

Computer Science A-level project

Pacman

Stanislaw Szymanowicz

Analysis of the problem	5
Problem Identification	5
Stakeholders	6
Research of the problem	6
Specification of the solution	9
Features of the solution	9
Limitations of the solution	11
Software and hardware requirements	11
Hardware requirements	11
Software requirements	11
Success criteria	12
Design	12
Decomposing the problem	12
Structure of the solution	17
Algorithms	20
Usability features	36
Key variables and structures	41
Test data for development	42
Test data for beta testing	43
Development – the first iteration	44
First iteration - introduction	44
First iteration - development	44
Declaring walls	45
Moving objects	52
Attributes and methods	52
Moving the objects	53
Collision detection	54
Turning algorithm	55
Tunnel	60

Coins	60
Enemies	62
Graphs	62
Node	62
Edge	63
Graph	63
Creating the graph of the map	65
Movement	70
Exit conditions	72
First iteration – summary	73
Consultation with the end user after the first iteration	73
Development – second iteration	74
Second iteration – introduction	74
Correction of the graph implementation	74
Colours and shapes in the game	78
Tunnel – bug	79
Score and lives	80
Depth First Search	84
Enemies using DFS	86
Second iteration – summary	86
Second iteration – consultation with the user	86
Development – third iteration	87
Third iteration – introduction	87
Starting screen and its functionality	87
Final screen	94
Top scores	103
Evaluation	107
Testing	107
Usability features	112
Controls explained on the start screen	112
Level select	114
Instructions on how to switch screens	114

High score shown	115
Evaluation	120
Matching the requirements	120
Changes to the design	122
Unmet criteria	123
Additional features	124
Maintenance	125
Corrective	125
Adaptive	125
Bibliography	125
Appendix A - Interview #1	126

Analysis of the problem

I knew my friend, Dmitry Rusanov, was looking for an alternative to a classical Pacman game. He liked the original arcade game, but was looking for something a bit different. I decided to create a game for him which would be an adaptation of the original. I started with conducting an interview which would give me information about what he is looking for in a game, what he would like it to look like, what should be the specific features of the game. I was looking for information on how he would like the enemies to behave, what he thinks about counting the score and storing it. The content of the interview is in the appendix.

Problem Identification

The end user said that he enjoys playing the classic Pacman game, but it is too easy for him. He also indicated that the patterns of movement of the enemies can be predicted and the solution to win the game every time can be found. Dmitry wants the enemies to move in an intelligent way, which in his understanding means that the enemies react to the moves of Pacman. This is not the case with current versions of Pacman, as the enemies always move in the same pattern and it can be predicted. The end user also mentioned he likes pushing himself, so keeping his top 10 scores is essential for him. Most of the current versions of Pacman are online, so every time browser is closed, all progress is deleted.

In summary, the problems with the current classic Pacman version are:

- It is too easy: this can be solved by for example increasing enemies' speed or moving them in an intelligent way, for example using shortest path algorithm. In the original version of Pacman the ghosts move in the general direction of Pacman but do not analyse the path. To make the problem more suitable for computational methods one can use path finding algorithms. Computational methods, for example Depth First Search (DFS) or Breadth First Search (BFS) algorithms enable finding shortest paths in graphs. They can also be applied to this case. Increasing enemies' speed can also be solved using Object Oriented Programming and assigning an appropriate value, suited to the user, for the speed of enemies. Using a path finding algorithm would differentiate the game from the classic version of Pacman, because the enemies in the existing version move in the general direction of Pacman, but do not analyse the path.
- The enemies move in a predictable way: this can be solved by applying path finding algorithms or artificial intelligence. Path finding algorithm will enable the enemies to react to the movement of the user, because the enemies will find, via an appropriate algorithm, a path to catch Pacman, which will be the shortest path available to his current, or predicted future position.

- High score is not kept in an online version: this can be solved by saving the high score to a file, retrievable offline, which would be used every time the user plays the game. This feature will require some memory on the hard drive, but every computer has one, so the problem is also solvable using computational methods.

Stakeholders

The solution is developed to suit the needs of my end user, Dmitry Rusanov, who is desperate for a more challenging version of Pacman, in which the enemies' movement is less predictable. He also wants his high score to be kept permanently.

Although the solution will be developed to suit his needs, it will also solve the problems of all users who are fans of classic arcade games, such as Pacman, but they are looking for a challenge and an innovation to them.

Another group which has a stake in this game is the group of highly competitive users, who would like their score (or their top 10 scores) to be stored and not lost every time a browser is closer, so that they can, for example, compare their high score with other gamers.

Moreover, this game would cater for the needs of a group which does not have a free internet access. Current versions of Pacman are online, and most of them need Adobe Flash Player plugin to run in the browser. There are some people however, who do not have a free access to internet and their hardware is not good enough to install Adobe Flash Player and run the game smoothly.

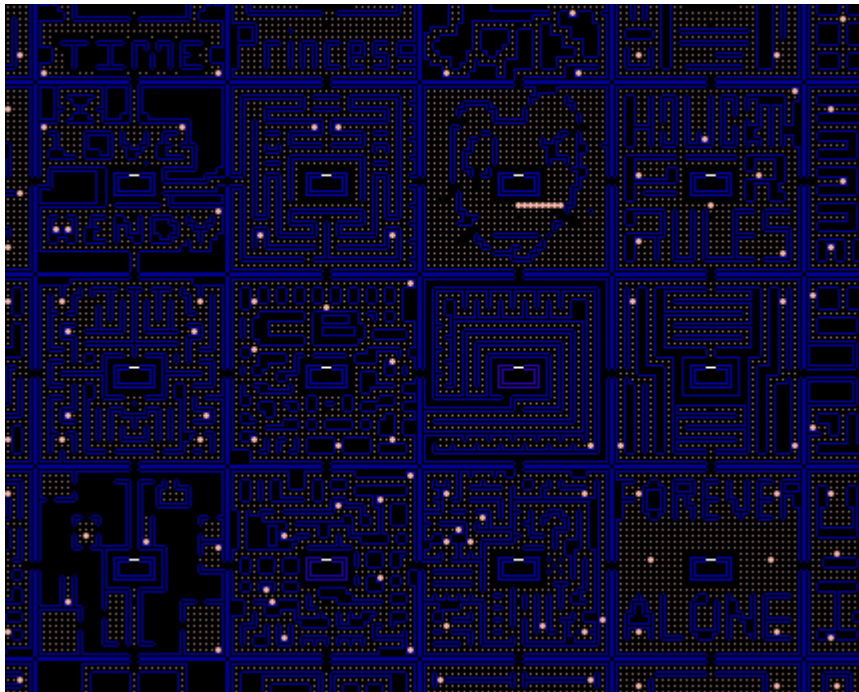
In summary, a typical person in the target audience is a high school or undergraduate student, with an interest in games, in particular classical arcade games. He/she is highly competitive, looks for innovation and challenge in games they are playing.

Research of the problem

There are already different versions of Pacman, for example Google's edition in browser, the one on <http://www.playpacmanonline.net/> or <http://worldsbiggestpacman.com/>, which offers over 200,000 different mazes.

Google home page, May 21, 2010

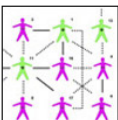




From the ones above, all are fairly similar to the original version. However, only the middle one provides the user with a classic map, display and bonuses. Moreover, all of them are online versions one can play in a web browser, so if one does not have an internet access, they cannot play Pacman. It is not available in shops, because normally people can access it online. My version of Pacman will solve this problem, because it will be an application working offline.

In terms of the algorithms, different implementation of graphs and search algorithms are available. For example, MIT shares their lectures and notes for course 6.001 Introduction to Programming and Computational Thinking on <https://ocw.mit.edu/index.htm>.

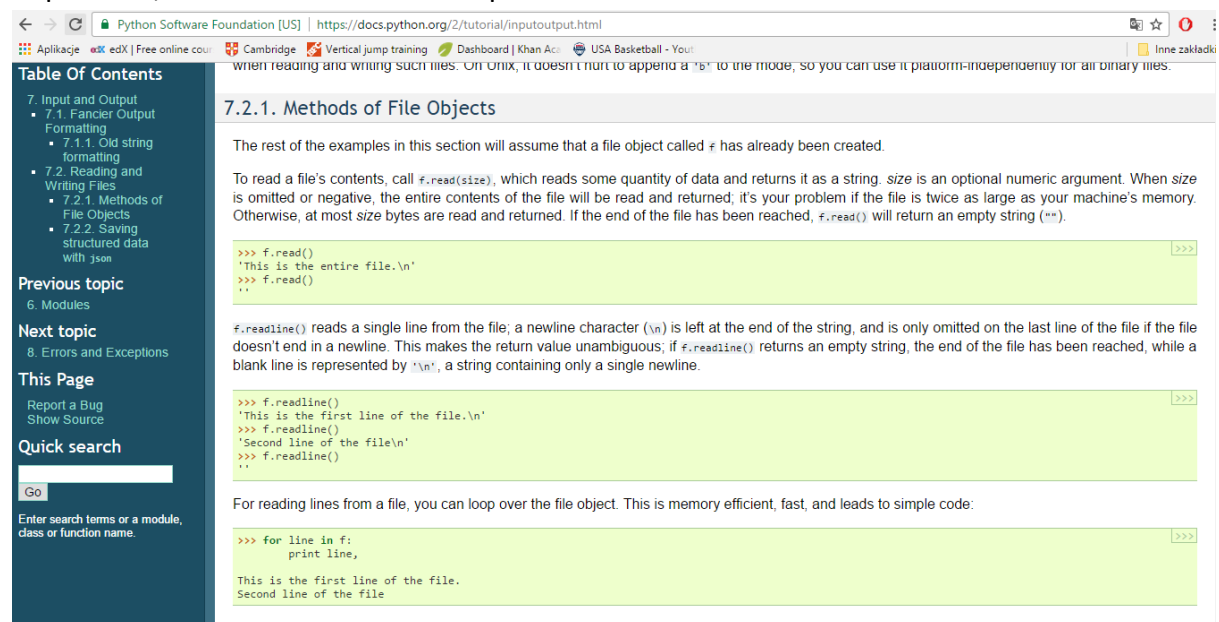
Using Graphs to Model Problems, Part 2

COURSE HOME		OCW Scholar	« Previous Next »
SYLLABUS	Session Overview		
SOFTWARE	 <p>This lecture returns to graph theory. It defines and gives examples of some classic graph problems: shortest path, shortest weighted path, cliques, and min-cut. It then shows how memoization can be used to speed up some algorithms.</p> <p>Centers for Disease Control and Prevention. This image is in the public domain.</p>		
REFERENCES	Session Activities		
UNIT 1	Lecture Videos		
UNIT 2	<p>> Lecture 22: Using Graphs to Model Problems, Part 2 (00:48:54)</p> <p>About this Video</p> <p>Topics covered: Dynamic programming, optimal path, overlapping subproblems, weighted edges, specifications, restrictions, efficiency, pseudo-polynomials.</p>		
UNIT 3	Resources		
MORE CLUSTERING	<p>> Lecture code handout (PDF)</p> <p>> Lecture code (PY)</p>		
USING GRAPHS TO MODEL PROBLEMS, PART 1	Recitation Videos		
USING GRAPHS TO MODEL PROBLEMS, PART 2	<p>> Recitation 9: Directed and Undirected Node Graphs (00:49:58)</p>		
DYNAMIC PROGRAMMING	About this Video		
ADVANCED STATISTICS			

On the website there are lectures and code for different standard algorithms, for example DFS. These algorithms can be amended by varying the implementation of the graph and the

search algorithms. This will allow for the map to be represented as a graph and the enemies to move in the shortest path, or a variation.

Finally, the availability of the high score can be solved by creating a text file in the folder with the game which would store the scores even when the game is turned off. Alternatively, I could use an RDBMS. However, there is no need for such, and implementing it would take more time than necessary on such an easy feature. Thus, using a text file will be a suitable approach, because it will solve the problem easily (using a very easy and short piece of code), not using up a lot of memory or processing power, because the size of such a file would be a few kilobytes. Python has ready functions to open, read and write to external files, and their descriptions are available in Python's documentation. I was also considering different file formats, for example .txt, .xml, etc. I considered different ways of storing the records, for example one line per record, JSON and other. I decided to use a .txt file and one line per record, because it is the simplest method; other methods would be much more difficult to implement, with no need for such sophistication.

A screenshot of a web browser displaying the Python documentation page for file objects. The browser's address bar shows the URL 'https://docs.python.org/2/tutorial/inputoutput.html'. The page has a dark blue sidebar on the left with a 'Table Of Contents' and links for 'Previous topic', 'Next topic', 'This Page', and 'Quick search'. The main content area is titled '7.2.1. Methods of File Objects' and explains the use of file objects. It includes two code blocks: the first demonstrates using the `f.read()` method to read the entire contents of a file, and the second demonstrates using the `f.readline()` method to read the file line by line. The code is shown in a light green background with a dark green border. The text explains that `f.read()` returns a string containing the entire file's contents, and `f.readline()` returns a string containing the next line of the file, including the newline character. It also mentions that `f.readline()` returns an empty string when the end of the file is reached.

Specification of the solution

Based on the interview conducted with the end user (Dmitry Rusanov) and research done I decided on the features my solution will have. The questions and answers from the interview are in the Appendix A at the end of the document.

Features of the solution

1. Walls are displayed on the screen
2. Pacman can move a set number of pixels every iteration of the program
3. The user can change the direction of Pacman using the arrow keys
4. The program can detect the collisions between Pacman and the walls
5. Pacman cannot go into walls

6. When an arrow is pressed which would cause a collision, Pacman takes the next turn possible in the maze
7. Pacman can go through the tunnel
8. There are coins displayed on the screen
9. There is one coin in each 20x20 px square of the area where Pacman can move
10. Pacman starts from the middle of the lower half of the screen
11. Each time Pacman touches a coin, it disappears
12. There are 4 enemies on the map
13. The enemies start from the corners of the map
14. The enemies can move in two different modes: easy and medium
15. In the easy mode, the enemies can move in random directions on the junction
16. In the middle mode, the enemies follow a path towards Pacman
17. The program creates a graph representing the map
18. The enemies use the graph when choosing the new random direction
19. The enemies use the graph to find a path towards Pacman
20. The game screen is changed when the user wins or loses
21. The user wins when he collects all the coins on the screen
22. The user loses when he dies with no available lives
23. Pacman loses a life every time he collides with an enemy
24. The score is displayed below the map
25. The lives available are displayed below the map in form of icons of Pacman
26. A path finding algorithm is used to find a path towards Pacman
27. The game has a start screen and an end screen which depends on whether the user won or lost
28. There is a starting screen
29. The user can choose the level of difficulty on the start screen
30. The top 10 scores are displayed on the start screen
31. The controls are explained on the start screen
32. The top 10 scores are displayed on the final screen
33. The user can go to the start screen from the end screen by pressing Escape
34. The top 10 scores are stored in an external text file – this is necessary for the stakeholders who are very competitive and want their score not to disappear
35. The design of the map is as in classic Pacman
36. The enemies react to the position and movement of Pacman
37. The enemies find a path to the current, or predicted future of the user, depending on the level of difficulty
38. The speed of enemies increases with increasing levels
39. When a pill is collected, it makes enemies vulnerable for a short time
40. Positions of the objects, the coins, available lives and score are reset to initial values if the game is restarted
41. The score increases by 1 every time Pacman collects a coin

I chose Python and Pygame as my programming language, because Pygame provides the user with a variety of useful features for game development. For example, it has a built-in class Sprite used for creating blocks of objects which are displayed on the screen with a function for detecting collisions, which in case of Pacman will be useful for for example collecting coins or finding collisions with enemies. It is also an interpreted language, so it will run on every machine with Python and pygame installed on it. This will enable users without internet (one of the stakeholders) to use it.

Limitations of the solution

1. The game does not have sound effects, because some computers may not use a speaker and this would put these users at a disadvantages. Moreover, I am not familiar with audio capabilities and this would need further research, not necessary for making the game playable;
2. Only top 10 scores are stored, because more would make achieving one of the top scores not a big achievement, making the game less attractive for the stakeholders who are highly competitive;
3. Enemies will be represented by blocks, rather than icons of ghosts, because the main part of the game is the algorithms behind the movement of the enemies, as well as storing the scores and catering for the needs of competitive users;
4. The walls will be rectangular and will not have curved vertices, because detecting collisions with them, creating junctions and implementing an efficient turning algorithm would be much more difficult;
5. Pacman's icon will not change orientation so that it is in the direction of movement, because it will be a square;

Software and hardware requirements

Hardware requirements

Keyboard, because arrows are used to control the movement of Pacman;

Screen with resolution min. 500x600px, because that is the size of the game screen;

RAM of at least 32MB, because the path finding algorithm will use a call stack which takes up a significant amount of memory;

Software requirements

Python version 3.1 or higher, because the code will make use of simplified syntax, which is correctly interpreted only by versions 3.1 or higher;

Pygame version 1.9.1 for Python version installed, because the code will make use of functions present only in version 1.9.1 of that library;

Operation system with a file system, for example Windows 7, 8, etc., because it is necessary for setting the interpreters correctly easily, which is necessary not to cause problems for the user;

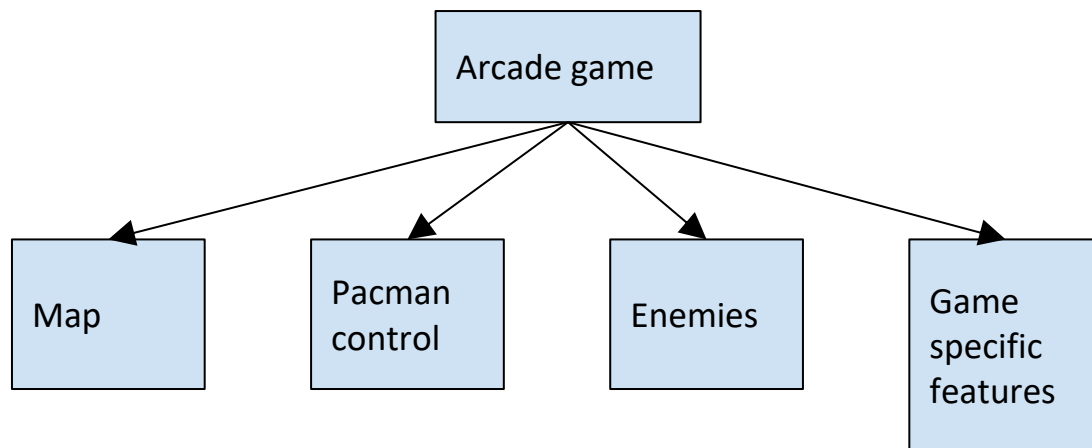
Success criteria

1. There is a map resembling the classic Pacman version – game map is an essential criterion for the success of the project, because without it the enemies and Pacman cannot function, making the game not usable;
2. The enemies move in a way depending on the level of difficulty
 - a. On the easy level, in random directions
 - b. On the medium level, finding a path to Pacman's position at a momentThis criterion caters for the need of a critical end user, Dmitry Rusanov, who is mainly looking for intelligent behaviour of ghosts;
3. Top 10 scores are stored offline on the device, and are retrieved when the game is started – this caters for the needs of the group of highly competitive stakeholders, so it has to be fulfilled;
4. User can use arrows to determine the direction of movement of Pacman and can do so easily in the maze, because it is necessary for the game to be playable;
5. The user can collect coins, because that makes the game have an aim which the user will try to achieve. Without it there is not much point playing the game;
6. Game terminates when it is lost or won – this creates a reward mechanism, making the game fun to play. Moreover, it is crucial that something happens after a game is won or lost, because there is no point for the user to move around on the screen with no coins to be collected, without it the game is not playable;
7. There is a set of screens enabling navigation between them – this key feature makes the game usable, because the user needs a start screen, game screen and end screen and should be able to move between them. This is a criterion for success because the game would not be usable otherwise;

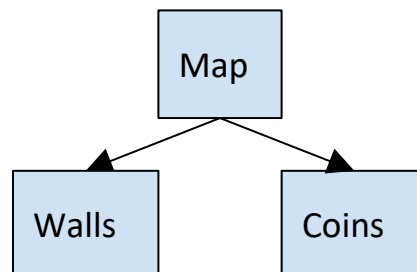
Design

Decomposing the problem

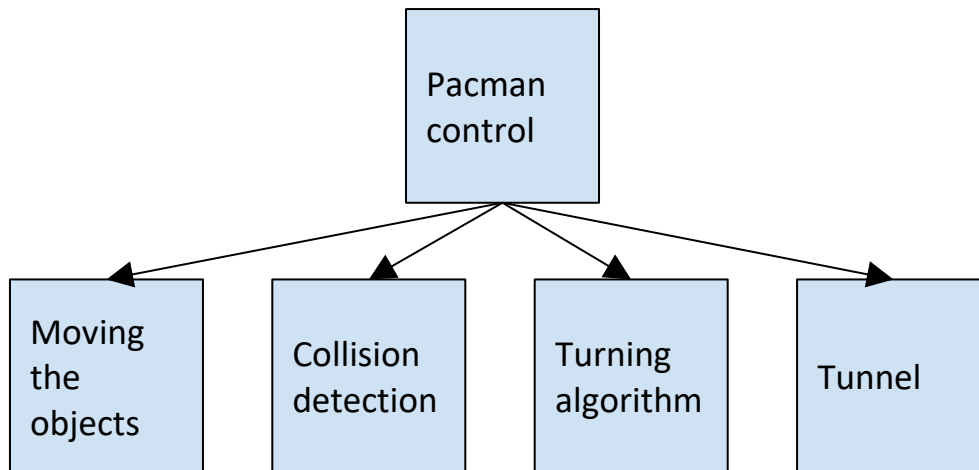
The problem can be broken down into the following sections:



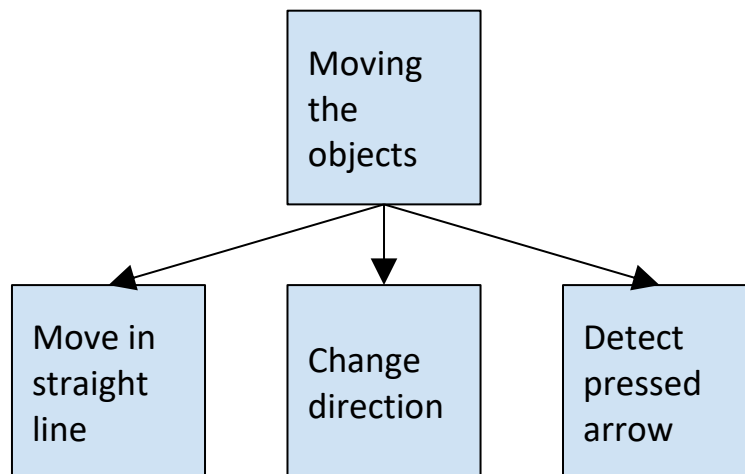
Game-specific features includes features like starting interface, score-counting, etc. Creating the map consists of creating a maze of walls and coins. The creation of the map has to be broken down into these two parts, because they will be different in terms of what they look like, their placement on the graph and their collision with Pacman. I could have divided it into different parts, for example moving objects, map and everything on it, navigation. However, I decided to divide the functionality for Pacman and enemies, because of how differently they move. I also decided to divide map and everything on it into two different parts, because map will be connected to graphs, whereas game-specific features will focus on more miscellaneous functions.



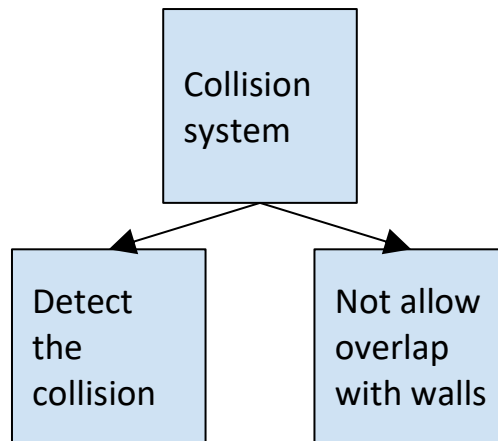
Pacman control consists of several parts: moving the object throughout the game and displaying the movement, detecting collisions with walls, turning so that Pacman takes the next turn possible, and 'teleporting' through the tunnel. I considered including the functionality of the tunnel in the map, because they are directly directed. However, I decided not to use that idea, because teleporting links directly to the position of Pacman, rather than the map, so I decided it would be better if it was included in Pacman control.



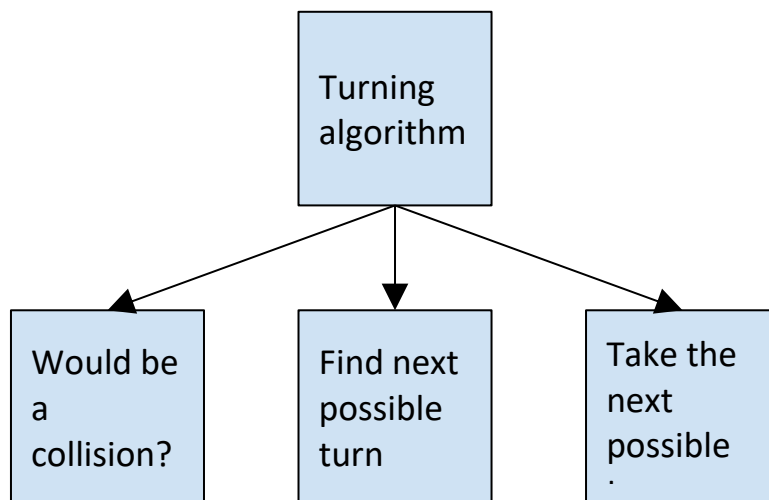
In order to move the object the program needs to move the Pacman in a straight line, and change direction when the arrow on the keyboard is pressed. I was also considering a different approach, which would allow moving in directions other than a straight line, for example a curve. I decided to choose the first approach, of moving up, down, left and right, because the maze is a set of horizontal and vertical lanes.



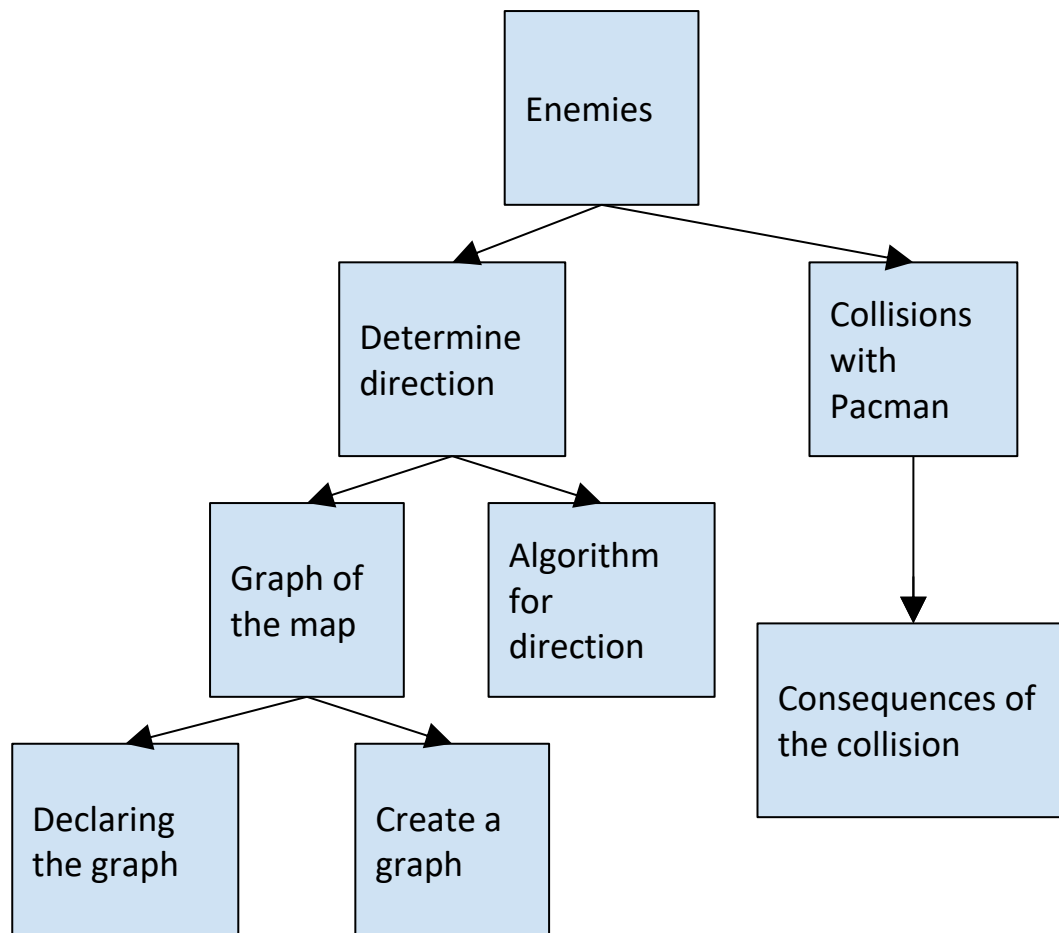
Detecting the collision consists of finding out when the collision happens and not allowing the object to overlap with walls. Initially I wanted to be able to detect when the collision does not happen and allow movement in these cases. However, the approach I thought of late was better, because it meant the instructions of allowing/not allowing the movement would have to be executed more often. By choosing this approach I limited the amount of computational power required.



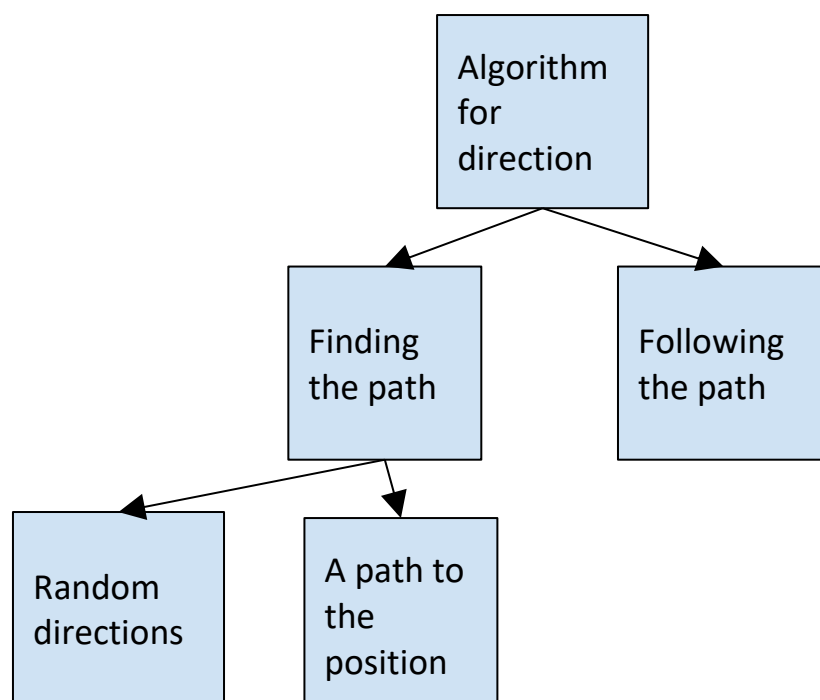
Turning algorithm will use the collision system, will detect whether a movement in a given direction would cause a collision and will find the next place where that turn can be taken, and take that turn.



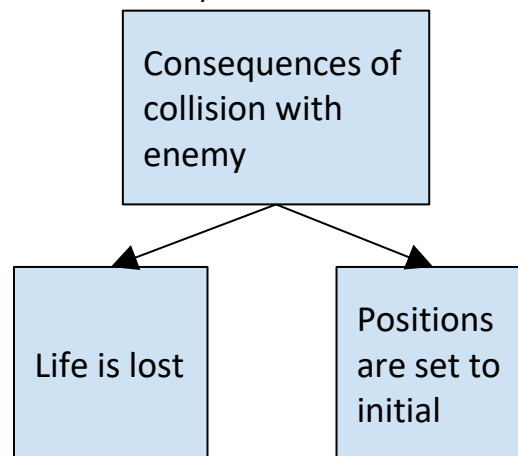
The enemies are the most complex part of the code, mainly because of the algorithm for controlling their movement. The enemies will build upon the Pacman control. Enemies will be moved with the same algorithms for moving the objects and will detect collisions the same way as for Pacman. They will also contain the algorithm for teleporting through the tunnel. The turning algorithm, however, will be different, because the enemies are not controlled by the user, but by the program. Therefore, the main additional part that will have to be done in order to implement the enemies is to determine their direction of motion. Additionally, there will have to be a system for detecting collisions with the user and its consequences (subtracting life, resetting the game, etc.).



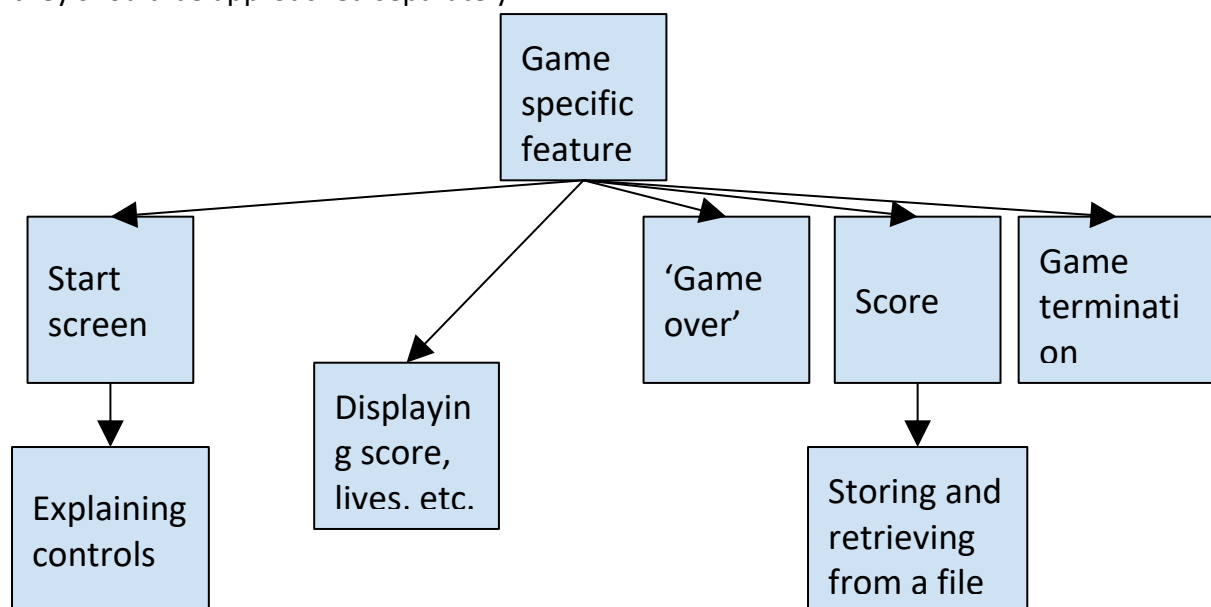
The algorithm for detection will consist of several parts for different levels. There will be one which will chose a random direction, one for finding the shortest path to Pacman's position at a given time, and one for finding the shortest path to the predicted future position. When the path is found, the program will have to control the enemies so that they follow that path.



