PycMan

Jan Dziedzic

March 9, 2018

Contents

Chapter 1

Project Definition and Analysis

1.1 Project Definition

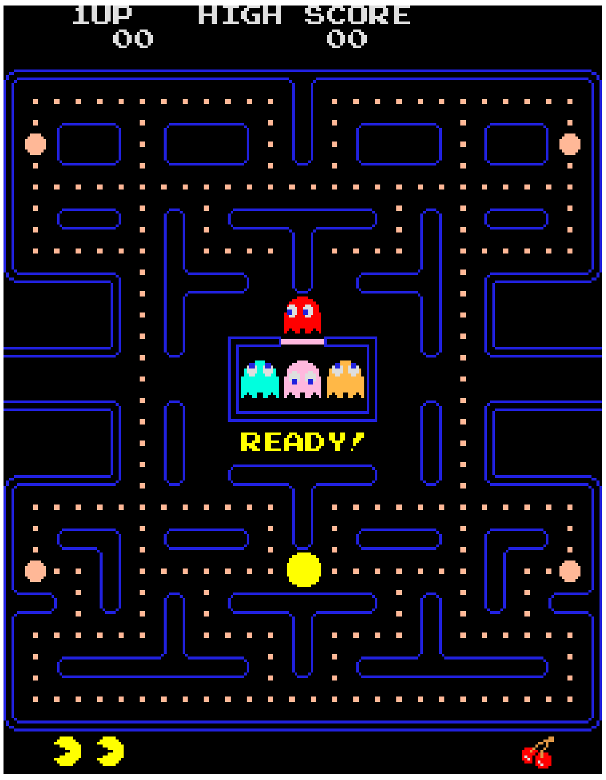


Figure 1.1: Original Pac-Man screen. Source is Reference 1.

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.

A good example of a stakeholder is my friend, Wojciech Wojtkowski, with whom I have conducted an interview to determine the requirements, see Section 1.4 for reference.

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
- LibreOffice Draw

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.

Libre Office Draw is a free office suite component meant to be used for developing multimedia presentations. I have used it for its great tool for developing flowcharts used in this document. See Figure 2.30 for example of such 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 Computer Science 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 might 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.
   (f) Player loses a life upon contact with a ghost.
   (g) Upon losing a life ghosts and the player return to their initial locations.
   (h) Each ghost behaves in a different way based on its color.
   (i) Ghosts do not reverse their direction of movement.

(j) Player progresses through a level with eating 'coins' left throughout some/all accessible places on the map.

(k) Upon eating a coin, it disappears and is not to be rendered again.

(l) Eating a coin increases point counter.

(m) Heart shaped Eatables are left in some places in some of the levels.

(n) Eating a heart grants the player extra one life.

(o) Upon completing the level (eating all 'coins') new level is loaded.

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

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

(r) 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.

(s) 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.

   (f) Player can move faster than the ghosts.

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. Each of the requirements listed is critical to either launch or play the game.

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.

Having the fact that my main tool will be the pygame module I have divided game design process into following subproblems.

- Preparing window environment for the game
- Preparing board and tiling system
- Creating player and ghost sprites
- Handling player input
- Adding Eatables and walls
- Preparing level storage-loading system
- Wall collision handling
- Eating routines
- Ghost movement system
- Advancing through levels
- Pop-up messaging

