PycMan

Jan Dziedzic

October 9, 2017

Contents

Chapter 1

Project Definition and Analysis

1.1 Project Definition

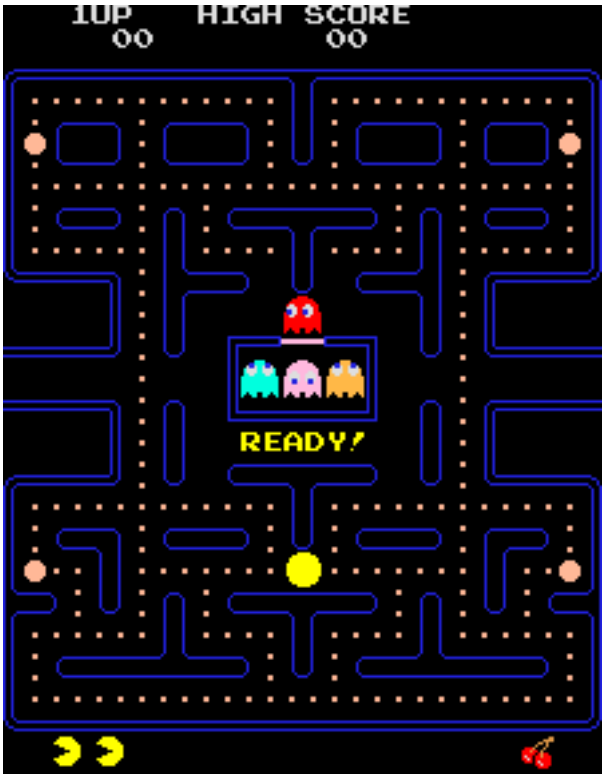

Figure 1.1: Original Pac-Man screen. Source is Reference 1.

As my project, I have decided to present a game written in Python, based around one of its modules - pygame. As I have not had as much time to prepare my project as my colleagues, I have decided that the concept of the game must be simple, yet allowing to demonstrate my programming and computational thinking skills. For this purpose I have decided to mimic the iconic PacMan game. For picture of the original game screen please refer to Figure 1.1

Called PycMan (utilizing the fact that many of Python modules have a prefix 'py' in their name), the game itself strongly resembles the original, yet with some meaningful changes. Like in the original - player can move in four directions (left, right, up and down), collecting coins and avoiding ghosts.

The game develops critical thinking skills and takes effect of human nature of taking risks and tackling unknown - with having more confidence when predicting ghost behavior, the player actually gets better when playing despite problems generally low complexity.

Actually the game itself may be referred as to playing tag in a transparent

maze. Such a play would be very hard for a human to enjoy as analyzing transparent walls while running and predicting opponent's next move is a far too complex problem for human cognition. That's why computational approach on this game is required - where player can see ghosts from above the board whilst ghost may exhibit algorithmically advanced behavior.

1.2 Stakeholders

Proposed end user group is really wide. As the game itself can be launched on virtually any device due to Python versatility, it may be used on a spectrum of devices with different controllers, from mobile phones and smart watches through PCs reaching as far as game console emulators mimicking the most authentic PacMan game experience. Such portability should satisfy most users.

The game should grab attention of children - with small challenges that don't require complex thinking and rather promote manual skills and reflex; those who seek quick brainteaser whilst attempting to play with more focus on tactics of ghost avoidance and finally, old gamers who played original PacMan but now they might want to try more modern version with a different approach to critical game mechanisms.

1.3 Software Challenges

All software used to develop the game but is not necessarily required to run it:

- Python3
  - Pillow module
  - Pygame module
- Pycharm Professional
- GIMP - GNU Image Manipulation Program
- LaTeX
- TeXstudio

The greatest advantage that convinced me to actually use Python for my game was its portability. As an interpretable programming language, the code of the game remains universal throughout different operating systems and processor architectures. My decision was also influenced by my previous experience with this language and my desire to actually get better around it. It's also free, meaning that everyone can take benefit of my game, without having to buy a license from a third party.

Also, whilst most of my experience was with Python2 I have decided that it's the right time to move on and actually start using Python3, which is reportedly faster and offers more functionality while providing greater stability.

Still, Python does not offer the speed one may expect from a programming language suitable for complex games. Advanced non-optimal solutions need to be thought of and substituted with faster algorithms in order for the game to run smooth on machines lacking processing power.

Happily Python is an objective language - which greatly helped with some designs.

As it's also an interpretable language, it's way easier to debug as one may actually run every single line of code step by step to see variable values changing. Python interpreters also provide a very generous error descriptions greatly helping with bug fixing.

The closest I could get with Python to making a full featured game conveniently was to use pygame. It's a rather good module supplying some basic functionality one may expect from assisted-design library for creating 2D games. Some functions were not supported out of the box, like level storing (suitable for my purpose) and loading and these had to be implemented by myself as an addition.

Pygame also doesn't have a tiling system I wanted for level designs, to make my game resemble original PacMan in terms of design. I had to write such myself. A similar situation occurred when I wanted to implement discrete "Time Segments" (see Section 2.5.1 for reference). For a game such as PacMan I find such solutions to be a very useful optimization and yet these are missing from the pygame module.

Pycharm by JetBrains was used as a Python IDE - it is a really convinient solution and comes with a lot of useful features. It was chosen due to my previous positive experience with it.

GIMP was used to produce graphics resources of the game as well as levels - described later. I found it to be a great deal of overkill for such basic tasks but I am very familiar with it's interface and tools which were of great help especially during the design of semi-transparent ghost graphics.

LaTeXand Texstudio were used to develop this document. As I identify as a pro-open-source-software person I wanted to use LaTeXinstead of word processors with a far advanced UI, like Microsoft Word. A practical reason behind this - LaTeXprovides an absolutely full flexibility. And even when it doesn't have a function I might want to use - I can always write such myself.

All development was done on an Ubuntu-running computer, therefore I must proudly admit that during the entire design process no non-free software was used. All software was either opensource, freeware or free for education.

1.4 The Interview

To assess interest and gain insight of the potential market for the game, I have interviewed my fellow schoolmate - Wojciech Wojtkowski on his opinion of my approach to redesigning PacMan.

-Hello Wojciech, may I interest you with my IT project - the PycMan, next generation of the classic PacMan with smarter ghosts and different mechanics?

-Sure mate, go ahead!

-Ok, so these are some concept graphics [first two levels were exhibited]...

-...but it looks just like the original!

-Yes, there is a strong resemblance in terms of the graphics, that's what I am aiming for. The biggest difference is how the ghost work.

-Oh, tell me more.

-So, in the classical PacMan, the ghost have their designated area they launch from, A.K.A. "The Ghost House", I want to get rid of that, instead all ghosts will start from predefined locations different for each level.

-That means they will chase you from the exact beginning of the level, isn't that going to make the game harder?

-Yes it will, that is the goal. But that's not the main change. They will be smarter than the originals.

-You mean that the way they move is going to be less predictable?