The consequences of the collision are as follows: a life is lost, positions of Pacman and enemies are reset to initial. All other consequences are a part of game-specific features, for example game is over when there are no more lives left, final score is displayed, etc. I decided to leave other consequences and include them in game-specific features and not in this part, because they depend directly on the number of lives available, rather than as a consequence of a collision with an enemy.



Finally, there is a number of game-specific features, for example the starting interface, where controls and rules are explained, displaying the lives available and the score on the game screen, bonuses for additional points and for making enemies vulnerable (they can be eaten for a short time), a 'game over' screen when the last life is lost, displaying the score, and saving the top 10 scores on an external text file. When all the coins are collected, the game is terminated. I considered having the start and end screens together, but decided it would be best to separate them, because they have a completely different functionality, so they should be approached separately.



Structure of the solution

The main structure of the program will be as follows:

1. Defining all parameters, objects with attributes and methods, and all the functions
2. Loop of the program, executed until the program is terminated
 - a. Checking if the program is terminated
 - b. Game logic
 - c. Resetting screen
 - d. Drawing
 - e. Displaying everything on the screen
 - f. Limiting the times per second when the screen is refreshed
3. Terminating the program

The most complex and the longest part will be the first one, because that is where all functions for different algorithms, and objects with their attributes and methods are defined.

It will have the the following structure of the pre-loop part of the program:

1. Properties of the screen
2. Map
 - a. Declaring the class for walls
 - b. Declaring the class for coins
 - c. Initiating the parameters of the map
 - d. Function for adding the map
 - i. Adding walls
 - ii. Adding coins
 - e. Adding the map
3. Pacman control
 - a. Declaring the class for moving objects
 - i. Constructor
 - ii. Moving in a straight line
 - iii. Changing the direction
 - iv. Moving without overlapping with walls
 - v. Collection the coins
 - vi. Teleporting through the tunnel
 - b. Initiating parameters for Pacman
 - i. Size and colour
 - ii. Position, direction and speed
4. Enemies
 - a. Graphs
 - i. Declaring the class
 - ii. Creating a graph which is equivalent to the map
 - b. Algorithms for finding the path
 - i. Random direction
 - ii. DFS for finding a path
 - c. Following the path

- d. Collisions with Pacman
 - i. Detecting the collision
 - ii. Consequences of the collision
 - 1. Life is lost
 - 2. Positions are reset to initial
- 5. Functions and procedures for game-specific features
 - a. Start screen
 - i. Controls explanation
 - b. Displaying lives, score on the game screen
 - c. 'Game over'
 - i. 'Game over' displayed
 - ii. Score displayed
 - iii. High scores updated
 - d. High scores stored
 - i. Stored and read from a text file
 - ii. A list of high scores displayed
 - e. Game is terminated when all the coins are collected

In terms of the program loop, the functions declared in the pre-loop part will be used. The signal from the keyboard for turning will be detected, Pacman will be moved, there will be an algorithm for taking the next turn possible, the enemies will find the shortest path to Pacman's position (not every iteration) and follow that path. Moreover, the terminating condition will be checked (lost all lives or collected all coins) and collisions of Pacman with enemies function will be used.

The structure will be as follows:

1. Signal from keyboard detected
 - a. Finding what key was pressed
 - b. If an appropriate key, in which direction the Pacman should be moving
 - c. Setting the new intended direction to the one indicated by the user
2. Next possible turn is taken
 - a. Checking if the user wants to turn
 - b. Checking at each place if there would be a collision was the turn to be taken
 - i. If not, moving in a straight line
 - ii. If yes, taking that turn
3. Teleport through tunnel
 - a. Check if Pacman is in the tunnel
 - b. Change Pacman's position
4. Enemies' movement
 - a. Finding the path using an appropriate algorithm
 - b. Following the path
 - c. Checking collision with Pacman
5. Coins are collected, score is updated

- a. Collected coins disappear
- b. Score is increased
- c. Collection of all coins is checked (to terminate the program if needed)

Terminating the program will happen if earlier a collision between Pacman and an enemy was detected or all coins had been collected.

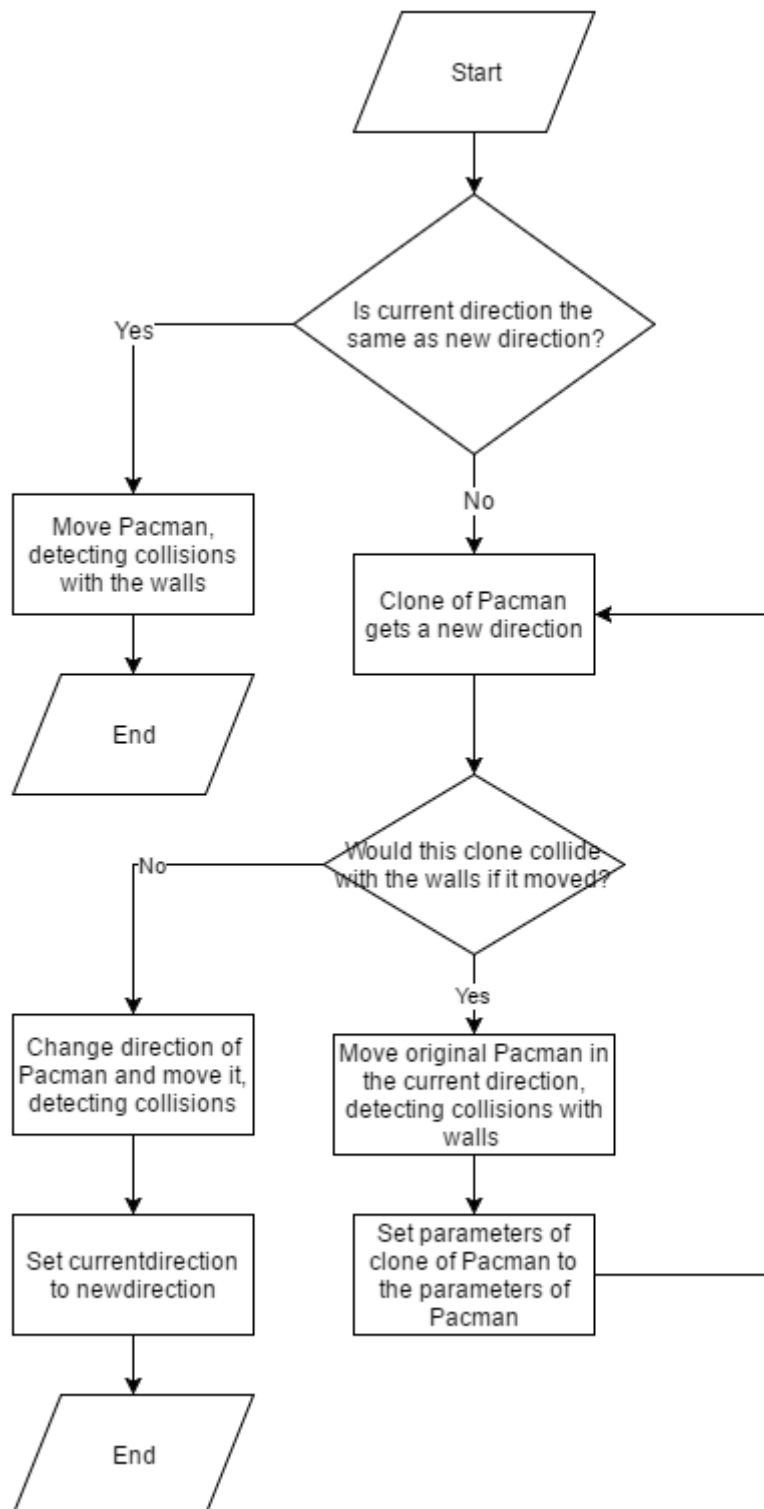
Algorithms

This is the list of main algorithms I will be using:

1. Turning algorithm
2. Creating a graph from the set of strings
3. A* algorithm – I decided to use this algorithm, because finding the shortest path to Pacman would in the best way cater for the needs of my end user – Dmitry Rusanov. Other algorithms would work as well, but this one finds the shortest path and is more efficient than Dijkstra's algorithm, making the solution more optimal computationally.
4. Predicting future position of Pacman
5. Collecting coins and incrementing the score
6. Teleporting through the tunnel
7. Storing the top 10 scores

This is how they will work:

1. Turning algorithm



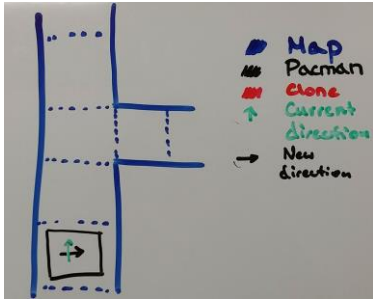
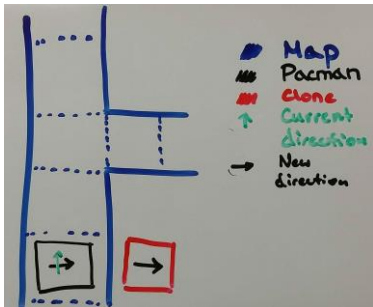
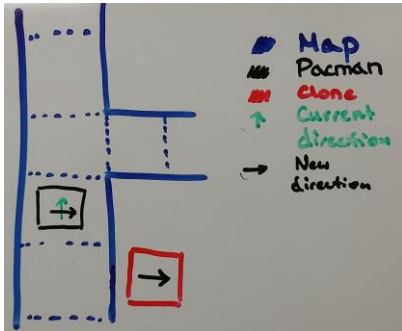
In pseudocode this will work as follows:

```

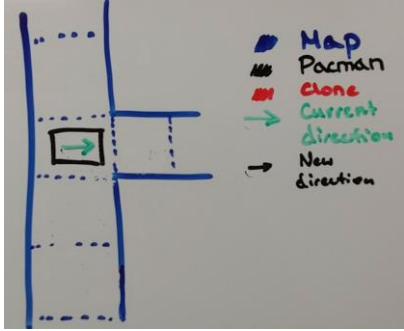
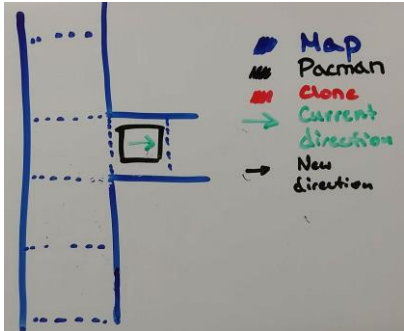
IF current_direction == new_direction THEN:
    move Pacman without it overlapping with the walls
ELSE:
    create a copy of Pacman
    copy of Pacman gets the new_direction
    WHILE copy_of_Pacman would collide if moved in the new direction:
        move Pacman in the old direction, detecting without overlapping with walls
        set direction and position of copy of Pacman to the ones of Pacman
    Pacman gets the new_direction
    current_dircciton = new_direction
END IF

```

I tested the working of this algorithm using the following hypothetical situation:

Current direction	New direction	Situation on the screen	Clone collides with walls?
Upwards	Right		
Upwards	Right		Yes
Upwards	Right		

Upwards	Right		
Upwards	Right		Yes
Upwards	Right		
Upwards	Right		
Upwards	Right		No

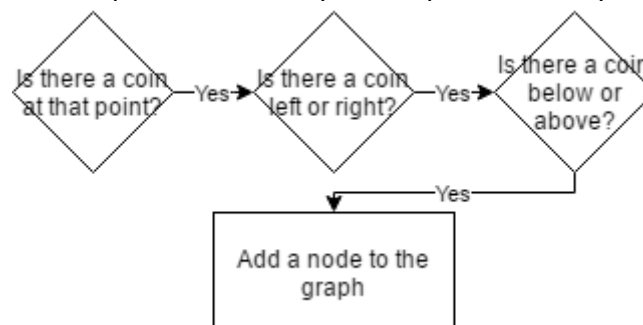
Right	Right		
Right	Right		

As shown with the trace table above, the Pacman takes the next turn possible, so the algorithm works as intended.

2. Creating a graph from the set of strings

a. Adding the nodes to the graph

There will need to be a node in the graph if in the map there is a junction. At the beginning, at regular places in the map there are coins, so they can be used to find all the junctions. A junction is defined in the following way: if there is a space without a wall, a free path horizontally, and a path vertically, there is a junction.



In terms of pseudocode:


```

WHILE row != final row:
    column = 0
    WHILE column != final column:
        IF map[row][column] == coin THEN:
            IF map[row+1][column] == coin or map[row-1][column] == coin THEN:
                IF map[row][column+1] == coin or map[row][column-1] == coin THEN:
                    name_of_new_node = str(row)+' '+str(column)
                    add node "name_of_new_node" to the graph
                END IF
            END IF
        END IF
        increment column
    END WHILE
    increment row
END WHILE

```

This algorithm checks if there is a coin at a given point, if there is a coin if moved vertically, and if there is a coin if moved horizontally. If there are, creates the name of the new node dependent on the current column and row, and subsequently adds that node to the graph.

The process is repeated for every place in every column and in every row.

This algorithm was tested using the following trace table and exemplar map. Walls are represented by crosses, coins by dots, nodes by dots in a green circle. A crossed coin means that it is not a node.

row	row != final row	Situation on the map	Nodes added
0	True		None
1	True		"Node1 1" "Node1 3" "Node1 5"

2	True		None
3	True		"Node3 3" "Node3 5"
4	True		None
5	True		"Node5 1" "Node5 3" "Node5 5"


```

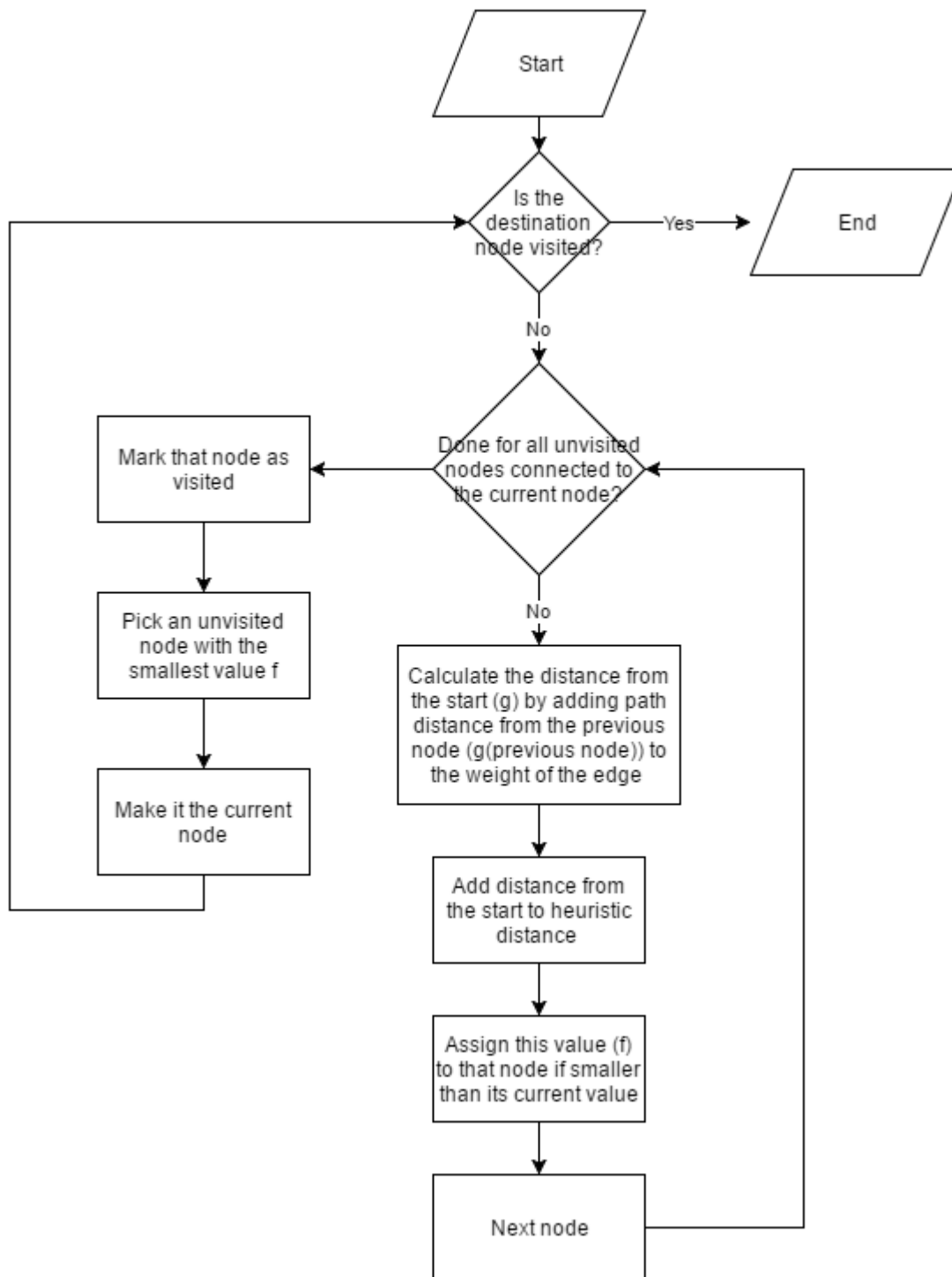
FOR each_node in the list DO:
    weight_edge = 0
    get row_of_node and column_of_node
    old_column = column_of_node
    old_row = row_of_node
    WHILE map[row_of_node][column_of_node+1] == coin:
        weight_edge += 1
        column_of_node += 1
    END WHILE
    newEdge = edge(each_node, node(row_of_node, column_of_node), weight_edge)
    weight_edge = 0
    WHILE map[row_of_node+1][old_column] == coin:
        weight_edge += 1
        row_of_node += 1
    END WHILE
    newEdge = edge(each_node, node(row_of_node, old_column), weight_edge)
NEXT each node

```

This algorithm relies on the fact that if there is an edge, it has two ends - left and right if horizontal, and top and bottom if vertical. In both cases, the left end or the top end has to be a node. This means that if there is an edge to be established, there is a node in the graph from which it starts, meaning we will find all the edges that are to be established. More importantly, we will not duplicate them, because each edge has only one left end or top end. After having found that node, we find the other end by going to the end of the end by following free spaces, both in case of a vertical and horizontal path on the map.

3. A* algorithm

A* algorithm will be used as a heuristic because it will be used often, so it will decrease the time of execution. The graph represents a physical map with distances, so the heuristic distance will be the sum of absolute coordinate differences. The trade-off will be that the path found may not be the shortest one, but it will be a good approximation, and since it will be used multiple times during the run of the program, so there is a high chance that the result found by the program will be the correct one. This is the flowchart for how an A* algorithm works:



In terms of pseudocode:

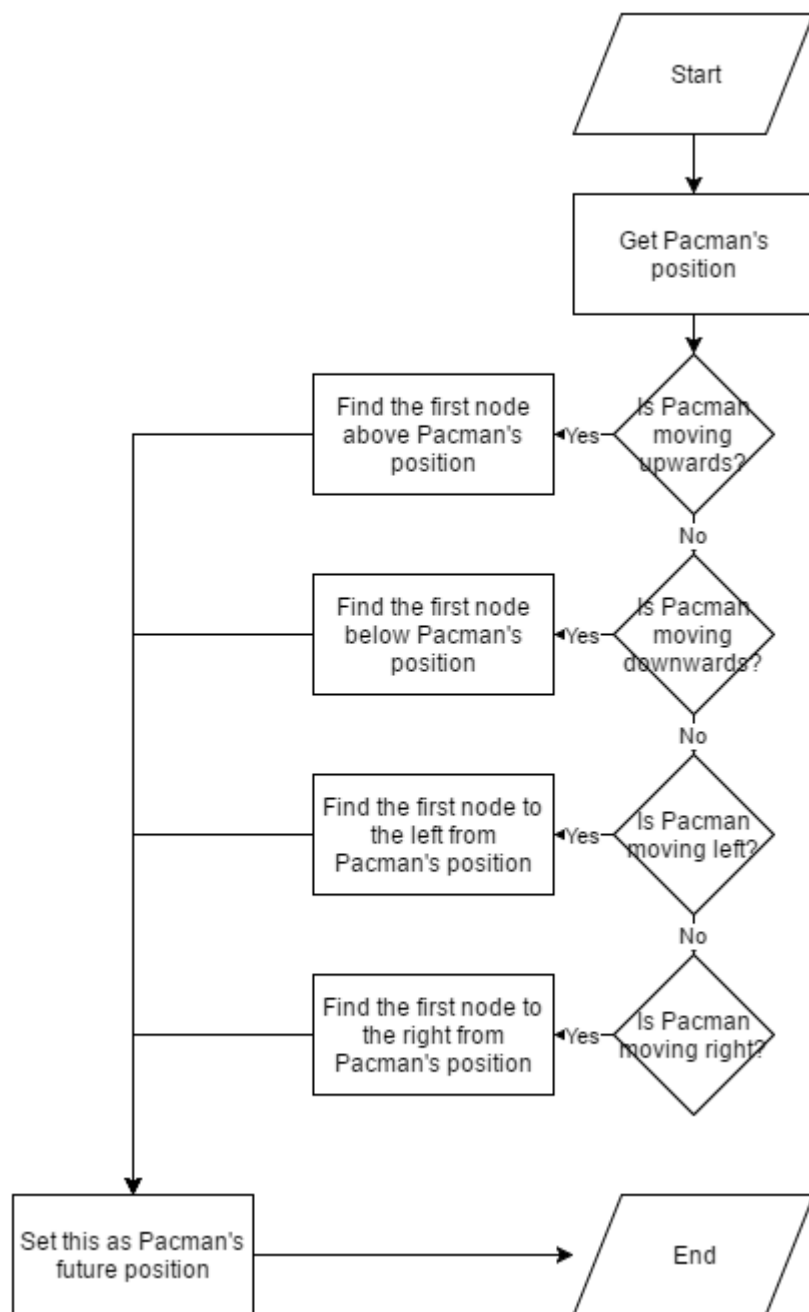
```

Begin at the start node and make this the current node.
WHILE the destination node is unvisited
  FOR each open node directly connected to the current node
    Add to the list of open nodes.
    Add the distance from the start (g) to the heuristic
    estimate of distance left (h).
    Assign this value (f) to the node if it is smaller
    than its current value.
  NEXT connected node
  Make the unvisited node with the lowest value the current node.
ENDWHILE

```

This algorithm uses heuristic to improve its efficiency by estimating the length of the path to a node by finding the absolute distance by subtracting coordinates of current position and destination, and subsequently adding them. Starting at the start node, assigns a value to all nearby nodes by adding the length of the path from the start to the estimate of the distance left. The algorithm then marks the current node as visited and goes to an unvisited node with the smallest value assigned. That node becomes the new current node, and the process repeats until the destination node is visited. For each node, previous node is stored, so when the algorithm gets to the destination, the estimate of the shortest path can be found by tracing previous nodes.

4. Predicting future position of Pacman



In terms of pseudocode:

Get Pacman's current position.

SWITCH (current direction):

CASE upwards:

Go upwards from current position until at a node

CASE downwards:

Go downwards from current position until at a node

CASE left:

Go left from current position until at a node

CASE right:

Go right from current position until at a node

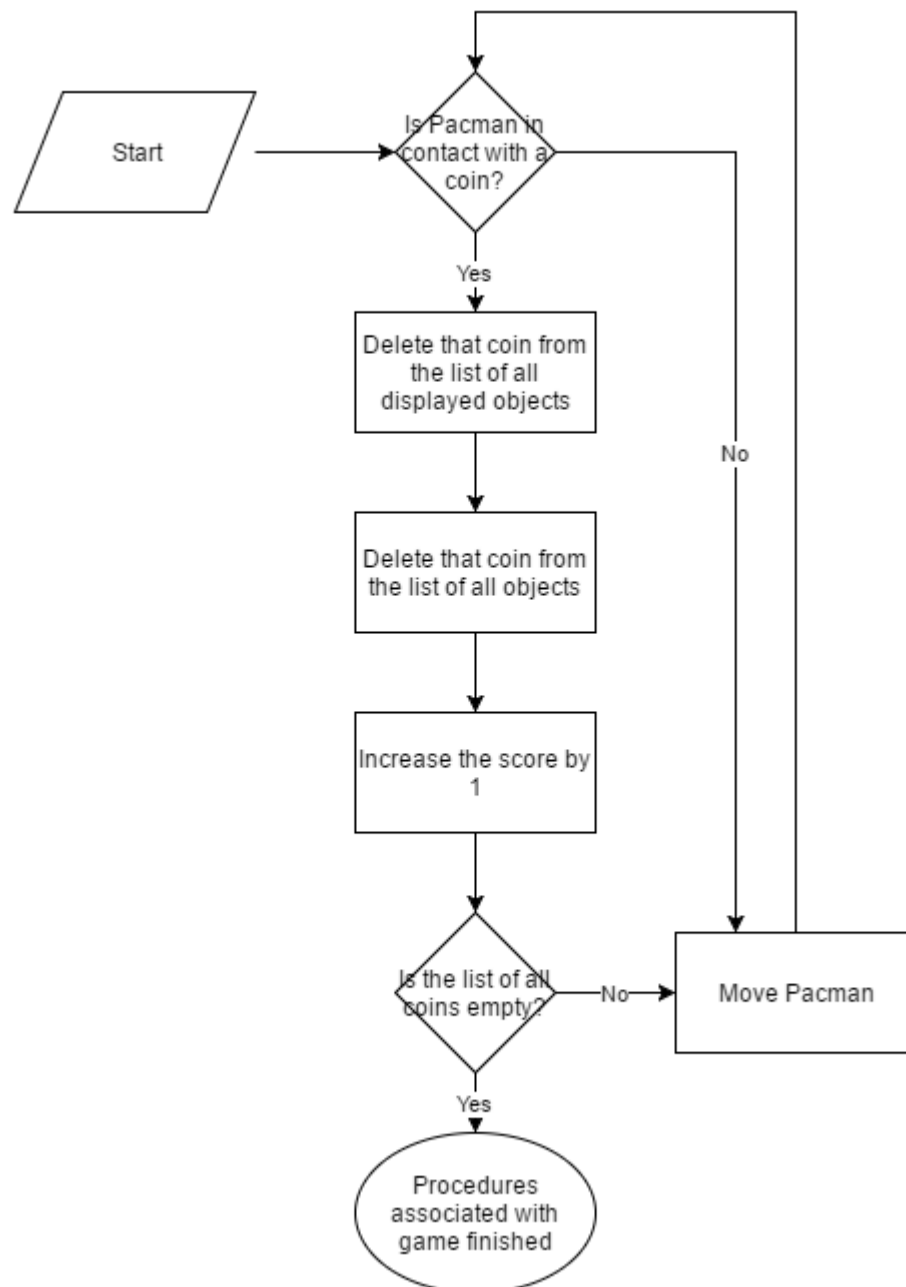
END SWITCH

Set Pacman's position to that node

This algorithm is relatively simple, as the prediction of Pacman's future position is based on an assumption that Pacman is more likely to move in a straight line than to turn. This is because for a user it 'feels better' to collect coins while they can be moving in straight line.

5. Collecting coins and incrementing the score

When Pacman is moving, it should collect coins from the screen, the score should be incremented, and game should be over after having collected all the coins.



In terms of pseudocode:

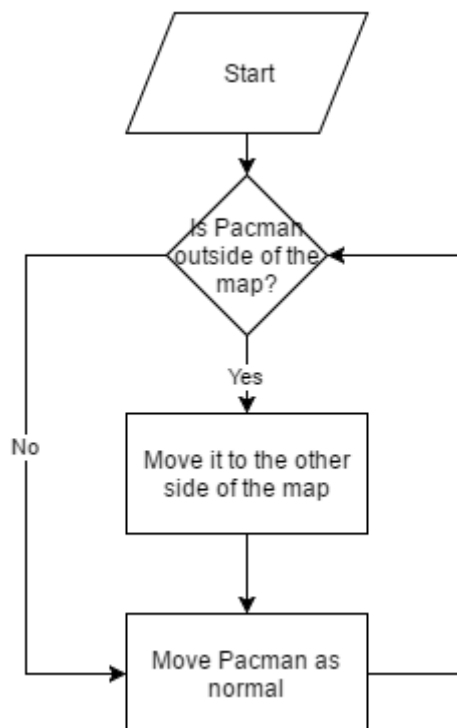

```

done = False
WHILE done == False:
    IF there is a collision between a coin and Pacman THEN:
        delete that coin from the list of coins
        delete that coin from the list of displayed objects
        delete that coin from the list of all objects
        score = score + 1
    END IF
    IF list of coins is empty THEN:
        done = True
    END IF
    Move Pacman
END WHILE

```

This algorithm will move Pacman until it collects all coins. When it collects all the coins, the program will exit the loop. Until that happens, whenever there is a collision between Pacman and a coin on the map, that coin is deleted from all the lists it is in, and the score is increased by one, or a different number, if needed. The program then checks if the program should be ended and does so, if necessary.

6. Teleporting through the tunnel



This algorithm is inside the main program loop, so it does not need to have an exit point, because it will not be executed anymore if the main loop is broken (all coins are collected or all lives lost).

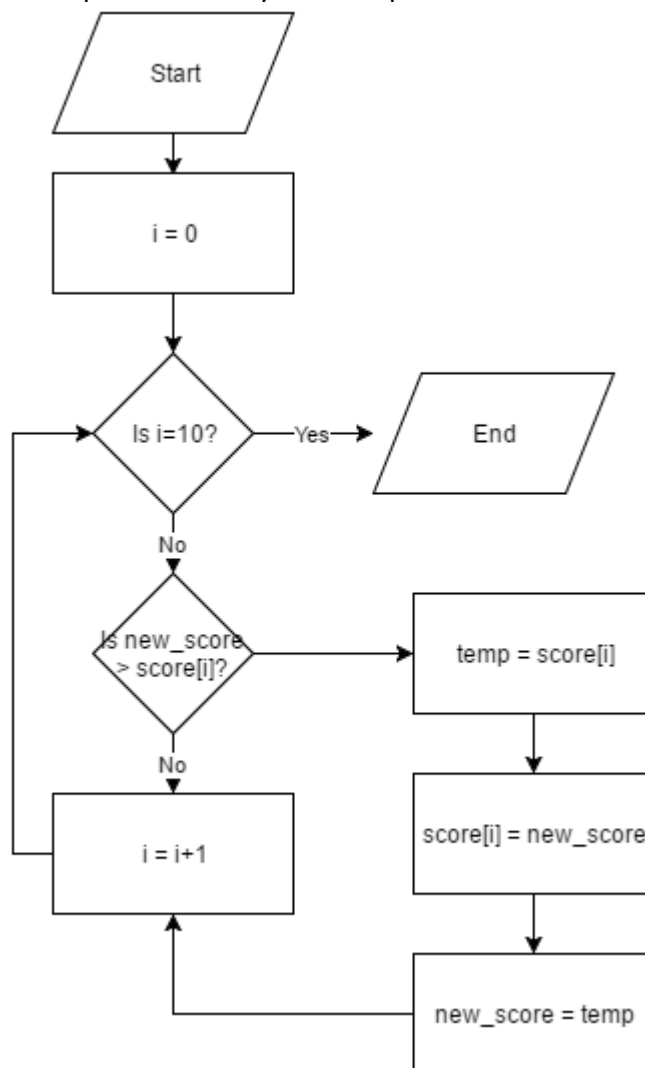
```

WHILE True:
    IF Pacman x coordinate > width of the screen THEN:
        Pacman x coordinate = 0
    END IF
    IF Pacman x coordinate < 0 THEN:
        Pacman x coordinate = width of the screen
    END IF
    Move Pacman
END WHILE

```

7. Storing the top 10 scores

Scoring the top 10 scores is a vital part of this game, because it targets the group of stakeholders, which are highly competitive. They would like their high scores to be recorded and kept so that they can compare their results with other gamers.



In terms of pseudocode:

```

FOR i = 0 to 9:
    IF new_score > score[i] THEN:
        temp = score[i]
        score[i] = new_score
        new_score = temp
    END IF
NEXT i

```

The program in pseudocode is very simple because of the use of a simple fact: scores in the list of score are ordered decreasing. That means that high score number i is bigger than high score number $i+1$. Thanks to that, we can simply go through the list and swap score at place i and new score if new score is bigger than the old score. The new score is now the score that used to be at place i . The next step will be swapping the new new score and score at place $i+1$, which will happen, because of the fact that scores are ordered in the list of scores.