2.1 Main window layout

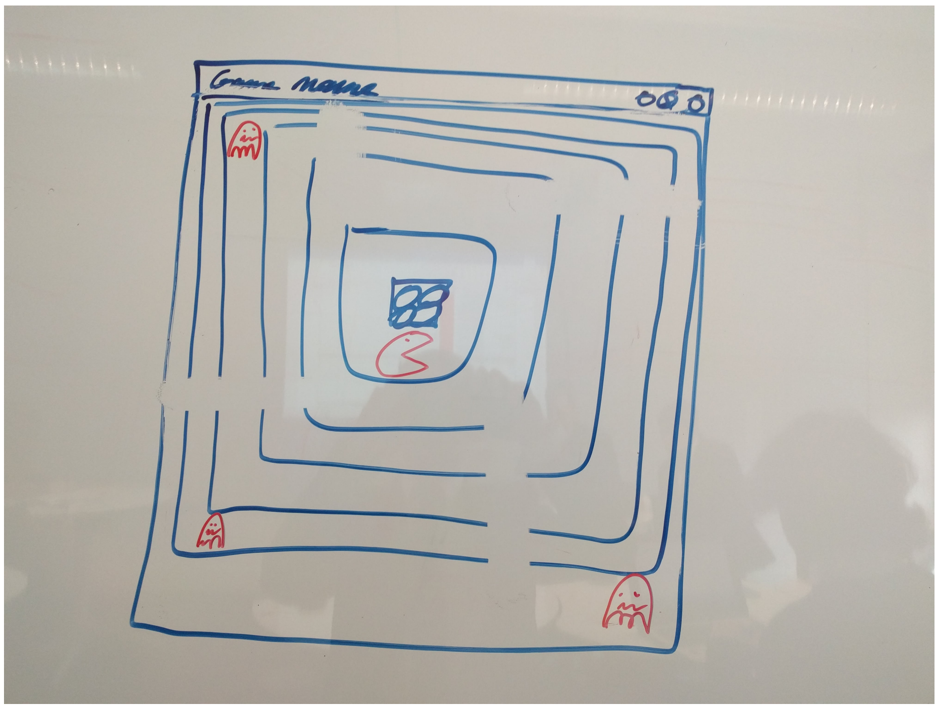


Figure 2.1: Very early design of the game window. Note the lack of counters.

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 as per requirement 1a.

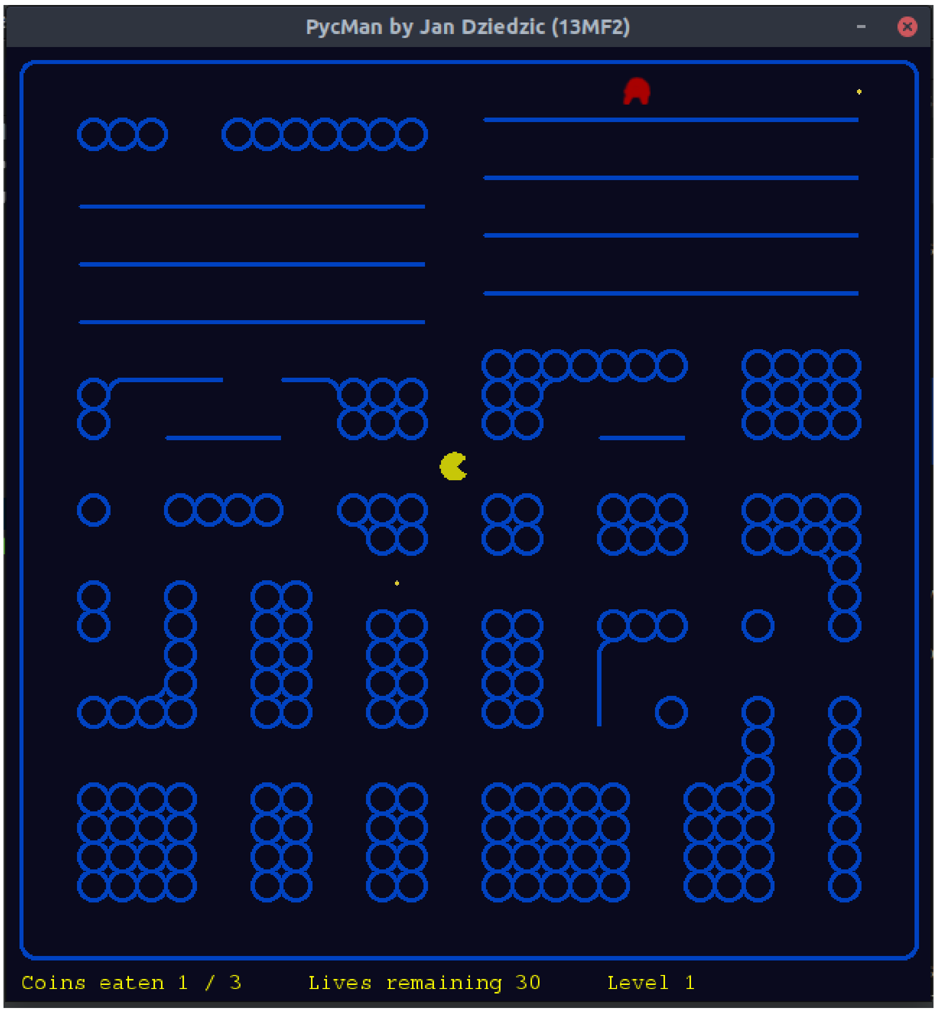


Figure 2.2: Game window rendered in Ubuntu Gnome graphical environment.

