac-Man sprite
Kill Ghost when colliding with pac-man sprite
Score-Count
append 1 to score when attained dot
append 200, 400, 800 or 1600 depending on how many ghosts killed

Programming

I am going to be programming my game using OOP (object-oriented-programming). This is where you created classes that you can create any number of instances of to create objects of the classes. These objects have attributes (which are variables connected to that object) that you can change easily by appending a dot to the end of the object name and then putting the attribute name. I will also be making use of methods is theses classes as it allows for easy control over them and makes the program more simple and reduces the risks of mistakes and errors as it means I only have to program it once and it is very easy to refer to the methods.

Below is a list of algorithms written out in order of how they will be run in the final program. There will be an algorithm detailing each part of my program. I have labelled each algorithm as a method or function based on whether I believe I will use them within a class or not.

Pac-Man Movement:

```
function pacManMove(moveSpeed, direction, move):
      if direction = left then
            Sprite = Left Pac-Man Image (Changes sprite based on direction)
      endif
      if direction = right then
            Sprite = Right Pac-Man Image
      endif
      if direction = up then
            Sprite = Up Pac-Man Image
      endif
      if direction = down then
            Sprite = Down Pac-Man Image
      endif
      if move = True then
            pacman_pos = pacman_pos + 1
      endif
      if collide with dot:
            newDot = True
      endif
end function
```

The above algorithm governs the movement and image of the Pac-Man sprite and will be a method in a Pac-Man class. This works by using an if statement to check which direction the sprite is going in. There is an if statement for each direction, depending on which one returns True, a different image will be picked so that the image used is facing the same direction as the sprite is moving. There is then another if statement that checks a move flag variable that says whether Pac-Man is allowed to move. If True then Pac-Man's position will be incremented by 1 in the given direction.

Map Generation:

```
function generateMap(file_list):
      for item in file_list
            for item2 in file_list[items]: (checks through all items in 2D list)
                  if value = 1:
                        display wall (displays)
                  endif
                  if value = 0:
                        replace with 2 (represents dot)
                        display dot
                  endif
            next item2
      next item
end function
```

This next algorithm that I have designed uses a text file to create a map, which is a feature that I want to add because I think that it may be too difficult to design each map into the games code and I want them to come from text files separately. It works by taking the file_list which has been imported from the text file and is a 2D linked list of what the text file contained. It is 2D to make sure that x,y values can be used. It starts by traversing the list using 2 for loops, meaning it searches each item in the list, if it finds a 1 then it draws a wall at that grid square location on the game display, if it's a 0 then it draws a black square to signify empty space.

Input Detection:

```
function detectInput(keyDetect):
      if keyDetect = w or up then (checks if w or up arrow is pressed)
            direction = up  (changes direction variable for later use)
      endif
      if keyDetect = s or down then
            direction = down
      endif
      if keyDetect = a or left then
            direction = left
      endif
      if keyDetect = d or right then
            direction = right
      endif
end function
```

This algorithm handles the inputs into the program, it detects when the user presses certain buttons on the keyboard. It checks for both input types (W,A,S,D and arrow keys). This code is also doing input validation by only looking for the correct inputs and completely ignoring incorrect ones, this means that if the user pressed the wrong button that the program isn't expecting, it will just ignore it and carry on running.

Ghost Path-Finding:

```
function ghostPath(pacManLocation):
      use A* to go direct to pacManLocation
      if collide with Pac-Man then
            Pac-Man lose life
      endif
end function
```

The above algorithm includes the A* algorithm to get the ghosts to follow the Pac-Man sprite. The A* algorithm will use the list generated from the text file to recognise the possible routes and find the best one. The next part of the algorithm uses an if statement to check if the position of Pac-Man and the ghost are the same (therefore colliding), if this returns true, then it will take a life away from Pac-Man.

Collision Detection:

```
function collisions(list from file):
      if next tile = 1 then
            move = False
      endif
      if next tile = 0 then
            move = True
      endif
end function
```

This algorithm governs the games collision detection, it will work by checking whether the space in front where Pac-Man or a ghost is about to move to is a wall or not. It does this using an if statement, if it returns true that it is a wall, then the move flag is set to False which will stop the code from moving Pac-Man forward. If the if statement testing for an available space returns True, then it sets move to True so that Pac-Man can move again.

Sprite Animations:

```
function animateSprite(frames):
      if frames MOD 5 = 0 then
            next sprite image in animations
      endif
      frames = frames + 1
end function
```

This next algorithm controls the Pac-Man sprite animations. It does this by taking a parameter of the frames counter (counts how many frames have been rendered since the game started). It then uses MOD to check if frames is divisible by 5 by checking if frames MOD 5 is equal to 0. If this returns True, then it will cause the code to switch to the next image in the animation. This means that the sprite animation image changes every 5 frames.

Power Up State:

```
function powerUp():
      ghost1 colour = Blue
      ghost2 colour = Blue
      ghost3 colour = Blue
      ghost4 colour = Blue
      switch ghostPath to go away from Pac-Man
      if collide with ghost then
            gotGhost = gotGhost + 1
      endif
end function
```

The above procedure is the code that will allow the power-up mode of the game to work. When this gets called, it sets all the ghost sprites colours to blue (like in the original game), then it tweaks the A* algorithm of the ghosts to go away from Pac-Man instead of towards. Then, using an if statement, when they collide with Pac-Man it adds 1 to the gotGhost variable, which counts how many ghosts have been killed within that power-up timer.

Score Count:

```
function scoreCount(newDot, gotGhost):
      if newDot is True then
            score added 1
            newDot set to False
      endif
      if gotGhost = 1 then
            score added 200
      endif
      if gotGhost = 2 then
            score added 400
      endif
      if gotGhost = 3 then
            score added 800
      endif
      if gotGhost = 4 then
            score added 1600
      endif
end function
```

This final algorithm uses the above one to count the score correctly. But starts by counting every pellet that is collected by the player, using an if statement to detect when each pellet is gotten, it will then add 1 to the score each time. It then checks the value of the gotGhost variable and adds the appropriate value to the score based on how many ghosts were killed in the time period of the power-up.

The above list contains an algorithm for each section of program showing, in detail, how it will work and in what order things are going to happen. Each of these algorithms is key to the other ones working as expected and they are all key in making sure the experience of the game is as cohesive and accurate to the original game as possible. This makes sure that the final program will have a polished feel and will have the least amount of bugs possible because I have pre-designed how I want to lay out the structure of my program and how I want each bit of it to work.

Data types:
in this program I will be using a wide variety of data types. Below is a list of each and a description of what they are as well as where any why I will be using them in my program.

Data Type	Explanation
Integer	There are many areas in my program where I need to use integers. In a few places I need to be counting some frames as they go by to calculate intervals between certain actions. I also use them to store the states of different tiles in the map. e.g. 1 (wall), 0(empty), 2(dot)
Boolean	I am mainly using these as markers to show when certain things have happened so that other parts of the program can take action knowing when theses certain things have occurred. For example when Pac-Man attains a dot it sends a marker to the score counting code to add 1 to the score and then it gets set back to False again to wait for another dot.
2D list	This is how my map is stored. This makes referring to a single tile on the map very easy as the first index refers to the x-axis and the second index refers to the y-axis.
Tuple	Tuples have lots of uses in pygame, which means I will be using them a lot to represent thing such as colour and dimensions of graphics. They are particularly useful for this as they can be an enclosed way to pass multiple values to a function.
Strings	I am mainly going to be using strings for debug purposes as I will get the program to print out to the console some values of certain variables to show me what is happening in the background to diagnose any issues. The string is so that I can label each of these outputs so that I know what they are.

Variables:

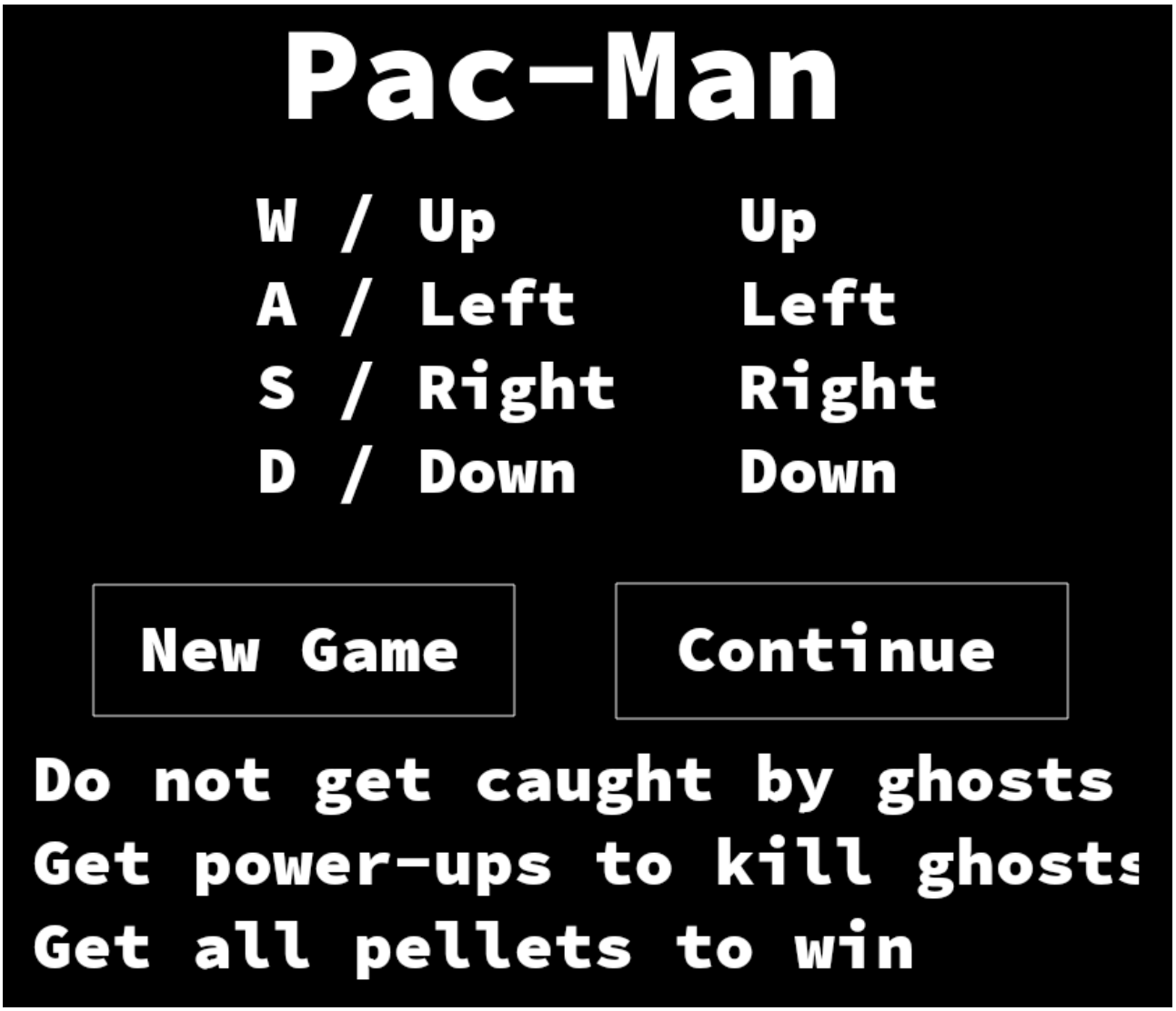
Variable name	Data type	Use
Direction	Integer	Shows the current direction Pac-Man is facing, 0, 1, 2, 3 for up, down, left and right.
file_list	2D Linked List	2D linked list that uses x y values to find any grid square of the maze and give its value
MoveSpeed	Floating Point	Floating point value of the speed Pac-Man should be moving.

GotGhost	Integer	Counts the amount of ghosts that have been killed by Pac-Man to manage how many points.
Score	Integer	Keeps track of the players score for the current game being played.

Overall Feature List:
I am going to have many features in my program. Below is a list of each of them and a few details on them

- Moveable Pac-Man Sprite that collides with walls on the screen to create a maze-type area that the player can navigate through easily by using controls on the keyboard.
- Path-Finding Ghosts that automatically make there way towards the Pac-Man sprite (the player) and will kill you and make you lose a life when collided with.
- Small dots (tokens) that the player can gather to gain points which will be displayed at the top of the screen.
- Power-ups which the player can use to change the mode of the game and kill the ghosts to gain a lot of points.
- Variable speed of the ghosts and Pac-Man to make the game more difficult as the player completes more levels.
- Allows for the creation of custom levels by any player/user as the program just reads the map from a text file.

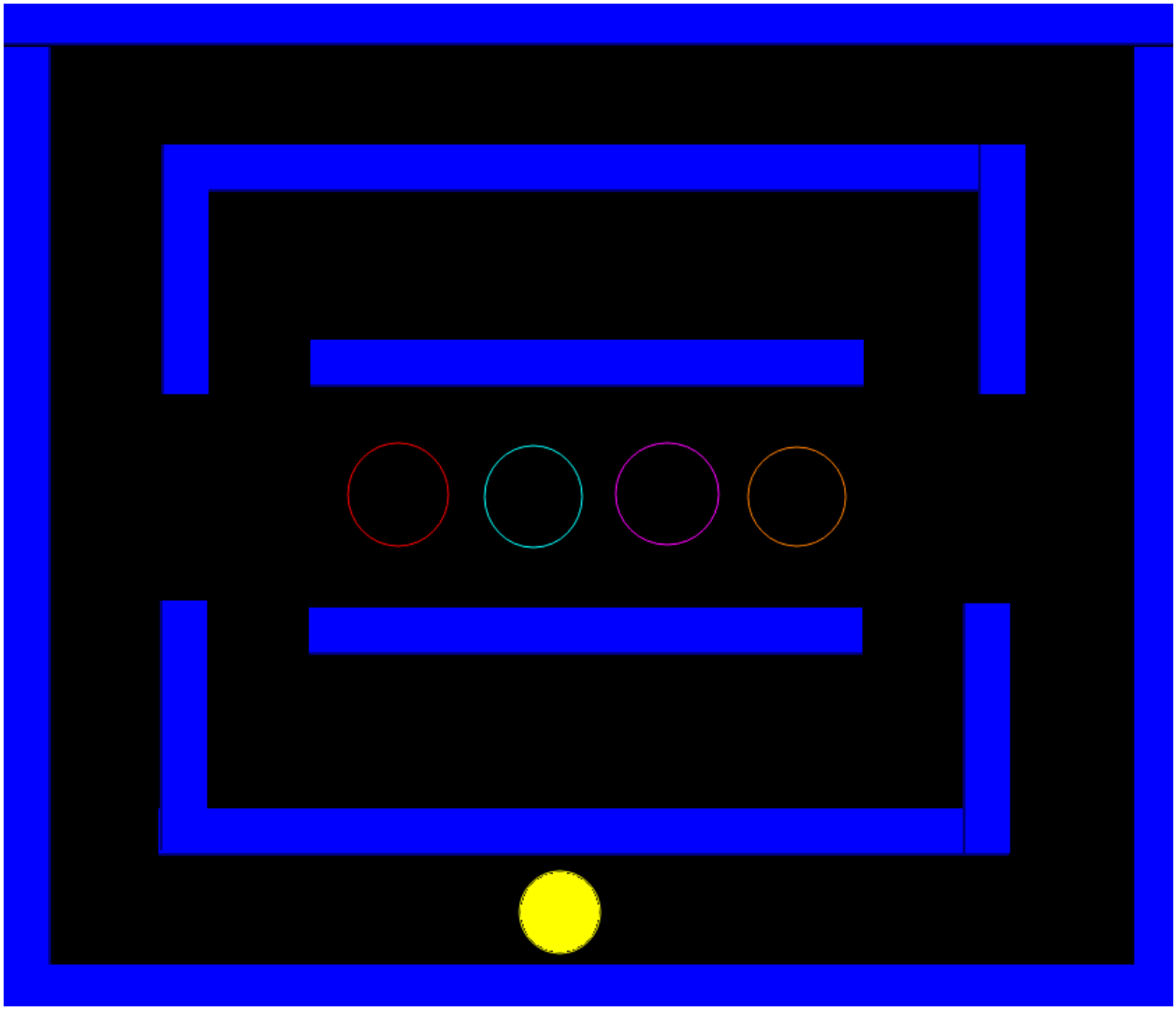
Usability/User Interface:
The usability of the game and the interface of the game is very important for making the user experience more enjoyable. This means that in preparation of creating my program I have designed a basic layout of the main menu and game interface so that I can more easily start my project and know how it should look and feel.