Testing the algorithm using trace table:

i	new_score	score[i]	new_score>score[i]	Score list
0	45	200	False	[200, 150, 100, 70, 60, 50, 40, 30, 20, 10]
1	45	150	False	[200, 150, 100, 70, 60, 50, 40, 30, 20, 10]
2	45	100	False	[200, 150, 100, 70, 60, 50, 40, 30, 20, 10]
3	45	70	False	[200, 150, 100, 70, 60, 50, 40, 30, 20, 10]
4	45	60	False	[200, 150, 100, 70, 60, 50, 40, 30, 20, 10]
5	45	50	False	[200, 150, 100, 70, 60, 50, 40, 30, 20, 10]
6	45	40	True	[200, 150, 100, 70, 60, 50, 45, 30, 20, 10]
7	40	30	True	[200, 150, 100, 70, 60, 50, 45, 40, 20, 10]
8	30	20	True	[200, 150, 100, 70, 60, 50, 45, 40, 30, 10]
9	20	10	True	[200, 150, 100, 70, 60, 50,

				45, 40, 30, 20]
--	--	--	--	-----------------

As the trace table above illustrates, the algorithm inserts the new score in the right place in the list of the top 10 scores.

The turning algorithm enables the Pacman to turn in a way which disables collisions between the Pacman and walls, which together with the algorithm for collecting coins enables the user to collect the coins, and eventually win the game if all coins are collected. During the run of the program enemies chase the Pacman making use of the graph which represents the map. This map is created using the algorithm for creating the graph representing the map. The enemies on the middle level find the shortest path to the position of Pacman at a given time, which is done using the A* algorithm. At a higher level, they find the shortest path to Pacman's future position, which is found using one of the algorithms above. When the game is ended, score is recorded and put into the right place in the table of top 10 scores, if appropriate. These algorithms fit together to form a solution of the problem.

Usability features

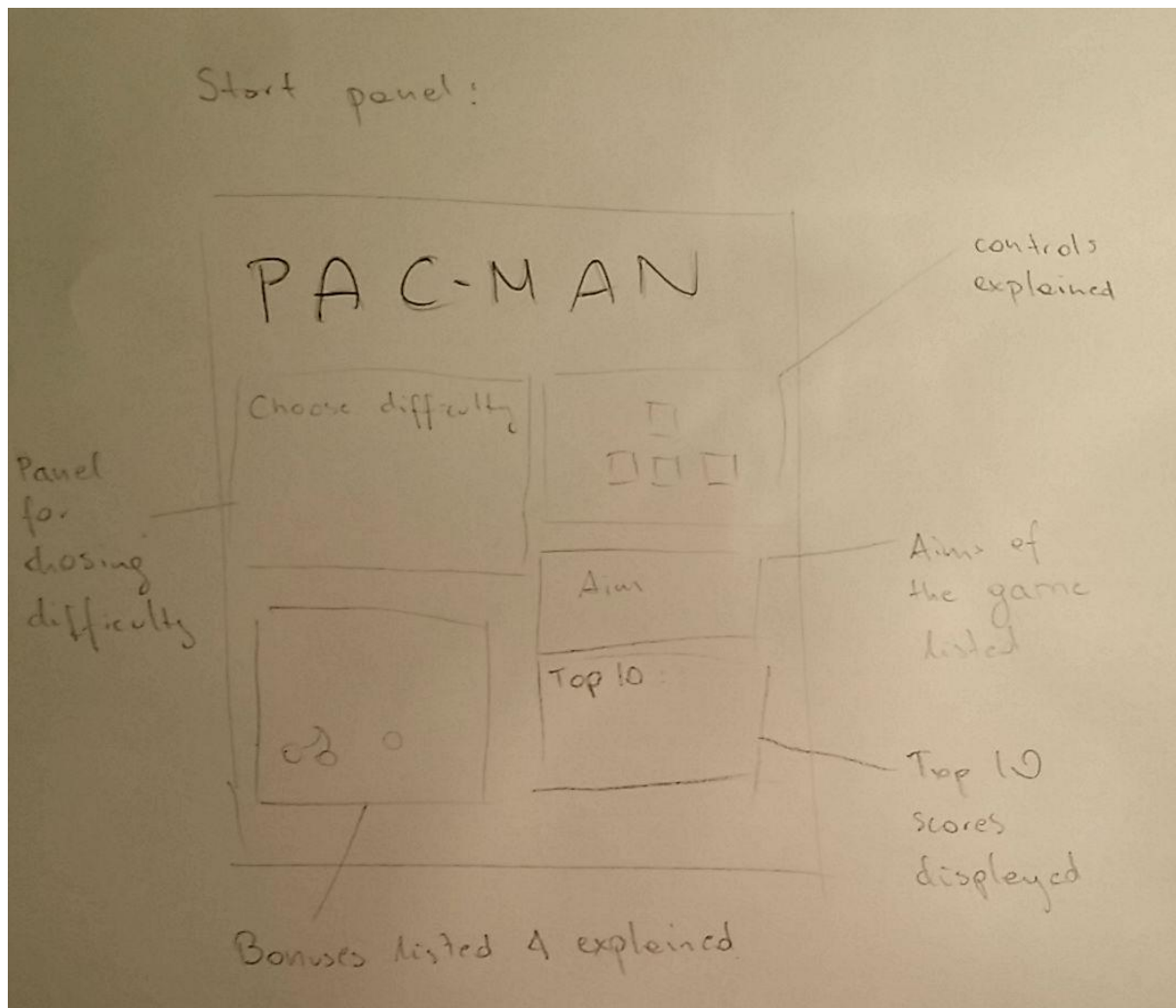
The game must have a number of usability features to make the game easily approachable and as easy to use as possible.

1. Controls displayed on the start screen, because the user needs to know how to control Pacman.
2. Pacman can be controlled with arrows. I decided to control it with arrows rather than WSAD, because it may be the case that in countries with a different alphabet other keys are used instead. Furthermore, arrows are instinctively used all around the world to indicate direction of movement, which justifies the choice of arrows rather than WSAD. I decided not to implement anything else, because it would just seem bizarre and make the user uncomfortable.
3. The possibility of switching to start screen on the after-game screen, because the user may not look at the on-screen help on the start screen and then not know how to play. In this case, the player will lose and should have the possibility of looking at the on-screen help on the start screen. If I do not implement these screens, the game would start the game straight away without any heads-up, resulting in confusion. Moreover, the user could play the game only once and would have to start the program every single time he wanted to play.

In terms of user interface, there are going to be 3 different user interfaces, depending on at what point in the game the user is. If the user has just started the game and is about to play for the first time, the start panel will be displayed. If the user is playing, the game interface

is displayed. If the user lost or won, the screen for a finished game will be displayed with an option to go to the start panel.

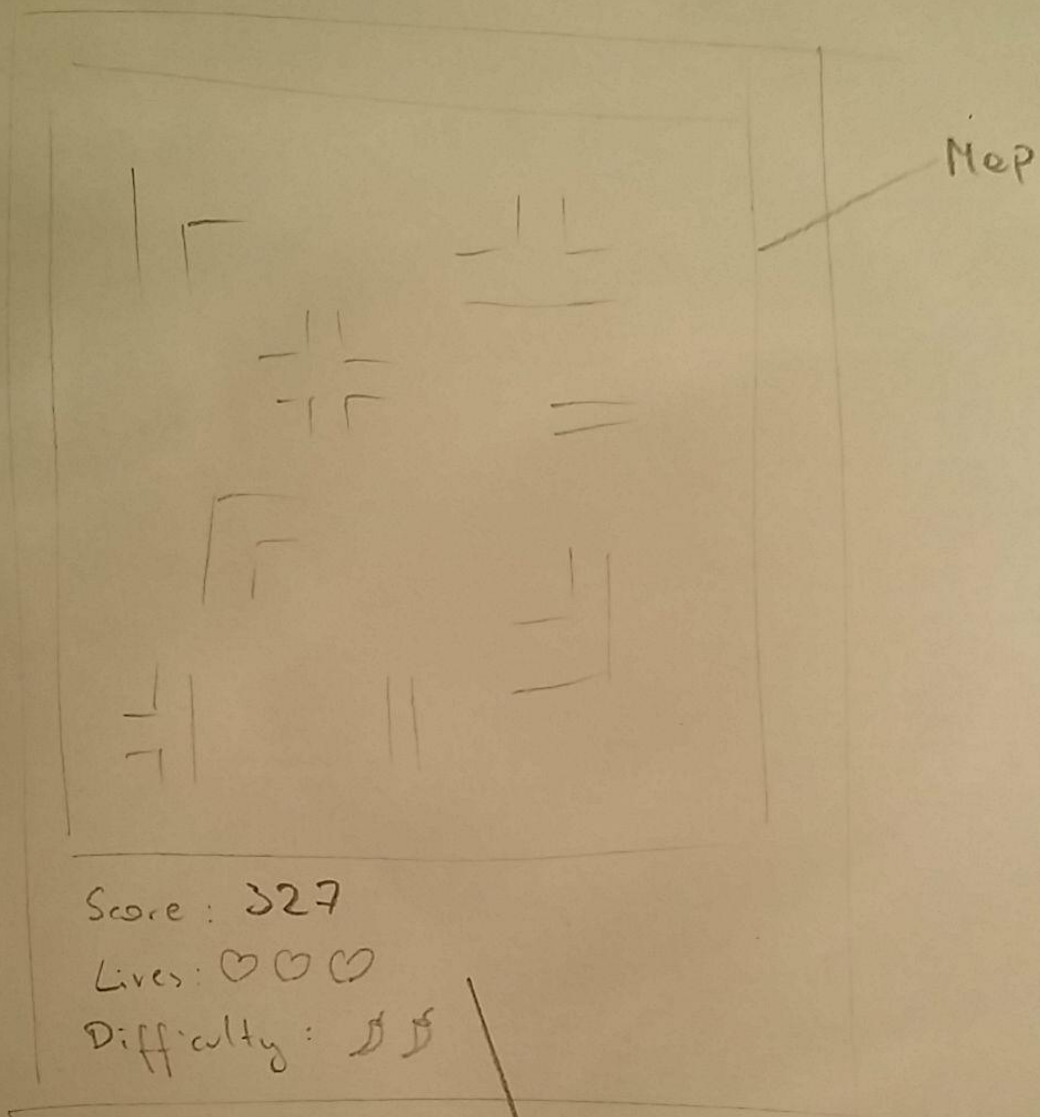
The start panel will look as follows:



On the start panel, there will be "Pac-man" written with a big font. Below, there will be an option for the user to choose a difficulty, because the game will operate on different levels of difficulty to cater for the needs of different users. On the right from that option there will be a piece of screen devoted to explaining the controls, because some people may not have played this game before. Moreover, it clarifies whether arrows or "WSAD" are used. In the bottom left corner there will be a part of the screen explaining different bonuses which the user can collect. This is essential, because in different versions of Pacman there are different bonuses and they are represented with different symbols. On the right side, the aims of the game are listed, so that the users who had not played Pacman before know what they should do. Moreover, in the right bottom corner there will be a list of top 10 scores on that machine. After the user chooses difficulty, the program will wait for a second to change to the game interface, where the game will start after a 3 second countdown.

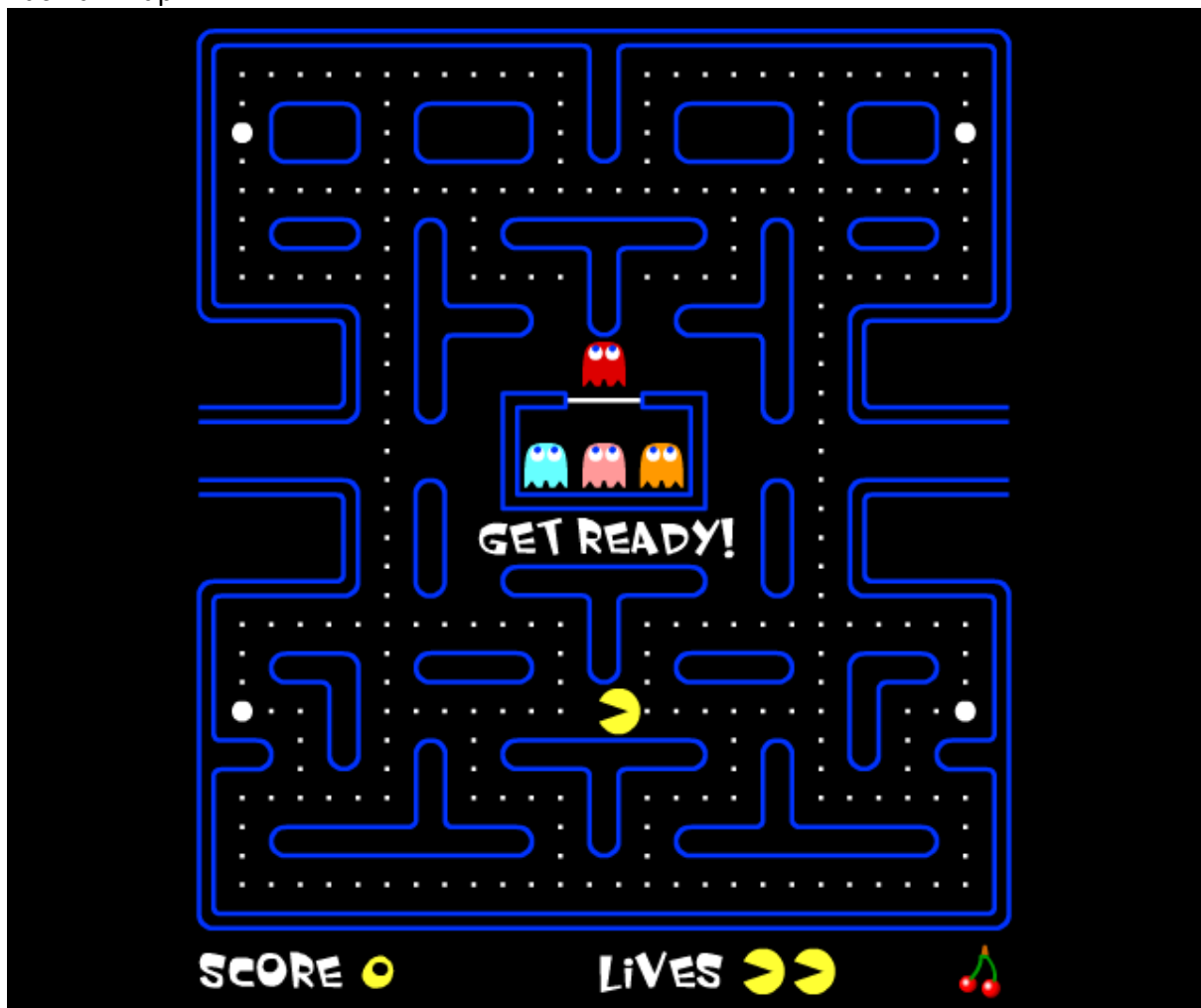
The game interface will look as follows:

Game interface :



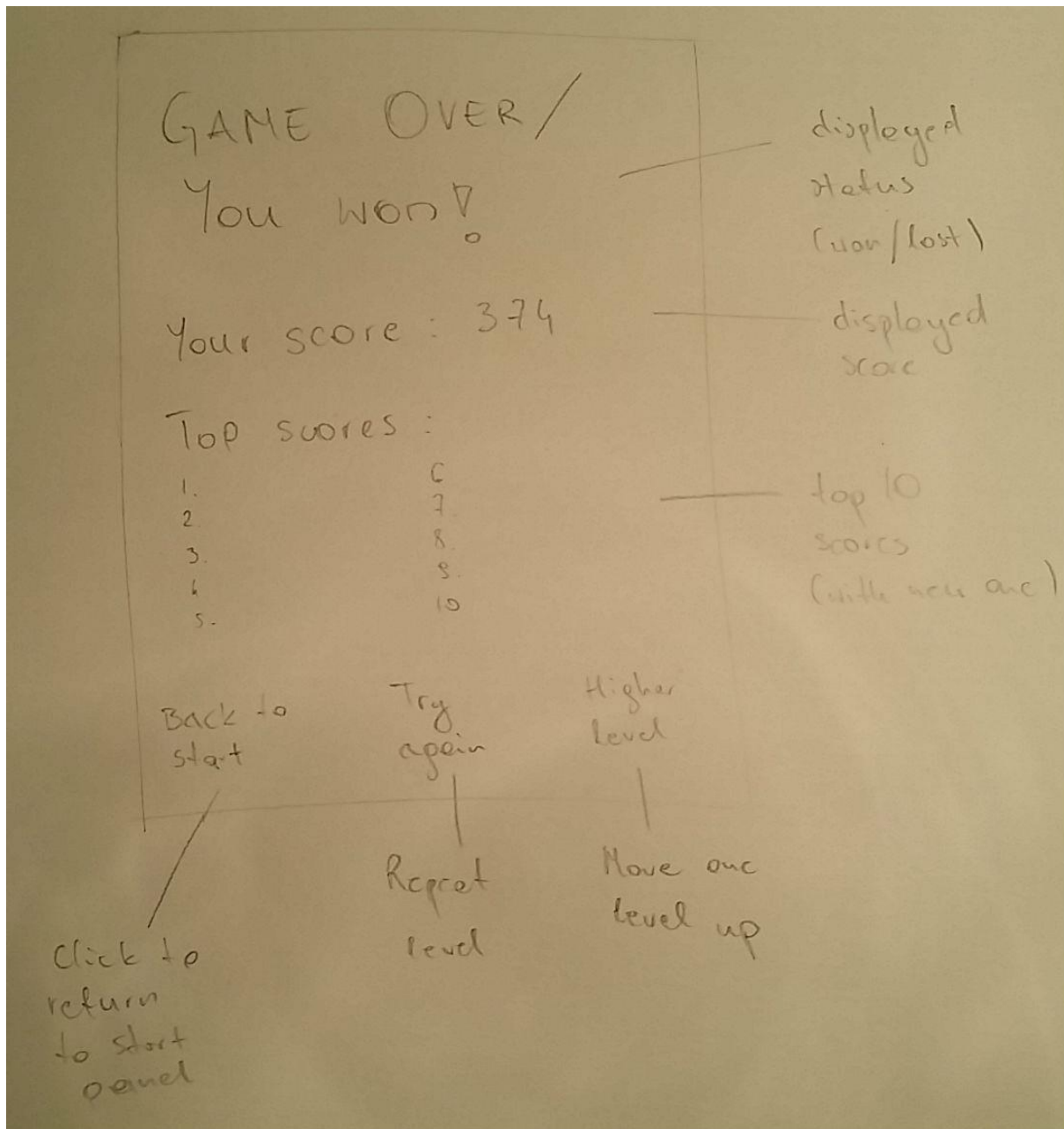
Panel displaying
the number of
available lives,
difficulty and
score

The main part, starting from the top, is the map, along with Pacman and enemies. This is where the game itself will take place. The map is included there, because it needs to be big and is the main part of the game. It contains walls, coins and bonuses. The map itself will have the same maze, that is junctions and paths connecting them, as the original version, because that is what my stakeholder wants. According to him, the map has to be a classic Pacman map.



Below the map itself, there will be a line indicating the score, number of available lives, and the difficulty of the current level indicated by the number of chilli peppers displayed. All of them have to be displayed, because the user needs to know what is their current score (for the competitive users), number of available lives, in order to manage the risks taken when playing (coming close to enemies or playing safe), and the difficulty level so that they can change it to lower if they feel it is too difficult.

When the game is finished (won or lost), the screen will get this design.



On top of the page there will be displayed 'Game over' or 'You won', depending on whether the player had won or lost. It will be displayed at the top of the page, because the player needs to be informed about his status. Moreover, it will intensify the user's emotions. That is also the reason for displaying the score below. Next, the top 10 scores will be displayed in order with the new entry highlighted in order to increase the reward for the most competitive players who beat their previous result. On the bottom of the page there will be three buttons which will allow the user to either return to the start panel, in case they want to review controls or had not understood the aim of the game. They are also able to repeat the same level, because some users may want to try and perform better than they did in their last try. Some players will be satisfied with their score, and will be looking for a bigger challenge, so there will be also an option of moving up by one level in difficulty (not available if already at the highest difficulty level).

Key variables and structures

There are several key variables and structures which will be used in the program. They are related to the algorithms which will be used in the game.

Variables and constants

- Difficulty, because depending on the chosen difficulty a different algorithm for controlling the enemies has to be used – it can only take values 1 or 2 and is validated every time the program goes through the loop
- Parameters of the screen (width and height), because it has to be adjusted to what is displayed on the screen
- Parameters of walls (height, width, colour), because there may be a need to change it in the development process
- Parameters of coins (height, width, colour), because the user may change his mind on how big and what colour they should be
- Pacman's properties (position, height, width, colour, speed), because its position changes, and speed has to be adjusted to user's preference (it will be found out during beta testing) – Pacman's position is validated every time the program goes the loop to check if it is on the screen, it can only take values within the width and height of the screen
- Enemies' properties (position, height, width, colour, speed), because their position varies, and their speed depends on the bonus status – position is validated every time the program goes through the loop, can only be within the screen
- 'Reverse roles' bonus collected - a boolean indicating whether a bonus had been recently collected, because the enemies need to run away from Pacman if that bonus had been collected

Classes

- Moving object, because there are several elements in the game, which need to move, for example Pacman or enemies. They will all use the same class moving object which changes position every time the program goes through the main program loop – this needs to be a class, because it needs to make use of collision detection used for turning algorithm and moving on the map
- Walls, because they have to be displayed on the screen and have certain properties – this needs to be a class because it requires collision detection not to allow Pacman go into a wall
- Coins, because they have to be displayed on the screen and disappear when collected – they have to be a class because collecting coins will require collision detection

Structures

- Graph of nodes and edges, because it is needed for A* algorithm to find the path for the enemies, and then follow that path

- List of top 10 scores, because there are 10 top scores that need to be stored, and it is the easiest to do using a list in Python.

The only input taken from the user is the signals from the keyboard during the run of the program indicating where it should be moving. These signals need validation, because only the arrows should have an effect of changing direction of Pacman.

Test data for development

During the development a variety of features has to be tested. The data that has to be used is:

Scenario	Test	Requirements
1 Does Pacman move correctly	1.1 does it go into walls, 1.2 does it move in the direction of the last arrow pressed, 1.3 does it take the next turn possible, 1.4 does it go through the tunnel)?	2, 3, 4, 5, 6, 7
2 Does the game initialise correctly	2.1 is Pacman in the right place, 2.2 are enemies in the right places, 2.3 is the map displayed correctly, 2.4 are all coins displayed, 2.5 is a correct number of available lives displayed?	1, 8, 9, 10, 12, 13, 35, 40
3 Do enemies behave correctly on the easy level	3.1 do they move in random directions, 3.2 do they change directions at the junctions, 3.3 do they not react to Pacman's behaviour?	14, 15, 17, 18
4 Do enemies behave correctly on the medium level	4.1 do they find a path towards Pacman's last visited node, 4.2 do they follow that path, 4.3 do they move faster than on the easy level?	14, 16, 17, 19, 26, 36, 37, 38
5 Do the enemies and Pacman interact correctly	5.1 does the Pacman lose a life when it collides with an enemy, 5.2 does the game terminate when all lives are lost?	22, 23
6 Do the screens work and display correctly	6.1 does the screen change when the user wins or loses, 6.2 is the score displayed on the game screen,	20, 21, 24, 25, 27, 28, 29, 30, 31,

	6.3 are the lives displayed below the screen, 6.4 is there a start screen and an end screen, 6.5 can the user choose the difficulty on the start screen, 6.6 are top 10 scores displayed on start and end screen, 6.7 can the user choose the difficulty, 6.8 can the user navigate between them?	32, 33, 35, 40
7 Does the game correctly indicate winning/losing	7.1 is the game won when all coins are collected, 7.2 is the game lost when all lives are lost?	21, 22, 27
8 Does collecting the coins work	8.1 Can Pacman collect the coins 8.2 does the score change accordingly?	11, 24, 41
9 Storing the scores	9.1 Are the top 10 scores stored properly?	34
10 Do the pills work correctly	10.1 Are they on the map, 10.2 do they make enemies vulnerable?	39

Test data for beta testing

- Is the score recorded well? Is it sufficiently good when the performance is good? Is it bad when Pacman takes a long time to collect all coins? Is the score lower when the lives are lost? Etc?
- Is the start panel intuitive enough? Are the controls explained well enough? Is the aim of the game explained well enough?
- Are the levels of proper difficulty? Is the easiest level relatively easy? Is the highest level difficult enough?
- Is the speed of Pacman and enemies appropriate? Does Pacman move fast enough compared to the enemies? Do the enemies move too fast? Do they move too slow?
- Is the after-game panel intuitive enough? Can the user easily switch to other screens?

Development – the first iteration

First iteration - introduction

In the first iteration I am aiming to produce a prototype of the game with basic functionality. First of all, I am going to create a map for the game. Secondly, I am going to create a shape representing the Pacman and moving in the maze. Next, I will implement the coins and enable Pacman to collect it. Finally I am going to create a graph representing the maze and create one enemy which is going to move in the maze in random directions.

First iteration - development

The main parts of the program have been copied from the Open Source file from website *programarcadegames.com* and are displayed below.

```

01. #import library called pygame
02. import pygame
03. #initialize the game engine
04. pygame.init()
05.
06. # Define some colors
07. BLACK = (0, 0, 0)
08. WHITE = (255, 255, 255)
09. GREEN = (0, 255, 0)
10. RED = (255, 0, 0)
11.
12. pygame.init()
13.
14. # Set the width and height of the screen [width, height]
15. width = 700
16. height = 500
17. size = (width, height)
18. screen = pygame.display.set_mode(size)
19.
20. pygame.display.set_caption("My Game")
21.
22. # Loop until the user clicks the close button.
23. done = False
24.
25. # Used to manage how fast the screen updates
26. clock = pygame.time.Clock()
27.
28. #initialise variables
29.
30. # ----- Main Program Loop -----
31. while not done:
32.     # --- Main event loop
33.     for event in pygame.event.get():
34.         if event.type == pygame.QUIT:
35.             done = True
36.
37.     # --- Game logic should go here
38.
39.     # screen cleared to white
40.     screen.fill(WHITE)
41.
42.     # --- Drawing
43.
44.     # --- Go ahead and update the screen with what we've drawn.
45.     pygame.display.flip()
46.
47.     # --- Limit to 60 frames per second
48.     clock.tick(60)
49.
50. # Close the window and quit.
51. pygame.quit()

```

Declaring walls

Walls need the following attributes: height, width, x-coordinate of left top corner, y-coordinate of left top corner, and color, because each of these values may vary with different walls. The only method needed will be drawing the wall. I decided to build the map from rectangular walls, so I drew them as rectangles, using the function from library pygame. I decided to start with that approach, because it seemed to be the simplest one.

```

30. # Define wall
31. class Wall(object):
32.     def __init__(self, corner_x, corner_y, width, height, color):
33.         self.corner_x=corner_x
34.         self.corner_y=corner_y
35.         self.width=width
36.         self.height=height
37.         self.color=color
38.     def draw(self):
39.         pygame.draw.rect(screen, self.color, [self.corner_x, self.corner_y, self.width, self.height])
40.

```

I tested the method at once after having declared it.

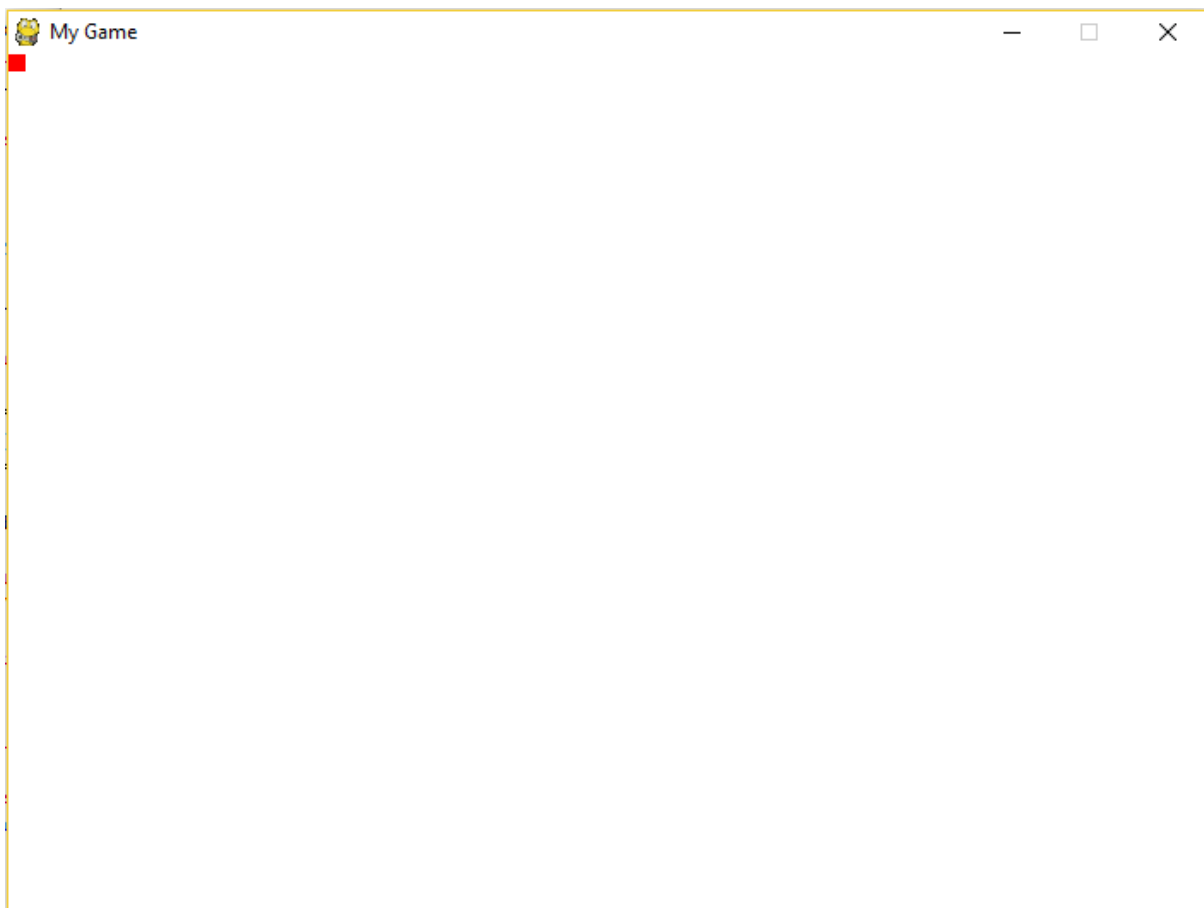
I declared a wall and used method draw() to test its working.

```

53. # --- Drawing
54.
55. wallblockingmovement=Wall(0, 0, 10, 10, RED)
56. wallblockingmovement.draw()

```

This was the result of this test:



A red square, 10 by 10 pixels appeared in the left upper corner, suggesting the declaration and drawing walls worked as intended.