-That's true, I want to utilize a rather complex algorithm to make them chase player more efficiently.

-Didn't the original have the most efficient solution?

-No, the target machines lacked processing power and memory in the past to actually use that with the game not slowing down. Now, when newer computers are available, I can actually use that.

-So if they were quite stupid then and the game was still hard, won't making them smart render the game impossible to win?

-That's what I am afraid of, I need to find a way to give the player some advantage. Do you, as an experienced gamer, have any idea how to do that?

-I think that making the ghost chase the player indirectly may be the way, how about them tailing the PycMan?

-Yeah, that migh be fine but this may eventually lead to them not catching it at all if it doesn't move.

-Oh, that might be the case.

-Actually I have one solution in mind - making them a little bit slower than the player.

-Seems okay, thought I think that one may run away from them, do a risky eating-maneuver then and regain the distance lost. Repeating that will make the game easy and very boring actually.

-Oh, true, I will have to think of these solutions. Probably final version will be designed during beta testing based on player satisfaction with each

of these methods.

-Good idea to let the players decide. Actually - on the player-decision thing. One thing that I always wanted with PacMan is to design my own levels. Can you make it possible?

-Yeah, I already have a solution in mind that will make it very easy for anyone to design their own maze, if you say that's going to interest players, I will surely include that.

-Cool! Thanks for letting me know, can I play that game later?

-Of course, as soon as I release the beta.

-Perfect, thank you then, I look forward to playing it.

-Thank you for the talk and insight. Bye.

-Bye.

1.5 Requirement specification (success criteria)

For the project to succeed the following criteria must be met:

1. The game must resemble the original PacMan in terms of graphical design and some of the mechanics.
   (a) Game graphics should be of similar color and shape to the originals.
   (b) Player sprite is an iconic yellow ball with 'mouth'.
   (c) Player sprite should rotate with 'mouth' towards the direction of movement.
   (d) Neither ghosts nor player are to pass through wall or be able to exit board borders.
   (e) Ghosts chase the player who loses a life upon contact with a ghost.
   (f) Upon losing a life ghosts and the player return to their initial locations.
   (g) Each ghost behaves in a different way based on its color.
   (h) Ghosts do not reverse their direction of movement.
   (i) Player progresses through a level with eating 'coins' left throughout some/all accessible places on the map.
   (j) Upon eating a coin, it disappears and is not to be rendered again.
   (k) Eating a coin increases point counter.
   (l) Heart shaped Eatables are left in some places in some of the levels.
   (m) Eating a heart grants the player extra one life.
   (n) Upon completing the level (eating all 'coins') new level is loaded.

(o) When player has less than one life the game finishes and the player loses.

(p) When player completes all the levels, the game finishes and the player wins.

(q) Text messages appear whenever a significant change in gameplay is to take place. E.g. Start of the game, level change, player's death, completing entire game.

(r) The game must be possible to win.

2. The game must be different from the original in these ways:

   (a) Majority of players must find the PycMan ghosts 'smarter' than the originals.

   (b) There is no 'ghost house' where ghosts start from. All sprites move from the beginning and ghost chase the player immediately.

   (c) There are no power-ups in the levels.

   (d) Ghosts cannot die.

   (e) Ghost don't change 'modes' and don't became frightened of the player.

3. Requirements independent from similarities to the original Pac-Man

   (a) Arrow keys used to control player's movement.

   (b) Spacebar used to dismiss on-screen messages.

   (c) Player sprite must not move without user input.

4. Game must work smoothly (30 frames per second is considered to be the standard of human perception of fluency) on contemporary medium-class laptop PCs. The following hardware and software requirements are to be met:

   (a) Operating system supporting Python3 interpreter

   (b) Python3 interpreter

   (c) Pillow module (PIL)

   (d) Pygame module

   (e) Color display of resolution of at least 600x600px

   (f) Keyboard

   (g) 500 MB of storage space

   (h) 512 MB of RAM

   (i) 700MHz processor

The game has been tested on described specification machine and has been found to meet fluency criteria. No testing on slower machines has been performed as these are not usually available on the market anymore.

Chapter 2

Design

As the game uses Pygame module it obviously derives some solutions natively implemented in it. All display solutions are actually handled using the module. Use of Tkinker was researched for pop-up messages but adding another module that doesn't bring any outstanding functionality above capabilities of Pygame has been ruled as an unnecessary waste of memory.

2.1 Main window layout

Main game window consists of a board where the actual game takes place. Below the board there is a set of informative counters kept in characteristic PacMan colours of gold-yellow on dark/royal-blue background.

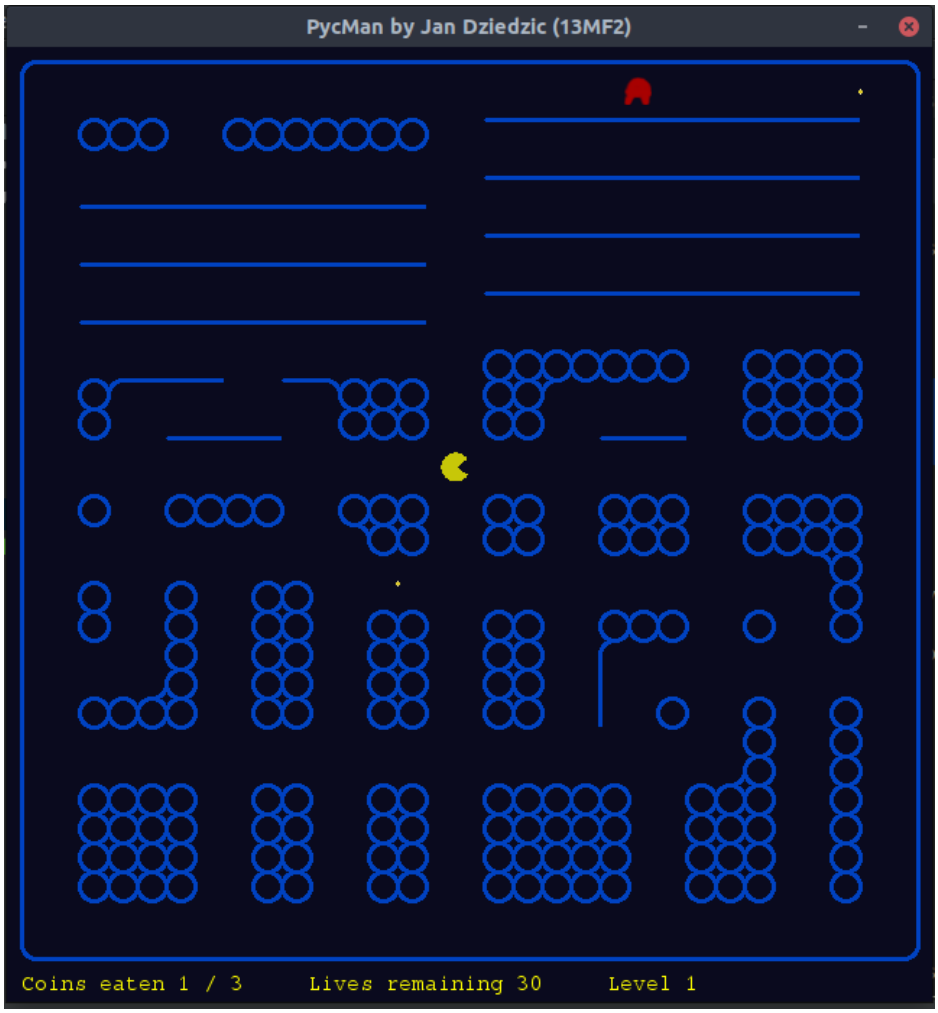

Figure 2.1: Game window rendered in Ubuntu Gnome graphical environment.