This is my design for the main menu interface and I think that having this will make the user a lot less confused and have a much better idea on how to play the game and what to do. It includes New Game and Continue buttons so the user can continue a previous game using the save feature.

This concept drawing shows the following usability features:

- Discrete buttons to take user input for what they want to do
- Large and clear font for easy readability
- Instructions to show how to play
- Mouse pointer can be used to click on buttons



Above, can be seen my basic design concept for the in-game portion. This is just an outline for the general colour-scheme and general idea for the aesthetics of how I want it to look in my finished program. Pac-Man Sprite at the bottom with ghosts all in centre (default starting locations)

Usability Features:

- Can use up/down/left/right or w/a/s/d keys to move the Pac-Man sprite
- High contrast between open space and walls so the user can easily see where they are allowed to move
- Large window size so that the image is clear and high resolution
- All colours are very contrasting to help distinguish between different elements displayed on the screen.

Test Design:

I will need to test my game after every version so that I can keep track of what issues there are so that I can fix them in following versions and also make an accurate evaluation of how well my program works and to what extent it has met my success criteria.

I will be doing testing during each iterative stage of my program, then testing as an evaluation of meeting the success criteria, as well as user testing to get an unbiased view from the target audience of my program.

The following is the test design for each of these test types:

Iterative/Evaluation:

Test	Expected Result	Justification	Test Type
Do sprites move correctly? (Using W,A,S,D keys as well as arrow keys)	Sprites face correct direction based on where they can go, Pac-Man can be controlled by user.	This is important so that the user is not confused about which way they are going in the game.	Normal
Is map/maze draw correctly to the display? (text file in a levels folder, 1 for wall, 0 for empty space)	The maze will show in the correct scale, rotation and location on the screen and correspond to where the sprites are able to move.	If the maze isn't correctly drawn, then the sprites may not move in the spaces where it looks like they're supposed to and the user will be very confused and the game will be unplayable.	Normal
Is the player's score shown and with the correct number (going up)? (use controls to get pellets)	The Score shows at the top of the screen and only goes up when the player collects the pellets, kills a ghost or gets items.	The player needs to be able to know what their score is in order to be able to keep track of how well they are playing.	Norma;
Are the pellets all places in the correct locations? (start program, check locations)	The pellets should only be in the empty grid square of the maze.	These need to be in the correct place as the player can't get them in the walls and the game won't work as intended and may not be winnable.	Normal
Do the collisions work with walls, ghosts and pellets? (User controls to direct the Pac-Man sprite into a wall, check if stops)	When Pac-Man collides with a wall the sprite should stop, with a ghost a life gets lost or ghost gets killed, with a pellet, it should disappear.	Without this the game will be easily winnable and won't have any strategy or skill component as the player can't collect points and can go through walls to avoid ghosts.	Normal
Does pressing buttons other than the normal controls cause issues? (press random buttons on keyboard that aren't W,A,S,D or arrow keys)	They should do nothing and the game should completely ignore them.	If this caused issues, accidentally pressing other buttons could cause the game to crash without saving the game.	Erroneous
What happens when Pac-Man goes off the	The sprite should re-appear on the direct	The sprite may disappear if this doesn't	Edge-Case

edge of the maze (use controls to direct sprite out of the bounds of the maze)	opposite side of the maze	work, this will cause the game to be unplayable.	

User Testing:

Test	Expected Result	Justification
Are the controls intuitive and understandable?	Yes, as I am going to use 2 sets of very standard control schemes (W,A,S,D and Up,Down,Left,Right keys)	They controls must be standard and intuitive because this ease the learning curve of the game and make it easier to play
Is the game sufficiently difficult?	The game should be about as difficult as the original but get more difficult the further you get	The difficulty needs to be solid so that the game is challenging enough to make it interesting and make the player want to win.
Is it recognisable to the original game?	The game should be easily recognisable as Pac-Man without even seeing what its called as I will make the aesthetics and assets much the same.	This is needed so that people who played the original game (my target audience) would get it.
Is it easy and intuitive to know what to do and how to win?	The instructions given before the game starts should show the user and make it easy to understand	This makes the game easier and less confusing to play as the user will know what to do from the beginning.
Are the aesthetics pleasant and GUI well designed	Yes, as I will be taking lots of notes from the original design which is nice and very user friendly and visually pleasing.	The user experience will be vastly improved if the GUI design looks nice and is neatly layed out
Is there anything that works differently than expected	No, as I am using very standard control schemes and using lots of the same mechanics of the old game, just expanded with extra features	This is important as it needs to be as least confusing as possible to make sure the user doesn’t need to spend time learning specifics and just play the game.

These tests will be very important after I have finished my iterative development of my game. This is because these will tell me how well I have managed to meet my original requirements and projections of how I wanted my game to be. I have designed these so that I can reference them later to link the final result back to my original design to help compare them.

Prototype 1 (Basic GUI, Sprite and controls):

Summary:
My first prototype is the first set-up of the GUI as well as an animated sprite of Pac-Man that can be moved across the screen using the controls on the keyboard. This is list of features, inputs and outputs:

- Pac-Man can be moved using WASD or arrow keys
- Pac-Man sprite will be rotated in whatever direction it is heading in
- Pac-Man sprite will always play animation in the correct rotation
- Pac-Man sprite will return at the other side of the screen when moved of the edge of the window
- a place-holder image is placed in the background to show how it will look with the map

Program Analysis:

```
import pygame

gameRun = True

fps = 120

clock = pygame.time.Clock()

pygame.init()

screen = pygame.display.set_mode((1024, 1024))
```

above is a screen-shot of the first part of my code. The first line simply imports the pygame library into my program so that I can use it to make my games GUI. After this I set various variables to their default or permanent state. Firstly I set gameRun to True which will be used later in the program to control when the game should keep running or quit. After this I set a variable called fps to 120, this is because most monitors/screens are at 60 fps, but setting this to 120 will mean I have more fine control of certain things like sprite speed as I have more operations per second and the game will feel more responsive as the screen will receive a more up-to-date frame. The clock variable set to the clock method from pygame which keeps track of the frame-rate later in the program. Then I run the pygame initialise command followed by setting the screen variable to be assigned to a 1024x1024 window. This window can be seen in image 1

```
#imports images for Pac-Man animation and maze background
pacMan1 = pygame.transform.scale(pygame.image.load("assets/images/pacman1.png"), (40, 40))
pacMan2 = pygame.transform.scale(pygame.image.load("assets/images/pacman2.png"), (40, 40))
pacMan3 = pygame.transform.scale(pygame.image.load("assets/images/pacman3.png"), (40, 40))
maze = pygame.image.load("assets/images/pacManMaze.png").convert()
```

In the next screen-shot above, the asset importing process is shown. This code (described by my comment in the code) gets all 3 images for each stage of the Pac-Man sprite animation and loads them as a 40x40 sprite. Next, the place-holder maze is leaded and the pygame convert method is used to improve performance.

```
#Pac-Man Object
class Pacman():
    def __init__(self, posX, posY):
        #default lives, starting position, and animation markers
        self.lives = 3
        self.posX, self.posY = posX, posY
        self.image = pacMan1
        self.currentImage = 1
        self.direction = 0 # 0 - Left, 1 - Right, 2 - Up, 3 - Down
        self.count = 0
```

Here, I have created a new OOP class called Pacman(). I have defined the constructor using parameters of posX and posY that are passed when an object is made using the class. I have also set many default attributes such as lives, image (current image of the animation cycle), currentImage (current stage of the animation cycle), direction (key for which is shown in my comments) and a general purpose counter named count.

```
def render(self):
    if self.count % 5 == 0:
        match self.currentImage:
            case 1:
                self.image = pacMan2
                self.currentImage = 2
            case 2:
                self.image = pacMan3
                self.currentImage = 3
            case 3:
                self.image = pacMan1
                self.currentImage = 1
```

This is the first section of a method called render. This code is used to make the current animation image used for Pac-Man change every 5 frames done by using an if statement and seeing if the counter is divisible by 5.

```
match self.direction:
    case 0:
        screen.blit(pygame.transform.rotate(self.image, 180), (self.posX, self.posY))
    case 1:
        screen.blit(self.image, (self.posX, self.posY))
    case 2:
        screen.blit(pygame.transform.rotate(self.image, 90), (self.posX, self.posY))
    case 3:
        screen.blit(pygame.transform.rotate(self.image, 270), (self.posX, self.posY))
```

This next part of the method uses the direction attribute to decide which way the Pac-Man sprite image should be rotated. Using a match case statement I am testing for which state self.direction is in. Based on this I use screen.blit (a pygame method on the screen variable I created earlier) to display the image and use pygame.transform.rotate to rotate the image if needed. This calls upon the image attribute meaning it will change based on the animation stage, meaning this code will work fine with the animation code also changing the sprite.

```
self.count += 1
if self.posX > 1024 and self.direction == 1:
    self.posX = -40
elif self.posX < -40 and self.direction == 0:
    self.posX = 1064
elif self.posY > 1024 and self.direction == 3:
    self.posY = -40
elif self.posY < -40 and self.direction == 2:
    self.posY = 1064
```

This is the final section of the render method. It starts by incrementing the count attribute by 1. Then I have added the code to test whether the Pac-Man sprite has been moved off screen. Using an elif structure I have code that determines in which direction it has gone out of bounds (-40 value is used because the Pac-Man sprite is 40x40) by getting its position and direction values. If the statement returns true it will change the position so that it comes back the other side.

```
def testMove(self):
    keys = pygame.key.get_pressed()
    if keys[pygame.K_a] or keys[pygame.K_LEFT]:
        self.direction = 0
    if keys[pygame.K_d] or keys[pygame.K_RIGHT]:
        self.direction = 1
    if keys[pygame.K_w] or keys[pygame.K_UP]:
        self.direction = 2
    if keys[pygame.K_s] or keys[pygame.K_DOWN]:
        self.direction = 3
```

I cannot take full credit for this next bit, in the testMove method, as I learned this method of getting key inputs using pygame from the internet back when I was learning how to use pygame. It works by setting the pygame method for receiving key presses to the keys variable, then using if statements and by using the pygame attribute of the key you are looking for in the list index of the keys value, if the key you want has been pressed. If this returns true I then set the direction attribute to the corresponding value.

```
match self.direction:
    case 0:
        self.posX -= 2
    case 1:
        self.posX += 2
    case 2:
        self.posY -= 2
    case 3:
        self.posY += 2
```

In this last bit of the testMove method is where the code that actually moves the Pac-Man sprite is. I use a match case statement to check which direction the sprite is currently facing, then move it by 2 pixels in that direction. When opening the game, I feel that this is a comparable speed to that of the Pac-Man sprite in the original game, achieving my goal of matching the original game.

```
pacMan = Pacman(512, 765)
```

This is where I create the pacMan object using the Pacman class. It provides 2 parameters, as seen from above where I defined the constructor, these will be the default x and y positions of the sprite. This position is just a place-holder as it puts it at the start position of the place-holder map that I used.

```
def renderBackG():
    screen.blit(maze, (0, 0))
```

Here is where I render the place-holder background into the window. It is a small procedure that will be called from the main loop of the program and uses the screen.blit method to render the maze image at position 0,0 which will make it fill the whole game window.

```
def renderPacMan():
    pacMan.testMove()
    pacMan.render()
```

I then created the procedure that can be called to call the testMove and render methods of the pacMan object. I have done this so that the main program loop doesn't get too over-complicated so that it is more easily readable. I think that this will make the further development that I do on my program much easier as I don't have to decipher what is happening in the main loop every time.

```
while gameRun:
    clock.tick(fps)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            gameRun = False
    renderBackG()
    renderPacMan()
    pygame.display.flip()
```

This is the main program loop. It starts by using the clock.tick method using the clock variable I created earlier. It then requires a desired frame-rate to be sent as a parameter. I have sent the fps value I set earlier which was 120. The next part, I also can't take full credit for as that is a standard way of checking if the user has closed the program that many others use with pygame, I learned it from the internet back when I was first learning pygame. It works by using a for loop using the variable event and traverses all events that are being detected by pygame. The if statement inside checks using the type attribute to see if the event is pygame.QUIT which is the event called when the user presses the X on the program. If this returns true then it sets the gameRun value to False, which will stop the main loop from happening as it works with a while loop reading the gameRun value. I then call the renderBackG and renderPacMan procedures I made earlier so that they run every frame. Finally, I call the pygame display flip method which updates the display with everything I have blitted. Program can be seen running in image 2

```
pygame.quit()
```

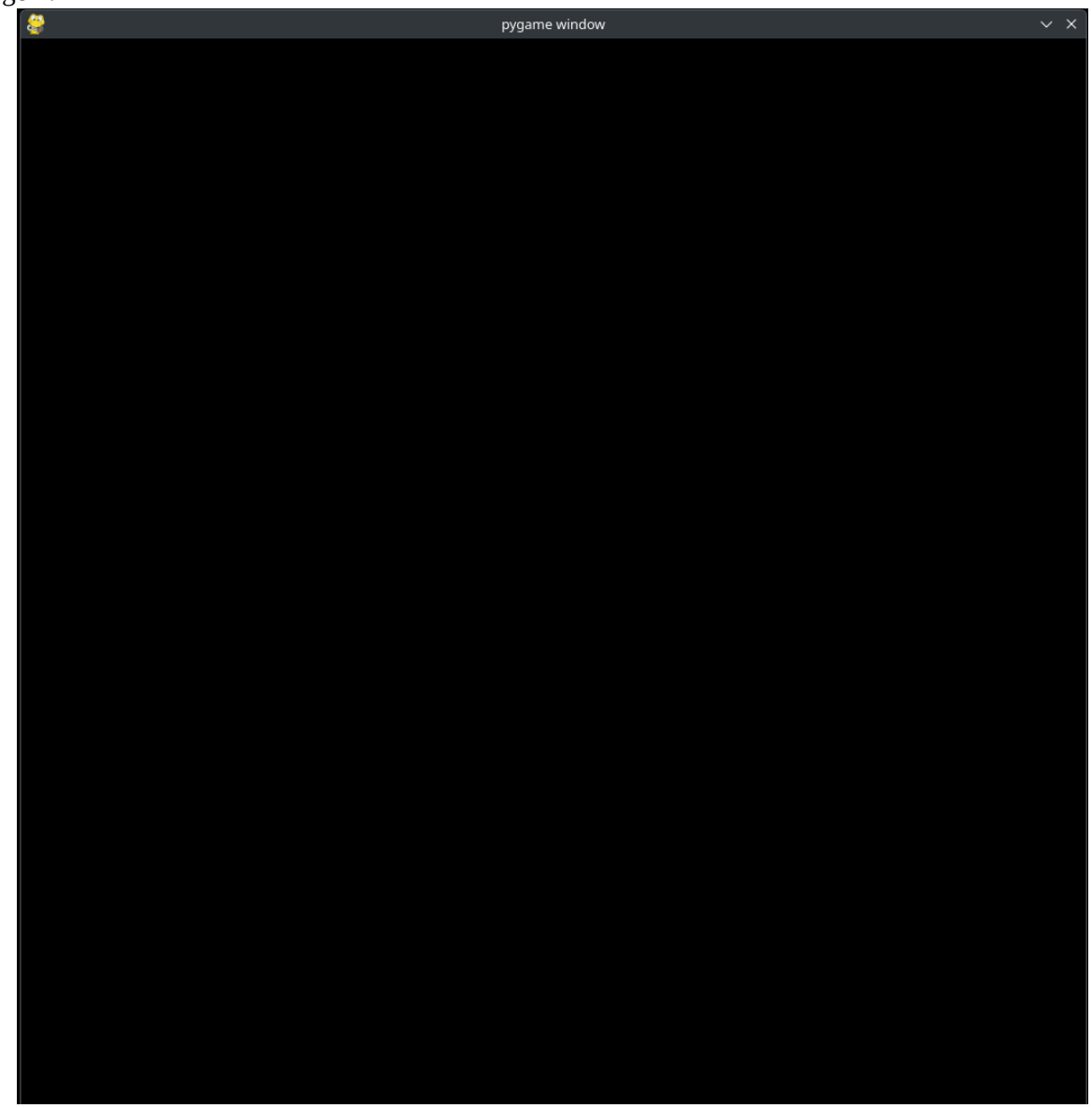
At the end of my program, outside the main loop (so that it only runs if the main loop is stopped) I call the pygame quit method so that the program closes properly.