Later, however I realised that it is difficult to detect collisions, necessary for requirement 5. I thus changed my implementation of walls to declaring them as sprites, as this enabled to create a list of all walls and all sprites in general to display all walls using one command and to detect collisions with all the walls using one list.

```

28. # List containing all sprites in the program to draw them
29. all_sprites_list = pygame.sprite.Group()
30.
31. # Define wall
32. class Wall(pygame.sprite.Sprite):
33.     #initialises parameters of a wall
34.     def __init__(self, corner_x, corner_y, width, height, color):
35.         super().__init__() #inherits all parameters of a sprite
36.         self.width = width
37.         self.height = height
38.         self.color = color
39.         #set the image of the wall
40.         self.image = pygame.Surface([width, height])
41.         self.image.fill(color)
42.         self.rect = self.image.get_rect() #finds the rectangle object that has the dimensions of the image
43.         self.rect.x = corner_x
44.         self.rect.y = corner_y
45.         #draws a wall
46.         def draw(self):
47.             pygame.draw.rect(screen, self.color, [self.rect.x, self.rect.y, self.width, self.height])
48.
49. #this is the list of all walls in the game
50. wall_list = pygame.sprite.Group()

```

For these purposes, I created a list to store all sprites in it for drawing. List *wall_list* will be later used for collision detection. This way of initiation of wall enabled to run all the procedures required for collision detection when initiating the wall. The method for drawing the wall was kept in case a wall had to be drawn separately from all other walls and sprites.

```

# --- Drawing
all_sprites_list.draw(screen)

```

This command was used for drawing all sprites on the screen. I tested the working of this method and initiation of walls using the following exemplary wall.

```

56. # Top horizontal wall
57. walltophor = Wall(0, 0, 500, 10, colorofwalls)
58. all_sprites_list.add(walltophor)
59. wall_list.add(walltophor)

```

This was the result of this test:

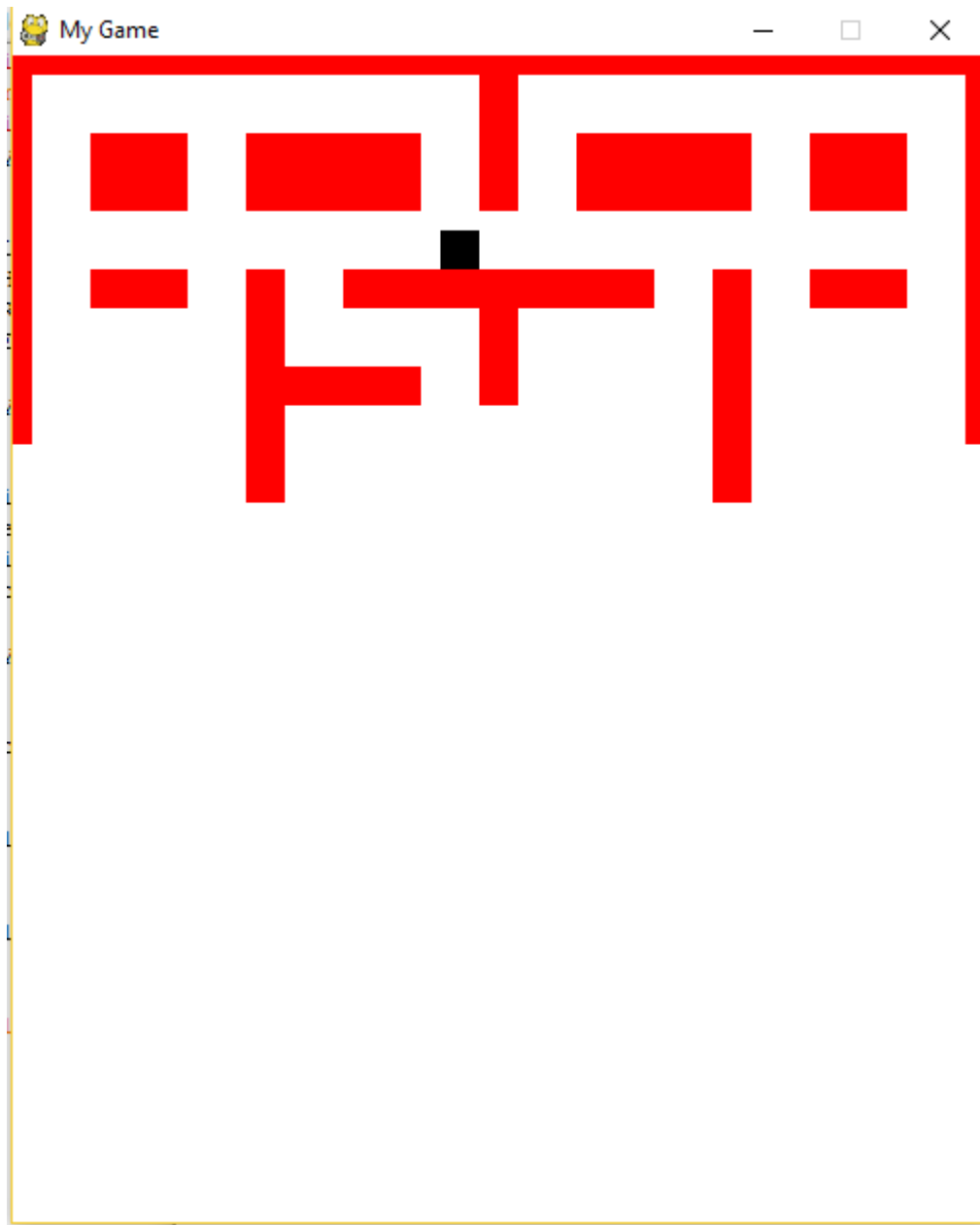


This meant my way for initiating walls worked, along with the method of drawing. After adding a few walls, as shown on the picture above, I have noticed this method for adding walls to the map is very inefficient in terms of time spent on adding all the walls.


```

56. # Top horizontal wall
57. walltophor = Wall(0, 0, 500, 10, colorofwalls)
58. all_sprites_list.add(walltophor)
59. wall_list.add(walltophor)
60.
61. # Top left wall
62. walllefttop1 = Wall(0,0, 10, 200, colorofwalls)
63. all_sprites_list.add(walllefttop1)
64. wall_list.add(walllefttop1)
65.
66. # Top middle wall
67. wallmidtop = Wall(240, 0, 20, 80, colorofwalls)
68. all_sprites_list.add(wallmidtop)
69. wall_list.add(wallmidtop)
70.
71. # Top right wall
72. wallrighttop = Wall(490,0, 10, 200, colorofwalls)
73. all_sprites_list.add(wallrighttop)
74. wall_list.add(wallrighttop)
75.
76. # Top left square 1
77. topleftsquare1 = Wall(40, 40, 50, 40, colorofwalls)
78. all_sprites_list.add(topleftsquare1)
79. wall_list.add(topleftsquare1)
80.
81. # Top left square 2
82. topleftsquare2 = Wall(120, 40, 90, 40, colorofwalls)
83. all_sprites_list.add(topleftsquare2)
84. wall_list.add(topleftsquare2)
85.
86. # Top right square 1
87. toprightsquare1 = Wall(410, 40, 50, 40, colorofwalls)
88. all_sprites_list.add(toprightsquare1)
89. wall_list.add(toprightsquare1)
90.
91. # Top right square 2
92. toprightsquare2 = Wall(290, 40, 90, 40, colorofwalls)
93. all_sprites_list.add(toprightsquare2)
94. wall_list.add(toprightsquare2)
95.
96. # Left mid square
97. leftmidsquare = Wall(40, 110, 50, 20, colorofwalls)
98. all_sprites_list.add(leftmidsquare)
99. wall_list.add(leftmidsquare)
100.
101. # Right mid square
102. rightmidsquare = Wall(410, 110, 50, 20, colorofwalls)
103. all_sprites_list.add(rightmidsquare)
104. wall_list.add(rightmidsquare)
105.

```



I have thus decided to create a new method of adding walls. This would be based on passing the design of the map using a list of strings, each string being a representation of a row.

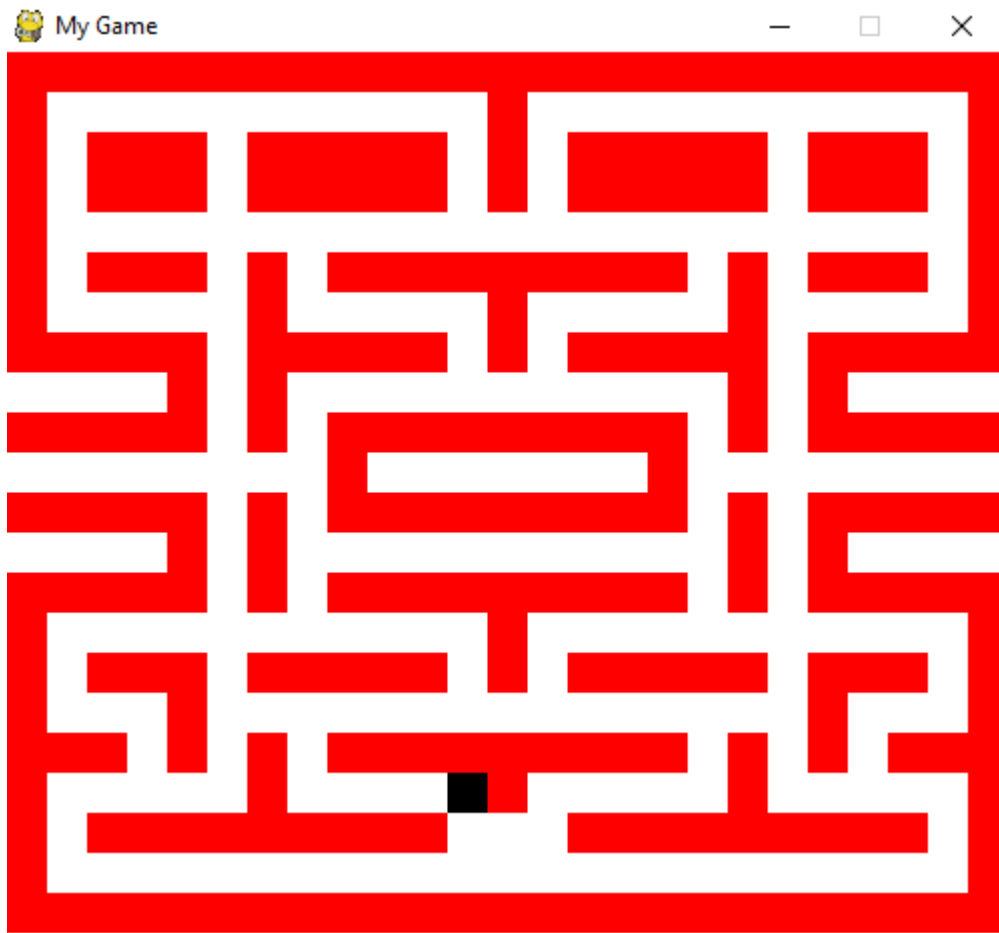
The input to the function creating a map is a list of strings. Each string is a description of a row. The reason for choosing a string is it will be easily understood by a human, because each element can have its representation, for example a wall can be represented by a “W” and a coin by a “.”. I have started with creating the loop and one case inside it. If a character in the string is a “W”, the function creates a wall at that place. This wall is immediately added to the list of all sprites to be drawn and list of walls to detect the collisions.

Testing

As a test, I passed my design of the wall, identical to the classical Pacman design, to the function and drew it.

```
57. # Function for adding the map from a given string
58. def adding_map(mapdescription, heightofwalls, widthofwalls, colorofwalls):
59.     # For each row, starting from 0
60.     current_row = 0
61.     for each_string in mapdescription:
62.         # For each column
63.         for i in range(0, len(each_string)):
64.             # Letter W means wall
65.             if each_string[i] == "W":
66.                 newWall = Wall(i*widthofwalls, current_row*heightofwalls, widthofwalls, heightofwalls, colorofwalls)
67.                 wall_list.add(newWall)
68.                 all_sprites_list.add(newWall)
69.             current_row += 1
70.
71. # Declaration of the maze
72. basicmap = ["#####",
73.             "W      W      W",
74.             "W WWW WWWWW W WWWWWW WWW W",
75.             "W WWW WWWWW W WWWWWW WWW W",
76.             "W      W",
77.             "W WWW W WWWWWWWWW W WWW W",
78.             "W      W      W      W W",
79.             "WWWWW WWWWW W WWWWW WWWWW",
80.             "      W W      W W      ",
81.             "WWWWW W WWWWWWWWW W WWWWW",
82.             "      W      W",
83.             "WWWWW W WWWWWWWWW W WWWWW",
84.             "      W W      W W      ",
85.             "WWWWW W WWWWWWWWW W WWWWW",
86.             "W      W",
87.             "W WWW WWWWW W WWWWWW WWW W",
88.             "W      W      W W",
89.             "WWW W W WWWWWWWWW W W WWW",
90.             "W      W      W W",
91.             "W WWWWWW      WWWWWW W",
92.             "W      W",
93.             "#####"]
94. adding_map(basicmap, heightofwalls, widthofwalls, colorofwalls)
95.
```

This was the result of the test:



As above, the function was working. The design will be contacted with the end user again, to see if it suits him.

Moving objects

Attributes and methods

I declared one object which will be later used for any moving objects in game, for example Pacman or enemies. Its attributes were *height*, *width*, *color* for drawing and coordinates, *speed* and *direction* for moving it. The methods were *move()* for moving, *new_direction()* for changing direction of movement. There will also be methods for retrieving data from an object in case it is needed.

```

41. # Define moving object
42. class moving_object(object):
43.
44.     #initialises theobject
45.     def __init__(self, speed, x, y, direction, height, width, color):
46.         self.speed = speed
47.         self.x = x
48.         self.y = y
49.         self.direction = direction
50.         self.height = height
51.         self.width = width
52.         self.color = color
53.
54.     #moves the object
55.     def move(self):
56.         #moves upwards
57.         if self.direction == 1:
58.             self.y -= self.speed
59.         #moves right
60.         if self.direction == 2:
61.             self.x += self.speed
62.         #moves down
63.         if self.direction == 3:
64.             self.y += self.speed
65.         #moves left
66.         if self.direction == 4:
67.             self.x -= self.speed
68.
69.     #allows the direction of movement to be changed
70.     def new_direction(self, newdirection):
71.         self.direction = newdirection
72.
73.     #returns direction
74.     def get_direction(self):
75.         return self.direction
76.
77.     #returns speed
78.     def get_speed(self):
79.         return self.speed
80.
81.     #returns coordinates
82.     def get_coordinates(self):
83.         return (self.x, self.y)
84.
85.     def draw(self):
86.         pygame.draw.rect(screen, self.color, [self.x, self.y, self.width, self.height])

```

Calling method move will move the object accordingly, changing direction will occur when there will be a signal from the user from the keyboard. Function draw will be used to draw the object.

Moving the objects

In each cycle of the game we are going to move the Pacman, so in the loop there will be a call *Pacman.move()*.

```

132. #Pacman moves every step in the game
133. Pacman.move()

```

Its direction can change, so if user presses down on the key, a new direction will be assigned to Pacman.

```

113.         #user pressed down on a key
114.         elif event.type == pygame.KEYDOWN:
115.
116.             #find if it was an arrow and adjust direction accordingly
117.             # Pacman now moves right
118.             if event.key == pygame.K_RIGHT:
119.                 Pacman.new_direction(2)
120.             # Pacman now moves up
121.             if event.key == pygame.K_UP:
122.                 Pacman.new_direction(1)
123.             # Pacman now moves down
124.             if event.key == pygame.K_DOWN:
125.                 Pacman.new_direction(3)
126.             # Pacman now moves left
127.             if event.key == pygame.K_LEFT:
128.                 Pacman.new_direction(4)

```

This change will change the direction of movement in the next loop.

Pacman has to be drawn at its current position in every run of the loop, so I used method `draw()` defined in `moving_object`.

```

145.         #draw the current position of pacman
146.         Pacman.draw()

```

Testing

I tested these methods in functions by initiating an object with the following values and moving it on the screen.

```

94.     #initialise variables
95.     initxPac = 0
96.     inityPac = 0
97.     initspeedPac = 1
98.     initdirectionPac=2
99.     heightPac = 10
100.    widthPac = 10
101.    colorPac = BLACK
102.
103.    Pacman = moving_object(initspeedPac, initxPac, inityPac, initdirectionPac, widthPac, heightPac, colorPac)

```

As a result of this test, a black square 10 by 10 pixels appeared on the screen and its movement followed what I was pressing on the keyboard, indicating my functions and the object work as intended.

Collision detection

The next step was detecting collisions between Pacman and walls. It turned out this would be difficult to do using the method I proposed, so I used a different method, that is declared walls and moving objects and sprites, as pygame has a function for detecting collisions between sprites.

My initial method was to check if there is a collision between Pacman and any wall and move Pacman if there is none.

```

406.         # Checks if there is a collision and moves Pacman if not
407.         if not pygame.sprite.spritecollide(Pacman, wall_list, False):
408.             Pacman.move()
409.

```

It turned out, however, after testing this function, that once the Pacman touched the wall, it could not move anymore. I had to thus amend the code. My approach was to move Pacman,

but not display it. If there was a collision, move it back by reversing its direction and moving it, then resetting the original direction. The code for this function is displayed below.

```
406. # Checks the new position of Pacman
407. Pacman.move()
408. # If it is such that there would be a collision, moves Pacman back
409. if pygame.sprite.spritecollide(Pacman, wall_list, False):
410.     if Pacman.get_direction() == 3:
411.         Pacman.new_direction(1)
412.         Pacman.move()
413.         Pacman.new_direction(3)
414.     if Pacman.get_direction() == 1:
415.         Pacman.new_direction(3)
416.         Pacman.move()
417.         Pacman.new_direction(1)
418.     if Pacman.get_direction() == 2:
419.         Pacman.new_direction(4)
420.         Pacman.move()
421.         Pacman.new_direction(2)
422.     if Pacman.get_direction() == 4:
423.         Pacman.new_direction(2)
424.         Pacman.move()
425.         Pacman.new_direction(4)
```

Testing

I tested the working of this function and Pacman was moving, it could not go through the walls and it could move after having touched the wall.

The problem I encountered was the fact that once user pressed a key such that the Pacman would move into a wall, the Pacman stopped moving. This problem made it very difficult for the user to turn if they wanted to take a turn in a place that was not a corner. It was also noted that in the original Pacman version, if a direction key is pressed before the available turn, Pacman moves with unchanged direction up to that place and then turns without any additional signal from the user. I thus developed a new method, and several “helper” methods in order to solve that turning problem.

Turning algorithm

The key idea behind the turning algorithm was to create a copy of Pacman, not displayed on the screen, which would be used for finding the place where Pacman can turn. When a direction key is pressed, the clone of Pacman is created. This clone takes the turn the user wants to make. If there would be a collision with the wall, Pacman’s direction remains unchanged, Pacman moves to a new position and the clone moves there as well. This happens until the turn the clone made does not create a collision with the wall. If this situation occurs, Pacman’s direction is changed and it takes that turn.

First of all, the method for moving Pacman and ghosts on the screen and the method for getting the new intended direction of Pacman were placed inside *moving_object* as a

method to encapsulate everything associated with moving objects inside them. The amended code for that part looked the following way:

```
185.     def moving_object_detecting_collisions(self, wall_list):
186.         # Checks what would happen if the object would be moved as intended
187.         self.move()
188.         # If there is a collision, moves the object back in the opposite direction
189.         # Object ends up in the same position as it was before the collision
190.         if pygame.sprite.spritecollide(self, wall_list, False):
191.             if self.get_direction() == 3:
192.                 self.new_direction(1)
193.                 self.move()
194.                 self.new_direction(3)
195.             if self.get_direction() == 1:
196.                 self.new_direction(3)
197.                 self.move()
198.                 self.new_direction(1)
199.             if self.get_direction() == 2:
200.                 self.new_direction(4)
201.                 self.move()
202.                 self.new_direction(2)
203.             if self.get_direction() == 4:
204.                 self.new_direction(2)
205.                 self.move()
206.                 self.new_direction(4)
```

```
208.     def signalfromkeyboard(self, event):
209.         # find if it was an arrow and adjust the next direction accordingly
210.         # Pacman now moves right
211.         if event.key == pygame.K_RIGHT:
212.             newdirection = 2
213.         # Pacman now moves up
214.         if event.key == pygame.K_UP:
215.             newdirection = 1
216.         # Pacman now moves down
217.         if event.key == pygame.K_DOWN:
218.             newdirection = 3
219.         # Pacman now moves left
220.         if event.key == pygame.K_LEFT:
221.             newdirection = 4
222.         self.new_direction(newdirection)
```

Using the methods for *moving_object* I created a new method for detecting whether there would be a collision if the object was moved. It moved the object and checked whether there was a collision. Because the object was passed as an argument, the method did not change the actual value of attributes of the object, so this way to create this method did not require moving the object back.

```
227.     def wouldcollide(self, wall_list):
228.         self.move()
229.         if pygame.sprite.spritecollide(self, wall_list, False):
230.             return True
231.         else:
232.             return False
```

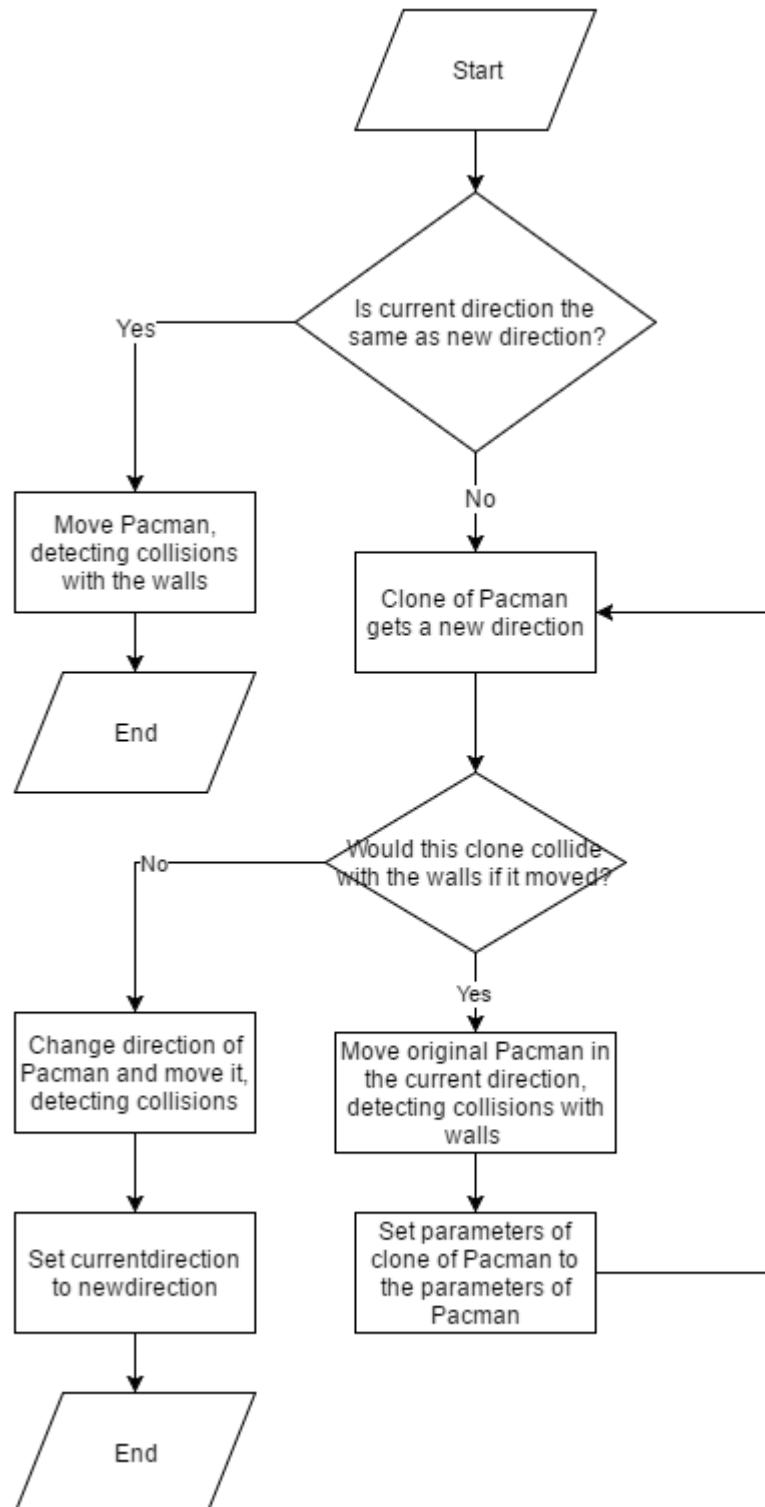
The next function I created was a function for creating a copy of Pacman that would later be used in the turning algorithm. It was straightforward, as its all parameters were the same of Pacman's. To safely access the parameters of Pacman, I used methods which returned value of appropriate attribute of Pacman. This was achieved with the code below.


```

248. # Will be used for implementing changing direction as it is in original Pacman
249. def createacopyofPacman(Pacman):
250.     Pacmanexampleforcollisions = moving_object(Pacman.get_speed(), Pacman.get_direction(), widthPac, heightPac,
251.                                                 colorPac)
252.     Pacmanexampleforcollisions.rect.x = Pacman.rect.x
253.     Pacmanexampleforcollisions.rect.y = Pacman.rect.y
254.     return Pacmanexampleforcollisions
255.

```

The actual algorithm for moving Pacman in the same way as it happens in the original version works in the following way:



This is repeated every time in the main loop of the program. The code looks in the following way:

```
280.     # Pacman moves every step in the game
281.     # If there was a change of the direction coming from the user
282.     if currentdirection != newdirection:
283.         # Then uses the example of Pacman to verify if turn possible
284.         Pacmanexampleforcollisions.new_direction(newdirection)
285.         # And checks if there would be a collision
286.         if Pacmanexampleforcollisions.wouldcollide(wall_list):
287.             # If so, moves Pacman as if there was nothing from the user
288.             Pacman.moving_object_detecting_collisions(wall_list)
289.             # And moves the example Pacman as well
290.             Pacmanexampleforcollisions = createacopyofPacman(Pacman)
291.         else:
292.             # If no collision, checks the direction of movement of Pacman
293.             Pacman.new_direction(newdirection)
294.             # And moves Pacman
295.             Pacman.moving_object_detecting_collisions(wall_list)
296.             # Now the Pacman moves in the direction in which the user wants it to
297.             currentdirection = newdirection
298.     # If there is no signal from the user, just moves the Pacman preventing collisions
299.     else:
300.         Pacman.moving_object_detecting_collisions(wall_list)
```

Testing

This was working well, apart from two situations:

1. Pacman did not take the first turn as intended

To solve this problem, I used the debugger to see what happens to the clone of Pacman inside the program. It turned out that when the direction key was pressed for the first time, clone of Pacman had not been initiated yet. A line added in the case when the user pressed on the direction key solved the problem:

```
272.     # user pressed down on a key
273.     elif event.type == pygame.KEYDOWN:
274.         Pacmanexampleforcollisions = createacopyofPacman(Pacman)
```

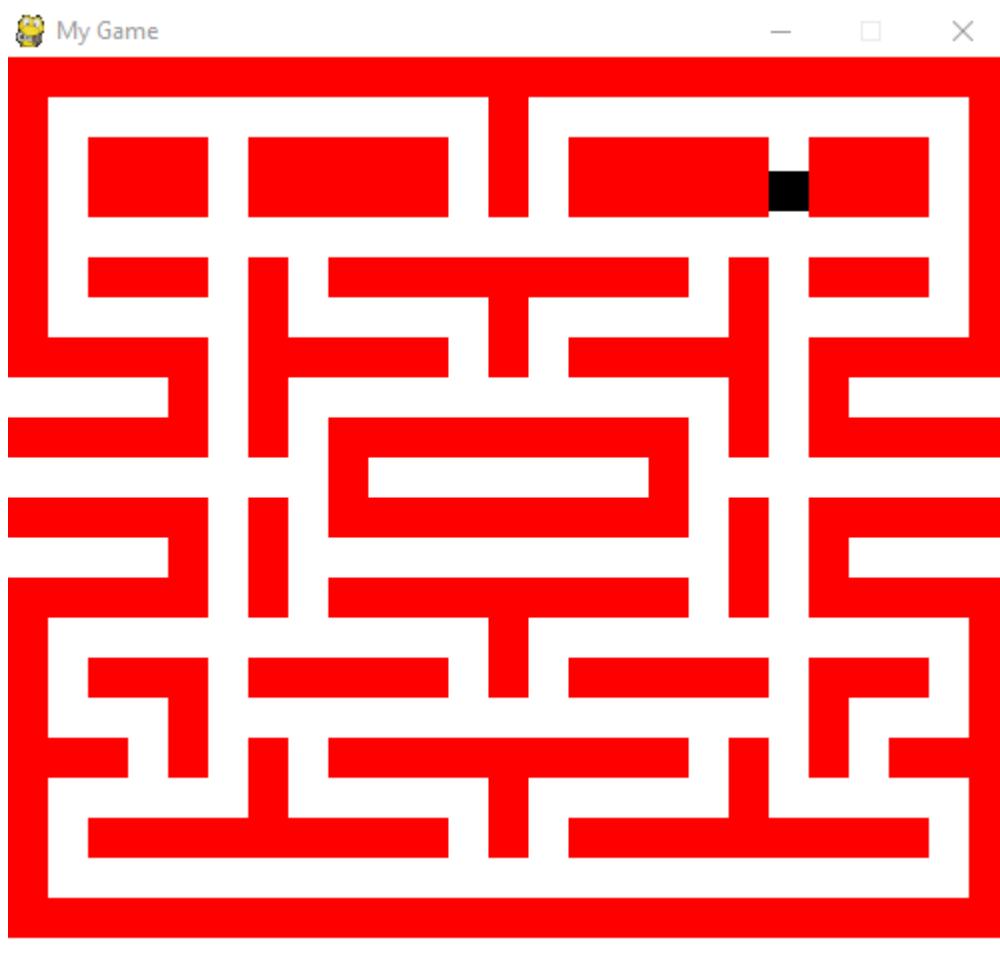
2. Pacman stopped when a lot of keys were pressed at the same time.

When I tested the working of this function, when pressing a key other than the arrow key Pacman stopped and the following error was returned:

```
>>> ===== RESTART =====
>>>
Traceback (most recent call last):
  File "C:\Users\Stanislaw\Documents\wszystko\Dulwich College\Projects\CS Project\arcade game5.py", line 230, in <module>
    Pacmanexampleforcollisions.signalfromkeyboard(event)
  File "C:\Users\Stanislaw\Documents\wszystko\Dulwich College\Projects\CS Project\arcade game5.py", line 184, in signalfromkeyboard
    self.new_direction(newdirection)
UnboundLocalError: local variable 'newdirection' referenced before assignment
>>> |
```

This meant there was an error in function which changed direction when there was a signal from the keyboard. The reason was, after inspection of the function, that the function had a line depending on newdirection. This line was executed regardless of which key was

pressed. `newdirection`, however, had a value assigned only when one of the arrow keys was pressed. If a key different to an arrow key was pressed, `newdirection` did not have a value, but was used. The result looked like on the picture below: Pacman not moving even though there is nothing stopping it.



To solve this problem, I moved line `self.new_direction(newdirection)` inside the conditional statements, meaning it would only be executed if one of the arrows is pressed.

```
208.     def signalfromkeyboard(self, event):
209.         # find if it was an arrow and adjust the next direction accordingly
210.         # Pacman now moves right
211.         if event.key == pygame.K_RIGHT:
212.             newdirection = 2
213.             self.new_direction(newdirection)
214.         # Pacman now moves up
215.         if event.key == pygame.K_UP:
216.             newdirection = 1
217.             self.new_direction(newdirection)
218.         # Pacman now moves down
219.         if event.key == pygame.K_DOWN:
220.             newdirection = 3
221.             self.new_direction(newdirection)
222.         # Pacman now moves left
223.         if event.key == pygame.K_LEFT:
224.             newdirection = 4
225.             self.new_direction(newdirection)
226.
```

After the amendment, the movement algorithm was working exactly like in the original version.

Tunnel

Another function to implement was for the Pacman to be able to go through the tunnel in the middle of the screen. The logic is as follows: if the Pacman *x_coordinate* is equal to the width of screen, it should be changed to zero. Similarly, if it is zero, it should be set to the width of the screen. The code looks as follows:

```
269     def tunnel(x_coord, width):
270         if x_coord > width:
271             x_coord = 0
272         if x_coord < 0:
273             x_coord = width
274         return x_coord
```

Testing

I tested it immediately by making Pacman go through the tunnel. It appeared on the other side, indicating this function works as intended.

Coins

The first step to create the functionality of coins is to declare a class for coins and declare a list for storing them. I decided to create coins in form of squares placed in the centre of free spaces. My method of initiation of this class was the same as the one for declaring walls.

```
48. # Define coins to collect
49. class coin(pygame.sprite.Sprite):
50.     # initialises parameters of the coin
51.     def __init__(self, corner_x, corner_y, width, height, color):
52.
53.         super().__init__()
54.         self.width = width
55.         self.height = height
56.         self.color = color
57.         # set the image of the coin
58.         self.image = pygame.Surface([width, height])
59.         self.image.fill(color)
60.         self.rect = self.image.get_rect() # finds the rectangle object that has the dimensions of the image
61.         self.rect.x = corner_x
62.         self.rect.y = corner_y
63.
```

If in the map there is a dot at some point, the function for adding map should add a coin to the *coin_list* and *all_sprites_list*, so I added the following code to the function *adding_map*.

```
90.         # A . means a coin
91.         elif each_string[i] == ".":
92.             x_coordinate = i * widthofwalls+(widthofwalls-widthofcoins)/2
93.             y_coordinate = current_row * heightofwalls+(heightofwalls-heightofcoins)/2
94.             newCoin = coin(x_coordinate, y_coordinate, widthofcoins, heightofcoins, colorofcoins)
95.             coin_list.add(newCoin)
96.             all_sprites_list.add(newCoin)
97.             current_row += 1
98.
```

I also want the coins to disappear whenever they are collected by my Pacman. To do that, I implemented a method of checking if Pacman collides with a coin, using the following method in *moving_object*.

```

228.     # This function checks whether Pacman collects a coin
229.     def collectcoin(self, coin_list):
230.         if pygame.sprite.spritecollide(self, coin_list, True):
231.             return True
232.         else:
233.             return False

```

Outside of the object, I defined a new function, called *scorecounter*, which increases the score by some number every time a coin is collected and makes the coin disappear. At the beginning, Pacman was disappearing as well, so at the end of the function Pacman is added *back to all_sprites_list* so that it is displayed. The code for this method looks as below. It changes the score, the list of coins and the list of all sprites and then returns all of them.

```

258.     # This function checks if the coin is collected and increments score
259.     def scorecounter(score, coin_list, Pacman, pointspercoin, all_sprites_list):
260.         if Pacman.collectcoin(coin_list):
261.             # Increments score
262.             score += pointspercoin
263.             # Delete the coin from the coin_list
264.             pygame.sprite.spritecollide(Pacman, coin_list, True)
265.             # Delete the coin from the list of all sprites so that it is not displayed
266.             pygame.sprite.spritecollide(Pacman, all_sprites_list, True)
267.             # Add Pacman to ensure it is displayed
268.             all_sprites_list.add(Pacman)
269.             return (score, coin_list, all_sprites_list)

```

In the main program this function is called once in every loop and the values it returns are passed using a tuple. The new values of *score*, *coin_list* and *all_sprites_list* are assigned to the values passed by the function.

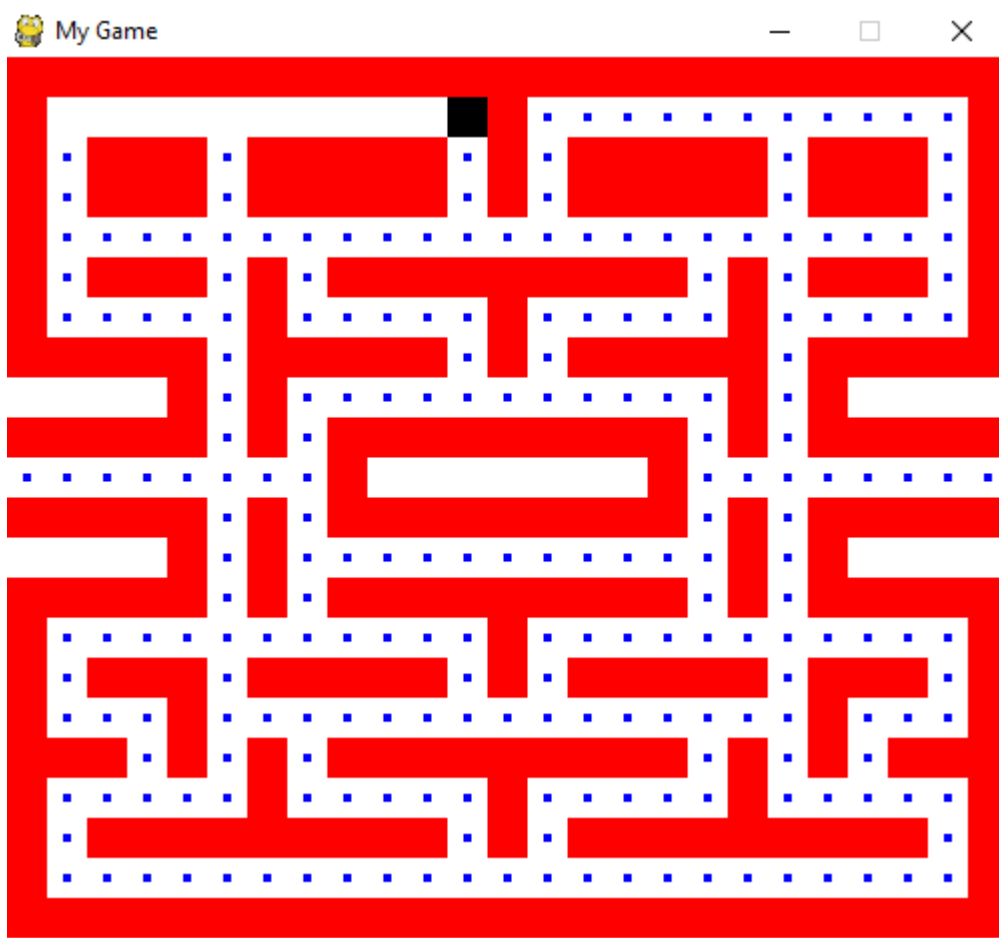
```

328.     # Update score and make the coin disappear
329.     newscorecoinlistandspriteslist = scorecounter(score, coin_list, Pacman, pointspercoin, all_sprites_list)
330.     score = newscorecoinlistandspriteslist[0]
331.     coin_list = newscorecoinlistandspriteslist[1]
332.     all_sprites_list = newscorecoinlistandspriteslist[2]

```

Testing

I tested the functioning of all of these functions by running the program, moving Pacman and printing the score to the shell. The program printed the score correctly to the shell, indicating the functions and methods work as intended.