2.1.1 'Board'

As each level has a different layout so look of the board may vary. Please refer to Figure 2.2 for an example of such board.

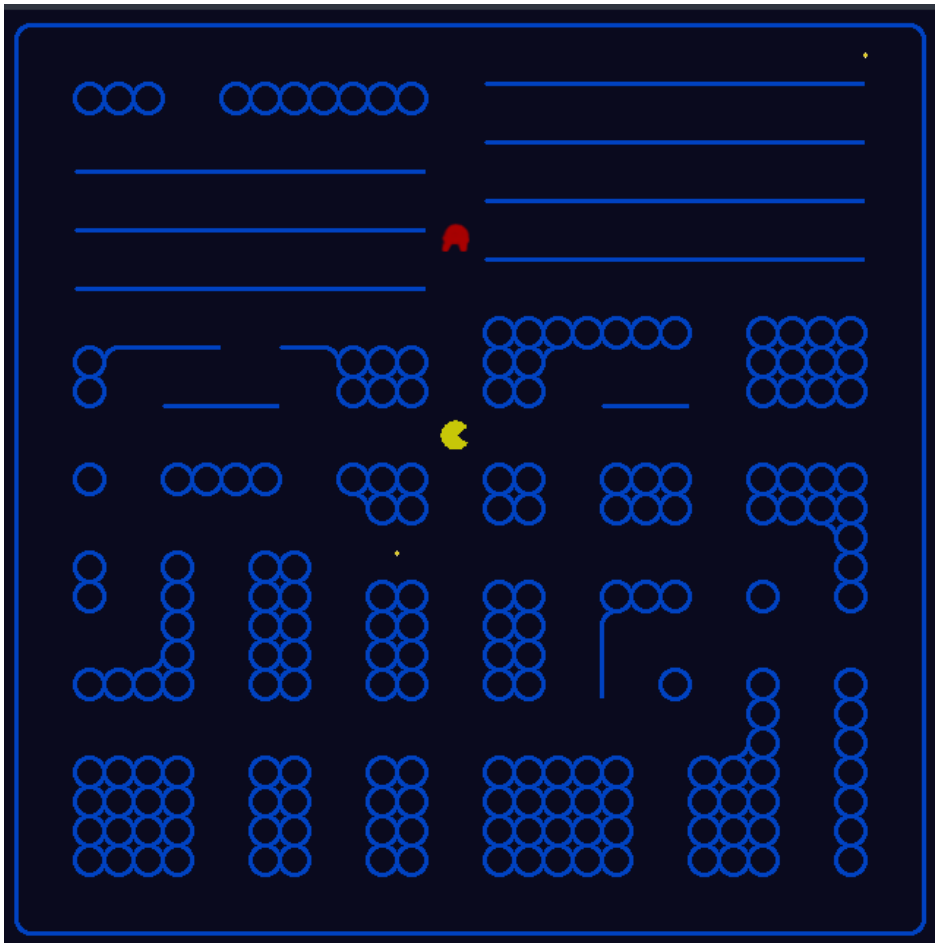

Figure 2.2: Example board. Walls, empty tiles, the player, coins and a red ghost are visible

General idea is that all border tiles of each level (tile system explained later) have to be walls, which creates a nice, outer border of the board with rounded edges.

2.1.2 'Counters'

Directly below the board, counters are located, these provide information on:

1. Number of coins eaten

2. Number of coins that are required to be consumed before progressing to the next level

3. Number of player lives

4. Current level number

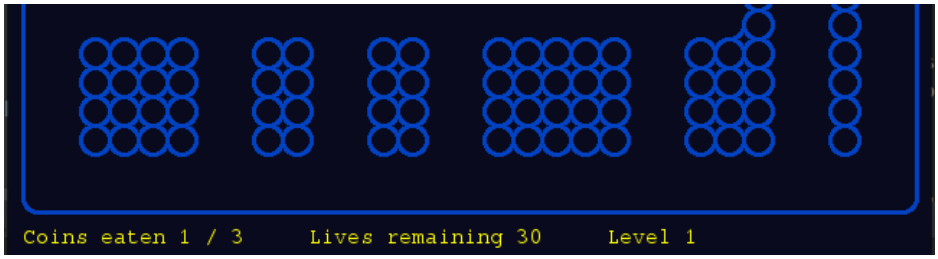

Figure 2.3: Example set of counters with part of the board included for position reference.

Every time the player eats a 'coin' the first counter increases. Throughout the level, this counter cannot decrease as even upon player's death, the already-eaten coins don't respawn.

Number of coins required to progress is constant through the level but may differ between levels. It is worth mentioning that as every coin must be eaten to progress, this acts as a total number of coins allocated in each level map. Coin placing algorithm described later in this document also proves that it equals to $32^2 - \#_{wall\_tiles} - \#_{empty\_tiles}$.

Each time a player comes in contact with a ghost(defined later in this document) a life is subtracted and every time the player eats a heart eatable, a life is added.

Level number increases from 1 (easiest level) up to ten (hardest level), as player advances through the levels.

2.2 Grid layout

Pygame provides a sprite attribute of its location but as I have decided to use a window of over 500x500px of size, its pixel-accurate positioning would be an overkill and could make programming harder as well as require more processing power to (later mentioned) pathfinding algorithms. Due to this I have turned to the original PacMan solution (as read in Reference 1), the grid system.