Skills used in this version:

Here is a list of the discrete programming skills that I have used in my first version:

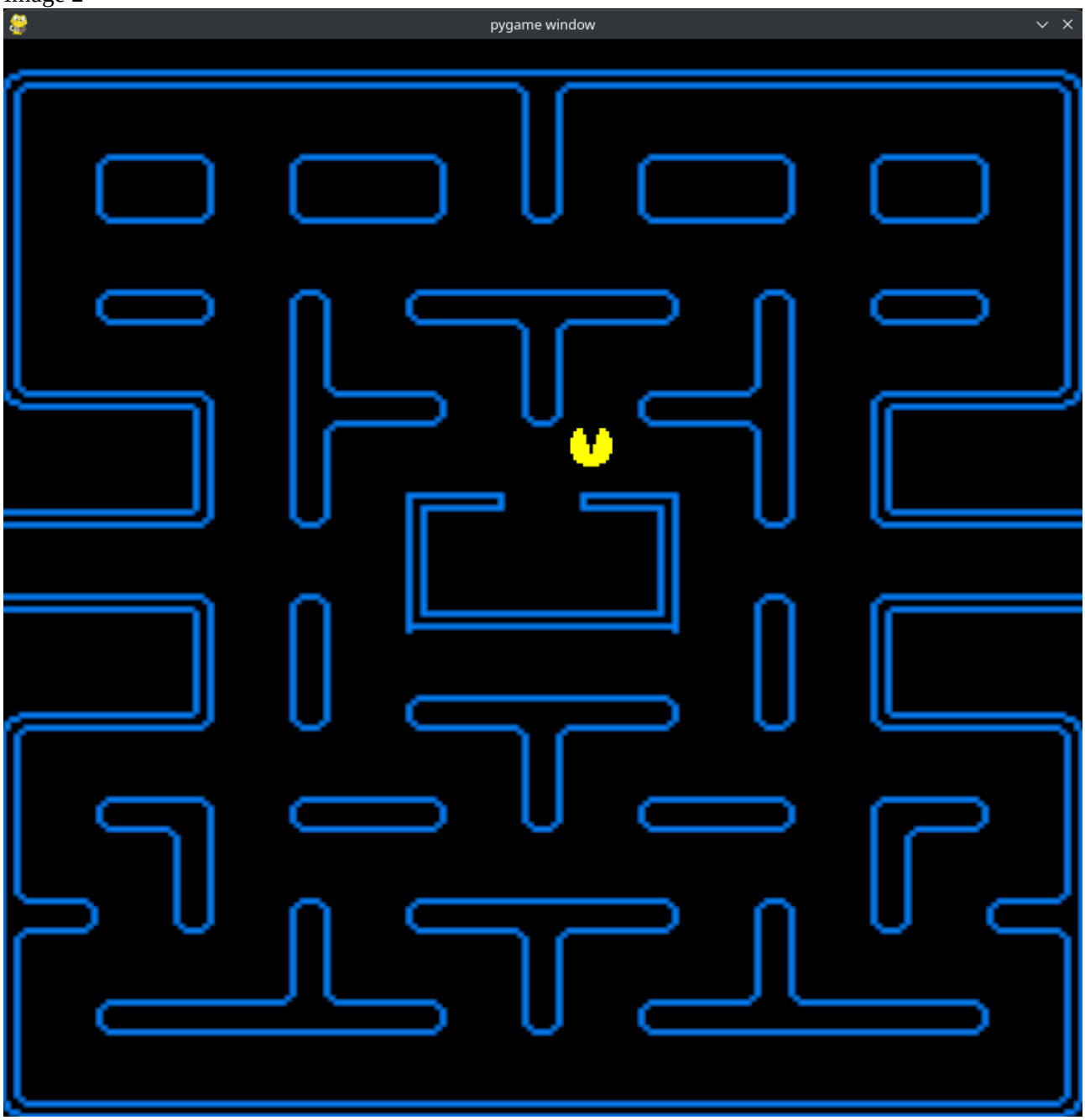
- Procedural Programming
    - I have created many procedures, functions and methods that are called from the main program making my program more modular
- OOP (Object-Oriented-Programming)
    - I have created an OOP class (Pacman()) using a constructor, attributes and methods.
- Validation
    - I have used validation in the keyboard detection code as it completely ignores and excludes any inputs that it isn't looking for, this means that the user can't send any invalid inputs to mess up the function of the program.

Image 1:



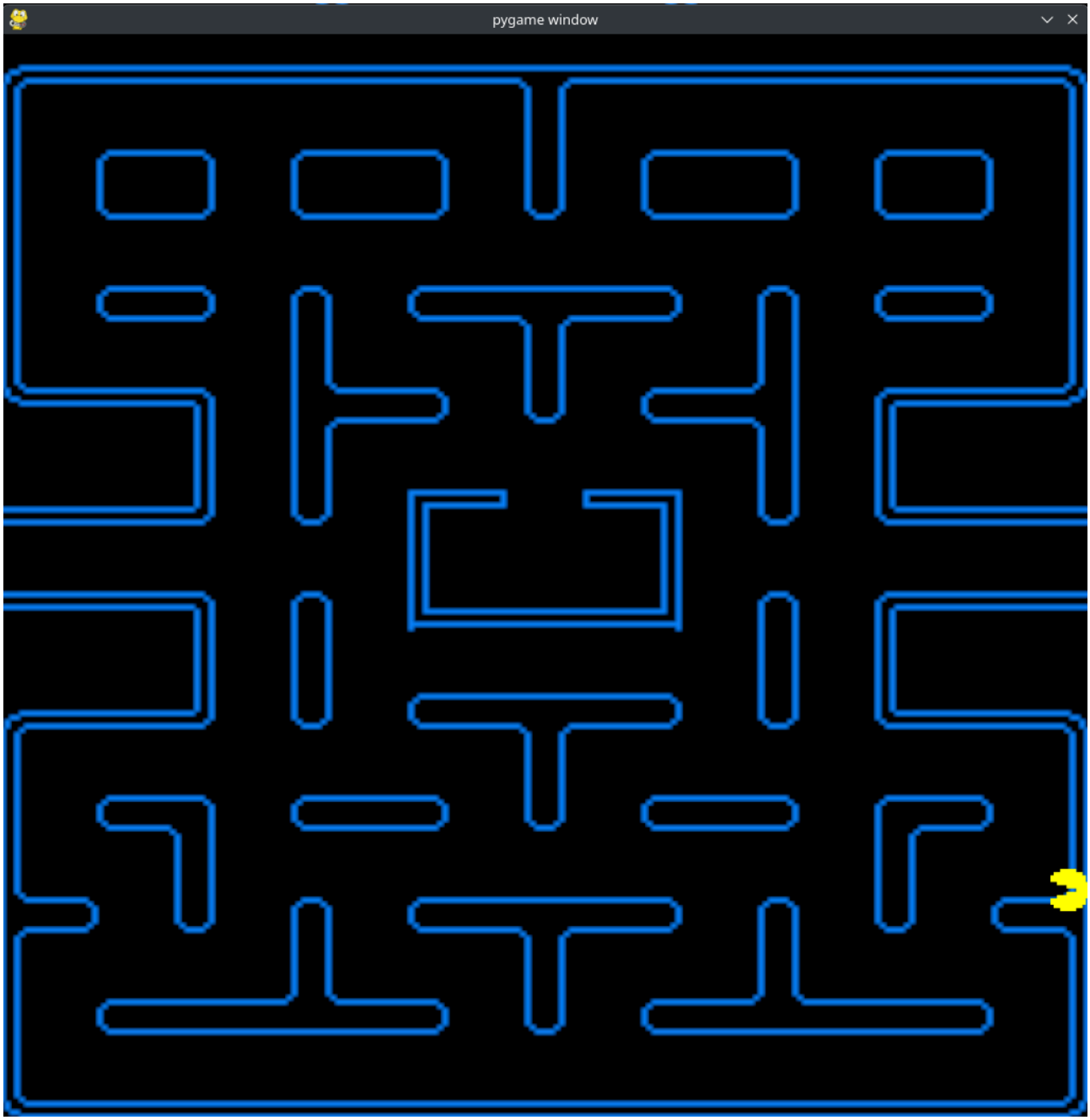
shows original windows opening...

Image 2



Shows the game open with Pac-Man mid-animation, facing upwards. When playing Pac-Man is actually moving and the animation plays quickly.

Image 3:



This image shows Pac-Man returning from the edge of the map. This is showing how it can go from one side to the other when crossing the edge of the game window.

Prototype 1 Final Testing:

Test No.	Test	Test Type	Action	Expected Result	Result	Analysis
1	Does GUI shows correctly	Normal	Open program and check for any anomalies or errors in the GUI	GUI should show Pac-Man moving and animating on top of a map image	Works partially as expected. Map image is stretched, not in proper scale. Can be seen in image 2.	The image can be altered using the pygame transform method to make it properly to scale.
2	Other key presses that aren't expected by the program	Erroneous	While game is open, press lots of different keys on keyboard and press mouse buttons	These buttons will do nothing due to the validation in my program	Works fully as expected	Needs no further fixes
3	Does program close correctly	Normal	Press X on window and check what happens	Program will close immediately with no errors	Works fully as expected	Needs no further fixes
4	Does window re-size	Erroneous	Grab edge of window and drag	Window should not re-size as that would ruin the scaling of the program	Option is not given to re-size. Works fully as expected	Needs no further fixes
5	What happens when multiple direction control keys pressed at once	Edge-Case	Press 2 or more of the WASD or arrow keys at once	Pac-Man should not move if the directions are opposing, if not move in the direction of the first one pressed	Pac-Man will not stay still when 2 opposing keys are pressed	This may not need to be fixed as it does not cause the game to be unplayable as there is no need to press these at the same time and you can continue as normal.
6	Map Border	Edge-Case	Going to edge	Go to other side	Expected. Can be seen in image 3.	N/A Already working as expected
7	Does Pacman move when keys pressed	Normal	Pressing Up/Down/Left/Right	Pacman sprite faces and moves in the specified	Works as expected	N/A Fully works as expected.

				direction with animations		
8	Does Pac-Man sprite have animations	Normal	Start Game	Pac-Man should have a 3 stage animation that plays while he moves	Works as expected. Shown in image 2.	N/A Already is fully working

Next Steps:

The next steps for my project in prototype 2 will be to add the following features to my program.

- Map Generation (creating a proper map instead of a place-holder image. This fixes the problem found in testing where the image isn't scaled properly)
- Map/Sprite collisions (Pac-Man and any other moving sprites need to stop before they go through a wall)
- Pellet Generation (When the game starts, all empty spaces in the map need to have a pellet, represented as a dot, in them)
- Pellet Collection (when the Pac-Man sprite collides with any of these pellets, the pellet needs to disappear)

Conclusion:

In terms of meeting all the requirements I had of this prototype before I developed it, I think I have done so quite well. The only issues being that the map background isn't scaled properly and the input detection not fully functioning when multiple inputs are pressed at the same time. This can be fixed in the next versions and I think that this first one makes a great base to now use to develop the further and more complex features that I set out to make in my success criteria from the analysis section.

Prototype 2 (Custom Maps, Collision and Pellets)

Intro:

For my second prototype I am going to be building upon the base program that I created in the first version. The following features are the ones that I outlined at the end of my prototype 1 conclusion that needed to be added to my program:

- Adding Collisions to my game (so that the user can only navigate through the maze)
- Adding the ability for the player to get points by collecting dots
- making the game generate the dots in the correct locations
- Add map generation

Adding Maze Generation from text file:

Making the game auto-generate maps was going to be above what I can do at this time, however I didn't just want to make a set amount of maps for the game and leave it there. Therefore, I have decided to make the maps generatable from a text file, this means that the user can easily change the map whenever they want so that they aren't stuck with the same one.

Decomposing this problem allowed my to highlight the steps I needed to take to solve it using computational methods. The following is what needs to happen in the program:

- create an empty list
- open the text file
- use iteration to read each row at a time
- append each to a 2D list that can be read as (e.g. theList[x][y])

```
levelList = []
with open("levels/level1.txt", "r") as file:
    for i in range(16):
            tempList = []
            with open("levels/level1.txt", "r") as file:
                for j in range(16):
                    tempList.append(str(file.readline())[i])
            levelList.append(tempList)
```

The screenshot above shows that I have started by making a linked list called levelList. I then use a with statement to import the text file using the variable "file". I needed to make the code import the text to the list in such a way that made it readable using x, y cooridinates. To do this I used 2 for loops, both embedded in their respective with statements. This allows me to append each row in order to the 2D list exactly how it was in the text file. It goes through each row, appending it to a temporary list. This temporary list gets appended to the main levelList as a row. it reaches the end. This only works with 16x16 maps due to limitations with how I would be able to make the program resize everything to scale for different sized maps.

Drawing the Maze:
being able to draw the maze will be a much simpler and easier idea to create. The process for doing this should be relatively simple:

- each square should be 64x64 pixels since I have now limited myself to 16x16 grid maps
- use 2 for loops to traverse the levelList created earlier
- test each index of levelList to see if it has a value of 1 or 0
- if the value is 1 render it at a size of 64x64 at 64 timesed by each coordinate used in the list

```
def renderBackG(): #draws background
    global debug
    keys = pygame.key.get_pressed()
    grid = 0
    if keys[pygame.K_HASH]:
        debug = not debug
    if keys[pygame.K_q]  and debug:
        GUI.fill((255,255,255))
        grid = 1
```

I started by creating a new procedure called renderBackG(). This will be the one that renders the maze using the levelList. Firstly I wrote a small bit of code that will allow me to debug my program (again, as said in prototype 1, using the keys list is a common way of doing this that I learned online while learning pygame, I don't take full credit for this). This code will allow me to see show the lines between all of the grid squares for when I create the Pac-Man movement code that locks it to the grid. I believe this debug technique will be very helpful further in the prototype and maybe the next.

```
col = 100
square = int(1024 // lineCount)
for i in range(len(levelList)):
    for j in range(len(levelList[i])):
        if int(levelList[i][j]) == 1:
            col += 1
            pygame.draw.rect(GUI, (0, 0, col), (i*square, j*square, square - grid , square -grid ))
        else:
            pygame.draw.rect(GUI, (0, 0, 0), (i*square, j*square, square -grid , square -grid ))
```

This is the part of the code that takes the levelList data and uses it to draw the maze onto the game window. Firstly I craete a variable called col, this is a colour value that will change for every collumn of the maze to create a gradient. I have done this because I believe it makes the game more aesthetically pleasing. I then traverse the list using 2 for loops like I outlined in my plan for this code. Whenever the traversal finds a 1 (tested using an if statement and listing I and J as the x and y coordinates) it will draw a square at those coordinates, timesed by 64 (square) so that it converts those coordinates into pixel coordinates that can be used to display them in the right location. Col is also incremented by 1 to create the gradient I am looking for. If it doesn't find a 1, it renders a black tile instead (this is done because in future I will render the pellets above these, and to be able to make them disappear I need to render the black tile back on top).