Enemies

The enemies will change their directions if they are at a point from which they can move in at least three directions. This will ensure they only change their direction at a crossing. Whichever method will be used for choosing the enemies' direction, I decided to use a graph in all of them, because that will be the most elegant representation of the map.

Graphs

The graph will consist of nodes and edges. There will be three classes: *node*, *edge* and *graph*. *Edge* will also have a subclass *weightededge*, which will allow for the graph to be weighted. *Graph* will be a subclass of a more general class, *digraph* (short for directed graph).

Node

Node requires only one attribute: *name*. The methods in node will be: constructor, a method for returning its name, and a function which will print its name whenever *print(node)* is called. This is used for *print* to be useful in the case of a node, because without it the address of the object would be printed, having no meaning for a human. The following code was used to implement the class *node*.

```

269. class node(object):
270.     # Has a name, x and y coordinates
271.     def __init__(self, name):
272.         self.name = name
273.     # Returns the name of the object
274.     def getName(self):
275.         return self.name
276.     # Returns the name of the object if print(node) is called
277.     def __str__(self):
278.         return self.name

```

Edge

An edge requires a *destination*, *source* and *weight*, in case of a weighted edge. Similarly to the node, an edge requires a method to return destination, a method to return source and print something meaningful when *print(edge)* is called. The following code is used to implement an edge.

```

280. class edge(object):
281.     # Initiates the object
282.     def __init__(self, src, dest):
283.         self.src = src
284.         self.dest = dest
285.     # Returns the source edge
286.     def getSource(self):
287.         return self.src
288.     # Returns the destination edge
289.     def getDestination(self):
290.         return self.dest
291.     # Returns the weight of the edge
292.     def __str__(self):
293.         return str(self.src)+ '->' +str(self.dest)

```

A weightedEdge will be a subclass of an edge, which will have a different constructor, it will have an additional method of returning weight, and a method for printing which will include its weight. The following code is used to implement a weighted edge.

```

295. # This edge is weighted
296. class weightedEdge(edge):
297.     # Overrides the constructor
298.     def __init__(self, src, dest, weight=1.0):
299.         super().__init__(src, dest)
300.         self.src = src
301.         self.dest = dest
302.         self.weight = weight
303.     # Returns the weight
304.     def getWeight(self):
305.         return self.weight
306.     # Prints the edge
307.     def __str__(self):
308.         return str(self.src) + '->(' + str(self.weight) + ')' \
309.             + str(self.dest)

```

Graph

I represent the graph with an adjacency list (a dictionary in Python). I decided to use an adjacency list, because it is easier to handle with dictionaries than an adjacency matrix. In order to be able to retrieve weights of the edges, it will be a dictionary of dictionaries. In order not to duplicate nodes, I will make use of a Python-specific data structure: set (an

unordered list, without indexes), which will store nodes. Thus, the constructor of a directed graph looks as follows.

```
311. # Creates a directed graph
312. class Digraph(object):
313.
314.     # Consists of a dictionary of edges and a set of nodes
315.     def __init__(self):
316.         # Contains all nodes, used to check node duplication
317.         self.nodes = set([])
318.         # Dictionary of dictionaries of edges coming out of a node
319.         self.edges = {}
```

In order to create a graph, we have to be able to add nodes and edges. When adding a node, the program will first check that it is not in the graph yet. If it is, the program should raise an exception. If it is not, a node will be added to the set of nodes. Moreover, an empty dictionary will be added to the dictionary of edges, with the key being the node. This is done with the following code.

```
321.     def addNode(self, node):
322.         # Checks if node is already in a graph
323.         if node in self.nodes:
324.             raise ValueError('Duplicate node')
325.         # If not adds node to the set of nodes and a list of edges for the node, initially empty
326.         else:
327.             # Node is added to the set
328.             self.nodes.add(node)
329.             # An empty dictionary of edges (to store their length) is added to that node in the dictionary
330.             self.edges[node] = {}
```

To add an edge we want to add a dictionary of the form *{destination: weight}* to the dictionary of edges, with the key being the source. In the end we will have a dictionary of the following form: *{source1: {destinationA: weightA, destinationB: weightB}, source2: {destinationX: weightX}}*. In order to do that, we have to know the weight, source, and destination of an edge, which we can do using methods in the definition of class *weightedEdge*. Next, we will call *self.edges[source][destination]=weight*, so that the data of the form above is added to adjacency list of the graph. This all is achieved with the following code.

```
332.     def addEdge(self, weightededge):
333.         # Gets the source node of the edge
334.         src = weightededge.getSource()
335.         # Gets the destination node of the edge
336.         dest = weightededge.getDestination()
337.         # Gets the weight of the edge
338.         weight = weightededge.getWeight()
339.         # Checks if both of them are in the graph
340.         if not(src in self.nodes and dest in self.nodes):
341.             raise ValueError('Node not in graph')
342.         # If they are, adds the destination node to the list of edges coming out of source edge
343.         self.edges[src].append(dest)
344.         # The weight of the edge is stored as an element in the dictionary inside the dictionary
345.         self.edges[src][dest] = weight
```

Finally, we need to be able to check whether a given node is in the graph and have a method which would return all children of a given node, in order to perform DFS and BFS. Moreover, we would like to have a method which would print the representation of a graph. To find whether a node is in the graph we use the following code.

```
351.     # Checks if a node is in a graph
352.     def hasNode(self, node):
353.         return node in self.nodes
```


When getting the children of a node, we also want to get the weight of the edge connecting them, which can be done using the following code.

```
347.     # Returns all nodes to which one can go from a given node
348.     def childrenOf(self, node):
349.         return self.edges[node]
350.
```

Finally, when printing we want the list of all edges in form which would present the source, destination and its weight.

```
355.     # Presents the graph as a string of all edges, with their sources and destinations
356.     def __str__(self):
357.         res = ''
358.         # For each node in the dictionary
359.         for k in self.edges:
360.             # For each edge print source, weight and destination
361.             for d in self.edges[k]:
362.                 res = res + str(k) + '->(' + str(self.edges[k][d]) + ')' + str(d) + '\n'
363.         return res[:-1]
```

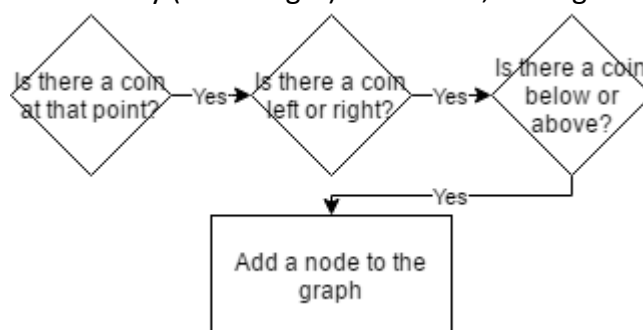
When implementing an undirected graph, we want all edges to be in both directions, which can be done by adding a reversed edge when adding a *weightedEdge*. This is done with the following code.

```
365.     # Declares a subclass of a digraph - an undirected graph
366.     class Graph(Digraph):
367.         # Overrides the function is the superclass
368.         def addEdge(self, weightededge):
369.             # Adds the edge to the graph
370.             Digraph.addEdge(self, weightededge)
371.             # Reverses the edge
372.             rev = weightededge(weightededge.getDestination(), weightededge.getSource(), weightededge.getWeight())
373.             # Adds the reverse edge
374.             Digraph.addEdge(self, rev)
```

This gives a full representation of an undirected graph, which will allow us to use Depth First Search or Dijkstra's algorithm when finding the shortest path to an object.

Creating the graph of the map

In order to be able to use the graph in a meaningful way in the game, the program has to construct a graph from a given list of strings, that is the map description. To do that, it has to create a graph, add nodes and edges to it. Nodes have to be added first, because without them no edges can be created. In order to add these nodes, the program will go through every element in the list and detect crossings, add the nodes. A node is where a path turns, so if at a point there is no wall, there is a path vertically (top or bottom) and a path horizontally (left or right). Therefore, the logic is as follows:



This was implemented using the following code.

```

376. # This is a function which creates a graph for a given string
377. def graphconstructor(mapdescription):
378.     mapGraph = Graph()
379.     # Goes through each row in the map
380.     for row in range(0, len(mapdescription)):
381.         # Goes through every field in that row
382.         for field in range(0, row):
383.             # For every point which may be a node
384.             if mapdescription[row][field] == ".":
385.                 # It needs to have a way down or up
386.                 if mapdescription[row+1][field] == "." or mapdescription[row-1][field] == ".":
387.                     # And a way left or right
388.                     if mapdescription[row][field+1] == "." or mapdescription[row][field-1] == ".":
389.                         # Then it's a node
390.                         mapGraph.addNode(node(str(row)+" "+str(field)))
391.

```

Testing

I then used the following command to find whether a correct list of nodes is produced.

```

422. for eachNode in graphconstructor(basicmap).nodes:
423.     print (eachNode)

```

Below is the output of the program.

```

C:\Python31\python.exe "C:/Users/SSZ/Documents/arcade game.py"
6 5
8 7
10 5
10 7
12 7
14 1
14 5
14 7
14 11
14 13
16 1
16 3
16 5
16 7
16 11
16 13
18 1
18 3
18 5
18 7
18 11
18 13
18 17
20 1
20 11
20 13
4 1
6 1

```

This set of nodes is incomplete, it does not return all nodes in the graph, as intended. After stepping through the code, I realised line 382. was incorrect. The range should have been the length of the string representing the row, not its number. I then changed the code so that it looked as follows.

```

376. # This is a function which creates a graph for a given string
377. def graphconstructor(mapdescription):
378.     mapGraph = Graph()
379.     # Goes through each row in the map
380.     for row in range(0, len(mapdescription)):
381.         # Goes through every field in that row
382.         for field in range(0, len(mapdescription[row])):
383.             # For every point which may be a node
384.             if mapdescription[row][field] == ".":
385.                 # It needs to have a way down or up
386.                 if mapdescription[row+1][field] == "." or mapdescription[row-1][field] == ".":
387.                     # And a way left or right
388.                     if mapdescription[row][field+1] == "." or mapdescription[row][field-1] == ".":
389.                         # Then it's a node
390.                         mapGraph.addNode(node(str(row)+" "+str(field)))
391.     return mapGraph

```

When the command for displaying all nodes was called, the program printed a complete set of nodes on the map.

Development - continued

The next step was adding edges to the graph. For each node on the graph, the program will check if it is connected to a node to its right or below. If there is, it will make a connection by creating an edge and adding it to the graph. The first step is to, for each node, get its coordinates, that is the row and the column. When creating the nodes, their coordinates were separated by space, so the program will iterate through chars in the string until it encounters the space. The row will be the number before the space and the column will be the number after the space. The program needs to convert the row and column from a string to an integer for it to be used later. For each node, the program will check if there is a path to the right or below. If there is, it will go along that path until it finds another node, find how far it is and add an edge between these nodes to the graph. I first used the following code to find out if the program would add the vertical edges.

```

376. # This is a function which creates a graph for a given string
377. def graphconstructor(mapdescription):
378.     mapGraph = Graph()
379.     # Goes through each row in the map
380.     for row in range(0, len(mapdescription)):
381.         # Goes through every field in that row
382.         for field in range(0, len(mapdescription[row])):
383.             # For every point which may be a node
384.             if mapdescription[row][field] == ".":
385.                 # It needs to have a way down or up
386.                 if mapdescription[row+1][field] == "." or mapdescription[row-1][field] == ".":
387.                     # And a way left or right
388.                     if mapdescription[row][field+1] == "." or mapdescription[row][field-1] == ".":
389.                         # Then it's a node
390.                         mapGraph.addNode(node(str(row)+" "+str(field)))
391.
392.     # Now adds edges to the graph
393.     for eachNode in mapGraph.nodes:
394.         # Gets the coordinates of the node to use it
395.         ycoord = ""
396.         iterator=0
397.         while eachNode.getName()[iterator]!=" ":
398.             # Gets y-coordinate, row number
399.             ycoord+=eachNode.getName()[iterator]
400.             iterator+=1
401.         # Makes it a number, rather than a string
402.         ycoord = int(ycoord)
403.         # Gets x-coordinate, column number, converts it to a string
404.         xcoord = int(eachNode.getName()[iterator+1:])
405.         # If there is space below to move then there must be a node somewhere on the right
406.         if mapdescription[ycoord+1][xcoord] == ".":
407.             # Will measure the weight of that edge
408.             distancecounter = 0
409.             # Moves while there is no wall on the right
410.             while mapdescription[ycoord][xcoord] != "W":
411.                 # Changes the weight of the edge
412.                 ycoord+=1
413.                 distancecounter+=1
414.             # Accounts for logic in the loop above
415.             ycoord-=1
416.             distancecounter-=1
417.             # Adds the edge
418.             mapGraph.addEdge(weightedEdge(eachNode, node(str(ycoord)+" "+str(xcoord)), distancecounter))
419.     return mapGraph

```

Testing

To test it, I used the following command.

```
422. print(graphconstructor(basicmap))
```

The result was as follows.

```

Traceback (most recent call last):
  File "C:/Users/SSZ/Documents/arcade game.py", line 421, in <module>
    print(graphconstructor(basicmap))
  File "C:/Users/SSZ/Documents/arcade game.py", line 418, in graphconstructor
    mapGraph.addEdge(weightedEdge(eachNode, node(str(ycoord)+" "+str(xcoord)), distancecounter))
  File "C:/Users/SSZ/Documents/arcade game.py", line 370, in addEdge
    Digraph.addEdge(self, weightededge)
  File "C:/Users/SSZ/Documents/arcade game.py", line 341, in addEdge
    raise ValueError('Node not in graph')
ValueError: Node not in graph

```

The program raised an exception from the implementation of the method for adding edges, indicating the node is not in the graph. To understand the cause of this error, I decided to print the coordinates of all nodes in the graph, along with the coordinates of the nodes that were supposed to be used to create the nodes. The result was that every node that was to be used to create an edge already was in the graph. This led me to the conclusion that the program was creating a new node in line 418 rather than using one which already existed in the graph. Therefore, I modified the program so that it went through all nodes in the graph

and checked which one had the intended coordinates. The new section of the program looked as follows.

```
418.         for thatNode in mapGraph.nodes:
419.             if thatNode.getName() == (str(ycoord)+" "+str(xcoord)):
420.                 mapGraph.addEdge(weightedEdge(eachNode, thatNode, distancecounter))
```

After I called the function for printing all the edges in the graph, the picture below illustrates what I got.

```
4 1->(2)6 1
14 19->(4)18 19
4 5->(14)18 5
14 23->(2)16 23
4 7->(2)6 7
16 1->(2)14 1
4 11->(3)1 11
16 3->(2)18 3
4 13->(3)1 13
16 5->(2)18 5
4 17->(2)6 17
16 7->(2)18 7
4 19->(14)18 19
18 5->(8)10 5
18 5->(2)16 5
18 5->(12)6 5
18 5->(14)4 5
18 5->(4)14 5
18 5->(17)1 5
16 11->(2)14 11
4 23->(2)6 23
16 13->(2)14 13
6 1->(5)1 1
6 1->(2)4 1
16 17->(2)18 17
6 5->(13)18 5
```

This was just a part of the output I got. It was much bigger, but I could see that all edges are displayed and that they were in both directions. The next step was to add horizontal edges. The principle behind that was the same as for the vertical ones, but this time the program had to go to the right instead of downwards. To do that, I added the following code to the function adding the edges, inside the loop going through all nodes.

```

422.         # Now horizontal edge
423.         ycoord = ""
424.         iterator = 0
425.         while eachNode.getName()[iterator] != " ":
426.             # Gets y-coordinate, row number
427.             ycoord += str(eachNode.getName()[iterator])
428.             iterator += 1
429.         # Makes it a number, rather than a string
430.         ycoord = int(ycoord)
431.         # Gets x-coordinate, column number, converts it to a string
432.         xcoord = int(eachNode.getName()[iterator + 1:])
433.         # If there is space to move to the right then there must be a node somewhere to the right
434.         if mapdescription[ycoord][xcoord+1] == ".":
435.             # Will measure the weight of that edge
436.             distancecounter = 0
437.             # Moves while there is no wall on the right
438.             while mapdescription[ycoord][xcoord] != "W":
439.                 # Changes the weight of the edge
440.                 xcoord += 1
441.                 distancecounter += 1
442.             # Accounts for logic in the loop above
443.             xcoord -= 1
444.             distancecounter -= 1
445.             # Adds the edge
446.             for thatNode in mapGraph.nodes:
447.                 if thatNode.getName() == (str(ycoord) + " " + str(xcoord)):
448.                     mapGraph.addEdge(weightedEdge(eachNode, thatNode, distancecounter))

```

However, the result displayed was as follows.

```

C:\Python31\python.exe "C:/Users/SSZ/Documents/arcade game.py"
Traceback (most recent call last):
  File "C:/Users/SSZ/Documents/arcade game.py", line 422, in <module>
    for eachNode in graphconstructor(basicmap).nodes:
  File "C:/Users/SSZ/Documents/arcade game.py", line 410, in graphconstructor
    while mapdescription[ycoord][xcoord] != "W":
IndexError: string index out of range

Process finished with exit code 1

```

The reason for that was the fact that there is a tunnel. Previously the function could use the fact that the map is bounded from above and from below. It is not bounded, however from the sides. To amend the program, it has to change its x-coordinate to zero if it is out of range. The amended section of code looked as displayed below.

```

442.         if xcoord >= len(mapdescription[ycoord]):
443.             xcoord = 0

```

This time the output was intended, all edges were listed, including the one corresponding to the tunnel, as shown by the picture below.

```
10 17->(15)10 7
```

Movement

The graph enables us to control the movement of the enemies in a meaningful way. The first algorithm I implemented was the one which chose the direction of the enemies at random at every node. Each time the program went through the loop it checked if an enemy is at a node. If so, it got the coordinates of the node and the nodes connected to that node with an edge. Afterwards, it randomly chose the destination node and determined the new position by comparing the coordinates of the source and destination node.

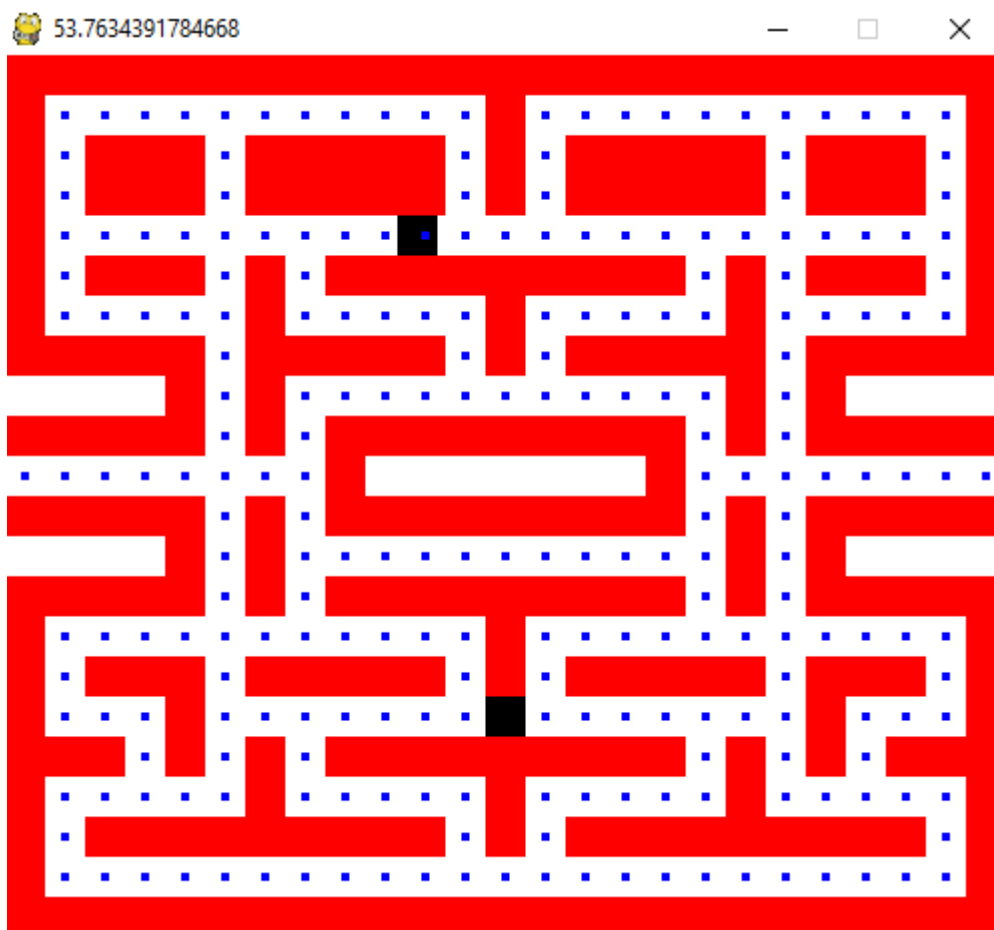
```

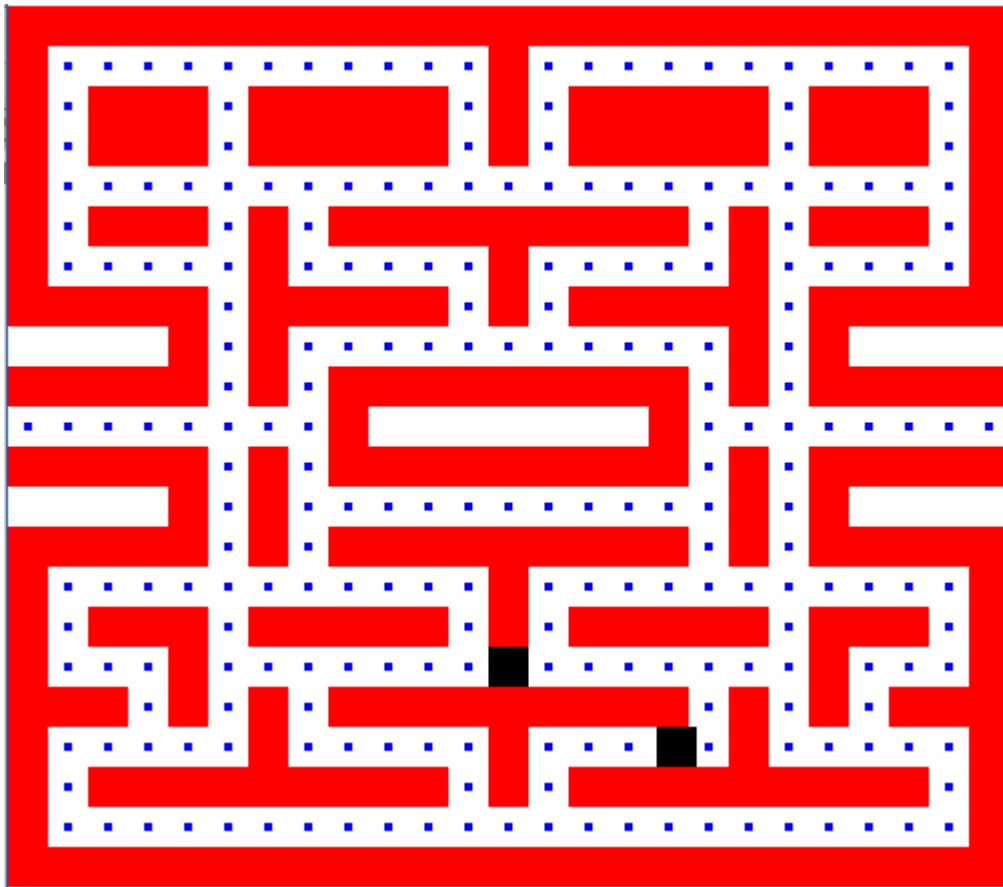
538     # Move the enemies
539     # If it is at a node, change direction randomly
540     # Check which node it is at
541     for eachNode in mapGraph.nodes:
542         # Checks which node it is at and if it is at any node
543         if eachNode.getName() == (str(int(EnA.rect.y/20))+ " "+str(int(EnA.rect.x/20))) and int(EnA.rect.y/20)*20 == EnA.
544             # Creates a list of possible destinations
545             possibleDestinations = []
546             for aNode in mapGraph.childrenOf(eachNode).keys():
547                 possibleDestinations.append(aNode)
548             # Choses the destination node randomly
549             destination_node = random.choice(possibleDestinations)
550             # Gets x and y coordinates of the destination node
551             ycoord = ""
552             iterator = 0
553             while destination_node.getName()[iterator] != " ":
554                 # Gets y-coordinate, row number
555                 ycoord += str(destination_node.getName()[iterator])
556                 iterator += 1
557             xcoord = destination_node.getName()[iterator + 1:]
558             ycoord = int(ycoord)
559             xcoord = int(xcoord)
560
561             # Changes destination accordingly
562             if xcoord*20 > EnA.rect.x:
563                 EnA.new_direction(2)
564             if xcoord*20 < EnA.rect.x:
565                 EnA.new_direction(4)
566             if ycoord*20 > EnA.rect.y:
567                 EnA.new_direction(3)
568             if ycoord*20 < EnA.rect.y:
569                 EnA.new_direction(1)

```

The code above was how the algorithm was implemented.

Testing





On the pictures above it can be seen that the enemy (one of the black squares) moves around the screen.

Exit conditions

There are two situations in which the game should end: either all coins are collected or Pacman collided with an enemy. The player wins if all coins are collected and loses if collides with enemy. Thus, every time the program goes through the main loop it should check these two conditions and if either is met, exit the loop and print appropriate statement on the console. This was done with the following code:

```
574         if pygame.sprite.spritecollide(Pacman, enemies_list, False):
575             print~("Game over")
576             done = True
577
578         if len(coin_list.sprites()) == 0:
579             print("You won")
580             done = True
```

Testing

After testing, the program correctly displays the statement indicating that game is lost or won and the pygame window closes.

First iteration – summary

In the first iteration of the development process the following success criteria were met: 1. – “A map resembling the classic Pacman map is created” and 2a. – “The enemies move in a way depending on the level of difficulty, On the easy level, in random directions”.

In the first of the iteration process the creation of graph – an abstract data structure representing an actual map – was a big success. Regardless of the actual details of the map, the program can create a graph without any programmer’s intervention.

Furthermore, the algorithm which makes Pacman take the next turn possible after a user presses a key is another big success of the first iteration. This algorithm makes the game very easy to play, preventing the program from causing unnecessary frustration that for example the user cannot take a turn when he wants to. This turning algorithm will be tested by the end user so that he can identify whether this algorithm makes turning easy enough for the player.

On the other hand, the implementation of the actual map, coins and enemies does not have a very aesthetic design. The colours look incoherent with each other and the white background creates a screen which is displeasing to look at. The program is created for the end user, so I need to contact him about the design and how he finds it the colours, shapes, etc.

Among the success criteria the following remain uncompleted: finding the shortest path to user’s position at a time when the algorithm is ran, finding the shortest path to Pacman’s predicted future position and saving the top 10 scores on the device, retrieving and amending them.

Consultation with the end user after the first iteration

After consultation with the end user, I got a few points of feedback, which should help me develop my game. Dmitry Rusanov played the game to test how easy it is to turn, whether the colours are appropriate, whether the enemies seem to move in the random directions. The following feedback was given to me:

1. It seems like the enemies are ‘trapped’ in the right bottom corner of the map. This was particularly visible when more than one enemy was in the game.
2. The colours should be as in the original version of Pacman
3. There is a bug involving the tunnel: the Pacman is allowed to turn while it is in the tunnel, making it go out of the map. Moreover, the enemies do not ever use the tunnel.
4. The turning algorithm is working very well. It is very easy to turn making controlling the Pacman extremely user-friendly.

5. The easy level is fun to play, but the higher levels with more intelligent ghosts should be implemented.
6. The score should be displayed on the screen, as well as the number of available lives.

Development – second iteration

Second iteration – introduction

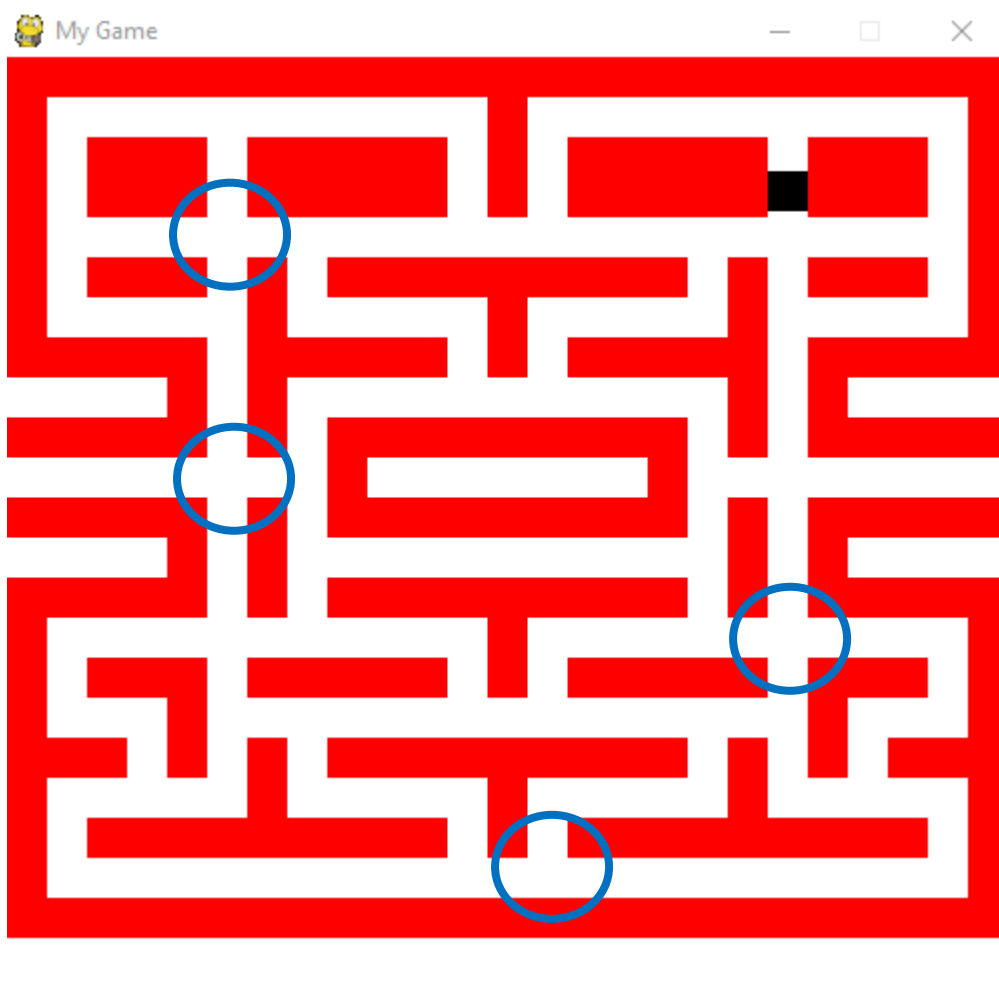
The aim of the second iteration was to create a much more usable prototype of the solution, by reacting to user's feedback and implementing the features he suggested.

Correction of the graph implementation

As the end user pointed out, the enemies seemed to be 'trapped' in the right bottom area of the map. In the first 30 seconds they moved to the area in the right bottom corner and did not leave it in the following minutes. Although they should choose random directions and it is not possible to tell for sure that some of the edges were not implemented, carrying out the test multiple times and for a long time led to the conclusion that most likely the graph was implemented in the wrong way. Below is how the graph was implemented initially, and more precisely, how the edges were implemented.

```
422.         # Now horizontal edge
423.         ycoord = ""
424.         iterator = 0
425.         while eachNode.getName()[iterator] != " ":
426.             # Gets y-coordinate, row number
427.             ycoord += str(eachNode.getName()[iterator])
428.             iterator += 1
429.         # Makes it a number, rather than a string
430.         ycoord = int(ycoord)
431.         # Gets x-coordinate, column number, converts it to a string
432.         xcoord = int(eachNode.getName()[iterator + 1:])
433.         # If there is space to move to the right then there must be a node somewhere to the right
434.         if mapdescription[ycoord][xcoord+1] == ".":
435.             # Will measure the weight of that edge
436.             distancecounter = 0
437.             # Moves while there is no wall on the right
438.             while mapdescription[ycoord][xcoord] != "W":
439.                 # Changes the weight of the edge
440.                 xcoord += 1
441.                 distancecounter += 1
442.             # Accounts for logic in the loop above
443.             xcoord -= 1
444.             distancecounter -= 1
445.             # Adds the edge
446.             for thatNode in mapGraph.nodes:
447.                 if thatNode.getName() == (str(ycoord) + " " + str(xcoord)):
448.                     mapGraph.addEdge(weightedEdge(eachNode, thatNode, distancecounter))
```

The reason why some edges were not added to the graph was that as line 437. indicates, the program moved until it found a wall. However, on the map there are some junctions which did not have any wall next to it, such as the ones in blue circles on the picture below.