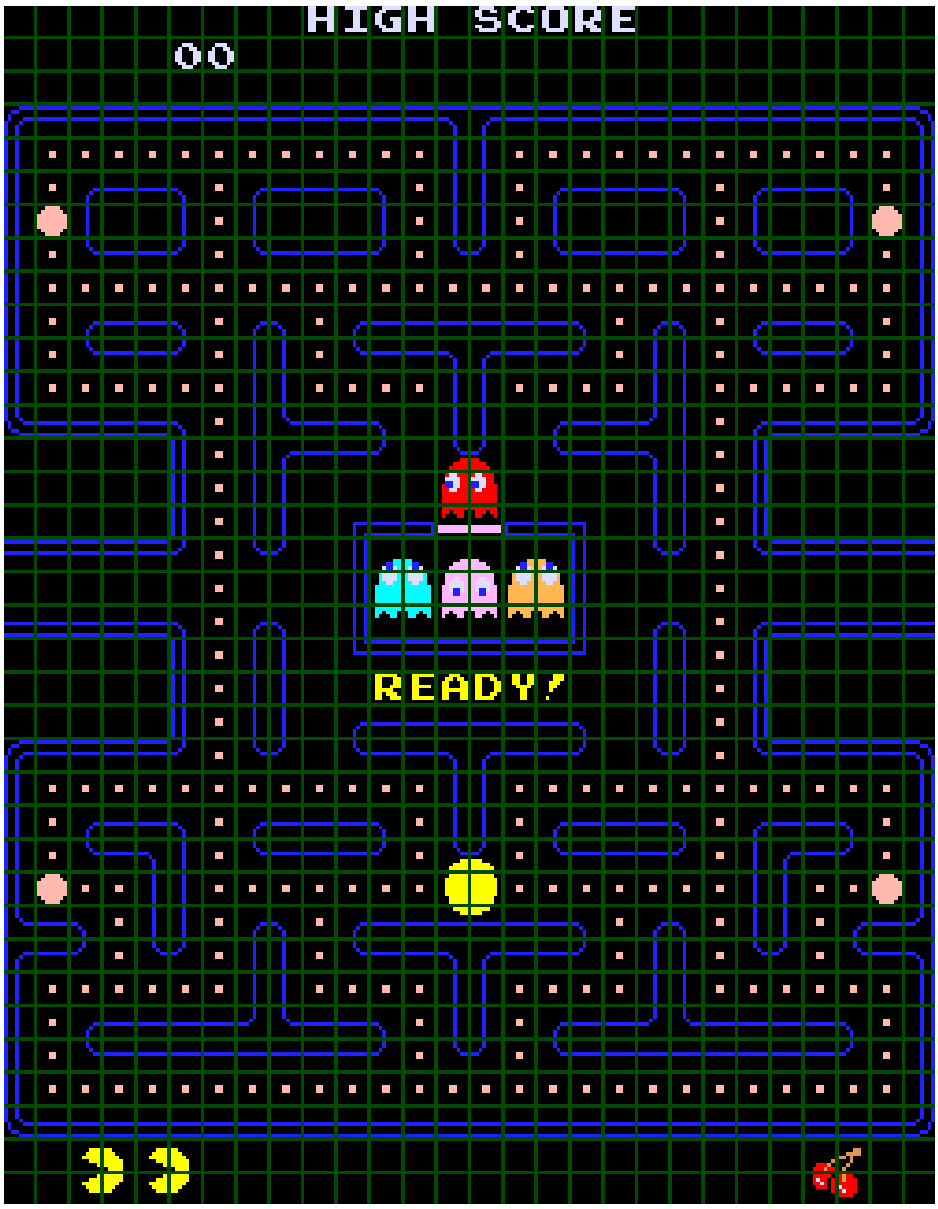

Figure 2.4: Original Pac-Man with grid layout depicted. Source: Reference 1

The PycMan board was initially made as a 64x64 grid, where each piece, namely a tile, was either a wall or a space a sprite may move on. First levels were designed this way and I found the level design a really hard task. Only later I have noticed that the grid of the original PacMan was barely $\frac{1}{4}^{\text{th}}$ of the size I have used, my board was just to big for a pleasant gameplay.

As I wanted the board to be square I have decided to move to 32x32

grid. Some testing later I have decided that this size not only nicely divides by 2 making perfectly symmetrical or even fractal levels possible to make but also is actually quite the one most similar in terms of number of tiles to the one the original PacMan had (for square boards).

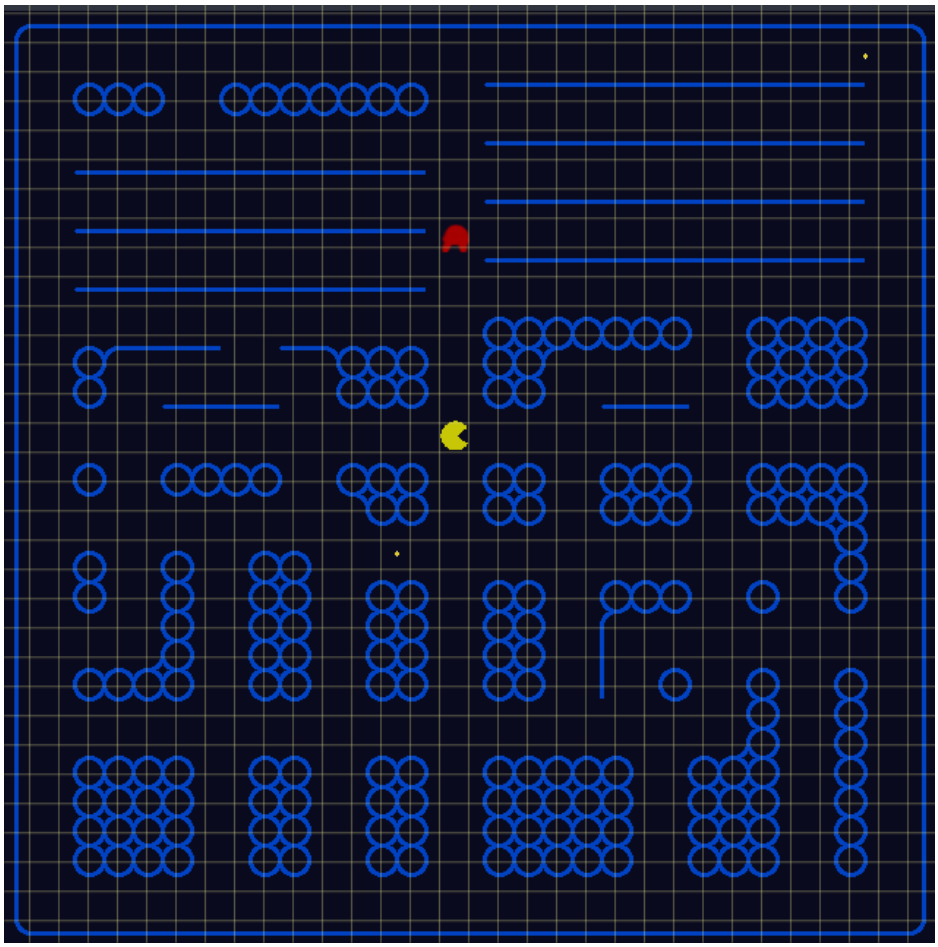

Figure 2.5: Example of a board with a grid applied (note that the grid was added just for demonstrative purposes and in not a part of the game.)

The following section describes types of different tiles used throughout the game.