The result of this code can be seen in image 1.

Generating Pellets in the correct locations:
This should be a simple feature to add. The following are the decomposed steps to create this code:

- traverse the levelList
- use and if statement to find any blank spaces (0's)
- render a small dot in the centre of this space

```
def createDots():
    square = int(1024 // lineCount)
    for i in range(len(levelList)):
        for j in range(len(levelList[i])):
            if int(levelList[i][j]) == 0 and i  > 0 and i < 15:
                pygame.draw.rect(GUI, (255, 255, 255), (i*square + 24, j*square + 24, 8 , 8))
```

I created a new procedure called createDots(). This traverses the levelList again to find all grid squares that have a value of 0 (meaning it's a blank space) and it needs to be 1 in from the border (this is why the if statement checks if I is between 1 and 14). if the if statement tests True then it will draw a small 8x8 square at the location of the grid timsed by 64, then plus 24 to make sure its in the middle. The colour is 255, 255, 255 to make it white.

The result of this code can be seen in image 2

Adding Collision detection:
Adding collision detection to my game will require me to heavily break down how to code it. This will be a very complex feature to add, aspecially because I am using a map that is generated meaning the collisions have to be based upon that instead of always being the same. The following process is what I will use.

- Convert the current position of the Pac-Man sprite to a grid square position (divide by 64)
- check whether the direction the use wants to change to allows at least one empty space ahead for the sprite to move
- do this by checking the direction the user wants to change to using an if structure. Use the and operator to also check the the grid square ahead if the current one, in the new direction, is free.
- Then write the code for another if structure that checks which direction the sprite is currently facing, looks 1 grid square ahead of that and checks if that space isn't a wall
- Then if this if statement returns True move Pac-Man forward in that direction, if it returns False then don't allow Pac-Man to move forward anymore.

```
square = 1024 // lineCount
posX = int((self.posX) // square)
posY = int(self.posY // square)
```

Above is the start of the new code in my testMove() method from the Pacman() class that calculates the current grid position of the Pac-Man sprite. This will then be used in the further parts of this section to calculate when Pac-Man is colliding with a wall. This works by using integer division to divide both the current x and y positions of the Pac-Man sprites by 64. Because the multiplier between the pixels and the grid squares is 64, the will convert it.

```
        match newDirection:
            case 0:
                #left
                if 0 < self.posX < 960 and 0 < self.posY < 960:
                    if self.posY / square == posY and self.dirCount > 0 and int(levelList[int((self.posX - 1) // square)][posY])  != 1:
                        self.direction = 0
            case 1:
                #right
                if 0 < self.posX < 960 and 0 < self.posY < 960:
                    if self.posY / square == posY and self.dirCount > 0 and int(levelList[int((self.posX) // square) + 1][posY] ) != 1:
                        self.direction = 1
            case 2:
                #up
                if 0 < self.posX < 960 and 0 < self.posY < 960:
                    if self.posX / square == posX and self.dirCount > 0 and int(levelList[posX][int((self.posY - 1) // square)] ) != 1:
                        self.direction = 2
            case 3:
                #down
                if 0 < self.posX < 960 and 0 < self.posY < 960:
                    if self.posX / square == posX and self.dirCount > 0 and int(levelList[posX][int((self.posY) // square) + 1]) != 1:
                        self.direction = 3
```

In the next part of the collision detection I use a match case statement to find the current state of the newDirection variable which is set to the direction that the user wants to change to using the controls. For each case it uses an if statement to check if the sprite is within the bounds of the map (1 in from each side) as the outer part is where the Sprite will roll-over to the other side of the maze. I then use another if statement to check that the attribute dirCount is more than 0. This is the variable that shows how many frames it has been since the user has tried to change direction. This is needed as the sprite wont always be aligned to a grid square, but it needs to be in order to change direction. So the user gets 1 second (120 frames) after they have pressed the button for the sprite to be in a valid spot to change to that direction. It then checks levelList at the current grid square of the Pac-Man sprite, adds or takes away 1 to either coordinate based on which direction the user wants to switch to. It checks if that levelList index is not equal to 1 (not a wall). Then it will let the user change direction if that returns True (sets the direction attribute to the newDirection).

```
match self.direction: #moves Pac-Man in chosen direction
    case 0:
        #left
        if 0 < self.posX < 960 and 0 < self.posY < 960:
            if int(levelList[int((self.posX - pacSpeed) // square)][posY])  != 1:
                self.posX = round((self.posX - pacSpeed), 1)
            if int(levelList[int((self.posX - 1) // square)][posY]) == 0 and self.posX == (((self.posX - 1) // square)*square + 7):
                levelList[int((self.posX - 1) // square)][posY] = "2"
        else:
            self.posX = round((self.posX - pacSpeed), 1)
    case 1:
        #right
        if 0 < self.posX < 960 and 0 < self.posY < 960:
            if int(levelList[int((self.posX) // square) + pacSpeed][posY] ) != 1:
                self.posX = round((self.posX + pacSpeed), 1)
            try:
                if int(levelList[int((self.posX) // square) + 1][posY]) == 0 and self.posX == (50 + (self.posX // square)*square):
                    levelList[int((self.posX) // square) + 1][posY] = "2"
            except:
                pass
        else:
            self.posX = round((self.posX + pacSpeed), 1)
    case 2:
        #up
        if 0 < self.posX < 960 and 0 < self.posY < 960:
            if int(levelList[posX][int((self.posY - pacSpeed) // square)] ) != 1:
                self.posY = round((self.posY - pacSpeed), 1)
            if int(levelList[posX][int((self.posY - 1) // square)]) == 0 and self.posY == (((self.posY - 1) // square)*square + 7):
                levelList[posX][int((self.posY - 1) // square)] = "2"
        else:
            self.posY = round((self.posY - pacSpeed), 1)
    case 3:
        #down
        if 0 < self.posX < 960 and 0 < self.posY < 960:
            if int(levelList[posX][int((self.posY) // square) + pacSpeed]) != 1:
                self.posY = round((self.posY + pacSpeed), 1)
            try:
                if int(levelList[posX][int((self.posY) // square) + 1]) == 0 and self.posY == (50 + (self.posY // square)*square):
                    levelList[posX][int((self.posY) // square) + 1] = "2"
            except:
                pass
        else:
            self.posY = round((self.posY + pacSpeed), 1)
```

Above is the final section of the updated testMove() method. Which calculates whether the Pac-Man sprite is running into a wall. I started by creating another match case statement that checks which state the direction attribute is in. In each case it starts by checking that the position of the sprite is within 1 grid square from the edge to make sure that it isn't transitioning to the other side of the maze. Then, using an if statement, it checks that the next pixel where Pac-Man will go is not part of a grid square that has a wall in it. This is done be checking the levelList hasnt got the pixel position added or minused by the speed of the sprite (how far forward it will go each frame) divided by 64. That gives the next grid square that its about to go to. It only does this for one of the coordinates because the sprite can only go in a single direction at a time. The if statements then check that the output of this test is not equal to 1, meaning that it is not a wall. If this tests True then it allows the Pac-Man sprite to keep moving forward, if False then not meaning Pac-Man will stop. The next if statement does the same test, but anded with another that checks whether the sprite has reached the middle of the grid square. This is calculated by checking if the pixel position is the same as itself divided by 64 (square) and timed by square again and summed with the appropriate value depending on the direction where the sprite would be facing the middle. This will then change levelList at that square from a 0 to a 2 to show that the pellet should no longer render there. All 4 of these cases do the same thing but with adjusted values to make it work with the different direction.

The result of this code can be seen in image 3

Testing:

Test Plan:

I am going to be testing my game by checking if all of the expected behaviour such as all collisions and movements work. I will also be testing some extreme case and erroneous cases.

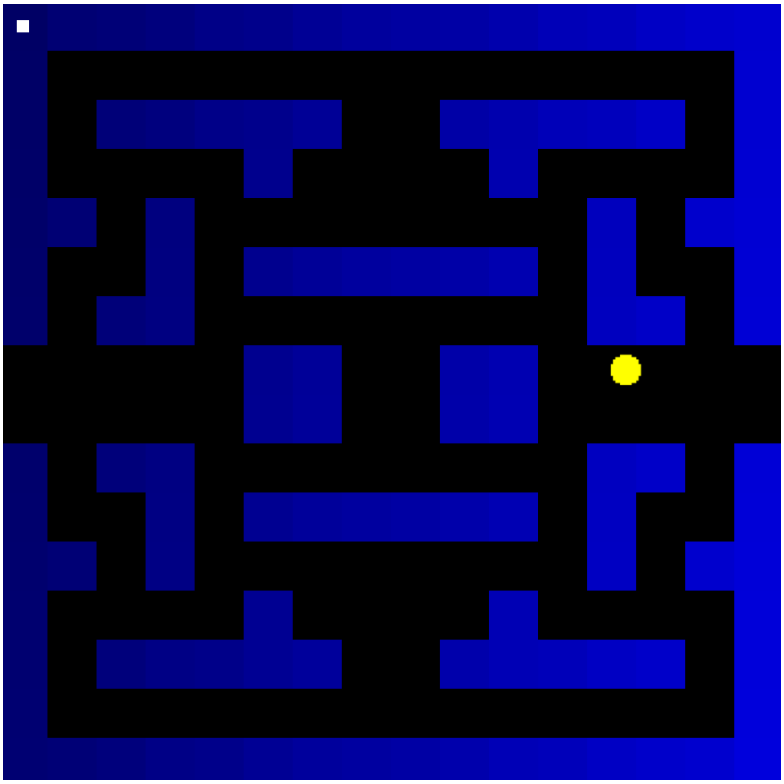
Test	Type	Action	Expected	Result	Next Steps
1	Normal	Running into all walls	Sprite stops moving	Sprite stopped in all cases	None, fully working
2	Normal	Run over all dots	All will disappear	All disappeared	None, fully working
3	Normal	Check if all dots are generated correctly	All show up	All did show up	None, Fully working
4	Extreme case	Check if collisions work while in roll over tunnel	Should only be able to turn into empty spaces	Could not turn at all until reached within regular map bounds	Need to make specific code to govern what the sprite is able to do while in the tunnel
5	Erroneous	Removing level text file	Program will crash	Program crashes	Add code to make the program generate a default level if none is present
6	Normal	Change directions very	Should just change	Does as expected	None, fully working

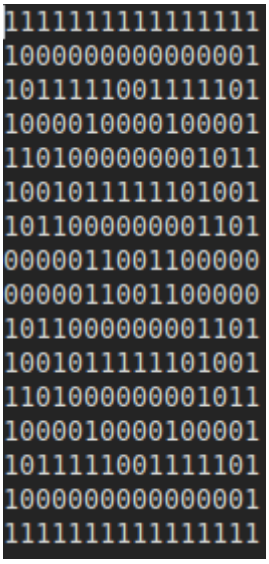
		quickly in succession	direction as expected unless a wall is in place		
7	Normal	Turn around and go back through tunnel before reaching in bounds area	Will just go back to the other side	Will only move once reached onto the boundery itself	Add code to make the program, Generating Dots in the correct locations: Adding this feature is fairly simple given the way that I have setup the rest of my code. All I need to do is make a loop that traverses levelList[][] and as long it is inside the outer wall. As can be seen in the screenshots above I made a double for loop to traverse the 2D list and find any empty spaces in the level and draw a dot on the middle location of each of the corrospoding grid squares on the level displayed in the pygame GUI. Allow movement during the time before it reaches the outer wall

Meeting Criteria:
At the top of this section for version 2 I made a list of requirements I wanted from this version. I believe I have met most of these to a satisfactory level. The following table details how well I have met these criteria:

Criteria	Met	Details	Justification
Map Generation	Partially	I decided to make it generate the map from the text file that the user can edit instead, this new functionality fully works	I was not able to implement this under theses time constraints and with my current knowledge as I ran into issues when trying to figure out a way to do it.
Collisions	Fully	Collisions are fully working and have all the expected functionality	N/A
Pellet Generation	Fully	Pellet Generation works fully as expected with full functionality	N/A
Pellet Collection /Points	Partially	I have made the Pellet collection fully working but at the minute it does not add any points.	I ran into constraints trying to implement this feature as to display the points the contents of the window would have to move the make it fit.

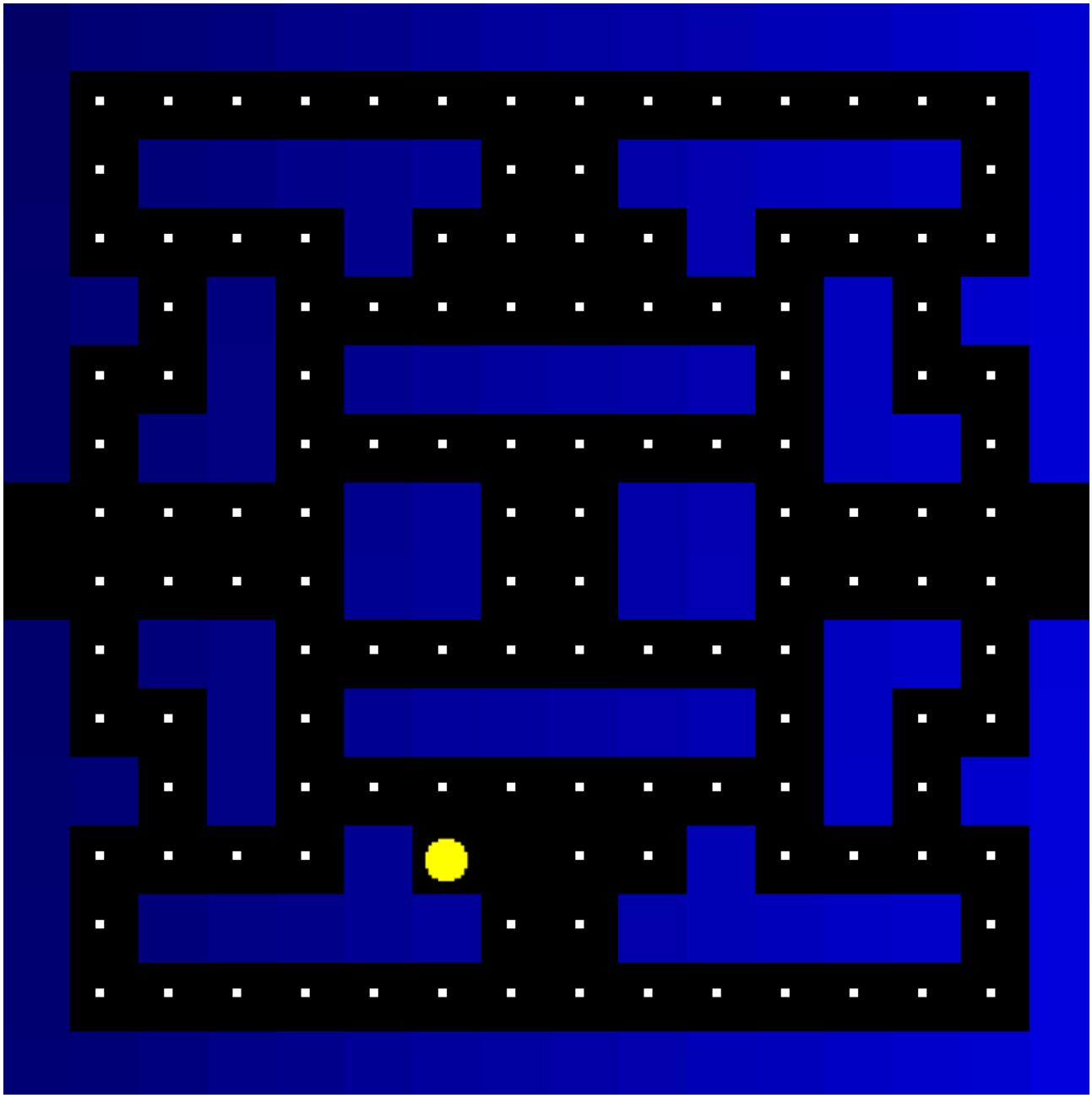
Image 1:





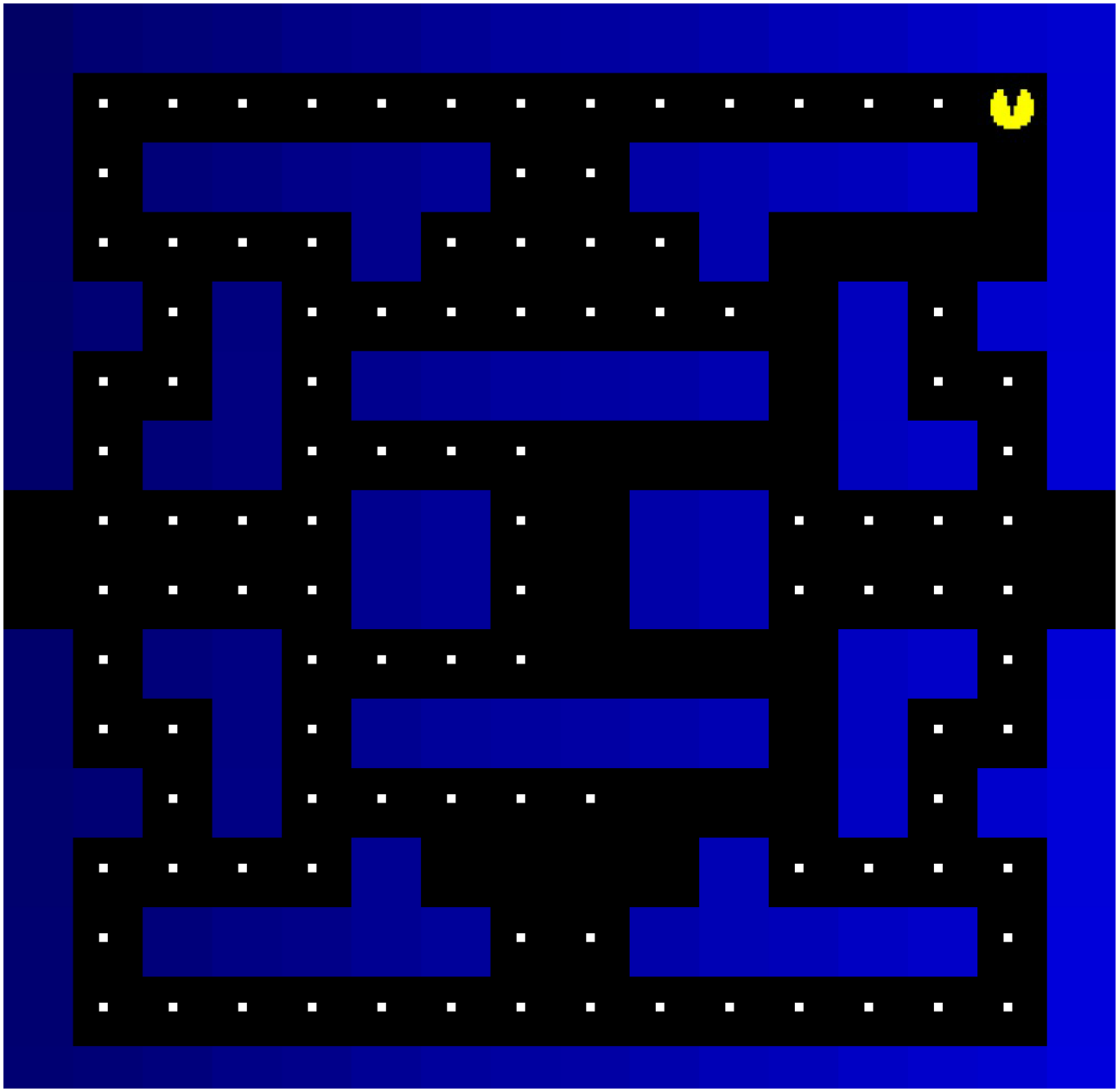
This is where I created the code to generate the maze, as you can see the maze on the left was generated from the text file on the right.

Image 2:



Above is the screenshot of what the game was like after I finished making the code that generates all the dots, as you can see the dots only go in the grid squares that have no walls. This is the expected functionality.

Image 3:



As can be seen in the screenshot above The code that allows collisions and the collection of pellets is in place and working. This can be seen as you can see the trail where the sprite has gone and collected the pellets. As well, Pac-Man is stopped from moving in the top right corner as it is hitting a wall.

My Conclusion:

I have added many features to my game since prototype 1. Such as new collision detection, dot rendering, dot collection, and custom map generation. These are all quite large and important features to my game that have significantly increased the playability and usefulness of my program. However there do appear to be a few errors and issues that I have found in my testing with a few edge case and erroneous situations. This means that there is still some work to be done on a few of my new features but definite progress has been made.

Final Version

Planning:

In my testing that I did for prototype 2, many needed improvements have been highlighted to me:

- Step 1: Add power-up dots to the map
  - add code that calculates where the corners of the map are and designate those places for power-ups. Make sure to add code in the createDots() procedure that detects designations and draws the power-up in this location.
- Step 2: add ghost class and program their movement
  - initialise class with needed attributes. Add movement method to class to calculate where the ghost is allowed to move and which way it will turn.
- Step 3: add limited lives
  - make sure that when the lives attribute of the Pacman object reaches 0, then the game will stop
- Step 4: add ghost and Pacman collision (Pacmal lose life)
  - calculate when the sprite of each ghost is touching the Pacman sprite, if this is true then decrease the lives attribute of Pacman by 1 and all sprites respawn back to their original posiion
- Step 6: add power-up mode (Pacman can kill ghosts)
  - in the testMove() method of the Pacman object, add code that detects whether Pacman has gone over a power-up. Also add code in the Ghost class that is called by this and changes is sprite image and makes it move away from Pacman.
- Step 5: add points counter
  - whenever Pacman gets a dot or kills a ghost, increase a points counter accordingly and display it at the top of the screen

Step 1 – Adding Power-up dots to the map:

```
global firstCorner, secondCorner, thirdCorner, fourthCorner
topRow, flag = 0, True
bottomRow = 0
firstCorner, flag1 = [0, 0], True
secondCorner = [0, 0]
thirdCorner, flag2 = [0, 0], True
fourthCorner = [0, 0]
```

This first section of code creates a topRow, flag, bottomRow, firstCorner, flag1, secondCorner, thirdCorner, flag2 and fourthCorner variables. It sets each of them to a default value of a list with [0, 0] in it for the corner variables, integer 0 for the row variables and True for the flag variables. All of these will be used in the next section of code.

```
for i in range(len(levelList)): #finds the top and bottom rows of map
    for j in range(len(levelList[i])):
        if int(levelList[i][j]) == 0 and flag and 0 < i < 15:
            topRow, flag = j, False
        if int(levelList[i][j]) == 0 and 0 < i < 15:
            bottomRow = j
```

In this next section, the for loop traverses the levelList from prototype 2. it then uses if statements to find the first row that has a clear space and the last row that has a clear space. For the topRow, it does this by checking that the current space is 0 and that that the x coordinate is between 0 and 15. Then it locks in the topRow value to the current value of j by setting the flag variable to False. For

the bottomRow it does the same, but without the flag variable, meaning the bottomRow will be the value of j for the final clear space that it finds.

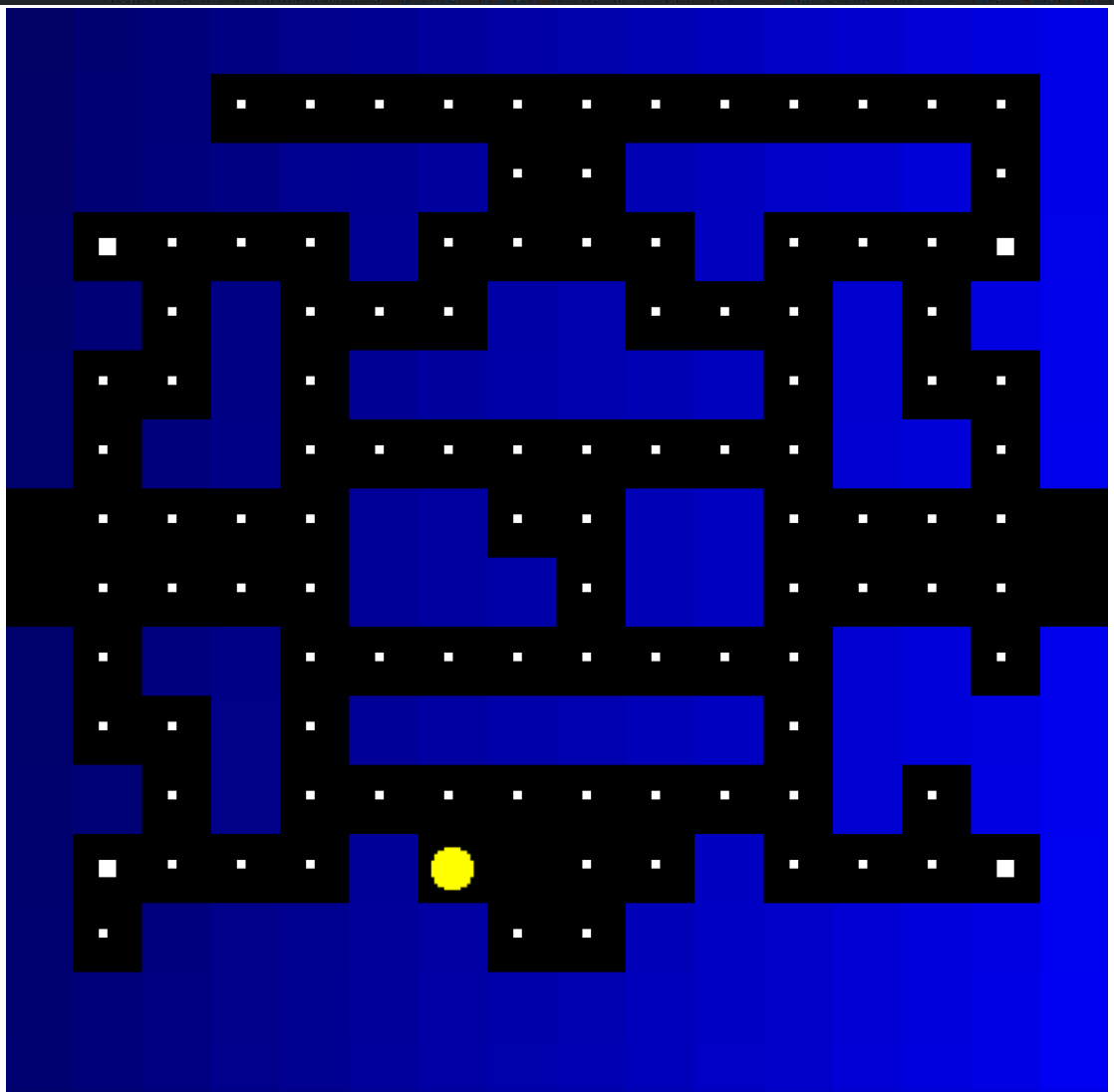
```
for i in range(len(levelList)): #finds the corners of the map
    for j in range(len(levelList[i])):
        if int(levelList[i][j]) == 0 and j == topRow and flag1 and 0 < i < 15: # top left
            firstCorner, flag1 = [i, j], False
        if int(levelList[i][j]) == 0 and j == topRow and 0 < i < 15: # top right
            secondCorner = [i, j]
        if int(levelList[i][j]) == 0 and j == bottomRow and flag2 and 0 < i < 15: # bottom left
            thirdCorner, flag2 = [i, j], False
        if int(levelList[i][j]) == 0 and 0 < i < 15: # bottom right
            fourthCorner = [i, j]
```

This section of code traverses the levelList for empty spaces again, but for each corner I checks that it is on the topRow or bottomRow from earlier by checking if the y coordinate is equal to it. It also makes sure that it doesn't use any spaces outside the map by making sure the x is above 0 and below 15. for the firstCorner (top left) and thirdCorner (bottom left) flag variables are needed as these will be the first ones that the loop finds and will need to lock in this value so that it cant change. It sets the value of each corner variable to a list of the coordinates of the empty spaces it finds

```
levelList[firstCorner[0]][firstCorner[1]] = "3" #sets all four corners to 3. Signifying power-up
levelList[secondCorner[0]][secondCorner[1]] = "3"
levelList[thirdCorner[0]][thirdCorner[1]] = "3"
levelList[fourthCorner[0]][fourthCorner[1]] = "3"
```

In this final section of code it then edits the levelList and changes the values for the corners that were found to 3, with signifies to the code in the createDots() procedure that it needs to render a larger dot (16x16). The code for this can be seen below.

```
        elif int(levelList[i][j]) == 3 and 0 < i < 15:
            pygame.draw.rect(GUI, (255, 255, 255), (i*square + 24, j*square + 24, 16, 16))
```



As can be seen in the screenshot above, the code is able to place the power up dots in each corner of the map, they are not generated all the way into each corner, but I don't see this as an issue because it doesn't affect gameplay. This is because, as long as they are somewhere in each corner and a far away from each other, it does not negatively effect how the game is played.