These junctions were ignored by my program when adding the edges to the graph, because the program expected a wall at the end of each edge. Therefore, the logic had to be changed so that when adding the edges and moving to identify nodes, the program has to check at each point after moving if it is at a node. This was done with the following code:

```

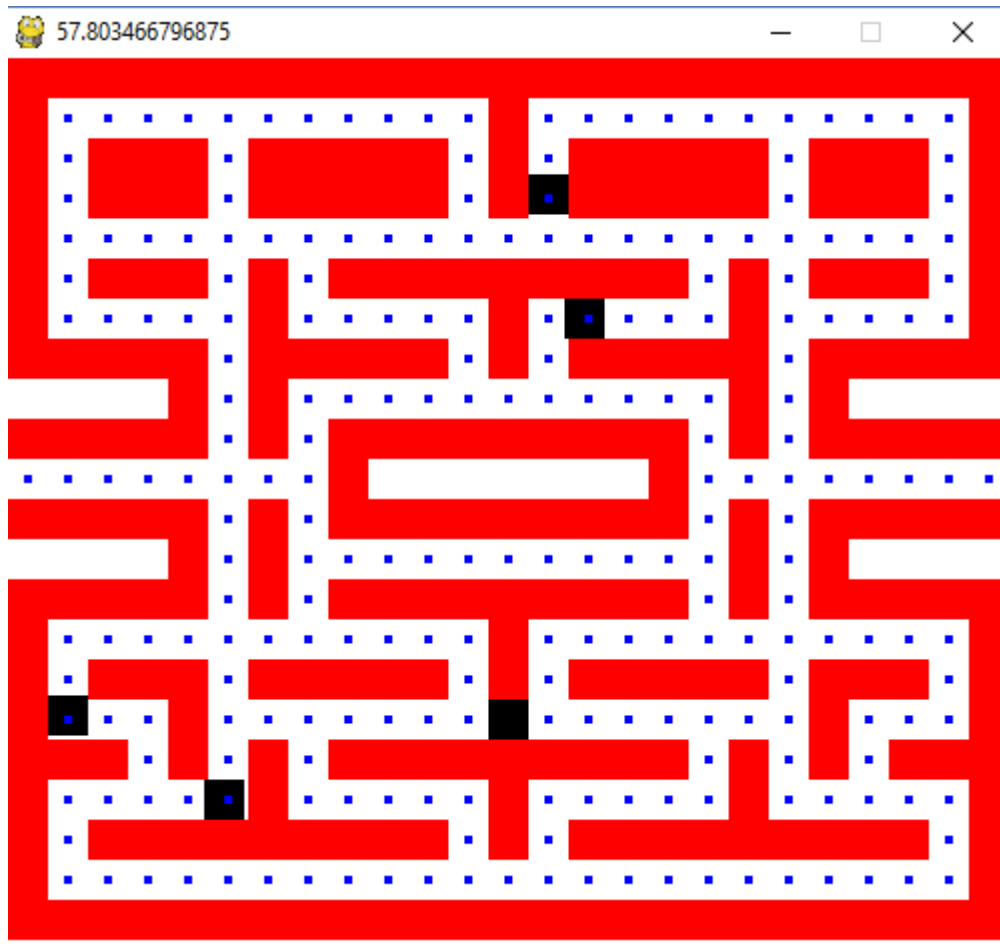
445     # If there is space to move to the right then there must be a node somewhere to the right
446     if mapdescription[ycoord][xcoord+1] == ".":
447         # Will measure the weight of that edge
448         distancecounter = 0
449         # Moves until there is a junction on the right
450         found = False
451         # Moves until it finds the node below
452         while not found:
453             # Changes the weight of the edge and moves down
454             xcoord += 1
455             distancecounter += 1
456             if xcoord == 25:
457                 xcoord = 0
458             # Checks if it is at a node
459             for thatNode in mapGraph.nodes:
460                 if thatNode.getName() == (str(ycoord) + " " + str(xcoord)):
461                     found = True
462             # Adds the edge
463             mapGraph.addEdge(weightedEdge(eachNode, thatNode, distancecounter))

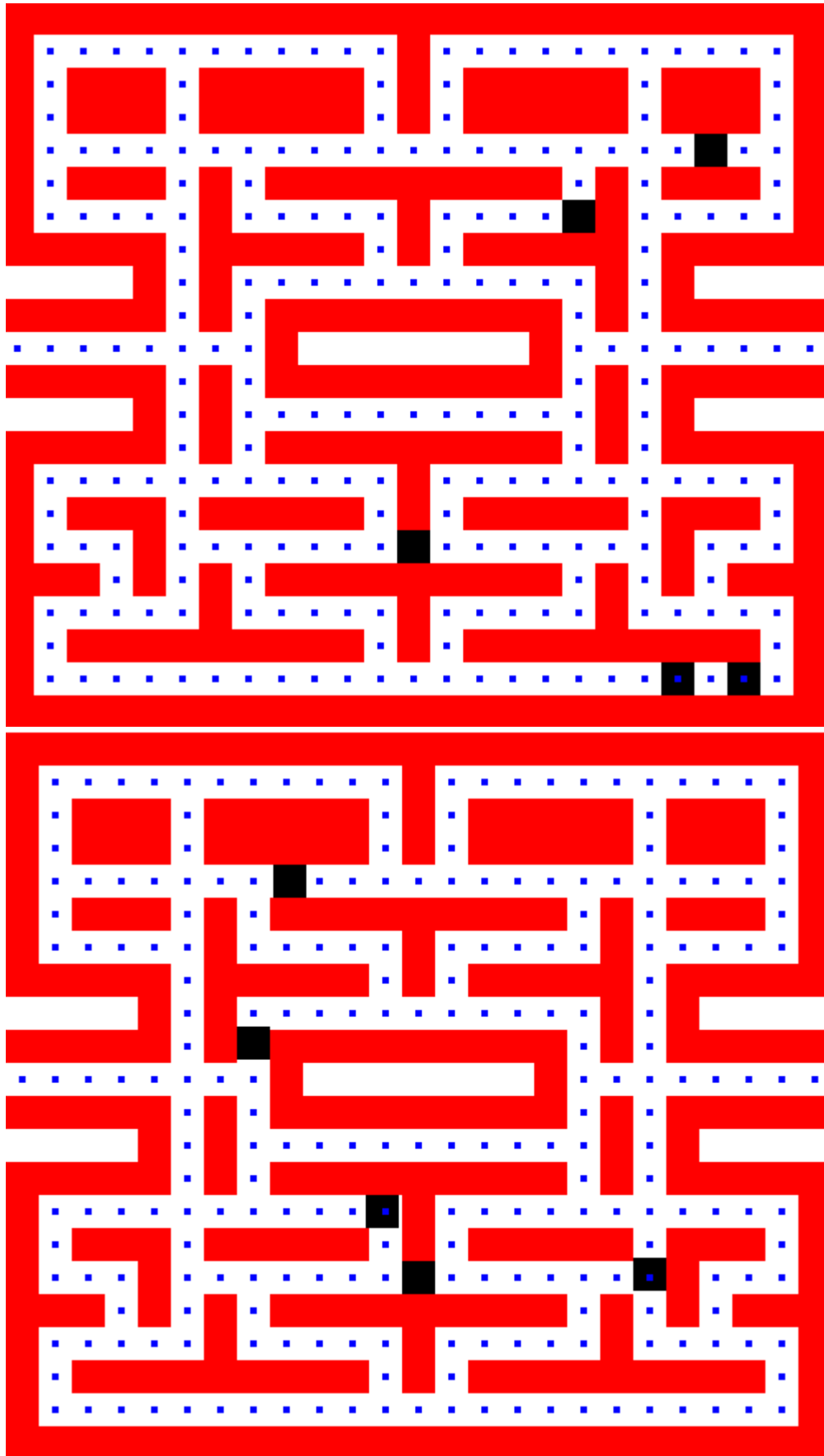
```

Code was amended analogically for the vertical edges. This means the logic of the solution, but instead of moving horizontally, the program looked for nodes vertically. Instead of checking space on the right, it checks the space below a node and looks for the space below, increasing *ycoord*.

Testing

Afterwards, the enemies in the game moved in the following way:

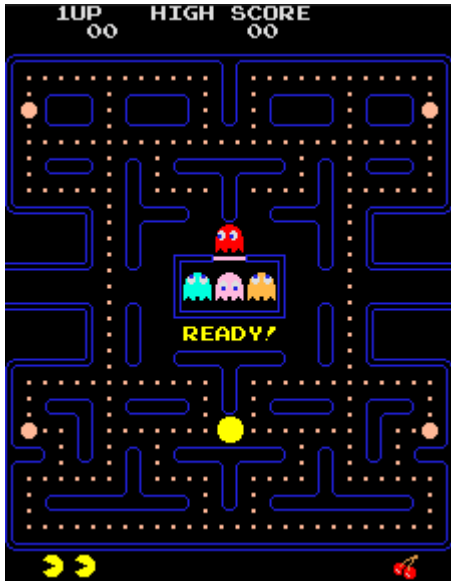




After a few minutes of playing, there seemed to be no clusters where the enemies seemed to be 'trapped'.

Colours and shapes in the game

The end user did not like the colours of the objects in the game. He said they should resemble the ones from the original version of Pacman. Thus, I did further research to identify what the colours in the original version of Pacman are.



From the picture above I came to the conclusions as follows:

1. Pacman has to be yellow
2. Background is black
3. Coins are light salmon
4. One enemy is red, one is cyan, one is light coral and the fourth one is magenta.

To implement them, I did research on how these colours are represented in hexadecimal from the website <http://www.discoveryplayground.com/computer-programming-for-kids/rgb-colors/>, which gave me the hexadecimal values for these colours, which I could use in my code.

```
7      # Define some colors
8      BLACK = (0, 0, 0)
9      WHITE = (255, 255, 255)
10     GREEN = (0, 255, 0)
11     RED = (255, 0, 0)
12     BLUE = (0, 0, 255)
13     YELLOW = (255, 255, 0)
14     LIGHT_SALMON = (255, 160, 122)
15     CYAN = (0, 255, 255)
16     ORANGE = (255, 165, 0)
17     LIGHT_CORAL = (240, 128, 128)
```

I then amended the code written before to change the colours in the game.

Testing

The resulting colour scheme was as follows:



This will be consulted with the user after the end of the second iteration of the development process.

Tunnel – bug

As my end user noticed, the Pacman left the map when an arrow down or up was pressed when going through the tunnel. The Pacman took the next turn possible, which is at the right or left border of the map. As a result, it was moving up or down outside the map, which was a bug making the user confused and helpless. Thus, I had to change the way Pacman was going through the tunnel. In the original version, it checked if it was too far left or right and set it to the new one. However, this was not working because the ‘teleportation’ happened one cycle too late for the tunnelling to work properly in all cases. Therefore, a change had to be made to the position of Pacman as soon as it was outside the map. I did it using the modulo function in python. The code changing the position of Pacman every time the program when through the loop was amended so that the modulo function was performed on the x-coordinate every time it was changed. The following code was used for that reason.

```

151         # moves the object
152         def move(self):
153             # moves upwards
154             if self.direction == 1:
155                 self.rect.y -= self.speed
156             # moves right
157             if self.direction == 2:
158                 self.rect.x += self.speed
159                 # Enables going through the tunnel
160                 self.rect.x = self.rect.x % width
161             # moves down
162             if self.direction == 3:
163                 self.rect.y += self.speed
164             # moves left
165             if self.direction == 4:
166                 self.rect.x -= self.speed
167                 # Enables going through the tunnel
168                 self.rect.x = self.rect.x % width

```

Testing

As a result, regardless what was pressed when the Pacman was in the tunnel it went through it successfully.

Score and lives

The end user notified me he would like the score to be displayed below the map, as well as the number of available lives in form of red hearts. In order to display the score below the map, I had to first initialise the font at the beginning of the program to avoid 'Font not initialised' error.

```

10         # initialize font; must be called after 'pygame.init()' to avoid 'Font not Initialized' error
11         myfont = pygame.font.SysFont("monospace", 15)

```

Next, I had to initialise the label which would be displayed.

```

491         # render text for displaying the score
492         label = myfont.render("Score: "+str(score), 1, (255,255,0))

```

Finally, every time the program goes through the loop the label has to be updated, because the score changes.

```

636         # update text for displaying the score
637         label = myfont.render("Score: " + str(score), 1, (255, 255, 0))

```

Finally, I had to draw the label.

```

642         # --- Drawing
643         all_sprites_list.draw(screen)
644         screen.blit(label, (10, 450))

```

Testing

As a result, a line was displayed below the map indicating the score.



Development - continued

The next step was to have icons of Pacman displayed on the right hand side below the map indicating how many lives there are left. The Pacman starts with three lives, 2 icons. Every time the Pacman collides with enemies one of the icons should disappear, the Pacman should be transported to initial position and so should enemies. The program should wait for a few second before enabling movement.

First, I downloaded the image of the icon of Pacman and placed it in the same folder as the arcade game.



Next, I loaded that image to the program.

```
43     # Load the image of Pacman
44     PacmanImage = pygame.image.load("Pacman.png")
```

I set the initial number of available lives to 2.

```
290     available_lives = 2
```

Next, when the Pacman collided with an enemy, the number of available lives was decreased by one. The positions of enemies and Pacman were set to initial ones.

```

632         # If Pacman collides with an enemy, it loses a life and enemies and Pacman are reset
633         if pygame.sprite.spritecollide(Pacman, enemies_list, False):
634             available_lives -= 1
635             # Reset position of Pacman
636             Pacman.rect.x = initxPac
637             Pacman.rect.y = inityPac
638             # Reset the position of enemies
639             EnA.rect.x = initxEnA
640             EnA.rect.y = inityEnA
641             EnB.rect.x = initxEnB
642             EnB.rect.y = inityEnB
643             EnC.rect.x = initxEnC
644             EnC.rect.y = inityEnC
645             EnD.rect.x = initxEnD
646             EnD.rect.y = inityEnD
647             # Indicate the game has just been reset
648             justStarted = True

```

I added a new variable, justStarted, indicating whether the positions were just restarted so that the program knows when it has to wait for a few seconds. If the number of available lives has gone below zero, the program should end, displaying “Game over”. If the program has just been restarted, it should be put to sleep for a couple of seconds. To do so, the following code was implemented.

```

650         # If there are less than 0 available lives, game is over
651         if available_lives < 0:
652             print("Game over")
653             done = True

```

```

684         # Wait if the game has just started or has just been reset
685         if justStarted:
686             # Wait 3 seconds before the game goes live
687             time.sleep(1)
688             justStarted = False

```

The correct number of icons was displayed with the following code:

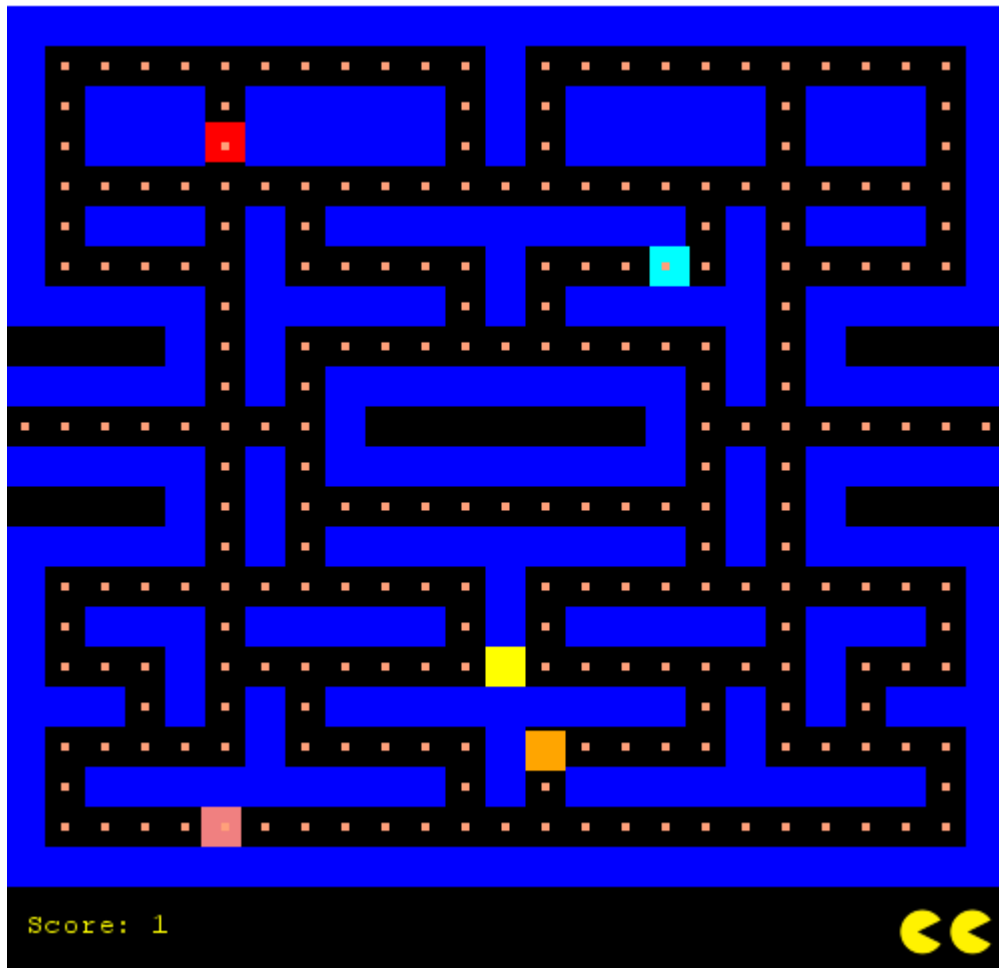
```

675         # Drawing the score on the left side below the map
676         screen.blit(label, (10, 450))
677         # Drawing as many icons of Pacman on the right side as many lives are available
678         for i in range(0, available_lives):
679             screen.blit(PacmanImage, (470 - 25*i, 450))

```

Testing

The following screens were observed in the game. The program was frozen for a few seconds after the Pacman has collided with the enemy and the positions were restarted.





Depth First Search

At the beginning, I was going to use A* algorithm and find the shortest path to the enemy. After the implementation of the graph I understood it is much more complex than predicted. Moreover, the user did not require finding the shortest path, so I decided to use a much simpler search in a graph – Depth First Search – which would create a path from an enemy to Pacman. This path may not be the shortest one, but the end user did not require the shortest path, any intelligent behaviour would be sufficient for him. Moreover, the game is still playable and all success criteria would be fulfilled if a DFS was used instead of A* algorithm. Therefore, I made a change to what was decided in the design stage and used a Depth First Algorithm instead of shortest path algorithms such as A* or Dijkstra's algorithm. I could have used a Breadth First Search, but DFS finds a path faster than BFS. It may not be the shortest path, but it is found quicker, which is crucial for the game to run smoothly. Moreover, in mazes DFS is known to be more efficient than BFS. The map can be treated like a maze, so using the DFS is more appropriate this case.

On the medium level, the Pacman is being followed by the enemies. They find a path from their current position to the position of Pacman and follow that path. If they don't catch

Pacman at that point, they call that algorithm again and follow that path. In order to find the path, Depth First Search algorithm is used.

The program registers the last node at which Pacman was by checking at each point whether it is at a node and updating a variable `last_visited` if it is. Every time the program goes through the loop, it checks for each enemy if it is at a node. If it is, it finds a path from that node to the last node Pacman visited.

The code below registers the last node at which Pacman was.

```
611     lastNodePacman = node("nameInitial")
612     # Note the last node visited by Pacman
613     for eachNode in mapGraph.nodes:
614         if eachNode.getName() == (str(int(Pacman.rect.y / 20)) + " " + str(int(Pacman.rect.x / 20))) and int(
615             Pacman.rect.y / 20) * 20 == Pacman.rect.y and int(Pacman.rect.x / 20) * 20 == Pacman.rect.x:
616             lastNodePacman = eachNode
```

The Depth First Search works in the way described in the design section. The following code is used to implement it using the recursive approach, because it is easier to code. The memory taken by the call stack is not an issue, so the approach does not impose any restrictions on the functionality of the game.

```
481     # DFS
482     def DepthFirstSearch(start_node, goal_node, aGraph, path_stack, visited_nodes):
483         # Adds the node to visited nodes
484         visited_nodes.append(start_node)
485         # If found, returns the path
486         if start_node == goal_node:
487             # After adding the current node to the path stack
488             path_stack.append(start_node)
489             return path_stack
490         # Executes the algorithm on each child
491         for child in aGraph.childrenOf(start_node).keys():
492             # Does not traverse if already visited
493             if child not in visited_nodes:
494                 # Appends the child and calls the algorithm recursively
495                 path_stack.append(start_node)
496                 p = DepthFirstSearch(child, goal_node, aGraph, path_stack, visited_nodes)
497                 if p:
498                     return p
499     return ""
```

To test whether the algorithm is working I used the following lines:

```
628     # Testing DFS
629     if lastNodePacman in mapGraph.nodes:
630         print(DepthFirstSearch(eachNode, lastNodePacman, mapGraph, [], []))
```

Which was written inside the loop for choosing the path for enemies under the conditional statement which checked if the enemy is at a node.

Testing

```
[<__main__.node object at 0x0573CB90>, <__main__.node object at 0x0573C910>, <__main__.node object at 0x0573C8D0>, <__main__.node
[<__main__.node object at 0x05739E70>, <__main__.node object at 0x0573C090>, <__main__.node object at 0x0573C290>, <__main__.node
[<__main__.node object at 0x05739DB0>, <__main__.node object at 0x05739F30>, <__main__.node object at 0x05739F70>, <__main__.node
[<__main__.node object at 0x0573C090>, <__main__.node object at 0x0573C290>, <__main__.node object at 0x0573C4D0>, <__main__.node
```

The program printed a list of nodes from the position of the enemy to the last registered position of Pacman.

Enemies using DFS

The next step was to use the list produced by Depth First Search for the movement of ghosts. To do so, there will be a dictionary storing the path which each enemy should follow. When a path for an enemy is empty, the program uses Depth First Search from that position to Pacman's last visited node to get a new path, which the enemy then follows. The path for each enemy is store in the dictionary. This is executed if variable *level* is 2, that is on the middle level of difficulty.

```
659 # Follows a path to the last registered node of Pacman
660 if level == 2:
661     # Finding the path using DFS
662     if lastNodePacman in mapGraph.nodes and len(list_nodes_to_Pacman[every_enemy]) == 0:
663         list_nodes_to_Pacman[every_enemy] = DepthFirstSearch(eachNode, lastNodePacman, mapGraph, [], [])
664     if len(list_nodes_to_Pacman[every_enemy]) > 0:
665         next_node = list_nodes_to_Pacman[every_enemy].pop(0)
666         xcoord = get_coordinates_from_name(next_node)[0]
667         ycoord = get_coordinates_from_name(next_node)[1]
```

The code above is what executed the description provided. If the level is not the easiest one, the enemy follows the path calculated earlier, unless the list for storing the path is empty. If it is empty, the program, uses the implemented Depth First Search to find a path from the position of enemy to the last node visited by Pacman.

Second iteration – summary

In the second iteration the following success criterion was completed: 2b. – Enemies find a path to Pacman's position at a time when the algorithm is run. Moreover, the following requirements are fulfilled: the score is recorded and displayed below the map, the number of available lives is displayed below the map in a form of Pacman's icons, Pacman can go through the tunnel (without bugs), the colour scheme is as in original version of Pacman.

In this iteration, I built up on the first iteration by using the implemented graph in order to fulfil further requirements. I used the weighted graph and Depth First Search algorithm to find a path from the enemy to Pacman. That was a major milestone in the project, because now the enemies have elements of intelligence and reaction to user's behaviour. This was the major assumption and reason for doing this project and it can now already be seen as successful because that key assumption has been fulfilled.

In the next iteration the rest of minor requirements have to be fulfilled, as well as the success criterion which includes storing top ten scores even when the game is closed.

Second iteration – consultation with the user

After the second iteration, I consulted my end user to see what he thinks about the game. His feedback was mostly positive, apart from a few indications he gave me with respect to the third iteration. Dmitry was very happy with the response to his feedback after the first iteration. He appreciated the fact that the movement through the tunnel does not have the

noticed bug anymore. He liked the colour scheme and suggested that no further changes are done with respect to colours and the design of the screen. He liked the fact that the score is displayed below the map, and that the number of available lives are displayed in form of Pacman icons.

Dmitry thought several elements of the game should be improved or created in the next iteration:

1. There should be a start screen and an end screen. On the start screen he would like to have controls explained as well as an option to choose the level of difficulty. It is crucial that when running the program, the game does not start straight away, but there is an intermediate screen beforehand. Similarly, after the game ends, the program should not close, but display a screen with top 10 scores instead.
2. Clicking on the screen to choose the level. Currently the level is indicated with a variable in the program and can be changed only in the code. Dmitry indicated that he would like to have the option of changing it on the start screen by clicking on an appropriate box on the start screen.
3. Top ten scores should be stored and retrieved every time the program is started. They should be displayed on the end screen after the game has finished.

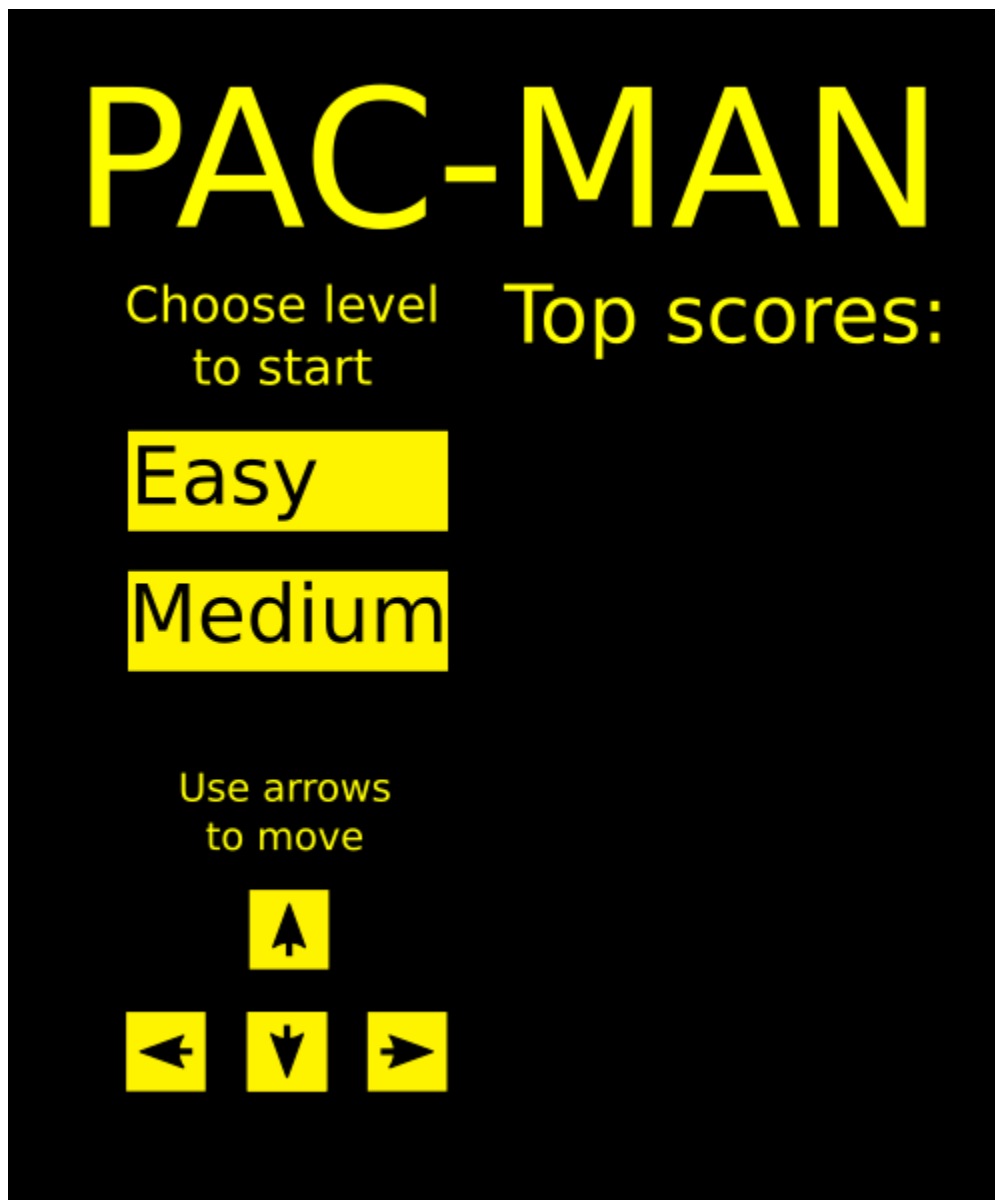
Development – third iteration

Third iteration – introduction

In the third introduction I am going to finalise my game by implementing screens and navigation between them, for the start of the game, game mode and final screen. Moreover, I am going to respond to user's feedback from the previous iterations.

Starting screen and its functionality

I decided to create the starting screen as an image in Inkscape, which would then be displayed. I decided to do that instead of creating the screen from sprites in pygame, because it is much easier to visualise the placement of elements in Inkscape. Moreover, Inkscape enables drawing of more complex elements, such as arrows. I created the starting screen with the placement of elements as in the design section. On top there is a text "Pac-Man", below in the left there is a choice of levels, in the bottom left corner there are arrows and text "use arrows to move". On the right side below the text "Pac-Man" there is a piece of text "Top scores". The top 10 scores will be displayed below using the program. As a result, the starting screen looks like this:



This screen has to be displayed if the game at a given point is in starting mode. To differentiate which screen should be displayed, I introduced a variable *screen_state*, which will get values of 0, 1, 2 or 3 depending on what state the game is in. Next, I divided the main program loop into two parts: one that is executed if screen should be in game state, and one that is executed if the screen is in starting state. Thus, before the piece of code in the main program loop I introduced the following lines.

```
623         # The following happens if the screen is in game state
624         if screen_state == 1:
625             # --- Main event loop
```

Next, I wrote the code which should be executed if the game is in start screen mode. For the start, I wanted to display the image of the starting screen. To do so, I loaded the image to the program using the following line.

```
46         # Load the image of starting screen
47         StartingScreen = pygame.image.load("starting_screen.png")
```


Then, in the loop executed continuously by the program, I added the statement which checked if the program should be in starting screen mode. Then I filled the screen with black for the start to then display the image of the starting screen. I did this using the code below.

```
594         # The following happens if the game is at start screen
595         if screen_state == 0:
596
597             # screen cleared to black
598             screen.fill(BLACK)
599             screen.blit(StartingScreen, 0, 0)
```

However, the program returned the following error.

```
C:\Python34\python.exe "C:/Users/Stanislaw/Documents/arcade game.py"
Traceback (most recent call last):
  File "C:/Users/Stanislaw/Documents/arcade game.py", line 599, in <module>
    screen.blit(StartingScreen, 0, 0)
TypeError: invalid destination position for blit

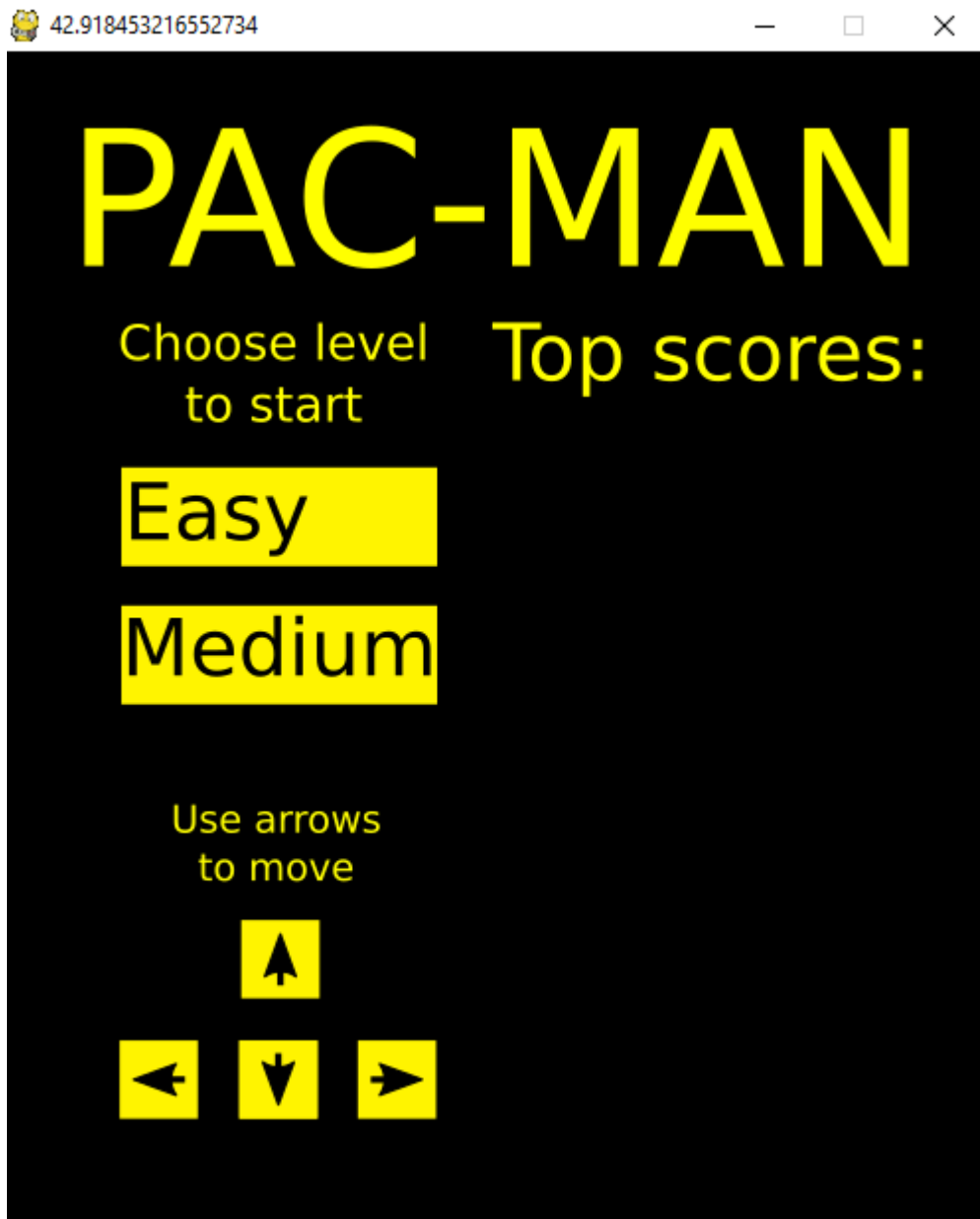
Process finished with exit code 1
```

After a bit of research in the internet to identify the reason for the error, I learned that in the function `screen.blit()` the position has to be passed as a tuple.

```
602         # The following happens if the game is at start screen
603         if screen_state == 0:
604
605             pygame.event.get()
606             # screen cleared to black
607             screen.fill(BLACK)
608             # Displays the image for the starting screen
609             screen.blit(StartingScreen, (0, 0))
```

Testing

The code above displayed the screen as intended.