2.3 Types of tiles

2.3.1 'Walls'

As wall tiles have to connect with neighboring (common edge) tiles, they need to be represented by different graphics depending on their surround-

ings. There is a total of 15 graphics depicted below. The system of numbering them is really intuitive, it consists of four digits each of which is either zero or one

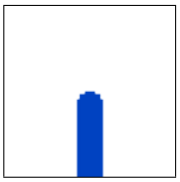

Figure 2.6: Wall 0001

Figure 2.7: Wall 0010

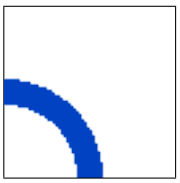

Figure 2.8: Wall 0011

Figure 2.9: Wall 0100

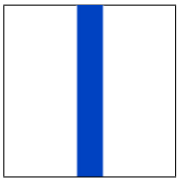

Figure 2.10: Wall 0101

Figure 2.11: Wall 0110

Figure 2.12: Wall 0111

Figure 2.13: Wall 1001

Figure 2.14: Wall 1010

Figure 2.15: Wall 1011

Figure 2.16: Wall 1100

Figure 2.17: Wall 1101

Figure 2.18: Wall 1110

Figure 2.19: Wall 1111

The first bit represents whether connection is to be made on the right, second whether on the top, third whether on the left and fourth whether on the bottom. As wall 0000 would be very small and hard to maneuver around for user, drawing of such is unsupported. I have actually tested moving around such and even my manual and keyboard skills are relatively good, I found it hard to move around such.

Algorithm deciding which type of wall is presented below:

```
def walltypecheck(location):
  r = t = l = b = 0
  if location[0] != 0:
    if level[location[0]-1][location[1]] == wall:
```

```
        l = 1
    if location[0] != 31:
      if level[location[0]+1][location[1]] == wall:
        r = 1
    if location[1] != 0:
      if level[location[0]][location[1]-1] == wall:
      t = 1
    if location[1] != 31:
      if level[location[0]][location[1]+1] == wall:
        b = 1
    return str(r) + str(t) + str(l) + str(b)

```

For reference of how level data is stored see section 2.4. This algorithm returns type of wall that would fit in the spot location provided as an argument while calling the function. Output format is a string corresponding with wall names described above.

2.3.2 'Player'

Player is represented by an iconic PacMan figure of my own design (to avoid direct copying of work of others). The tile rotates depending on the direction of player's movement so that the 'mouth' is always facing forward. Color has been slightly darkened as I find this design a little bit nicer than the original.

Figure 2.20: Player sprite

2.3.3 'Ghosts'

There are three types of ghosts in the game, red, green and blue. They are similar in design, though they behavior is different. (described later)

Figure 2.21: Red ghost sprite

Figure 2.22: Green ghost sprite

Figure 2.23: Blue ghost sprite

2.3.4 'Eatables'

Coins

Coin spawning works a little bit different than spawning other tiles, which is described later. They actually appear in every tile **not** designated as 'wall', 'heart' or 'empty'. They are static and their sprites are killed when player enters their tile resulting in coins-eaten counter value increasing.

Figure 2.24: Coin graphics

Hearts

Hearts are similar to coins in terms of being eaten, but they spawn in designated places and their consumption increases lives counter.

2.3.5 'Empty tiles'

These tiles are actually not drawn. They only act as an abstract concept to hold information that coin is not to be drawn on this particular grid tile. Player and ghost can move through this tile. For user these appear as plain dark blue space - the color of the background.

2.4 Levels

2.4.1 'Permanent storing'

It took me a while to actually figure out how to store level blueprints. The container had to be a 32x32 matrix of values. Python lists of lists of vari-

ables would actually be the most efficient way to store types of tiles on each position in the grid. I have actually tried this approach back when my original board was still 64x64 tiles and it didn't take me 10 tiles of brackets and commas to figure out, that maybe this approach is most efficient for storing, but it is absolutely to slow to prepare the level of 4096 tiles. I have decided that I need an editing software for these grids. Then I understood that my levels, grids of values are actually bitmaps. I have prepared an exemplar of a level in GIMP and using my previous experience with Pillow module I have prepared a script to load these bitmaps to my Python program. It all took less time that it would take to fill $\frac{1}{10}^{\text{th}}$ of that list of lists of values manually.

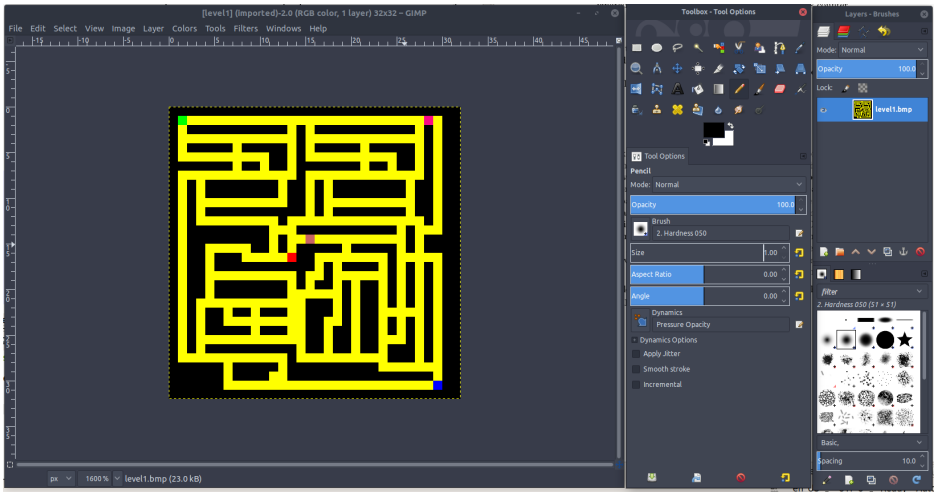

Figure 2.25: An exemplar of a level being edited in the GIMP software

Each tile has its representation in a particular color in RGB notation:

Empty tile	(255, 255, 255)
Wall	(0, 0, 0)
Coin	(255, 255, 0)
Player	(255, 0, 0)
Red Ghost	(200, 100, 100)
Green Ghost	(0, 255, 0)
Blue Ghost	(0, 0, 255)

2.4.2 'Storing while in game'

Do you remember my first concept for the most optimal storage of levels mentioned earlier? Well, that didn't work for storing levels as files... But in game while loaded in RAM - works like a charm. List of lists of integers,

just like that:

[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
[1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
[1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
[1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 2, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1],
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
[1, 4, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

Where every value represents a different type of tile.

Empty tile	0
Wall	1
Coin	2
Player	3
Red Ghost	4
Green Ghost	5
Blue Ghost	6

2.4.3 'Interpreting'

Figure 2.26: First tests of interpretation of level files. Note that coins are formed into letters LU, RU, LB, RB as in words Left, Right, Upper, Bottom to load the tiles in the right orientation. Old 64x64 grid is used.

These tables are then interpreted for program to know where to place which sprites, therefore the following algorithm is used:

```
if level[column][row] = wall
  spawn wall at location column, row
elif level[column][row] = player
  spawn player at location column, row
elif level[column][row] = coin
  spawn coin at location column, row
elif level[column][row] = ghost
  gtype = check ghost color
  spawn ghost of color gtype at location column, row
elif level[column][row] = heart
  spawn heart at location column, row

```

2.4.4 'Designing the levels'

Process of level design is quite easy and only limitation one needs to remember is that all coin - having tiles must be accessible by a player - not surrounded by a wall. The rest is just a subjective approach to the difficulty. As not all ghosts have to be used, the first few levels have one or two ghosts at most, making them easier. Through alpha testing I have noticed that long passages are traps, as one ghost may approach player from one side and other from the other and there is nowhere to hide. Also it's easier for ghosts to navigate through complex maze and these might be finger-tangling even for advanced players. See Figure 2.25 for a screenshot of a design process.