Step 2:

```
class Ghost():
    def __init__(self, image, posX, posY):
        self.image = image
        self.posX = posX
        self.posY = posY
        self.ogPosX, self.ogPosY = posX, posY
        self.currentImage = 0
        self.count = 0
        self.dirChange = True
        self.direction, self.newdirection = 0, 0
        self.oldX = 0
        self.oldY = 0
```

In the above section of code I have created and initialised the Ghost() class and added all the required attributes that is needed for the movement method. The class has parameters of image, posX and posY to signify the image for the sprite and the default position of the sprite. It stores this in both, self.posX/posY and self.ogPosX/ogPosY because it needs to keep a copy in an attribute that wont be edited as well as the normal one. CurrentImage is set to 0 and will be changed with the code later on that adds the other mode where Pacman kills the ghosts as they have a different sprite.

```
def move(self): #handles ghost movement
    square = 1024 // lineCount
    posX = int((self.posX) // square)
    posY = int(self.posY // square)

    leftCheck, rightCheck, upCheck, downCheck = False, False, False, False #checks where the ghost is allowed to turn
    if 0 < self.posX < 960 and 0 < self.posY < 960:
        if self.posY / square == posY and int(levelList[int((self.posX - 1) // square)][posY])  != 1:
            leftCheck = True
        if self.posY / square == posY and int(levelList[int((self.posX) // square) + 1][posY]) != 1:
            rightCheck = True
        if self.posX / square == posX and int(levelList[posX][int((self.posY - 1) // square)]) != 1:
            upCheck = True
        if self.posX / square == posX and int(levelList[posX][int((self.posY) // square) + 1]) != 1:
            downCheck = True
```

This next section of code is the start of the move method for the ghosts and controls how and where they move. It starts by setting the value if posX and posY, being the current grid squares that the ghost is in. This is used in the next bit where it uses the same code from the Pacman class to calculate whether or not it can move to the next grid square in a given direction. It tests all 4 directions by checking levelList to see if the next square in that direction is clear. If it is, it will set that directions check variable to True to signify to later bits of the code that it is an available direction to turn.

```
if leftCheck and random.randint(3,4) == 4 and self.direction != 1: #randomly selects a direction to turn
    self.newdirection = 0
elif rightCheck and random.randint(3,4) == 4 and self.direction != 0:
    self.newdirection = 1
elif upCheck and random.randint(3,4) == 4 and self.direction != 3:
    self.newdirection = 2
elif downCheck and random.randint(3,4) == 4 and self.direction != 2:
    self.newdirection = 3
elif leftCheck and self.direction != 1:
    self.newdirection = 0
elif rightCheck and self.direction != 0:
    self.newdirection = 1
elif upCheck and self.direction != 3:
    self.newdirection = 2
elif downCheck and self.direction != 2:
    self.newdirection = 3
```

Above is the next bit of code that controls where the ghost will turn next. My original plan from my design section was to use the A* algorithm to govern where the ghosts will turn to make sure they go towards Pacman. However, I have changed my mind about this as that would increase the complexity of my program by quite a lot and make the game far to difficult as the original Pacman has other, also very complex, algorithms to govern where they turn. This is why I have decided to make them turn randomly, this still makes the game quite difficult as you need to get all the dots whilst avoiding the ghosts and you cant predict where they will go. The code above shows how I have achieved this. I used the check variables from earlier to see the options of where it could turn, then made used a random integer (checking if it is equal to one of the possible outputs) to randomise the turns as well making sure it doesn't turn 180 degrees as that could make the ghosts repeatly turn back and forth. The next 4 if statements lack the randomiser bit as all of the if statements above it could return False, meaning the ghost would not turn, so the ghost will then turn with a priority order of left, right, up, down, depending on the possible ways it can turn. Inside each of the if statements it sets the newdirection attribute to a number corrisponding to the direction it changes to, which will be used later in the method.

```
if self.posX == self.oldX and self.posY == self.oldY: #makes sure ghosts goes back if reaches end of hallway
    self.dirChange = True
self.oldX = self.posX
self.oldY = self.posY
if self.dirChange:
    self.newdirection = random.randint(0, 3)
    self.dirChange = False
```

This bit checks to see if the ghost hasnt moved, it does this by using an if statement to see if the current position is the same as the old position, the variables of which are set after the if statement for the next time the code loops around. If this is True, then it sets the attribute dirChange to True which activetes the next bit of code to set a new direction that is random and sets dirChange back to False. This stops the ghost from getting stuck at the end of a hallway.

```
match self.newdirection: #checks if direction is set to be changed by the above code
    case 0:
        #left
        if leftCheck:
            self.direction = 0 #direction change
    case 1:
        #right
        if rightCheck:
            self.direction = 1 #direction change
    case 2:
        #up
        if upCheck:
            self.direction = 2 #direction change
    case 3:
        #down
        if downCheck:
            self.direction = 3 #direction change
```

This next bit uses a match case statement to see what newdirection is (which was set above) and checking if it is able to go in that direction, then (if it is True) setting the actual direction attribute to the corrosponding number. This will be used in the next section of the code.

```
match self.direction: #moves Ghost in chosen direction
    case 0:
        #left
        if 0 < self.posX < 960 and 0 < self.posY < 960:
            if int(levelList[int((self.posX - pacSpeed) // square)][posY])  != 1:
                self.posX = round((self.posX - pacSpeed), 1)
        else:
            self.posX = round((self.posX - pacSpeed), 1)
    case 1:
        #right
        if 0 < self.posX < 960 and 0 < self.posY < 960:
            if int(levelList[int((self.posX) // square) + pacSpeed][posY] ) != 1:
                self.posX = round((self.posX + pacSpeed), 1)
        else:
            self.posX = round((self.posX + pacSpeed), 1)
    case 2:
        #up
        if 0 < self.posX < 960 and 0 < self.posY < 960:
            if int(levelList[posX][int((self.posY - pacSpeed) // square)] ) != 1:
                self.posY = round((self.posY - pacSpeed), 1)
        else:
            self.posY = round((self.posY - pacSpeed), 1)
    case 3:
        #down
        if 0 < self.posX < 960 and 0 < self.posY < 960:
            if int(levelList[posX][int((self.posY) // square) + pacSpeed]) != 1:
                self.posY = round((self.posY + pacSpeed), 1)
        else:
            self.posY = round((self.posY + pacSpeed), 1)
```

The code in the screenshot above shows how the change of direction translates to where the ghost ends up moving. It starts by using a match case statement to see which direction was chosen in the above code. It then checks, using an if statement, to see if the ghost is within bounds of the map, if True then it will only move the sprite if the next space is clear, if False it will move the sprite regardless. It moves the sprite at the same speed as Pacman, as can be seen with the use of the pacSpeed variable. It moves the sprite by that amount in the given direction then rounds that value to reduce any glitches or errors. It moves the sprite by overriding the posX attribute which determines the location that the sprite will be rendered at. This code has also been mostly reused from the Pacman testMove() method.

```
self.count += 1
if self.posX > 1024 and self.direction == 1: #loops sprite back round
    self.posX = -40
elif self.posX < -40 and self.direction == 0:
    self.posX = 1064
elif self.posY > 1024 and self.direction == 3:
    self.posY = -40
elif self.posY < -40 and self.direction == 2:
    self.posY = 1064
```

This next bit is simply the same code from the testMove method of Pacman from prototype 2 reused to make sure that the ghosts can loop back to the other side of the map when it goes out of bounds.

```
def render(self):
    GUI.blit(self.image, (self.posX, self.posY)) #renders the ghost sprite onto the display
```

Here is the method of the Ghost() class that renders it onto the screen, it takes the image assigned to it and puts it in the position dictated by the posX and posY attributes used in previous methods.

```
blinky = Ghost(redGhost, 448, 448)
pinky = Ghost(pinkGhost, 448, 448)
inky = Ghost(blueGhost, 448, 448)
clyde = Ghost(orangeGhost, 448, 448)
```

Here is where all the instances of the Ghost() class are made in the main program it give the image and defualt position. The names are the same as the ghost names in the game.

```
redGhost = pygame.transform.scale(pygame.image.load("assets/images/redGhost.png"), (64, 64))
blueGhost = pygame.transform.scale(pygame.image.load("assets/images/blueGhost.png"), (64, 64))
pinkGhost = pygame.transform.scale(pygame.image.load("assets/images/pinkGhost.png"), (64, 64))
orangeGhost = pygame.transform.scale(pygame.image.load("assets/images/orangeGhost.png"), (64, 64))
```

Here is where I imported the assets for the ghost into the program and resized them to the same 64x64 size of the Pacman sprite.

```
def renderGhosts(): #calls all render ghost methods
    blinky.move()
    blinky.render()
    pinky.move()
    pinky.render()
    inky.move()
    inky.render()
    clyde.move()
    clyde.render()
```

I have also created a renderGhosts procedure in the main program that calls the methods in the Ghost() class.

```
renderGhosts()
```

Then I also called this procedure from the main loop of the program.

Step 3:

This step is very simple, I just need to make sure that the lives attribute for Pacman is set to 3 and then check in the main program that it is 0, if True, then quit the game. I also need to add code that counts up the remaining dots to also check if the player has won.

```
class Pacman():
    def __init__(self, posX, posY):
        #default lives, starting position, and animation markers
        self.lives = 3
```

Here, you can see the bit I added the the Pacman initialisation to make the lives attribute equal to 3.

```
def countRemainingDots(): #counts how many dots are left to see if the player has won
    count = 0
    square = int(1024 // lineCount)
    for i in range(len(levelList)):
        for j in range(len(levelList[i])):
            if (int(levelList[i][j]) == 0 or int(levelList[i][j]) == 3) and 0 < i < 15:
                count += 1
    if count <= 0:
        return False
```

This code traverses levelList using very similar code to what I have used many times to see how many of the spaces are still dots, if there is none left the function returns False back the the main program.

```
if frames % 10 == 0:
    if countRemainingDots() == False:
        win = True
if pacMan.lives == 0:
    lose = True
```

This is the part of the main program that checks whether the player has won or lost. As shown, every 10 frames (to make sure it doesn't have to run a loop every single time the game loops) it calls the countRemainingDots() function and checks if the output is Flase using an if statement. If it does return False it sets the win variable to True. As well as this is checks if pacMan.lives is equal to 0 using another if statement, if so, it sets lose to True.

```
win = False
lose = False
```

This is where the win and lose variables are defined in the main program and by default are both set to False.

```
if gameQuit: #breaks out of loop
    break
if win or lose:
    gameQuit = True
```

Here, in the main game loop it checks a variable called gameQuit and breaks out of the main loop (which will close the program). The next if statement checks whether lose or win is True, if either are, then it sets gameQuit to True, which will quit out of the game.

Step 4:

```
def setOgPos(self): #respawns in center
    self.posX, self.posY = self.ogPosX, self.ogPosY
def allGhostRecall(self): #respawns all ghosts
    blinky.setOgPos()
    inky.setOgPos()
    pinky.setOgPos()
    clyde.setOgPos()
```

I start by creating 2 methods in the Ghost() class, setOgPos() and allGhostRecall(). The setOgPos sets the ghosts current position to the original one set in the parameters of the object.

AllGhostRecall() calls this method for every ghost object that I created so that I can easily make them all go back to their default locations. I also added the exact same setOgPos() method to the Pacman() class.

```
def checkCollide(self): #checks if Pacman and the ghost collide with each other
    if -40 < pacMan.posX - self.posX < 40 and -40 < pacMan.posY - self.posY < 40:
        pacMan.lives -= 1
        pacMan.setOgPos()
        self.allGhostRecall()
        print("life lost")
```

Then I created a checkCollide method in the Ghost() class which includes an if statement that checks if the difference between the pacMan objects x and y position with its x and y position is less than 40. If this is True then it decreases the lives attribute of pacMan by 1 and calles the pacMan object setOgPos() method that I made earlier. Then calls the allGhostRecall() method to send them all back to their original positions. Then prints to the console “life lost”.

Review of Final Version:

Steps	Description	Completed?
1	Add power-up dots to the map	y
2	add ghost class and program their movement	y
3	add limited lives	y
4	add ghost and Pacman collision (Pacmal lose life)	y
5	add power-up mode (Pacman can kill ghosts)	n
6	add points counter	n

I have completed 4 of my 6 steps for my final version. I was not able to do the last 2 as when creating my plan I under-estimated how much time the steps would take and how complex they would be (mainly step 2). This means that I was not able to fully complete my final version as I had planned it to be, however I think I have still made a very large amount of progress since prototype 2, so I am still happy with the amoun that I was able to complete. If I were to have another version then I would definitely make sure to complete the power-up mode and points counter in that version.

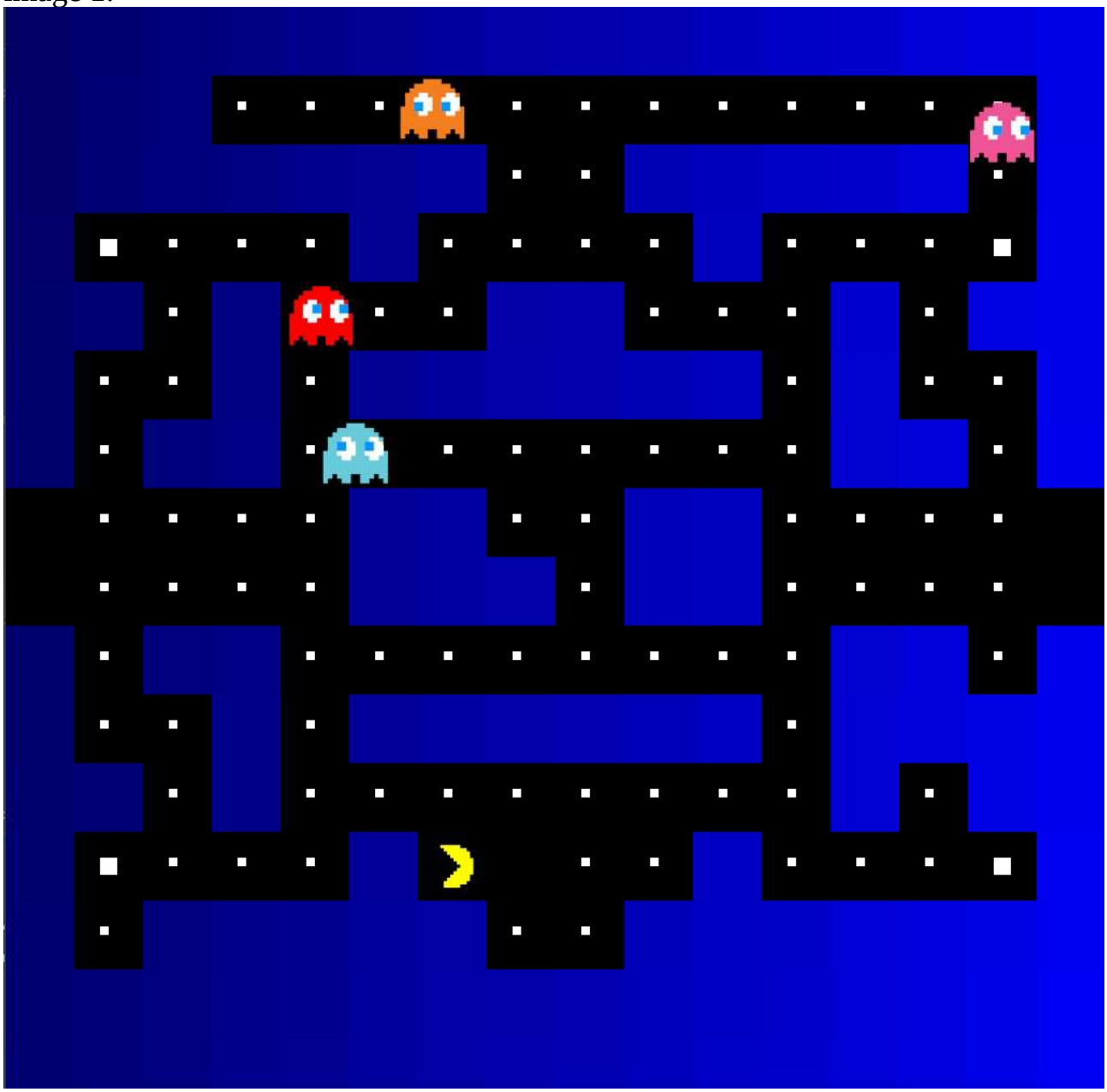
Testing of Final Version:

My testing strategy includes testing the normal functions of the program, extreme function (edge cases) and erroneus functions (cases that the program is not designed to deal with).