However, after implementing that part, the program could not be closed by clicking the cross in the right top corner. The reason was that I included checking the events in the conditional statement checking if the program is in game mode. The program was not checking the events if it was in starting screen mode. Hence, I copied the part checking the events and quitting the program if it is closed to the conditional statement for the screen being in starting mode.

```
602         # The following happens if the game is at start screen
603         if screen_state == 0:
604             for event in pygame.event.get():
605                 if event.type == pygame.QUIT:
606                     done = True
```

Now the program could be successfully closed.

Development - continued

The next piece of functionality I had to add was being able to choose the level. In order to do so, the program needs to check if the mouse cursor is over the block for easy or medium, choose the appropriate level and start the game if the mouse is pressed. To check if the mouse is over one of the blocks, I created a new function *inarea* which checks if a point is within a block.

```
518 def inarea(x_coor_pic, y_coor_pic, wid_pic, he_pic, x_mous, y_mous):
519     result = True
520     if x_mous < x_coor_pic or x_mous > (x_coor_pic + wid_pic):
521         result = False
522     if y_mous > (y_coor_pic) or y_mous < (y_coor_pic - he_pic):
523         result = False
524     return result
```

In the part of the program loop executed for starting screen, I checked if the mouse is over one of these blocks using the function above and coordinates of the blocks from Inkscape and changed the variable *level* accordingly. If the mouse was pressed while being over one of these blocks, the *screen_state* variable was set to value for the game screen. To execute the logic described, I used the functions provided by pygame for checking the position of the mouse.

```
616 # If it is over the block with "easy", sets level to 1
617 if inarea(60, 339, 160, 50, pygame.mouse.get_pos()[0], pygame.mouse.get_pos()[1]):
618     level = 1
619     # And starts if the left mouse button is pressed
620     if pygame.mouse.get_pressed()[0]:
621         screen_state = 1
622 # If it is over the block with "medium", sets level to 2
623 if inarea(60, 269, 160, 50, pygame.mouse.get_pos()[0], pygame.mouse.get_pos()[1]):
624     level = 2
625     # And starts the game if left mouse button is pressed
626     if pygame.mouse.get_pressed()[0]:
627         screen_state = 1
```

Testing

However, when I ran the program, it did not start even if pressed on a box. After white-box testing, I realised the coordinates provided by Inkscape are from the left bottom, while pygame counts them from left top corner.

Development – continued

Thus, I modified the function *inarea* to the following.

```
518 def inarea(x_coor_pic, y_coor_pic, wid_pic, he_pic, x_mous, y_mous):
519     result = True
520     if x_mous < x_coor_pic or x_mous > (x_coor_pic + wid_pic):
521         result = False
522     if y_mous > (600 - y_coor_pic) or y_mous < (600 - y_coor_pic - he_pic):
523         result = False
524     return result
```

Testing

Afterwards, I tested if that functionality is working. When I ran the program, the screen was as following.

PAC-MAN

Choose level
to start

Top scores:

Easy

Medium

Use arrows
to move



When I clicked on easy, it changed to the screen below.



When I clicked on the cross in the right top corner, the program terminated. When clicked on the button “Medium”, the game screen was as following.



The screens above suggest that the functionality of the start screen is as intended, apart from the top scores being displayed, but that will be implemented later in the iteration.

Final screen

The final screen should be displayed after the game is lost or won. I created two different images, one in case the game is won and the second in the case the game is lost. I used Inkscape to create them and they are displayed below.

GAME OVER

You lost

Top scores:

[Press Esc to return to start screen]

SUCCESS!

You won

Top scores:

[Press Esc to return to start screen]

In order to display them, the *screen_state* has to be set to 2 when the number of available lives is smaller than 0 or 3 when all coins are collected. In the main program loop, two additional conditional statement were added to be executed in these cases. They are very similar, apart from the image that has to be displayed.


```

779     # The following is executed if the game is lost
780     if screen_state == 2:
781
782         for event in pygame.event.get():
783             if event.type == pygame.QUIT:
784                 done = True
785             if event.type == pygame.KEYDOWN:
786                 if event.key == pygame.K_ESCAPE:
787                     screen_state = 0
788         screen.fill(BLACK)
789         screen.blit(GameOverScreen, (0, 0))
790
791     # The following is executed is the game is won
792     if screen_state == 3:
793
794         for event in pygame.event.get():
795             if event.type == pygame.QUIT:
796                 done = True
797             if event.type == pygame.KEYDOWN:
798                 if event.key == pygame.K_ESCAPE:
799                     screen_state = 0
800         screen.fill(BLACK)
801         screen.blit(YouWonScreen, (0, 0))

```

The images were loaded using the following commands:

```

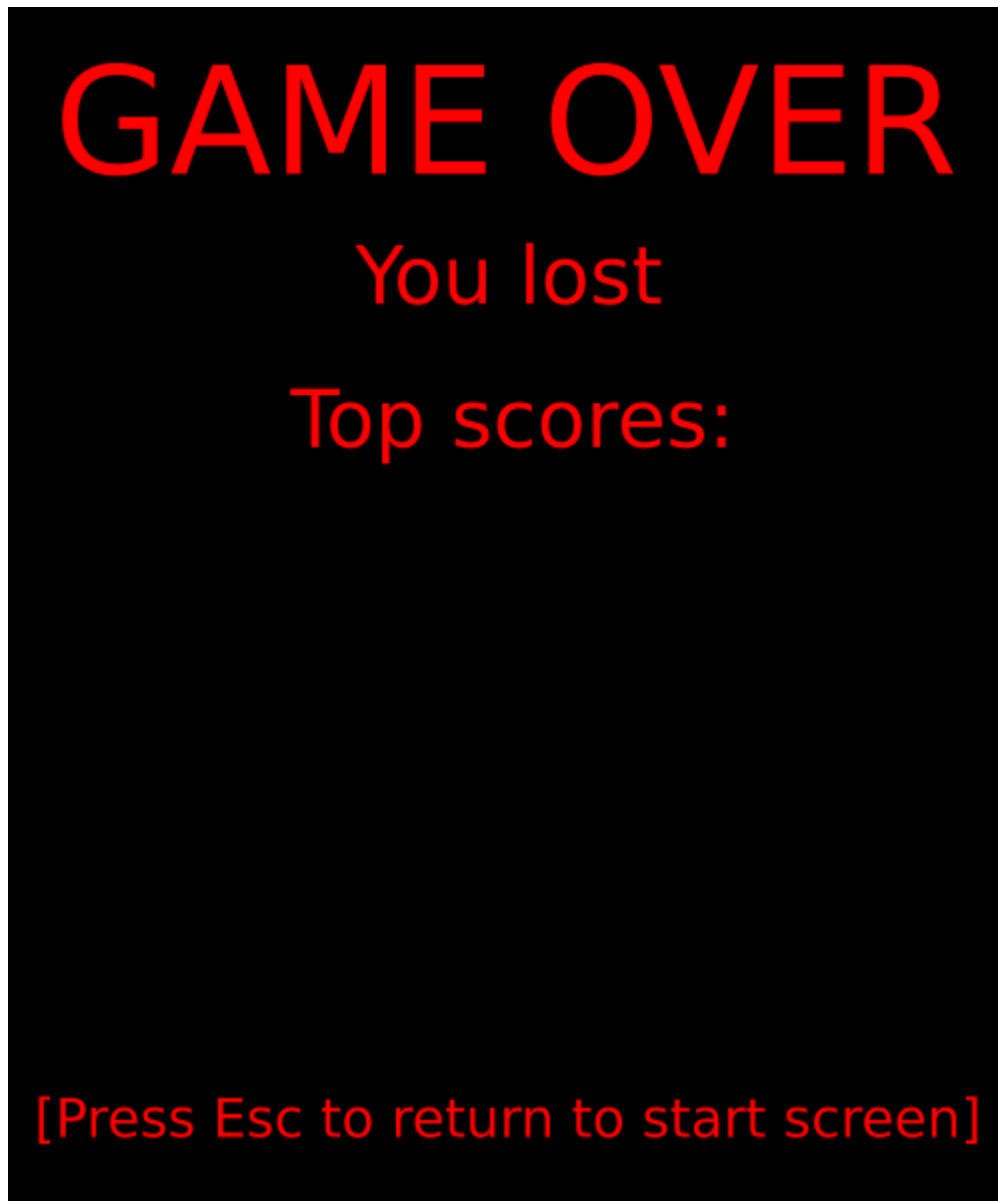
49     # Load the image of the screen when the game is over
50     GameOverScreen = pygame.image.load("game_over.png")
51
52     # Load the image of the screen when the game is won
53     YouWonScreen = pygame.image.load("you_won.png")
54

```

In order to come back to the start screen, when the game is in post-game mode, it checks if the user presses escape and changes screen state to the pre-game state if it is.

Testing

I immediately tested the working of this program, and when I lost the game it displayed the following image.



When I won, it displayed the image below.

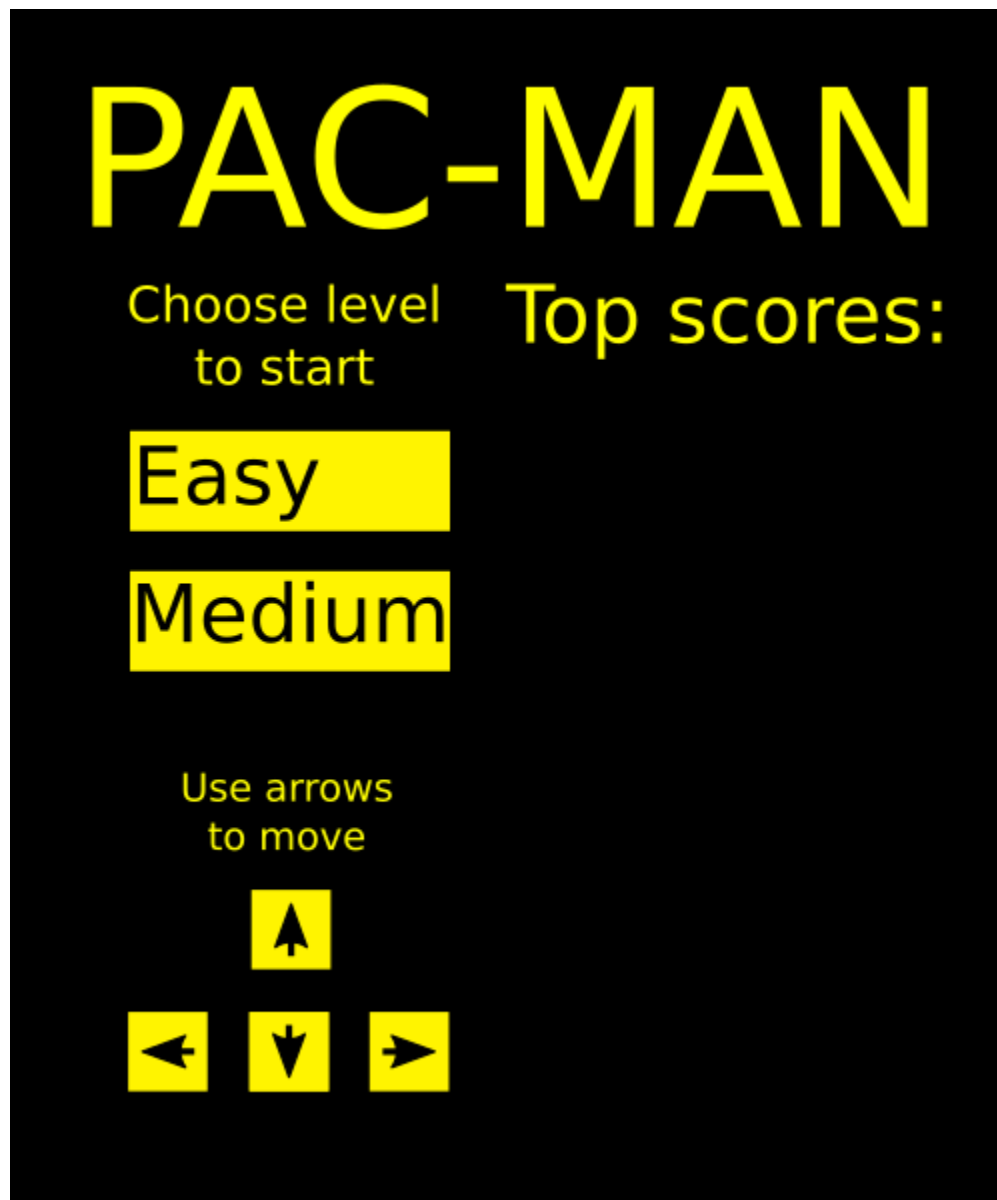
SUCCESS!

You won

Top scores:

[Press Esc to return to start screen]

When I pressed Escape, the screen displayed the screen below.



This indicated that the functionality of the final screen is as intended. However, when I pressed the button “Easy” after losing the game, it immediately displayed the image of the game over screen. I realised that I did not reset the game after the game finished. Not only did I not reset the number of available lives, but also the coins displayed and the positions of Pacman and enemies.

Development - continued

Therefore, I added a piece of code under a conditional statement which checked if the game was finished, which had to set the values of all relevant parameters to the initial ones. I identified the following variables as the ones that have to be reset: coordinates of enemies and Pacman, coins, score and name of available lives. All of them can be directly modified to be the same as the initial values. To reset all coins, I decided the easiest way would be to clear all the sprites in lists of walls and coins and add all of them again, as it is done when the game is started. I also had to empty the `all_sprites_list`, and add enemies to it. This was executed by the code below.

```

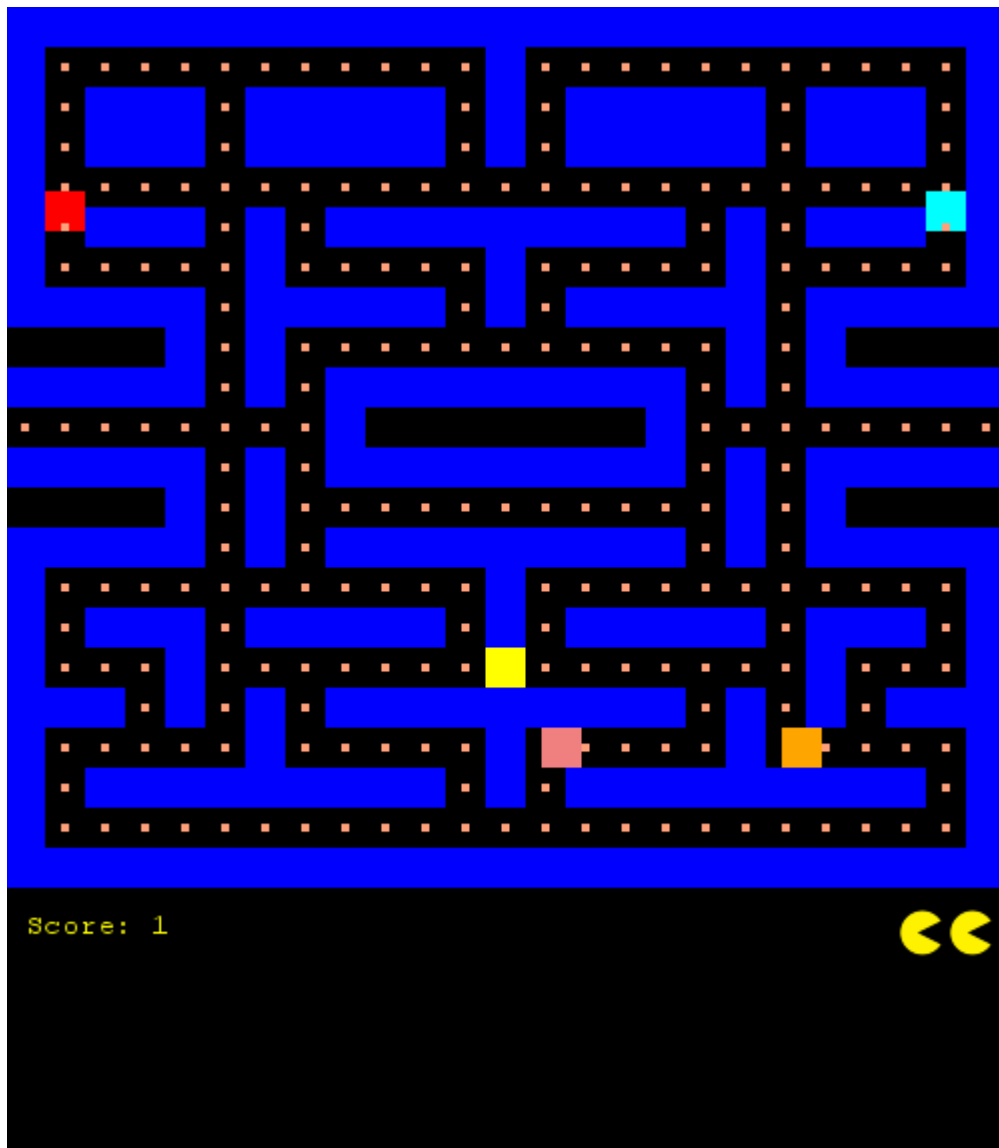
780         # If the game is lost or won, it is reset to initial state
781         if screen_state == 2 or screen_state == 3:
782             # Set last_score to score in order to use it for the top 10 scores
783             last_score = score
784             # Reset Pacman's position
785             Pacman.rect.x = initxPac
786             Pacman.rect.y = inityPac
787             # Reset the position of enemies
788             EnA.rect.x = initxEnA
789             EnA.rect.y = inityEnA
790             EnB.rect.x = initxEnB
791             EnB.rect.y = inityEnB
792             EnC.rect.x = initxEnC
793             EnC.rect.y = inityEnC
794             EnD.rect.x = initxEnD
795             EnD.rect.y = inityEnD
796             # Reset the lives available and the score
797             available_lives = 2
798             score = 0
799             # Empty the lists of coins and walls
800             wall_list.empty()
801             coin_list.empty()
802             all_sprites_list.empty()
803             # And fill them in again
804             adding_map(basicmap, heightofwalls, widthofwalls, colorofwalls, colorofcoins)
805             for enemy in enemies_list:
806                 all_sprites_list.add(enemy)

```

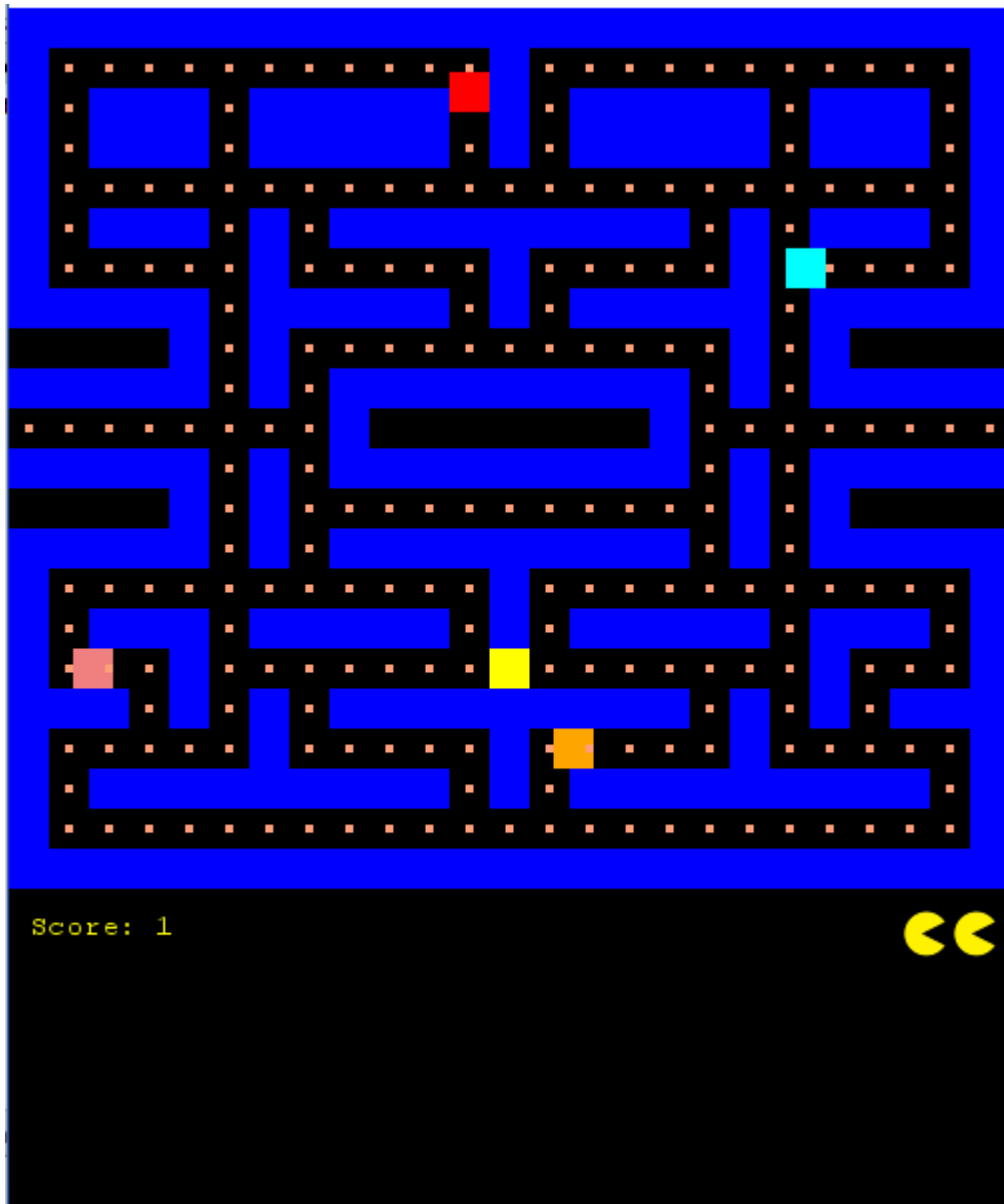
Testing

I tested the working of the code above by playing the game, either winning or losing, going back to the start screen and starting it again.

This is the screen when the game was started for the first time:



And this is the screen when the game was started for the first time.



These two display all the walls, lives available, score and coins correctly. This suggests that the functionality of the final screen is as intended.

Top scores

The final piece of functionality was storing the top 10 scores and storing them in a durable way, so that they can be retrieved once the program is closed. Moreover, that list has to be displayed. To start with, I created a function which would amend the list of high score when a new score is achieved. It checks all the values in the list and checks if the new score achieved should be added to the list. If so, it swaps the value in the list and the new scores, and proceeds to the next element in the list. This way, the elements should be shifted correctly if a new score is placed in the middle of top 10 scores. The function was written as follows.

```

522     # Function for updating the list of top 10 scores
523     def newhighscores(highscorelist, newscore):
524         for i in range(0, len(highscorelist)):
525             # Checks if the new score is bigger
526             if highscorelist[i] < newscore:
527                 # And swaps them if neccessary
528                 temp = highscorelist[i]
529                 highscorelist[i] = newscore
530                 newscore = temp
531         return highscorelist

```

The working of the function was tested by creating a list, initially with all ten places containing 0. The function was run inside the program using the code below.

```

827     # Check if the new score should be on the leaderboard
828     list_of_high_scores = newhighscores(list_of_high_scores, last_score)

```

Testing

Every time the game was won or lost, the content of the list was printed. I achieved three scores: 73, 38, 237, in that order. The program returned the following three lists: [73, 0, 0, 0, 0, 0, 0, 0, 0, 0], [73, 38, 0, 0, 0, 0, 0, 0, 0, 0], [237, 73, 38, 0, 0, 0, 0, 0, 0, 0]. The program correctly placed the scores in the list.

Development - continued

The next step was to have the list in an external text file which would be modified by the program. I created a .txt file called *high_scores* which contained ten 0s, one below another. When the program is run, it has to create a list of high scores based on the content of the text file. This is done using the functions provided by Python for manipulating files. The program opens the file in reading mode, splits the lines and converts the content to integers with the code pasted below.

```

618     # Gets a list of high scores from the file
619     with open('high_scores.txt') as f:
620         list_of_high_scores = f.read().splitlines()
621         # Convert them to integers
622         for i in range(0, len(list_of_high_scores)):
623             list_of_high_scores[i] = int(list_of_high_scores[i])

```

Testing

To verify if this code is working I printed that list and got a list of ten 0s, indicating the code is working as indicated. The list was modified using the function described above. Before the program was closed, it opened the file in the write mode and wrote the content of the list to the file, after converting the contents of the list to strings.

```

868     high_scores_file = open('high_scores.txt', 'w')
869     for i in range(0, len(list_of_high_scores)):
870         high_scores_file.write(str(list_of_high_scores[i]))
871     high_scores_file.close()

```

However, after playing a couple of times and closing the game, the content of the file was "163243432000000". This was an obvious bug, which was caused by the fact I did not separate the lines when writing to the program. The code was amended to look as below.


```

868     high_scores_file = open('high_scores.txt', 'w')
869     for i in range(0, len(list_of_high_scores)):
870         high_scores_file.write(str(list_of_high_scores[i])+'\n')
871     high_scores_file.close()

```

I tested the program by playing a couple of times, closing the game, starting it again, and repeating this cycle a couple of times. Every time I closed the game I checked the content of the file with the high scores, and it was as intended, indicating the code was working properly.

Development - continued

The last piece of functionality related to high scores was printing them on the start screen and the end screens. To do so, I used *screen.blit()* function which printed individual labels on the screen. I started with rendering a font in the program so that it is the same as in the picture created in Inkscape.

```

12     # initialize font; must be called after 'pygame.init()' to avoid 'Font not Initialized' error
13     myfont = pygame.font.SysFont("monospace", 15)
14     font_scores = pygame.font.SysFont("sans-serif", 30)

```

Next, I created a function for printing the high scores on the end screen. It went through the list and printed the content of each cell on the screen. For the end screen I needed two columns and one column for the start screen. The placing of scores on these two types of screens was different, hence two different functions. I checked the coordinates where the scores should be placed in Inkscape. The coordinates of the top score had to be (330, 200). I decided to space each score by 40 pixels, so the coordinates of an i^{th} score had to be (330, $200+40*i$). I used the function screen blit, to produce a complete function for printing the scores on the start screen, with the code pasted below.

```

634     def print_high_scores_start(list_of_high_scores, color):
635         for i in range(0, 10):
636             score_to_print = font_scores.render(str(str(list_of_high_scores[i])), 1, color)
637             screen.blit(score_to_print, (330, 200 + 40*i))

```

Printing the scores for the end screen was a bit more complex, because they did not have the same x-coordinate and the y-coordinate was the same for pairs of scores. To find the y-coordinate I used the modulo function to give the same coordinate for 1st and 6th, 2nd and 7th, etc. scores. To get x-coordinate, I tried passing $120 + (i/5)$ as x-coordinate of the i^{th} element. This was executed by the code below.

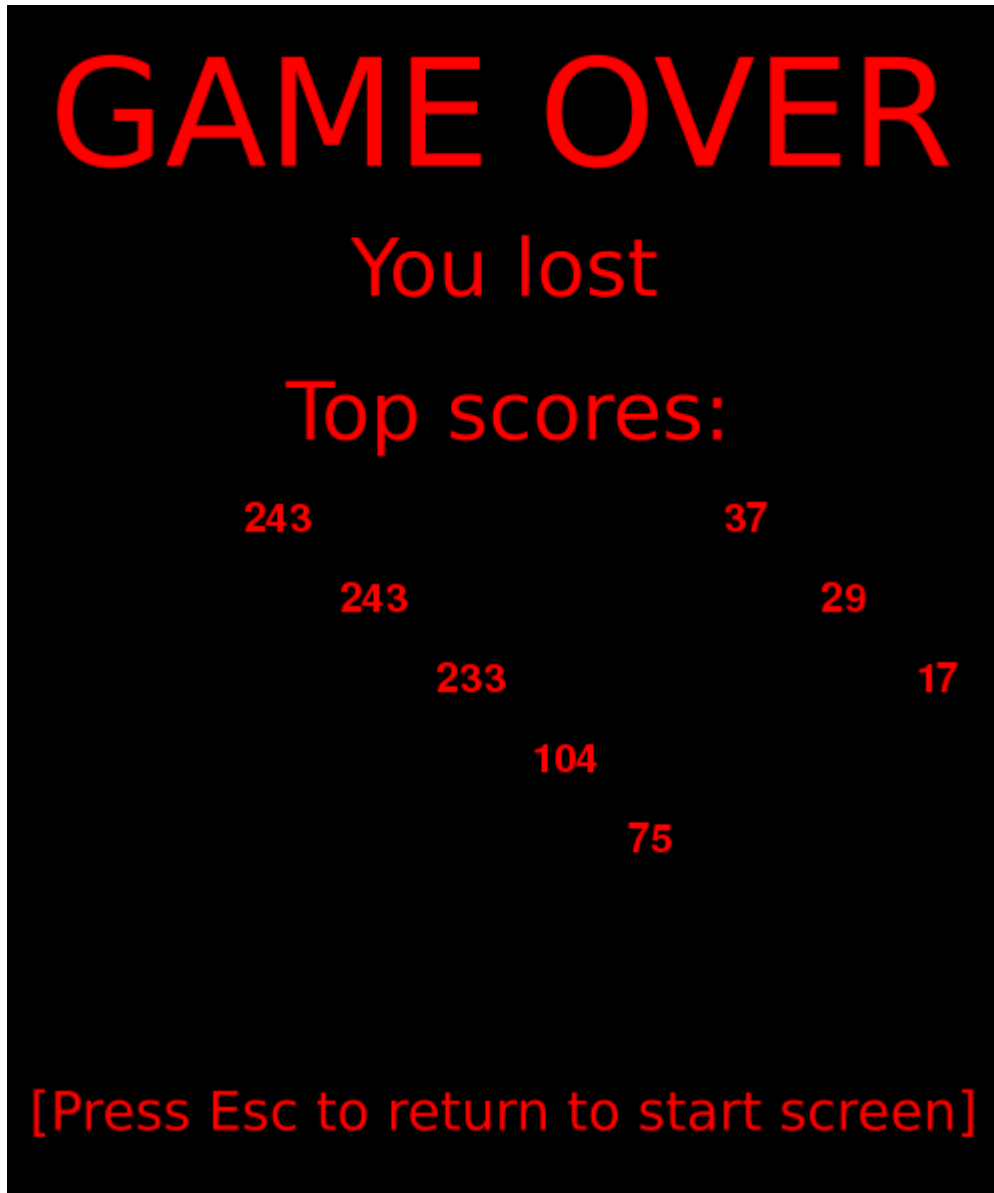
```

629     def print_high_scores(list_of_high_scores, color):
630         for i in range(0, 10):
631             score_to_print = font_scores.render(str(str(list_of_high_scores[i])), 1, color)
632             screen.blit(score_to_print, (120 + 240*(i/5), 250 + 40*(i%5)))
633

```

Testing

The result I got is as on the picture below.