2.4.5 'User-defined levels'

As levels can be created using a very basic bitmap editing software, user might actually add their own levels. And use them instead of the built-in ones.

2.4.6 'Progressing to the next level'

When player eats all coins, they advance to the next level. At this stage a message screen (described later) appears and all level - loading procedures are called. Also counters of coins eaten and total are reset at this point.

If the level completed was the last level, a congratulations screen is displayed and the game eventually quits.

See Section 2.6 for reference on messages signaling level change and completing the game.

2.5 Gameplay scheme

The game works on a concept of a loop, that (when run on a sufficiently fast machine) executes 60 times per second. During that time multiple conditions are checked and different procedures are called.

2.5.1 'Introduction to the concept of time segments'

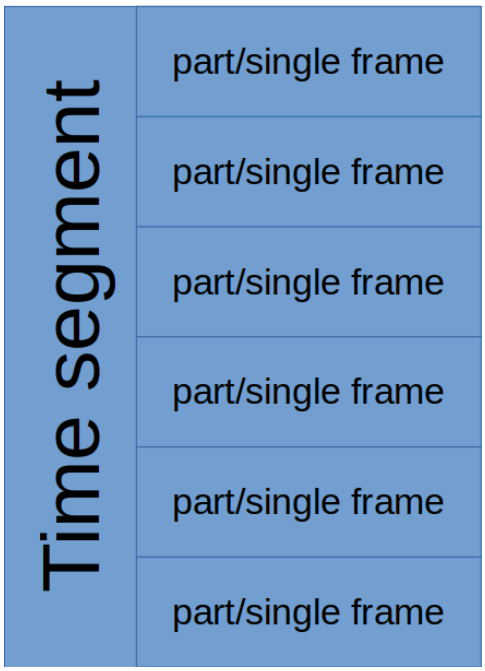

Figure 2.27: Scheme depicting relation between a time segment and parts

It's actually not required to compute some values every each of these 60 loop repetitions per second. An example - ghost moves $\frac{1}{10}^{\text{th}}$ of a tile in that time, it still can't turn so its path to the unchanged player location will remain the same. That's why I have introduced "Time segments". These are discrete representations of time period which is required for a ghost or a player to move a length of one tile. As ghost are slower, they have 5 of these in a second and player has 6.

Whilst processing single frames is still needed, these were called parts, whilst introducing 'part' variable determining which part of time segment is currently executed. See Figure 2.27 for reference.

2.5.2 'Player movement'

Every player's time segment the following code is called:

```
1 # ----- checking pressed keys -----
```

```
keys = pygame.key.get_pressed()
# ------ wall collision check ------
surroundings = walltypecheck(player.location)

```

It's worth noting that I have reused walltypecheck to look for surroundings of the player for walls. See section 2.3.1 for reference on how this algorithm works.

Then check is performed whether player's move is legit (wall does not obstruct it).

```
if keys[pygame.K_LEFT] and surroundings[2] == '0':
  player.move('left', player_part)
elif keys[pygame.K_RIGHT] and surroundings[0] == '0':
  player.move('right', player_part)
elif keys[pygame.K_UP] and surroundings[1] == '0':
  player.move('up', player_part)
elif keys[pygame.K_DOWN] and surroundings[3] == '0':
  player.move('down', player_part)

```

Please note usage of the player_part variable. It is the value of frames that have passed since last player segment begun (as described in Section 2.5.1). It's useful to determine whether a new time segment is to be started and for player's moving routine to know at which part of tile it should position player's sprite inducing a fluency of movement.

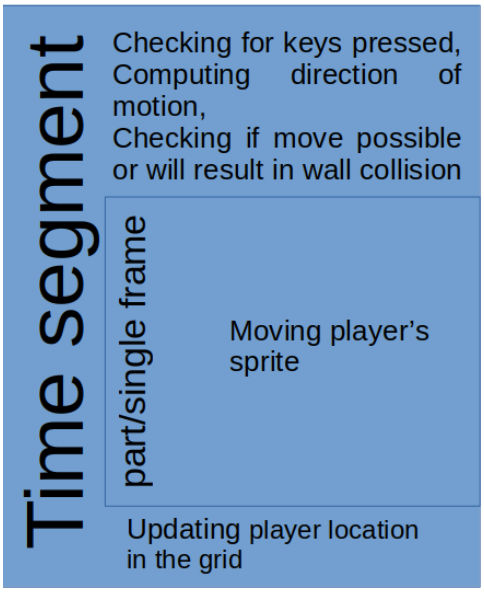

Figure 2.28: Diagram showing how tasks are executed either once per time segment or once per part

2.5.3 'Ghost movement'

Ghost movement is a far more complex algorithm than player's movement though it uses same concept of time segments.

As mentioned in Success Criteria 2a, I wanted the ghosts to be more intelligent than the originals. I have read (Reference 1) that the original algorithm is really primitive. It checks from which of surrounding tiles ghost will be straight-line closest to the player and moves there.

The graph

First idea of mine was to make the ghost actually pick the shortest path to the player. That would have to involve implementation of a graph and a pathfinding algorithm.

As I already had my level-keeping structure in place I have decided to use a similar one to store my graph. It is represented as a list of lists of lists of location tuples. It looks confusing so...
A list (index i) of lists (index j) is used so that all nodes accessible from node of location (i, j) can be stored. Then that next list stores tuples (x, y) of locations accessible from that node. Object is named 'thegraph' throughout the program (as it is the only graph implementation in it). Note that it is a global object.

It's obviously individual for each level and is built every time a level is loaded. Through the following routine:

```
1  def graphbuilder():
2    global level, thegraph
3    thegraph = [[[] for i in range(32)] for j in range(32)]
4    for i in range(32):
5      for j in range(32):
6        if level[i][j] != wall:
7          location = [i, j]
8          if location[0] != 0:
9            if level[location[0] - 1][location[1]] !=   wall:
10             thegraph[location[0]][location[1]] += [[location[0]
     - 1, location[1]]]
11         if location[0] != 31:
12           if level[location[0] + 1][location[1]] != wall:
13             thegraph[location[0]][location[1]] += [[location[0]
     + 1, location[1]]]
14         if location[1] != 0:
15           if level[location[0]][location[1] - 1] != wall:
16             thegraph[location[0]][location[1]] += [[location[0],
      location[1] - 1]]
17         if location[1] != 31:
18           if level[location[0]][location[1] + 1] != wall:
19             thegraph[location[0]][location[1]] += [[location[0],
      location[1] + 1]]
20
```

Pathfinding

This part I found actually the hardest. All my previous experience with graphs and pathfinding was actually from old times when my favorite language was C++. Apparently when I switched to Python, my rusty knowledge on the topic had to be refreshed. My first approach (a stupid one) was to use DFS search to check all possible paths and decide which one to use.