Test	Type	Test for...	Expected	Result	Next steps
1	Normal	Do Ghosts move correctly?	Move in random directions around the map	As expected	No next steps. Test succeeded see image 1.
2	Extreme	Do ghosts loop back when out of bounds?	Ghosts appear on other side of map when they go out of bounds where no walls	As expected	No next steps. Test succeeded

3	Normal	Do the power-ups appear?	Power-ups appear in corners of map	They do appear not exactly in corner, but close enough as to not affect gameplay	Not fully necessary, but could fix the code that finds where the corners are to find the exact corner – see image from step 1.
4	Normal	Does life get lost when touching ghost	When Pacman sprite collides with any ghost sprite, all sprites get put back to default locations and life is lost	As expected	No next steps. Test Succeeded see image 2
5	Erroneous	Make less than 4 spaces so that not all power-ups will be generated	Will only generate 1 in the single space	As expected, though Ghosts and Pacman are stuck inside walls as well	Make sure warning is given when there is no space where the sprites are supposed to be and tell the user to change how they have the map configured.
6	Erroneous	If only 1 axis is aligned with a ghost, what happens?	Nothing, Pacman will not lose a life	As expected	No next steps. Test Succeeded

Test Screenshots:
image 1:



as seen in the screenshots all the ghosts have gone from their starting place, in random directions, proven by the fact that they arent all in the same place even though they have the same starting position

Image 2:

As can be seen, the console has printed life lost. From the code this happens when I collide with a ghost, which I did (could't screenshot fast enough to show the collision).

Testing Overview:

The program seems to be very robust, except for when I reduce the amount of free square to 1 and it stops the game from being playable as the sprites cannot move when they are in a wall by default as there are walls all around them. However playing with a very small map like this would be unlikely, but I do think the program could be improved by including a warning to say that certain areas need to be kept open to make sure the sprites can move in their default locations.

Evaluation Section

Success Criteria:

The table below is from my analysis section detailing the success criteria that I had for my final program before I started developing it. Each criteria has its own number so that I can reference to it later in my evaluation.

Number	Requirement	Criteria
1	Moveable Pac-Man sprite	User can control Pac-Man sprite using WASD and arrow keys Sprite will turn in corresponding direction Sprite will move in new direction
2	Sprite Collides with map/maze	Sprite can move freely back and forth through corridors Sprite won't turn toward or move into walls Sprite can only change direction left/right when directly on grid square of map
3	Proper Ghost Movement	Ghosts need to go towards Pac-Man, taking the shortest route through the maze Ghosts cannot turn around 180 degrees at any point (mechanic from the original game)
4	Pac-Man and Ghost Collision	In normal mode (without power-up) Pac-Man colliding with Ghost causes life lost and all sprites re-spawn. In Power-Up mode, collision will cause ghost to re-spawn and extra points to the player
5	Ghosts re-spawn	When Ghost or Pac-Man dies, or the game is starting, Ghosts need to spawn in centre of maze.
6	Dots disappear when colliding with Pac-Man	When Pac-Man sprite collides with any dot in the maze it needs to disappear. At this point, the player needs to gain points.
7	Sprite maze roll-over	When the Pac-Man or Ghost Sprite moves past edge of map it should come back at the other side
8	power-up/items correct spawn locations	When the game starts, the power-ups need to be slightly larger versions of the dots and need to be in the 4 corners of the map items need to appear at intervals during the game in the middle of the map
9	Win detection, new level	When the player has gotten all the pellets, including the power-up ones then the game stops and it moves onto the next level
10	Sound effects	Make sure that the Pac-Man animation sound effects are always playing, and that the power-up, item

		and ghost kill sound effects all play correctly with no delay.
11	Main menu before game starts	Menu with games name, instructions on how to play, and play button. Game should start when the user clicks on the play button
12	Save game functionality	When the user closes the game while in the middle of a game, the state should be saved and an option to continue that specific game should appear when they start up the game the next time
13	New maps / mazes	The game should have new mazes for the player to play in rather than just the one from the original. This will give the game a bit more variation and improve on the original

Testing Program with respect to success criteria:

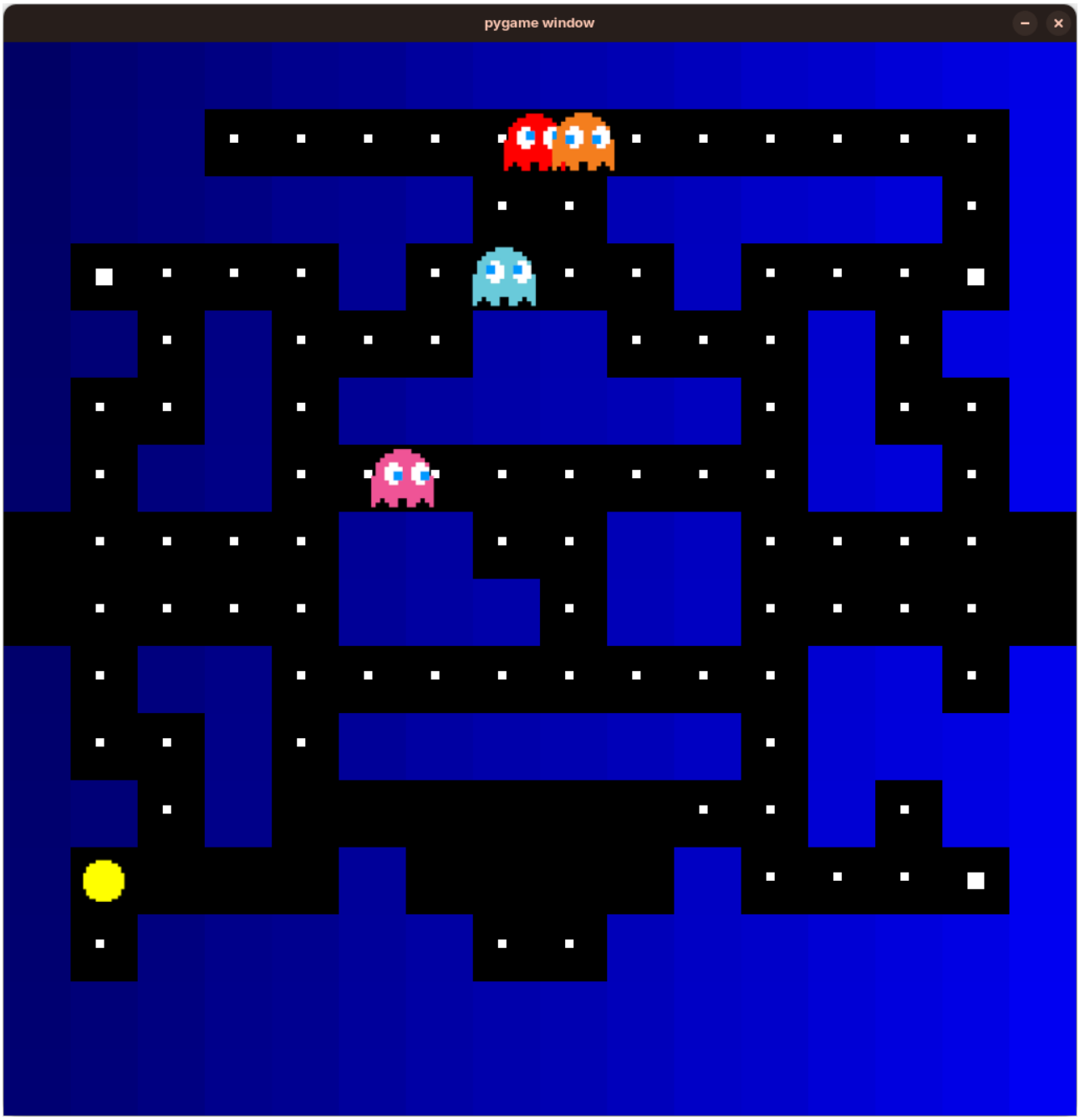
because not all of my success criteria have been met, I have created I final test table against my original success criteria to show what has, partially has, or hasn't been met.

Test No.	Criteria	Plan	Result	Analysis
1	Moveable Pac-Man sprite	User uses WASD as well as arrow keys and check whether the Pac-Man sprite moves in the corresponding directions	Pac-Man moves, turns and animates correctly in given direction when changed by user. Works with both WASD, and arrow keys.	The Success criteria was fully met
2	Sprite Collides with map/maze	User tries to direct the Pac-Man sprite into walls of maze. Also watches to see if any ghost sprites go through any walls for 5 mins.	Pac-Man sprite is always stopped in place before going through any walls, no ghosts went through any walls after moving around for 5 mins.	The Success criteria was fully met
3	Proper Ghost Movement	Check that the ghost always goes towards Pac-Man and never turns 180 degrees backwards unless hitting a dead-end.	As expected from limitations I ran into while creating my final version, Ghosts only move in random directions. But they never turn 180 degrees unless at dead end.	The success criteria was partially met as the 180 degree rule in functional but the navigation algorithm is less complex than was originally described in success criteria.
4	Pac-Man and Ghost Collision	To check this, the user purposefully sends the Pac-Man sprite towards a ghost and sees what happen when they collide. Does this 3 times to check if game quits when 3 lives lost.	When Pac-Man sprite collides with ghost sprite all sprites get reset to their starting position correctly. When done 3 times the game closes to show that the life count is functioning	The Success criteria was fully met

5	Ghosts re-spawn	User starts game, checks whether the ghosts spawn in centre. Also checks that they spawn in the same place when re-spawning after a life is lost.	Spawn locations work as expected as long as the map is one which has an opening in the centre, doesn't work if not.	Success criteria partially met. Fully works as long as the map has a opening in the centre, but does not if the user has made extra maps that don't have this.
6	Dots disappear when colliding with Pac-Man	User simply plays the game, checking whether dots disappear when colliding with the Pac-Man sprite	This is working fully as expected, dots disappear as Pac-Man goes over them, creating a seamless transition as you cant see them while they disappear	Success criteria fully met, while also being improved on to make a more seamless transition. This is shown in screenshot 1
7	Sprite maze roll-over	User attempts to go off the edge of the map and checks whether the Pac-Man sprite re-appears parallel on the other side	This works as expected, however one thing to note is that when on the outer edge the sprite cannot rotate 90 degrees in either direction, only forward or backward	Success criteria fully met. But ideally if I had more time I would add the ability to turn left and right on the maps outer edge. This is shown in screenshot 2
8	power-up/items correct spawn locations	Check that the Pac-Man power ups show up at the 4 corners of the map, even when the map is configured differently. Check that the items show up in the centre at regular intervals	The power-up placement works fully as expected even when map is configured differently. But the items feature is completely absent, as is known from my final version analysis.	Success criteria partially met. Power-ups work correctly. Items completely absent. This is shown in screenshot 2
9	Win detection, new level	User wins the game by collecting all pellets, checks game win detection. Checks if new level shows up afterwards	The games win detection works fully as expected but closes the game currently as I was unable to add the feature to go to subsequent levels.	Success criteria partially met, no extra levels.
10	Sound effects	User plays game and listens to see if sound effects play at correct times (when collecting pellets, losing etc.)	No sound effects play as I ran into time constraints trying to add this feature.	Success criteria not met.
11	Main menu before game starts	User simply opens the game and checks if the main menu graphical user interface shows up.	This does not happen as the feature is absent.	Success criteria not met
12	Save game functionality	Game continues when user closes and re-opens the game	This does not work as this feature is absent from my game	Success criteria not met
13	New maps / mazes	Check that the user can choose new and different maps/mazes	This specific feature is absent but, I have instead added a feature so that users	Success criteria not met. But general idea has been implemented in a different way.

			can create their own and change maps themselves easily using a text file. They need no knowledge of how to code	Can be seen in screenshot 3

Screenshot 1:

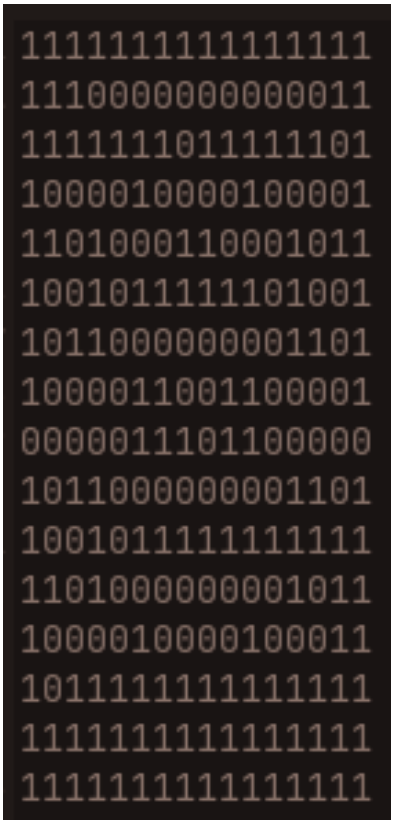


As can be seen, where the Pac-Man sprite has gone, the pellets have disappeared showing how this feature is working as expected.

Screenshot 2:

At the current location of the Pac-Man sprite. The user is unable to make it turn left or right (up or down relative to the axis) when in the outer border location as shown here. Also, for test 8, you can see all of the power-ups have spawned in the 4 corners of the map correctly so that they line up with each other.

Screenshot 3:



As can be seen with the 2 images above, in the text file a 1 corresponds to a wall and a 0 corresponds to a blank space. The left image is the text file that is used to create the map shown in-game on the right image.

Limitations:

Below is a list of the limitations of my program. It shows all of the original success criteria from the analysis section that I have not met in my final version of the program.

Criteria	Details	Evaluation
Sound effects	Make sure that the Pac-Man animation sound effects are always playing, and that the power-up, item and ghost kill sound effects all play correctly with no delay.	I was unable to add this feature as I had prioritised features that more effected the gameplay rather than extras. This would have also required getting the original Pac-Man sound effects as .wav files which would have taken up a lot of time with finding and converting the files.
Main menu before game starts	Menu with games name, instructions on how to play, and play button. Game should start when the user clicks on the play button	This feature has not been implemented in my final version but does not effect the actual gameplay in any way, it just makes the game seem less polished and finished.
Save game functionality	When the user closes the game while in the middle of a game, the state should be saved and an option to continue that specific game should appear when they start up the game	I was also unable to implement this part which does partially effect the functionality and usability of the game. It's easy to see how the lack of this feature could affect the user

	the next time	experience of the game as you can't stop and continue your game again. This could cause annoyance and inconvenience for the user.

Below is a table showing a list of all the original success criteria that have only been partially met in my final version.

Criteria	Details	Evaluation
Proper Ghost Movement	Ghosts need to go towards Pac-Man, taking the shortest route through the maze Ghosts cannot turn around 180 degrees at any point (mechanic from the original game)	My implementation only has the ghosts turn in random directions at every intersection. This can affect the complexity and fun of the gameplay as the ghosts do not actively chase you like in the original game. The game is still playable and does still strongly resemble the experience of the original game
Ghosts re-spawn	When Ghost or Pac-Man dies, or the game is starting, Ghosts need to spawn in centre of maze.	I have fully implemented this, but it has a small issue where if the user creates a map where the center is an unavailable space, the ghosts will be stuck and the game wont function properly. This does not affect the gameplay very much as it only happens when you create a specific map with no available space in the centre
power-up/items correct spawn locations	When the game starts, the power-ups need to be slightly larger versions of the dots and need to be in the 4 corners of the map items need to appear at intervals during the game in the middle of the map	The power-ups go to the correct location, providing a good user experience as there are no issues with that part. But there are no items at all in my game which will reduce the variation of the experience.
Win detection, new level	When the player has gotten all the pellets, including the power-up ones then the game stops and it moves onto the next level	Win detection fully works, meaning the player gets the validation of winning and is able to win the game which is a key feature of the game. However I didn't implement the code to take the user to another level as I haven't made any pre-made levels apart from the first one
New maps / mazes	The game should have new mazes for the player to play in rather than just the one from the original. This will give the game a bit more variation and improve on the original	I have not implemented any new maps to the game but the user is able to make them themselves which means that the user experience isn't very degraded and I believe that it allows the user to have more control over their own experience.