2.1.1 Board

As each level has a different layout so look of the board may vary. Please refer to Figure 2.3 for an example of such board.

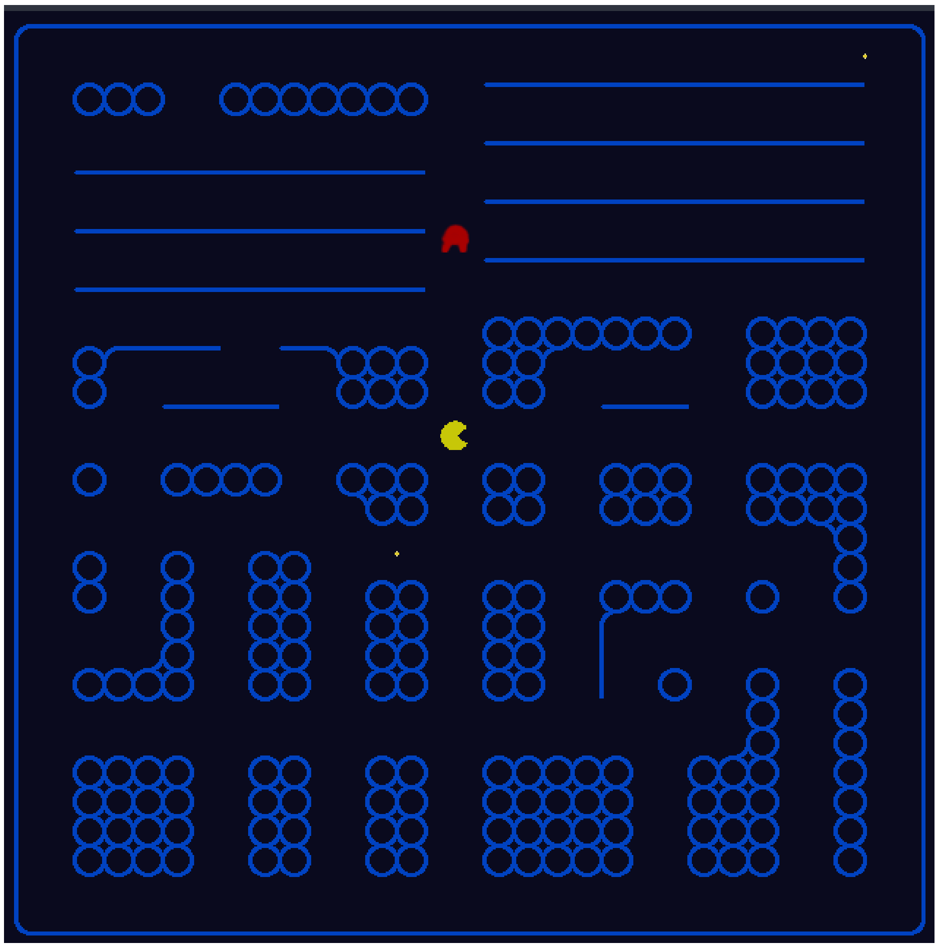


Figure 2.3: Example board. Walls, empty tiles, the player, coins and a red ghost are visible