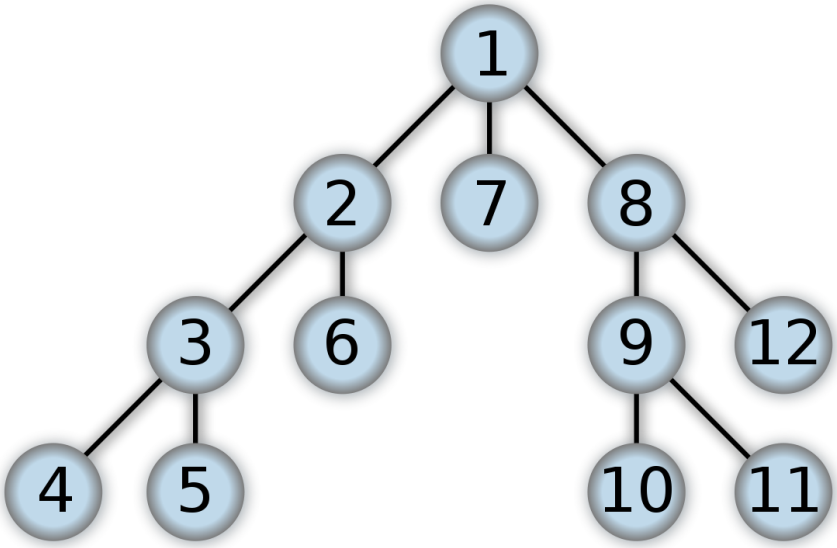

Figure 2.29: Diagram showing order of nodes visits starting from node 1 using DFS algorithm Source: wikipedia.org

It was a rather simple recursion.

```
def findpath(start, end, path):
  append start to path
  if start == end:
    return path
  else:
    for neighbour in neighbours(start): # neighbours easily
    obtained from thegraph
      findpath(neighbour, end, path)

```

That was stupid and very slow, as I have noticed that my graph is not actually a tree and paths may have cycles, this process never actually worked.

And then I realized that a more appropriate solution was to use BFS and implement whether a new node was actually visited before.

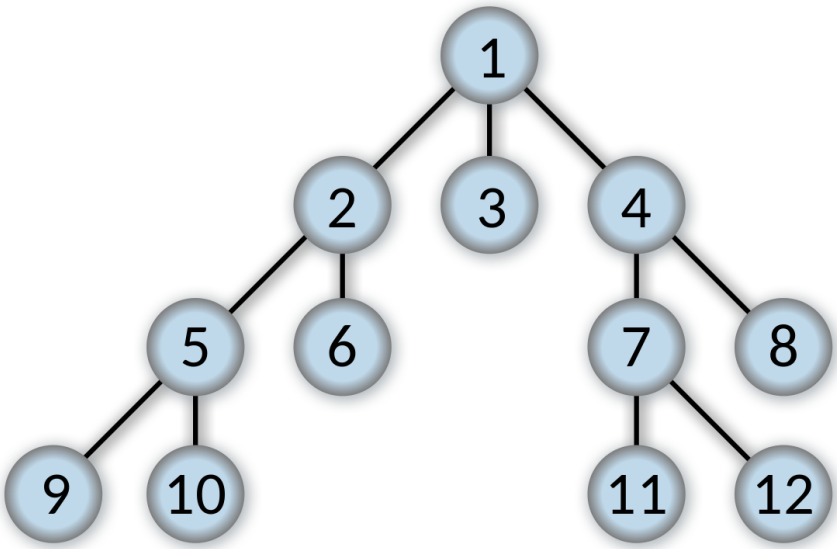

Figure 2.30: Diagram showing order of nodes visits starting from node 1 using BFS algorithm Source: wikipedia.org

I also dropped recursion as advised by online forums... Maybe recursion is a way in C++ but in Python it apparently is really slow. Also my own testing of recursive approach sometimes lead to stack overflows, which disqualified this method completely.

A queue of nodes to visit was implemented and new algorithm came to life. It was fast but its output of a complete path to player was actually an overkill - why would I need it if it will become obsolete in the next time segment when player location changes?

The next iteration actually begins BFSing from the player, until it finds any of the tiles surrounding the ghost. It then returns location of such tile - next tile ghost has to move to.

```
1  def find_next_move(start, end, forbidden):
2    visited = []
3    queue = []
4    neighbours = thegraph[start[0]][start[1]].copy()
5    queue.append([end[0], end[1]])
6    visited.append(end)
7    for point in queue:
8      if point not in visited:
9        visited.append(point)
10     if point in neighbours:
11       return point
12     for node in thegraph[point[0]][point[1]]:
13       if node not in visited:
14         queue.append(node)
15
```

As this approach worked, I was satisfied with its speed and overall performance. Then in alpha testing I found a strange occurrence I haven't noticed in the original PacMan - ghosts could reverse.

I didn't like it, so the algorithm had to exclude ghost's previous location from its neighboring tiles. The next version looked like this:

```
def find_next_move(start, end, forbidden):
    visited = []
    queue = []
    neighbours = thegraph[start[0]][start[1]].copy()
    neighbours.remove([forbidden[0], forbidden[1]])
    queue.append([end[0], end[1]])
    visited.append(end)
    for point in queue:
        if point not in visited:
            visited.append(point)
        if point in neighbours:
            return point
    for node in thegraph[point[0]][point[1]]:
        if node not in visited and node != [forbidden[0],
    forbidden[1]]:
    queue.append(node)

```

Difference between ghosts

As I have generated some levels and started testing, I found out that if two ghost share location, they effectively become one as their optimal paths will be the same and they will always move the same way. That of course wasn't the effect I wanted, I had to figure out how to make the ghosts behave different. But how to make ghosts both smart and not going straight for the player? Internet for the win, meme pages came with help.

Figure 2.31: Popular meme. Source: pinterest.com

Now enlightened by a piece of digital artwork originating from Star Wars (see Figure 2.31), I decided that my ghosts have to behave like they were to surround the player. How to achieve such effect? I turned to the original PacMan solution - not moving to the exact tile the player occupies. A decision was made to leave the red ghost originally smart while blue and green ghosts were to move to tiles shifted by (-2, -2) and (2, 2) vectors from the player. The result was amazing, ghosts seemed to move separately while away from the player, when they approach it, I had a sense that this is not what I programmed, **They hunt in a pack!** I had an authentic experience of ghosts setting a trap, green and blue were blocking my exit routes, when red went straight for me.