Testing for Robustness:

Test	Plan	Result
Test for program crashes that may occur during normal use	Use the program as the user would, experiment when changing the map configuration	Validation is build directly into my program as it will only let you turn when you press the key a within a few

		seconds before that turn is available, if not it is ignored. This means that it does not crash with invalid inputs. However I have found that if you do an invalid map as part of the text file, this is not validated and will (in some cases) cause the program to crash. For example if I create a map where the ghosts and Pacman both cant show up, it does seem to crash.
Test for any instability issues that cause the game to run unexpectedly	Try using the program as normal and (ignoring crashes) look for any anomalies in sprite movement, pellet collection etc.	Found no such anomalies, program ran as expected with no instability issues that cause any glitches

Further Maintenance:

maintenance for programs is the process of keeping the program working, fixing issues that are found and making sure you remove bugs and glitches as you find them during the programs life cycle.

This process would be required for fixing some of these issues:

- updating the program to support power-up mode
- updating to support ghost chase mode (algorithm to make ghosts move towards Pac-Man sprite)
- Fix issue where invalid map text file where there are no spaces for any sprites to go causing app to crash (implement validation to tell the user why it has happened and how to fix it, then reset the map to the default)

The use of modularity (functions, procedure, OOP, etc.), explanatory identifiers, and comments is very helpful when trying to maintain and update a program as it allows you (or someone else who didn't write the code themselves) to be able to look at it and understand what is happening. This will make it a lot easier to implement new features and fix bugs or crashes. I think that my use of a common library (pygame), use of lots of OOP methods, separate functions and code comments would make maintenance of my game very easy as it wouldn't take very long to get familiar with how the code works and how certain features have been implemented. However, there are some parts that are quite complex, such as the collision detection. This could definitely be something that could hurt the maintainability of my game. This means that if the collision detection ever needs to be changed or updated, it may end up being quite difficult.

I also think that since I have used pygame in my project, that will help the maintainability of my game. This is because it is one of the easiest to use 2D game engines for python that can easily be installed as a library.

Project Overview:

(green = fully met, amber = partially met, red = not met)

No.	Criteria	Met?
1	Moveable Pac-Man sprite	
2	Sprite Collides with map/maze	
3	Proper Ghost Movement	
4	Pac-Man and Ghost Collision	
5	Ghosts re-spawn	

6	Dots disappear when colliding with Pac-Man	
7	Sprite maze roll-over	
8	power-up/items correct spawn locations	
9	Win detection, new level	
10	Sound effects	
11	Main menu before game starts	
12	Save game functionality	
13	New maps / mazes	

I had originally made 13 success criteria in my analysis section. After creating 2 prototypes and a final version, I have fully met 6, partially met 4, and not met 3.

Program Usability Test:

I have chosen a person to test my program and I am going to give them a questionnaire and let them give a response about how their experience playing the game including the strengths of the game and the difficulties found when testing it. I will be using the tests I designed in the design section.

The user is a very experienced software engineer and likes quick and simple games like what I have made. I think that this will make them a very good match to the ideal stakeholder of my game. This will make their feedback very useful and strongly representative of what many of the others part of my target audience.

Questionnaire:

1. Do you think that the controls are intuitive and easy to understand?
    - Yes. I was able to use the up, down,left and right without being instructed. The other set of control buttons of w, a, s, d are also quite standard controls in this type of game.
    - The movement related to the controls seems very responsive.

2. How difficult would you say the game is, and do you think it should be any more or less difficult?
    - The game is not difficult once you get used to the layout.
    - The game would be more challenging if the monsters were to chase the pacman.

3. does it resemble the original game enough so that it is easily recognisable?
    - Yes, the game has the look and feel of the original pacman.

4. how would you rate the experience of changing the games map using the text file?
    - This is a good feature but it is not easy to create a new map.

5. do you find it easier to understand how to win the game?
    - Yes, it was clear the the player needs to clear all the dots to win.

6. what do you think to the aesthetics and design of the GUI / game?
    - The window is a good size but could be a bit bigger or full screen to make the player feel fully immersed in the game.

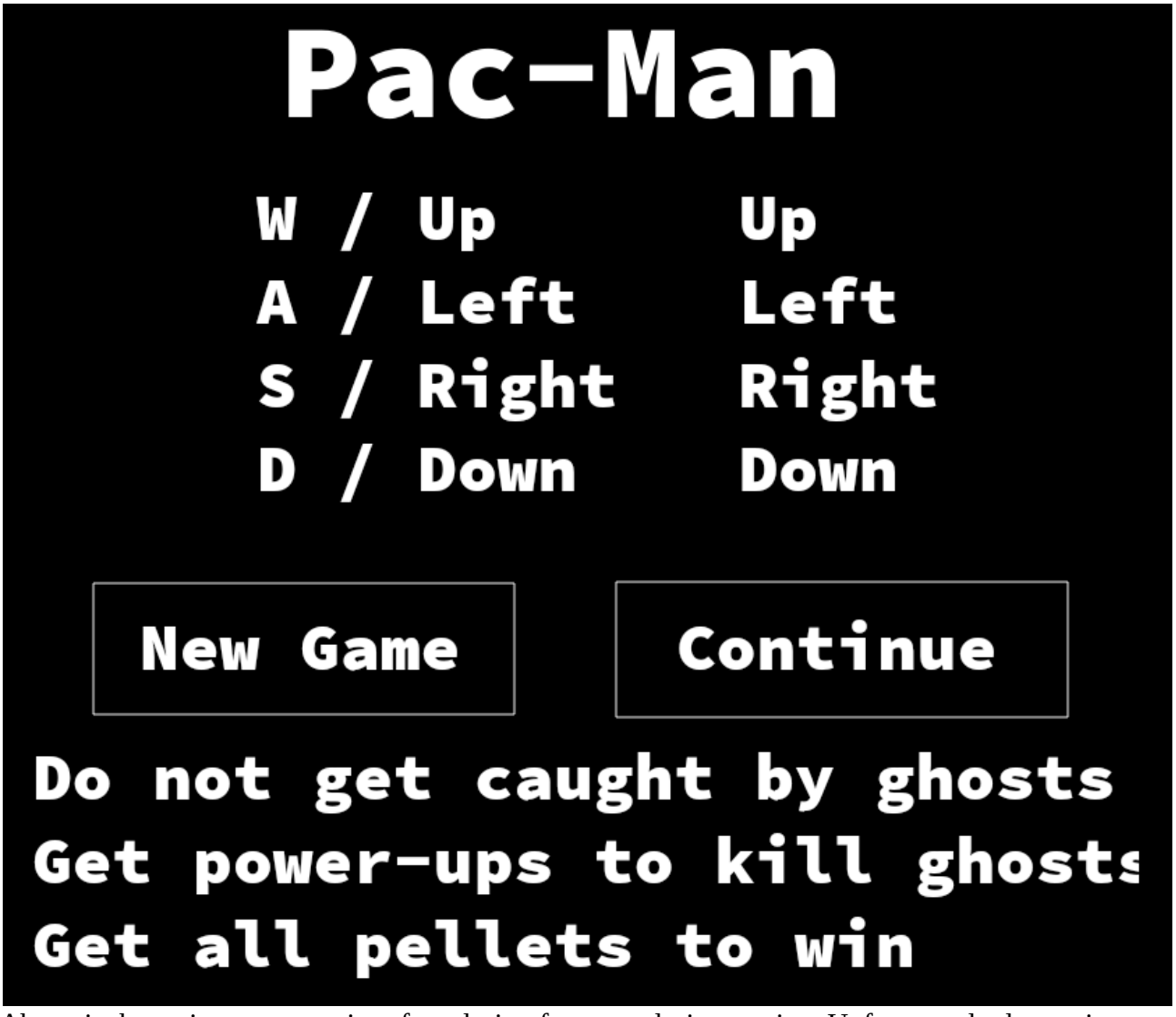
7. is there anything that worked differently as to what you expected?
    - As a player I would have expected the ghosts to me more attracted to the pacman.
    - Wasn’t sure what the larger dots did or were for.

Overall Experience	Strengths	Difficulties
The game was easy to play and enjoyable.	The graphics are clear and controls responsive. Good reproduction of the basics of the original game.	Needs more interaction from the ghosts to keep the player interested.

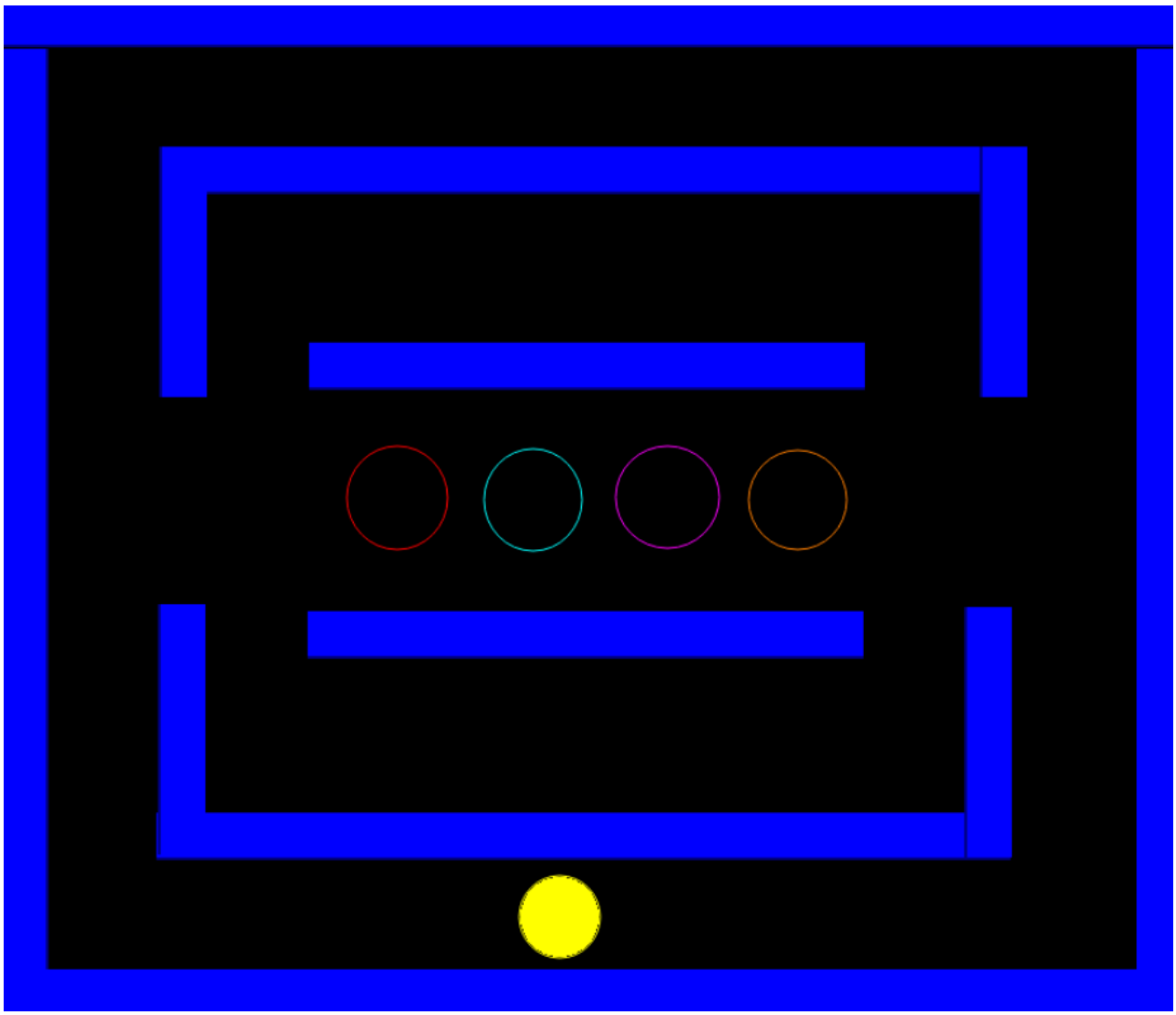
Users Suggestions for new features and updates:

- A scorebord to show points scored.
- Ghosts to follow the pacman more.
- Ability to go into attack mode to kill the ghosts
- A gameover screen rather than jumping out of the program

Usability design overview:



Above is the main menu user interface design from my design section. Unfortunately due to time constraints, I have been unable to create this in my final version of my program. However this would be something that I would definitely want to do in future for my program as it would really boost the user experience of my program as it has many usability features as outlined in my design section that I am now missing.



Above can be seen the in-game design concept from my original design section. I believe I have achieved the usability and design of this and more. In comparison, my final program looks much more polished and has a lot more space for the user to move around in. My final version also includes a gradient that makes the user interface look less bland and more inviting. The usability features of the UI in my final program are much better than what I wanted originally. I now have a much more interesting map with many more different corridors and more to do.

My final program hasn't got all of the usability features I outlines in my analysis and design section. This include the main menu, the life counter and points counter. This means that in these areas my final version isn't as usable as I had initially wanted it to be before I started creating my project.

The main menu would have given a significant boost to the usability of my program as it would have given the user some information about how to play the game and what you are supposed to do. It would have also meant the the game doesn't abruptly start right as you open the program, this can be overwhelming and possibly hurts the usability of my game.

As for the life and points counters, the points counter alone wouldn't have made it much more usable as there is only one level, but the lives counter definitely would have upped the usability because it would allow the user to easily know how many lives they have left instead of having to count how many lives they have lost like in the current state of the program.

Further Development:

After evaluating my program in this section I have shown that there are additions and improvements that could be made to my program so that it could be better and meet more of my success criteria.

Extra Features:

- Add the power-up mode so that Pac-Man can temporarily kill the ghosts (improves gameplay)
- Add Sound effects when getting pellets, power-ups and items
- Add items so user can get more points and adds a new layer to the gameplay
- Add points and life counters (improves usability)

Improve Features:

- make the win detection code send the user to the next level
- make extra pre-made levels for the game
- make ghosts use A* algorithm to always go after Pac-Man

Further Development Review:

Feature	**Action**	**Difficulty Level**	**Impact on program**
Power-Up mode	Requires additional sprite image for ghosts (dark blue). Create code to detect when power-up is gotten then change modes for 10 seconds then change back. When in this mode Pac-Man/Ghost collisions need to kill ghost instead of Pac-Man and add extra points	Med	Vastly improves the gameplay and give a lot more content for the user.
Sound Effects	Pygame has built in way to add sound effects. Would only require sourcing the sound effects as .wav files and adding them to the program using pygame. Single command to play them when needed	Low	Somewhat improves gameplay as it will make it more immersive and fun for the user
Items	Add extra sprite to the game using pygame. Make appear in center at regular intervals over top of pellets	Med	Adds extra content and makes the game more interesting
points/life counters	Simply increase the height of the game window, add text at the top using pygame that displays the point counter, add counter variable that increases for every pellet. Make as many Pac-Man images appear at the bottom based on the value of the lives attribute	Med	Shows the user how they are doing so they can better strategise their gameplay
Next level upon win	Put the text file loading code into a function and call it upon win with the second levels text file as a parameter and reset the game.	High	Lets the user carry on playing and possibly move on to more difficult maps

Extra levels	Simply just add text files with new levels in them using 1 for wall and 0 for space.	High	Gives the user much more extra content to play through.
Ghost A*	Code the A* algorithm in python to the ghost class using the map generated from the text file showing all the paths it can go and Pac-Man as the target destination which is always changing. Algorithm needs to run very often to make sure it is always going towards Pac-Man's current position.	Very High	Makes the game much more interesting and difficult requiring much more strategic ways of playing that will make the user experience much better.