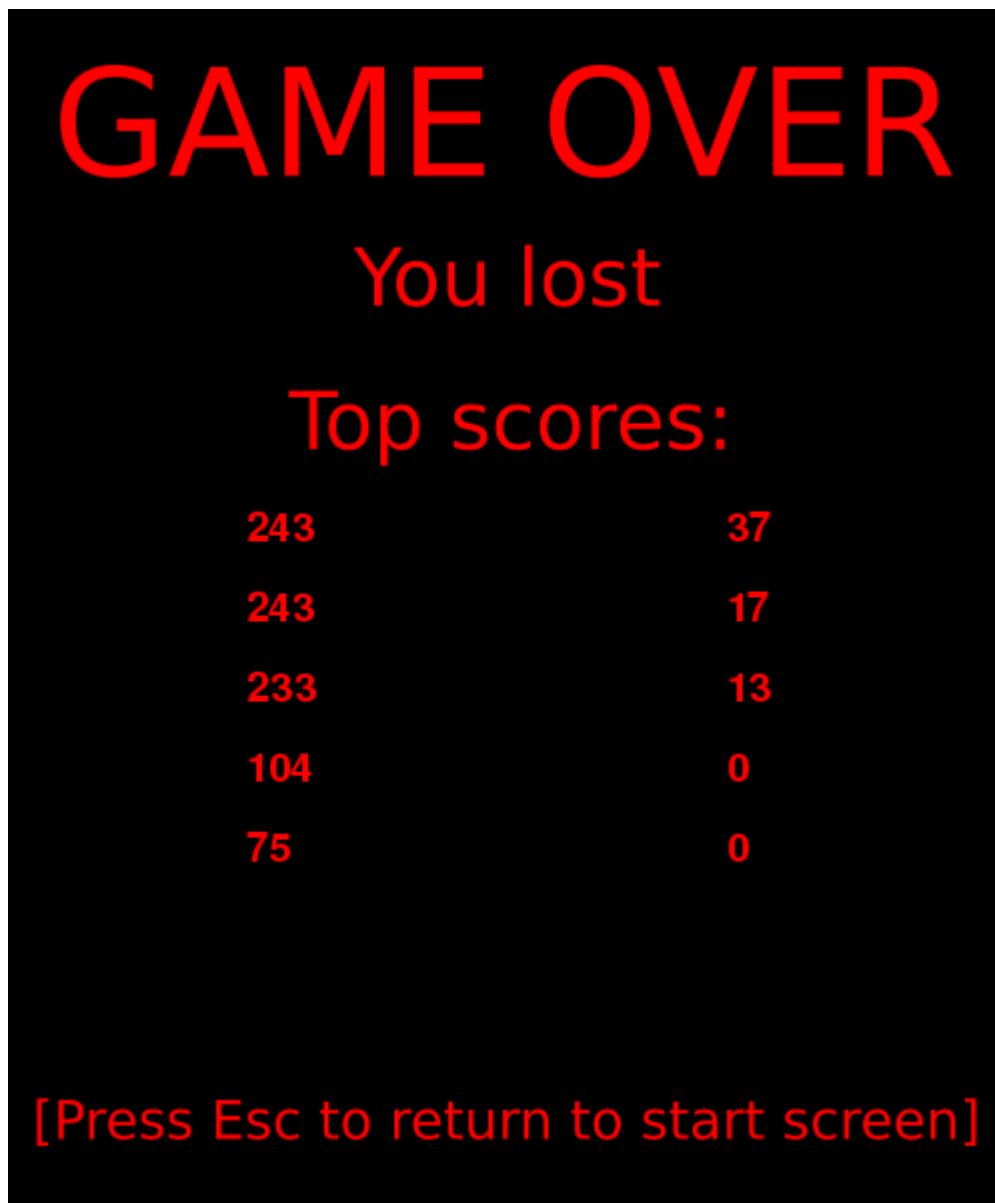
I realised that I should make use of the floor function in the math library, which returns the biggest integer smaller or equal to a passed value. The modified code was as shown below.

```

629     def print_high_scores(list_of_high_scores, color):
630         for i in range(0, 10):
631             score_to_print = font_scores.render(str(str(list_of_high_scores[i])), 1, color)
632             screen.blit(score_to_print, (120 + 240*(math.floor(i/5)), 250 + 40*(i%5)))
633

```

As a result, I obtained the following screen when I lost the game.


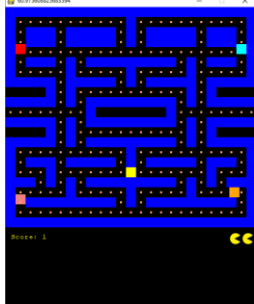




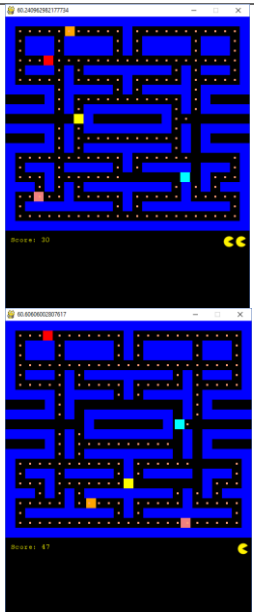
This suggests printing the list of high scores on the screen works as intended.

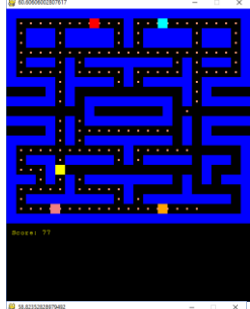
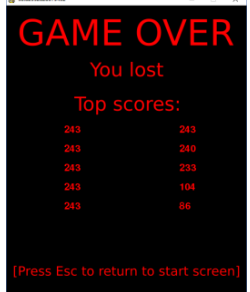
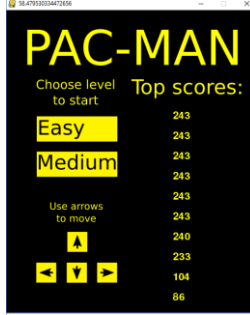

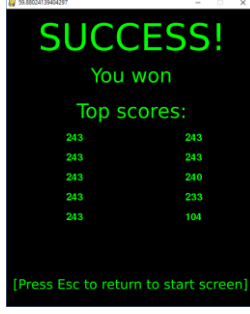
Evaluation

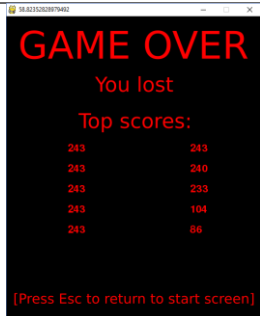
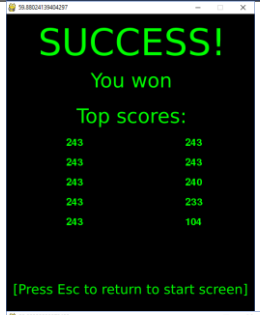
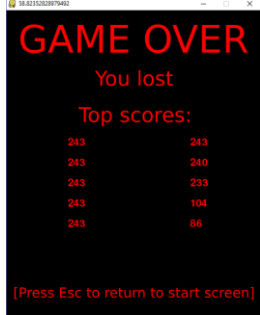
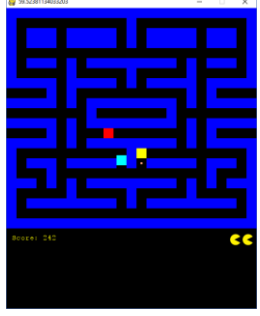
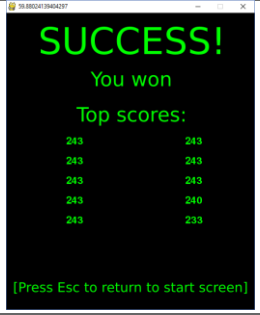
Testing

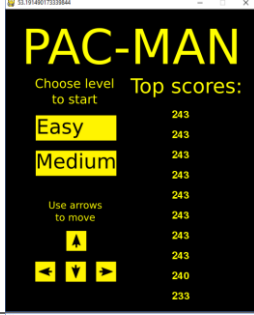

After the development section, I did a complete test of the solution against the requirements. This is shown in the table below. The left column is the requirement, the second column is the description of the test, the third column is the expected outcome. The last column indicates if the expected outcome is the same as the actual outcome. The set of tests carried out is taken from the design section.

Number	Test	Requirements	Expected result	Screenshot of the screen(s)	Passed
1	10 Does Pacman move correctly (does it go into walls, does it move in the direction of the last arrow pressed, does it take the next turn possible, does it go through the tunnel)?	2, 3, 4, 5, 6, 7	Pacman does not go into walls, moves in the direction of the last arrow pressed if possible, takes the next turn possible and goes through the tunnel without leaving the map unexpectedly.	 A screenshot of the Pacman game. Pacman is a yellow triangle moving through a blue maze. He is currently in a tunnel. There are several red dots (coins) and one blue square (ghost) visible in the maze. The score 'Score: 24' is displayed at the bottom left, and two lives are shown at the bottom right.	Yes
2	11 Does the game initialise correctly (is Pacman in the right place, are enemies in the right places, is the map displayed correctly, are all coins displayed, is a correct number of available lives displayed)?	1, 8, 9, 10, 12, 13, 35, 40	Pacman is in the middle of the bottom half of the screen, the enemies are in the corners of the map, all coins are displayed, one in each 20x20 px square. There are two lives at the bottom.	 A screenshot of the Pacman game at the start. Pacman is a yellow triangle located in the bottom center of the screen. Four red dots (coins) are positioned at the corners of the maze. Four blue squares (ghosts) are also at the corners. The score 'Score: 1' is at the bottom left, and two lives are at the bottom right.	Yes

3	12 Do enemies behave correctly on the easy level (do they move in random directions, do they change directions at the junctions, do they not react to Pacman's behaviour)?	14, 15, 17, 18	Enemies move in random directions, change directions only at the junctions and seem to move independent of Pacman's movement		Yes
4	13 Do enemies behave correctly on the medium level (do they find a path towards Pacman's last visited node, do they follow that path, do they move faster than on the easy level)?	14, 16, 17, 19, 26, 36, 37, 38	Enemies seem to follow a path to Pacman's initial position and update their path every time they reach the destination, they move faster than on the easy mode		Yes, but they do not move faster than on the easy level
5	14 Do the enemies and Pacman interact correctly (does the Pacman lose a life when it collides with an enemy, does the game terminate when all lives are lost)?	22, 23	Pacman loses the game when it collides with an enemy, the game terminates after losing all available lives		Yes

				 	
6	<p>15 Do the screens work and display correctly (does the screen change when the user wins or loses, is the score displayed on the game screen, are the lives displayed below the screen, is there a start screen and an end screen, can the user choose the difficulty on the start screen, are top 10 scores displayed on start and end screen, can the user</p>	<p>20, 21, 24, 25, 27, 28, 29, 30, 31, 32, 33, 35, 40</p>	<p>There are three screens: a start screen with controls explained, top 10 scores displayed, available choice of level; a game screen with the lives and score displayed below the map; an end screen which is displayed when the user wins or loses with displayed top 10 scores and an option to go to the start screen by pressing Escape</p>	  	Yes

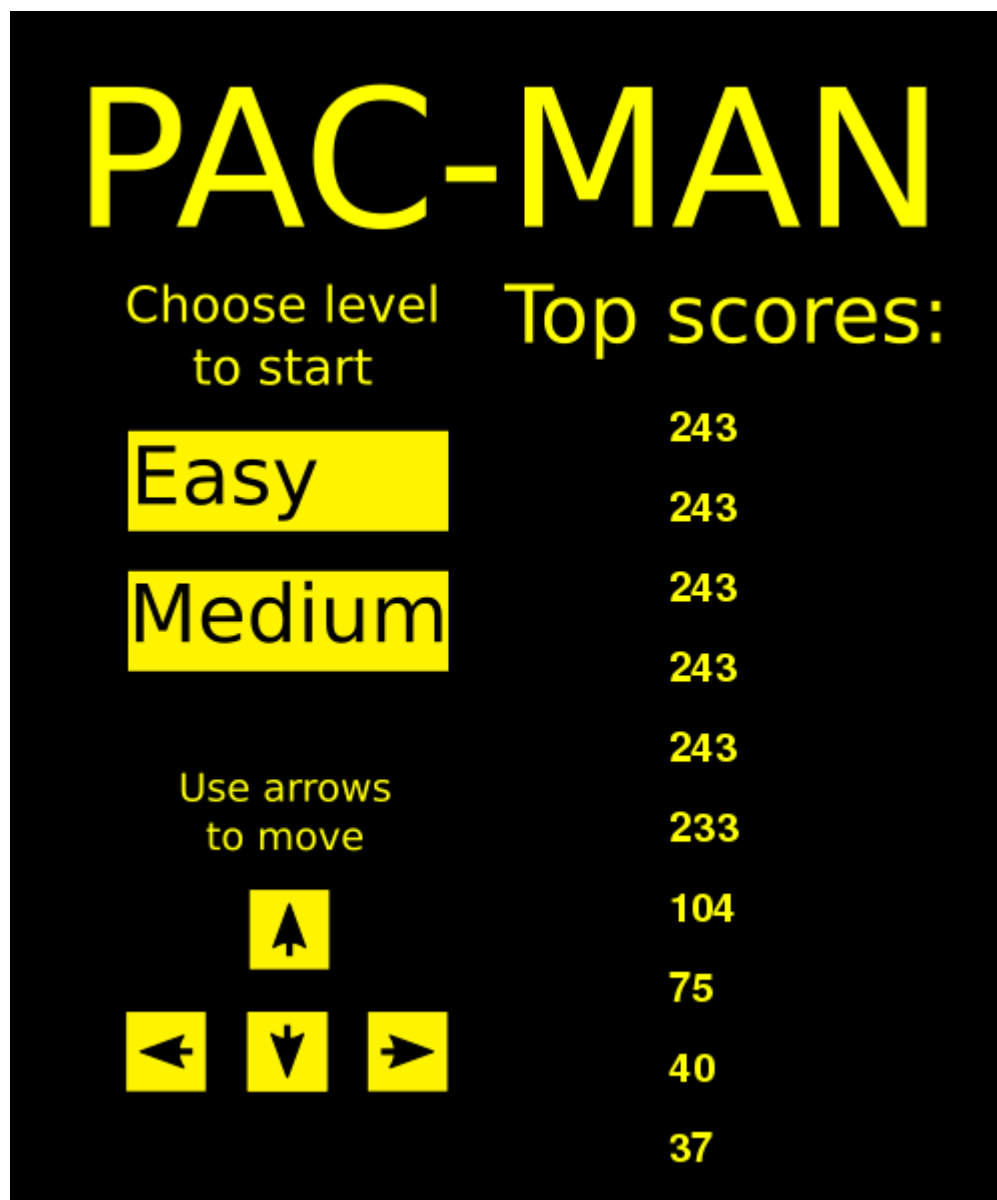
	choose the difficulty, can the user navigate between them)?				
7	16 Does the game correctly indicate winning/losing (is the game won when all coins are collected, is the game lost when all lives are lost)?	21, 22, 27	If Pacman collides with enemies three times, the displayed screen is red and indicates 'Game over'; If Pacman collects all the coins the displayed screen is red and states 'Success'	 	Yes
8	17 Can Pacman collect the coins and does the score change accordingly?	11, 24, 41	Every time Pacman collides with a coin, the coin disappears and the displayed score increases by 1.		Yes
9	18 Are the top 10 scores stored properly?	34	Score is retrieved even after closing the game window and restarting the program.		Yes

					
10	19 Do the pills work correctly (are they on the map, do they make enemies vulnerable)?	39	There are multiple pills on the map which can be collected. After one of them is collected, a collision between Pacman and enemy kills the enemy and gives bonus points		No

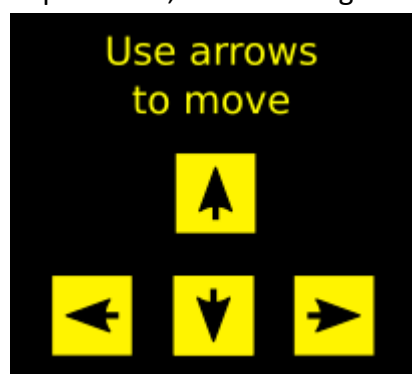
Usability features

Controls explained on the start screen

The game was expected to have controls explained on the start screen. When the program is started, the screen looks as following.



In particular, the following section is worth attention in context of usability features.



This part of the start screen indicates that arrows have to be used to move. It is explained in two ways: by words and by arrows for the users who do not speak English. They are combined to ensure the message is conveyed effectively.

Level select

On the start screen, the following section of the screen is what fulfils the usability feature of being able to choose the levels on the start screen.

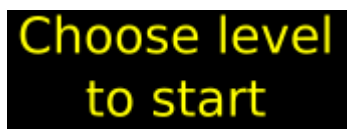


By clicking on one of the two boxes the user notes that the game screen opens, the game starts. Depending on which box was clicked the user can see that the enemies move in a different way. When clicking on easy the user can note that the motion of the enemies is completely unpredictable and cannot be predicted. The enemies move independently and there is no similarity in their behaviour. When clicking on medium it is visible that the enemies follow a path, sometimes a few enemies follow the same route. This clearly indicates that two different levels can be chosen from the start screen, meaning the usability feature of being able to select a level on the start screen has been fulfilled.

Instructions on how to switch screens

There are two ways in which the user is allowed to switch screens. There is a way to switch from the start screen to game screen and from end screen to start screen. The instructions of how to do that are as following.

On the start screen:



On the losing end screen:



On the winning end screen:

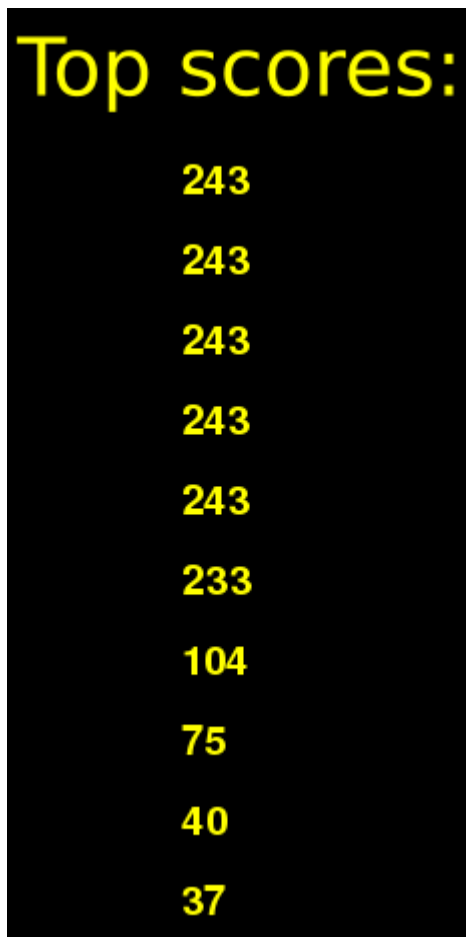


The test for this usability feature has thus been successful.

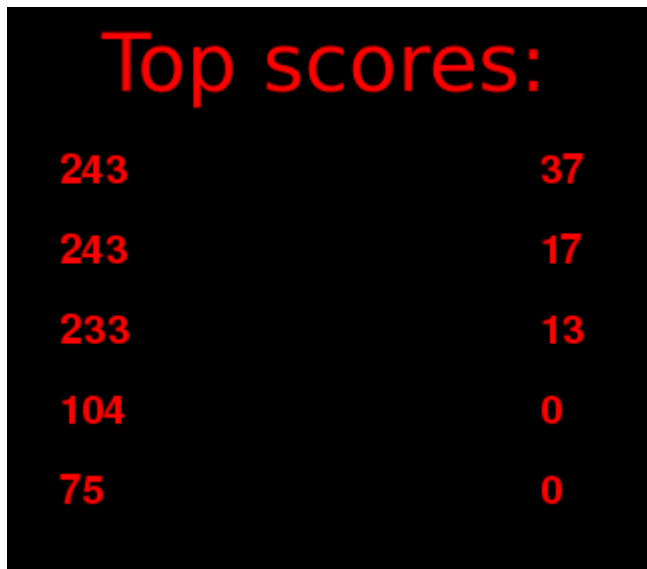
High score shown

This feature is essential for the project, because it caters for the needs of one of the stakeholders – the competitive users – who care a lot about their scores and would like it to be displayed on the screen. The high score should appear on the start and end screens. This usability feature was tested by running the game, winning it or losing it. The following parts of the screens were observed.

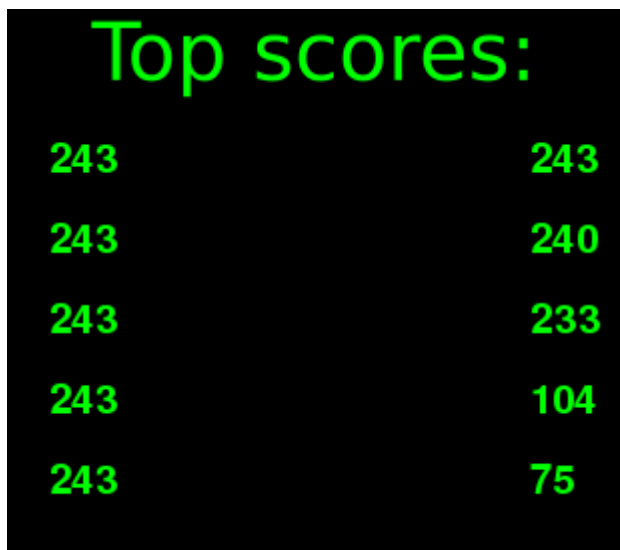
On the start screen:



On the losing end screen:



On the winning end screen:



These three screenshots from the game are evidence that displaying high scores was tested and works as intended.

Contrasting colours

The last usability feature were contrasting colours. This was tested by simply running the game, winning and losing. The following screenshots of the screens were taken:

PAC-MAN

Choose level
to start

Easy

Medium

Use arrows
to move



Top scores:

243

243

243

243

243

233

104

75

40

37



GAME OVER

You lost

Top scores:

243	37
243	17
233	13
104	0
75	0

[Press Esc to return to start screen]



On all of the following it can be seen that the colours contrast with the background colour, because black has been used as the background colour. It may be more problematic in distinguishing the colours of Pacman and coins, but this is intuitive due to different shapes.

Evaluation

Matching the requirements

Requirement		Met/not met	Test
1	Walls are displayed on the screen	Yes	2
2	Pacman can move a set number of pixels every iteration of the program	Yes	1
3	The user can change the direction of Pacman using the arrow keys	Yes	1

4	The program can detect the collisions between Pacman and the walls	Yes	1
5	Pacman cannot go into walls	Yes	1
6	When an arrow is pressed which would cause a collision, Pacman takes the next turn possible in the maze	Yes	1
7	Pacman can go through the tunnel	Yes	1
8	There are coins displayed on the screen	Yes	2
9	There is one coin in each 20x20 px square of the area where Pacman can move	Yes	2
10	Pacman starts from the middle of the lower half of the screen	Yes	2
11	Each time Pacman touches a coin, it disappears	Yes	8
12	There are 4 enemies on the map	Yes	2
13	The enemies start from the corners of the map	Yes	2
14	The enemies can move in two different modes: easy and medium	Yes	3, 4
15	In the easy mode, the enemies can move in random directions on the junction	Yes	3
16	In the middle mode, the enemies follow a path towards Pacman	Yes	4
17	The program creates a graph representing the map	Yes	3, 4
18	The enemies use the graph when choosing the new random direction	Yes	3
19	The enemies use the graph to find a path towards Pacman	Yes	4
20	The game screen is changed when the user wins or loses	Yes	6
21	The user wins when he collects all the coins on the screen	Yes	6, 7
22	The user loses when he dies with no available lives	Yes	5, 7
23	Pacman loses a life every time he collides with an enemy	Yes	5
24	The score is displayed below the map	Yes	6, 8
25	The lives available are displayed below the map in form of icons of Pacman	Yes	6
26	A path finding algorithm is used to find a path towards Pacman	Yes	4
27	The game has a start screen and an end screen which depends on whether the user won or lost	Yes	6, 7
28	There is a starting screen	Yes	6
29	The user can choose the level of difficulty on the start screen	Yes	6
30	The top 10 scores are displayed on the start screen	Yes	6
31	The controls are explained on the start screen	Yes	6
32	The top 10 scores are displayed on the final screen	Yes	6
33	The user can go to the start screen from the end screen by pressing Escape	Yes	6
34	The top 10 scores are stored in an external text file – this is necessary for the stakeholders who are very competitive and want their score not to disappear	Yes	9
35	The design of the map is as in classic Pacman	Yes	2, 6
36	The enemies react to the position and movement of Pacman	Yes	4

37	The enemies find a path to the current, or predicted future of the user, depending on the level of difficulty	No	4
38	The speed of enemies increases with increasing levels	No	4
39	When a pill is collected, it makes enemies vulnerable for a short time	No	10
40	Positions of the objects, the coins, available lives and score are reset to initial values if the game is restarted	Yes	2, 6
41	The score increases by 1 every time Pacman collects a coin	Yes	8

As indicated by the table above, the solution matches the requirements very well. All of them were tested with the 10 different tests carried out on the game. As a result, one can see that the solution fulfils the vast majority of requirements. It does not, however, fulfil a couple of requirements, 2 of which are related to enemies and their behaviour directly, and the other is about having pills which would make enemies vulnerable.

Success criterion	Requirement number	Test proving that met
There is a map resembling the classic Pacman version	1, 8, 9, 10, 12, 13	2
The enemies move in a way depending on the level of difficulty On the easy level, in random directions On the medium level, finding a path to Pacman's position at a moment	14, 15, 16, 17, 18, 19	3, 4
Top 10 scores are stored offline on the device, and are retrieved when the game is started	34	9
User can use arrows to determine the direction of movement of Pacman and can do so easily in the maze	3	1
The user can collect coins	11, 41	8
Game terminates when it is lost or won	20, 21, 22	7
There is a set of screens enabling navigation between them	20, 21, 24, 25, 27, 28, 29, 30, 31, 32, 33, 35, 40	6

As indicated by the table above, the tests 1 – 9 have proven that the success criterion are met. The fact that all of them are met means that the game is perfectly usable and can be played, catering for the needs of the stakeholders. Hence, the unmet requirements do not impede upon the playability of the game and not fulfilling them is not a problem.

Changes to the design

There were multiple changes made to the design during the development stage. First of all, the colours changed drastically after the first iteration. This change was made after a consultation with the end user, who suggested making the change. Because it did not require significant workload, I implemented the suggestions. Both the end user and I find the colours more attractive now.

There was a big change done in terms of algorithms used for controlling the enemies. At the beginning, I was aiming for the best option possible and decided to use an A* algorithm in the design. As I progressed, however, the details of the implementation turned out to be a difficult part of implementing the graph. I analysed that implementing an A* algorithm would take a lot of time, which I did not have. Thus, I looked for other path finding algorithms. I could have used Dijkstra's algorithm, but it would take even more time to use in the program, which would jeopardise smoothness of the game. Moreover, the user did not require the path finding algorithm to find the shortest path. Any path would satisfy him, as he stated in his first interview. Thus, I could use DFS or BFS. I decided to choose DFS because it requires less memory, is likely to find a path faster and is more suitable for mazes. This change in the end resulted in the enemies following a path which was visibly inefficient. The enemies tended to miss Pacman when it was very close and take a route all around the map when Pacman was not in an immediate vicinity. However, it still meets user needs and caters for the needs of stakeholders.

The design of the start, game and end screen changed during the development stage due to the general changes in the game.

Firstly, there were three changes to the start screen. Now the bonuses are not explained, aims of the game are not stated explicitly, and the placement of different parts of the start screen changed. The bonuses are not explained, because there are no bonuses or pills in the game. It was decided during the development process that there is a good difficulty of the game. Additional bonuses would infringe on this equilibrium, making the game less fun to play. Secondly, the aims are not explained because that would make the start screen too congested and the user is most probably aware what one should do in Pac-Man. Finally, due to the fact that the previous two changes were made, the placement of different parts changed so that the start screen looks more spacious and aesthetic.

Furthermore, the end screen has slightly different functionality. In the design, there was an option to play again, go to start screen or try a harder level from the end screen. In the game there is only one option of going to the start screen. I decided to make that change, because it does not change functionality (the program can be run again or a harder level can be run from the start screen) and reduced the number of functions I had to implement. I therefore saved memory that the program occupies in RAM, improved speed of execution without losing any functionality.

Moreover, the design of the game screen changed: the difficulty of the level is no longer displayed below the map. This is because there are only two levels and the user chooses it, so they know which level of difficulty they are at. As a result, displaying the difficulty in form of chili peppers would be useless for the game.

Unmet criteria

The unmet criteria are:

37. The enemies find a path to the current, or predicted future of the user, depending on the level of difficulty

38. The speed of enemies increases with increasing levels

39. When a pill is collected, it makes enemies vulnerable for a short time

37.: One of the success criteria required the enemies to find the shortest path to Pacman's current or future position. In the game the enemies work a bit different. They register the last node Pacman visited, find a path from their position to that node, follow that path until they reach destination node, repeat the cycle. In the testing phase it turned out that this is not a very effective way for the enemies to follow Pacman, because Depth First Search finds a path, not necessarily the best one. Depth First Search was used instead of Breadth First Search, because it is more likely to find a path quicker and takes less memory to execute. Given more time this requirement could be met by updating the path of an enemy towards Pacman every time an enemy reaches a node. This would allow to follow the path to Pacman's current position. In order to move to Pacman's future position, the enemies could predict which node Pacman is going to be in a couple of cycles by getting direction and finding the first node in that direction from Pacman's position. I could also use A* algorithm to find the quickest path to Pacman

38.: The speed of enemies does not change between different levels. The reason for that is that without any bonuses or pills the game is already fairly difficult. If the enemies were even faster on the medium level, the user may perceive the enemies as moving too fast on the screen, making the game confusing, harder, and less fun to play. This requirement could be fulfilled by changing the value of the variable `speed_of_enemies` to a higher one if level is 2. This could be easily changed, but would not benefit the game.

39.: Pills were not included in the game. In the original Pacman there are pills which Pacman can collect. In my version of game there are no such pills. This requirement has not been met, but was due to the implementation of the movement of enemies. The enemies were finding a path towards Pacman using DFS. It would be hard to use the same algorithm for finding the path of escape. Moreover, the game has a very well balanced difficulty, so adding pills would make the game too easy and not fun to play. If there was more time this could be fulfilled by making the enemies move in general direction away from Pacman on medium level or in random directions on the easy level.

Additional features

In the future, there is a variety of features which can be added.

First of all, A* algorithm, Breadth First Search or Dijkstra's algorithm can be used for finding the path towards Pacman by the enemies. This could make use of the graph already implemented.

Next, more sophisticated ways of counting the score can be implemented, as well as point bonuses appearing on the screen. This would make the game more competitive and fun to play even after winning it. Currently, there is a maximum score the user can obtain when winning the game. Making the score dependent on time taken to collect all coins, on the lives lost, point bonuses collected would increase the likelihood of the game being played regularly.

On the final screen, the score achieved by the user could be displayed, as well as highlighted on the list of scores. At the moment, it is ambiguous for the user if he achieved one of the top high scores. He does not know, unless he looks under the map, what his score was. Displaying the score would make the game more rewarding.

Maintenance

Corrective

First of all, when going in the same direction as an enemy and narrowly escaping from it, the user feels he could escape even if the enemy is very close. However, as soon as Pacman and the enemy turn, Pacman and enemy collide on the junction, disappointing the user because of an unsuccessful escape, which should be successful. The aim of corrective maintenance in this context should be improving the collision detection between Pacman and enemy on the junctions in a way which would enable narrow escapes.

Adaptive

In the future, the user may get bored with the game, because there are only two levels of difficulty and one map. To account for this in the maintenance, further levels should be developed. Moreover, new maps should be created in order for the user to face new challenges and find the game interesting.

Furthermore, as the user does not find the game difficult enough anymore I could improve the performance of the path finding algorithm. I could change the way assigning the start and destination node works. I could update the position of Pacman as soon as it gets to a new node and find a path to that node. Moreover, I could use A* algorithm, as predicted by the design to find a shorter and more efficient path. I would have to watch out, however, not to make the game too difficult.

Subsequently, I could engage the user in adapting the speed of the enemies. I could conduct multiple trials on different people to manipulate the speed of the enemies to achieve the intended ratio of wins for a statistical user.

Bibliography

1. <http://www.playpacmanonline.net/> - example of Pacman
2. <http://worldsbiggestpacman.com/> - world's biggest collection of Pacman mazes
3. Pacman in Google browser
4. Open Courseware - <https://ocw.mit.edu/index.htm>
5. Python documentation: <https://docs.python.org/2/contents.html> - documentation on Python 2.7 used in the project
6. Pygame documentation: <https://www.pygame.org/docs/ref/sprite.html>

Appendix A - Interview #1

Q1: Do you prefer moving Pacman with arrows, WASD or both?

I do not have a personal preference, but moving with arrows would be perfectly fine.

Q2: What should be the colour of the walls?

Any colour would be good, but I think red will be the best.

Q3: What should be the colour of Pacman?

Yellow, definitely, as in the classic version.

Q4: How many enemies would you like to have? Would you like to have the same number as in the classic version or more?

I think the classic version has a good number enemies, so I would like to have the same number in my game.

Q5: What should be the scoring system?

1 point for a coin, 10 points for every enemy destroyed would be a good proportion of points. Additional bonuses should also be a possible source of 20 points per collected bonus.

Q6: Would you like your highscore to be kept?

I am highly competitive and I like to push myself, so I saving high score is a necessary feature for me.

Q7: Would you like to have a list of top 10 scores?

This is not as important as keeping my high score, but a list of top 10 score would be a good feature to have.

Q8: Would you like to have controls explained on the starting screen?

Although it is not necessary for me, if someone else wanted to use this game it would be very useful if the controls and rules were explained on the starting screen.

Q9: Would you like to have randomly appearing bonuses on the map?

That is a very good idea! The game could be more interesting if some of the bonuses were negative (i.e. subtracting points).

Q10: How many lives would you like to have?

I think the number on the classic game is too low, but too many would make the game too easy. Hence I would like to have 3 lives on each stage, reset every time the player goes to the next stage.

Q11: How many stages would you like to have?

I think 3 would be optimal.

Q12: What would be the difference in different stages?

They should have an increasing difficulty, for example enemies could move faster or in a different, more difficult and intelligent way.

Q13: Do you prefer playing on the classic map or other maps?

Classic map, definitely. I have played on other maps as well, but the classic map appeals to me more.

Q14: What do you think about the difficulty of the classic version of Pacman?

I think it is too easy. You can find patterns in the Internet indicating how your Pacman should move in order to clear a stage.

Q15: Do you think the enemies should move in a set pattern or should they move in an intelligent way?

As I said, it is a big problem that you can find patterns in the Internet indicating how you should move. That would be the case if the enemies were moving in a set pattern. Thus, I believe intelligent algorithms for steering the ghosts are necessary.

Q16: Do you think shortest path algorithm would be enough, or should they have artificial intelligence?

AI is not necessary in this case, any algorithm which would be dependent on the moves on the player would be sufficient for me to see that as an intelligent algorithm.

Q17: Do you think the choice of difficulty would be useful for you?

Yes, I would like to be able to choose easy, medium and hard version of Pacman.

Q18: Would you enjoy playing against enemies with increased aggression?

Yes, definitely, for example ones that would follow me on the shortest path available.

Q19: Do you think the ability to create and destroy walls would be an interesting and innovative feature?

I like the classic version of Pacman, the only problem is too low difficulty. This may be an interesting feature, but it is not necessary for me.

Q20: Would you enjoy playing with or against other people?

No, for me Pacman is definitely a single-player game, and I would like it to stay that way.