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

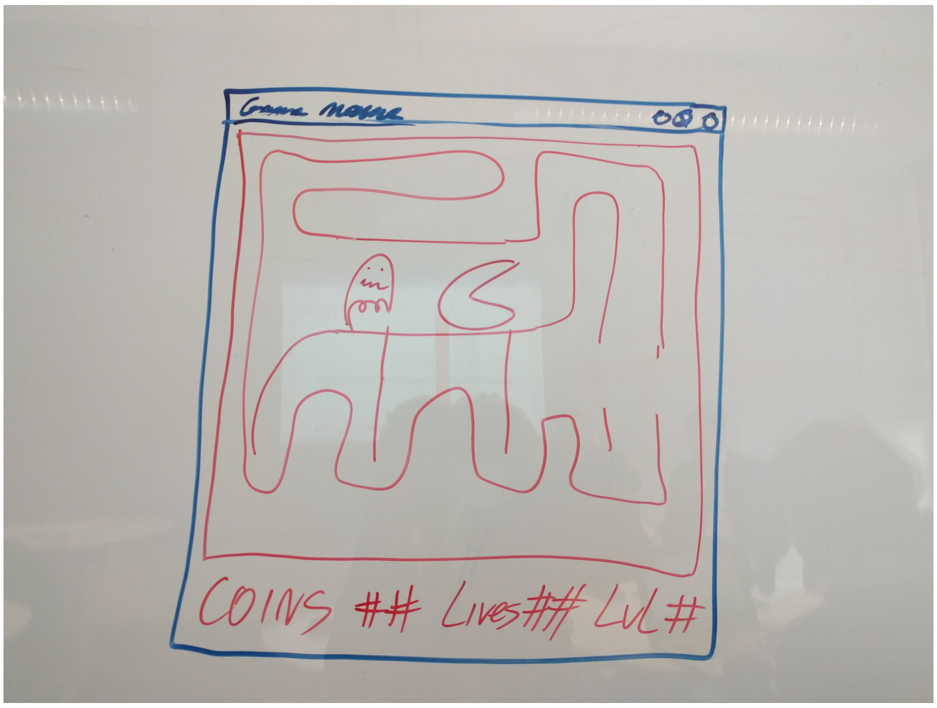


Figure 2.4: Counter placement design in its earliest stage.

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

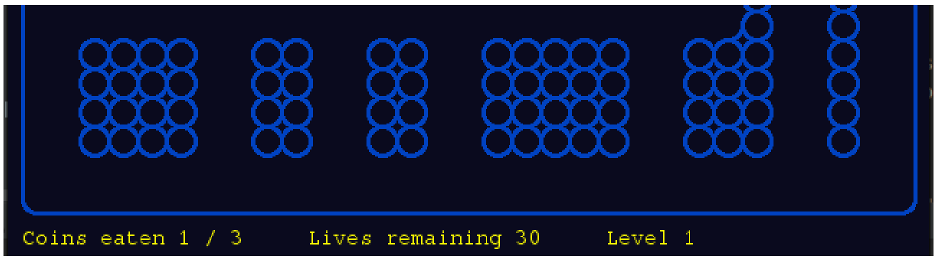


Figure 2.5: Example set of counters with part of the board included for position reference.

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.