But it sometimes crashed, I wondered what was the case and then I found out that the ghosts were actually trying to go to a tile, which is either a wall (inaccessible) or outside the board (object level was referenced with an invalid index, either negative or exceeding it's length). I had to write a piece of algorithm to look for a nearest tile that is actually not wall.

```
1 def find_nearest_not_wall(point):
2     if point[0] > 31:
3         point[0] = 31
4     if point[0] < 0:
5         point[0] = 0
6     if point[1] > 31:
```

```
7           point[1] = 31
8       if point[1] < 0:
9           point[1] = 0
10      radius = 1
11      if level[point[0]][point[1]] != wall:
12          return point
13      while True:
14          for i in range(point[0]-radius, point[0]+radius):
15              for j in range(point[1]-radius, point[1]+radius):
16                  try:
17                      if level[i][j] != wall:
18                          return [i, j]
19                  except IndexError:
20                      pass
21          radius += 1
22
```

It first moves the point to the nearest one on the board (lines 2 to 9). Then it checks whether such point is a wall, if not, returns it (lines 11 and 12). If it was a wall it starts searching surroundings of such point in a fixed radius, starting from 1. First point it finds not to be wall is then returned.

Later it had to be implemented in the pathfinding algorithm, which now looks like this:

```
1  def find_next_move(start, end, forbidden):
2      visited = []
3      queue = []
4      end = find_nearest_not_wall(end)
5      neighbours = thegraph[start[0]][start[1]].copy()
6      neighbours.remove([forbidden[0], forbidden[1]])
7      queue.append([end[0], end[1]])
8      visited.append(end)
9      for point in queue:
10         if point not in visited:
11             visited.append(point)
12         if point in neighbours:
13             return point
14     for node in thegraph[point[0]][point[1]]:
15         if node not in visited and node != [forbidden[0],
       forbidden[1]]:
16     queue.append(node)
17
```

Note line 4. where final point is substituted with a one closest to it being actually accessible.

Calling pathfinding algorithm

Now, every ghosts' time segment (note that all ghosts share a single time segment), a new destination tile is calculated for each ghost and pathfinding is called to determine which tile should the ghost move to throughout the time segment.

```
for ghost in ghosts_list:
    if ghost.color == 'red':
        ghost.nexttile = find_next_move(ghost.location, player.
    location, ghost.previouslocation)
    elif ghost.color == 'blue':
        ghost.nexttile = find_next_move(ghost.location,
                                        [player.location[0] + 2,
                                         player.location[1] +
    2],
                                        ghost.previouslocation)
    elif ghost.color == 'green':
        ghost.nexttile = find_next_move(ghost.location,
                                        [player.location[0] - 2,
                                         player.location[1] -
    2],
                                        ghost.previouslocation)

```

Note how find_next_move function takes arguments of:

1. Ghost location
2. Target location
3. Previous ghost location - forbidden tile as ghosts can't reverse.

Ghosts speed

As I had ghosts chasing me, I have figured out that the player actually can't outrun them. And every suboptimal move of the player led to red ghost getting closer and closer. I had the smartest ghost possible, but it had an advantage of having no means of being killed. It was a too powerful opponent. I didn't want to add power-ups for the player to either become faster or to be able to frighten/kill the ghosts. Instead, as derived from the Interview (see Section 1.4), I decided to make the ghosts move at $\frac{5}{6}^{\text{th}}$ of the player's speed. I found the gameplay to be quite optimal and actually started playing the game which at this stage was only running from the ghost. But with a stopwatch I was challenged to keep my distance from them for quite a long time. I finally knew that this is what I was aiming for.

2.5.4 'Handling movement animation'

This section is common for both ghosts and a player as it actually derives from the same code. As mentioned in Section 2.5.1, the part part is a smaller piece of time segment. During a single part no routes are computed for ghosts, they only move from one tile to another, being able to actually be rendered in between these tiles to make the Movables (ghosts + player sprites) move fluently.

```
1  def move(direction,
2          segmentsize,
3          part):
4    # handling multiple direction formats
5      if direction == 'up' or direction == (0, 1):
6          speed = (0, -1)
7      elif direction == 'down'or direction == (0, -1):
8          speed = (0, 1)
9      elif direction == 'left' or direction == (-1, 0):
10         speed = (-1, 0)
11     elif direction == 'right' or direction == (1, 0):
12         speed = (1, 0)
13     part2 = (part+1) / segmentsize #as parts are numbered from 0
14     #rect parameters of a Movable as implemented in pygame
15     rect.x, rect.y = (location[0] + speed[0] * part2) *
       tile_width,
16                      (location[1] + speed[1] * part2) *
       tile_width
17     if part == segmentsize - 1:
18     #if movement came to the end update location
19         location = location[0] + speed[0],
20                    location[1] + speed[1]
21
```

1. Speed vector is derived from parameter 'direction' (lines 4-12)
2. part2 is calculated - fraction of progress of the movement to be completed in this part (line 13)
3. Movable sprite is moved to new position (lines 14-16)
4. If the actual movement came to an end and a Movable is in a center of a next tile location is updated (lines 17-20)

2.5.5 'Eating'

Eating of both coins and hearts (Eatables) was easy to implement in terms of detection, but I had to look up how to destroy eaten sprites in the pygame documentation. I found that Sprites may be killed, which seemed to be an optimal solution to that problem. Therefore every player's time segment the following routine is called.

```
1  for eatable in eatables:
2    if eatable.location == player.location:
3      if eatable == heart:
4        lifes += 1
5      if eatable == coin:
6       coins_eaten += 1
7      eatable.kill()
8
```

For every eatable is checked for occupying the same tile as player.
Then if it is a heart, player gains a life,
if it was a coin, coins eaten counter increases.
Eatable sprite is killed.

2.5.6 'Getting killed'

Killing a player is fairly similar to eating... It's just checking for collision with a Ghost instead of an Eatable. It's also called every player's time segment **or** every ghosts' time segment. Following routine is then called:

```
for ghost in ghosts:
  if ghost.location == player.location:
    lives -= 1
    if lives == 0:
      pull Game Over screen
    Respawn every Movable
    Reset parts
    Wait 5s
    Continue game

```

For every ghost a collision with player is checked, if such is detected, process of dying begins.
Life is subtracted from lives counter.
If player has no more lives the game ends.
If there are more lives, all movables return to their original position (described in Section 2.5.7) and part counters are reset.
Game waits 5 seconds to let the user cope with the loss, and continues.

2.5.7 'Respawning after a loss of life'

This is a relatively simple process of putting all Movables to their initial position and making them not preserve their speed. It can be easily described with a following code:

```
for Movable in Movables:
    Movable.location = Movable.initiallocation
    Movable.speed = (0, 0)

```

2.6 Types of message screens

2.6.1 'Tutorial'

TODO: Photo of the screen

2.6.2 'Loss of life'

TODO: Photo of the screen

2.6.3 'Completing the level'

TODO: Photo of the screen

2.6.4 'Loosing the game'

TODO: Photo of the screen

2.6.5 'Completing the entire game'

TODO: Photo of the screen

2.7 Testing

I have decided to use two stages of testing, namely alpha (α) and beta (β).

2.7.1 'α-testing'

Chapter 3

References

1. http://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior
2. https://www.pygame.org/docs