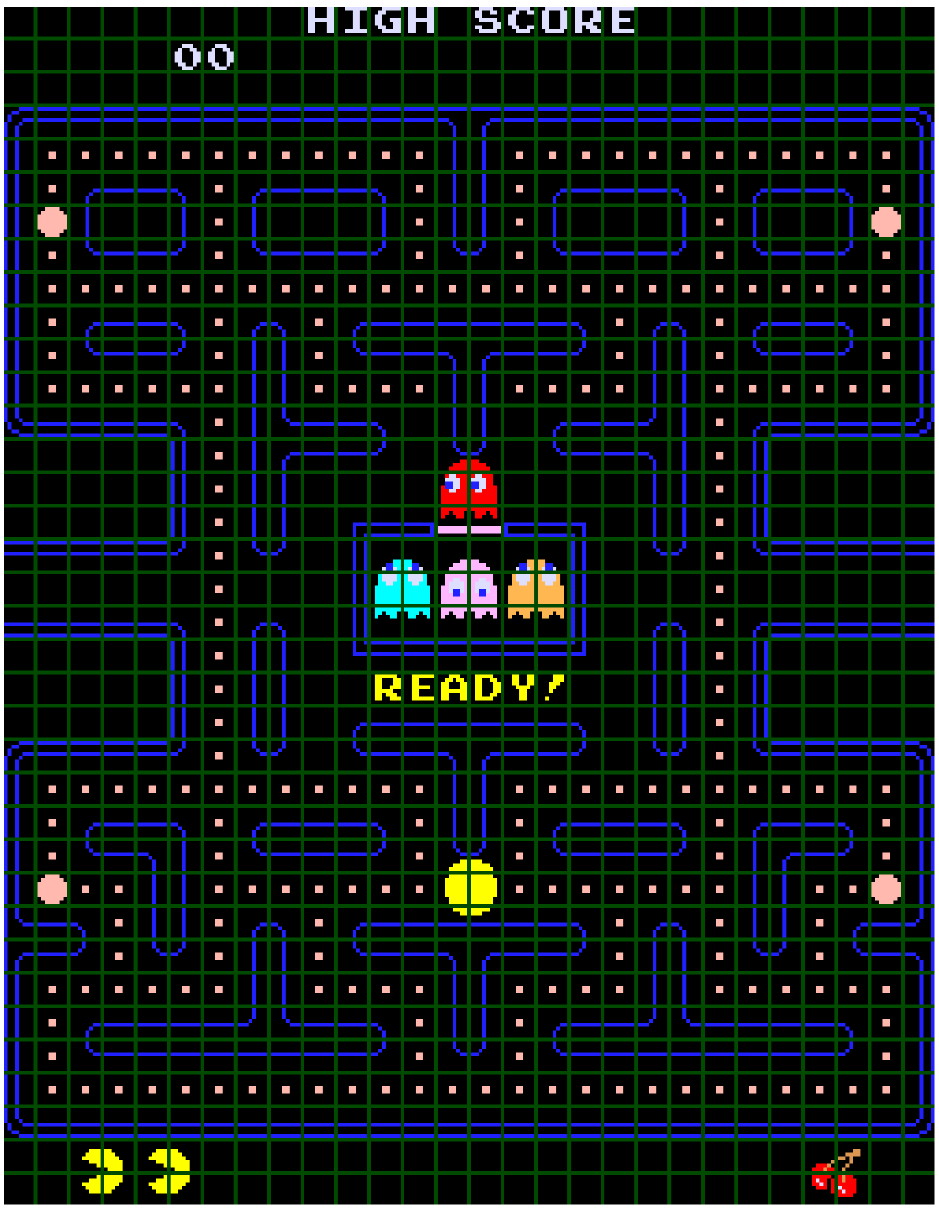


Figure 2.6: Original Pac-Man with grid layout depicted. Source: Reference 1

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).

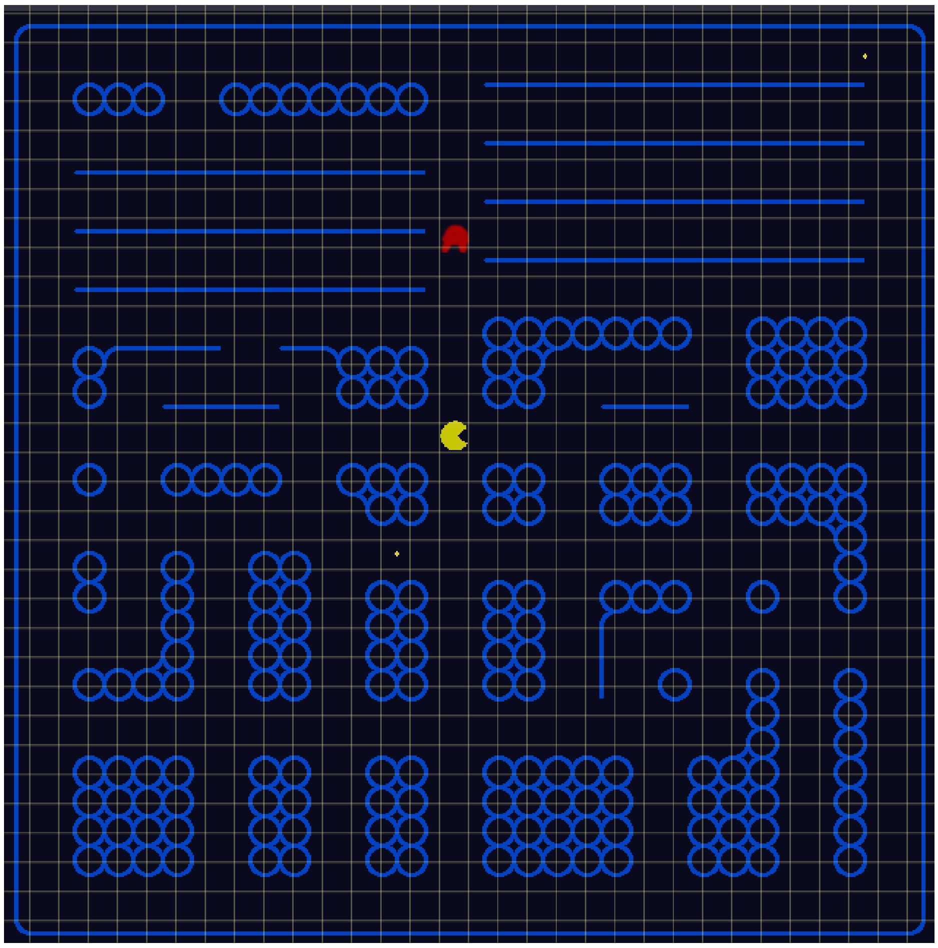


Figure 2.7: Example of a board with a grid applied (note that the grid was added just for demonstrative purposes and in not a part of the game.)

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 surroundings. 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

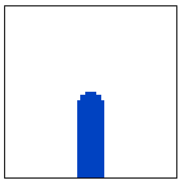


Figure 2.8: Wall 0001

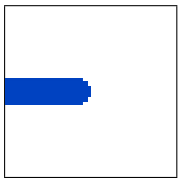


Figure 2.9: Wall 0010

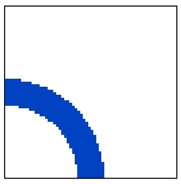


Figure 2.10: Wall 0011

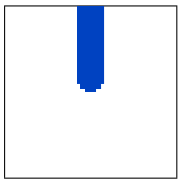


Figure 2.11: Wall 0100

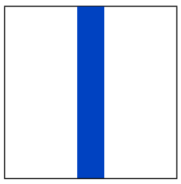


Figure 2.12: Wall 0101

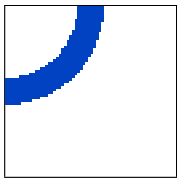


Figure 2.13: Wall 0110

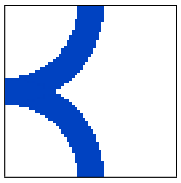


Figure 2.14: Wall 0111

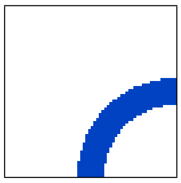


Figure 2.15: Wall 1001

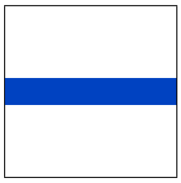


Figure 2.16: Wall 1010

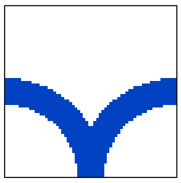


Figure 2.17: Wall 1011

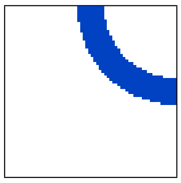


Figure 2.18: Wall 1100

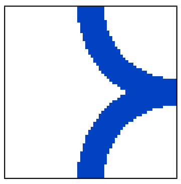


Figure 2.19: Wall 1101

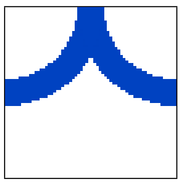


Figure 2.20: Wall 1110

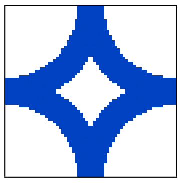


Figure 2.21: Wall 1111

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:

```python
def walltypecheck(location):
  r = t = l = b = 0
  # note that letters correspond to Right Top Left Bottom
   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 as per requirement 1c. Color has been slightly darkened as I find this design a little bit nicer than the original while still complying with requirement 1b.

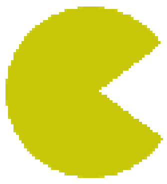


Figure 2.22: Player sprite

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)

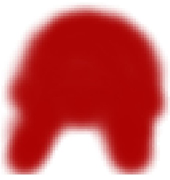


Figure 2.23: Red ghost sprite

Figure 2.24: Green ghost sprite

Figure 2.25: 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.

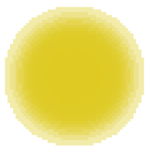


Figure 2.26: Coin graphics

Hearts

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

Figure 2.27: Heart graphics

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 variables 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.

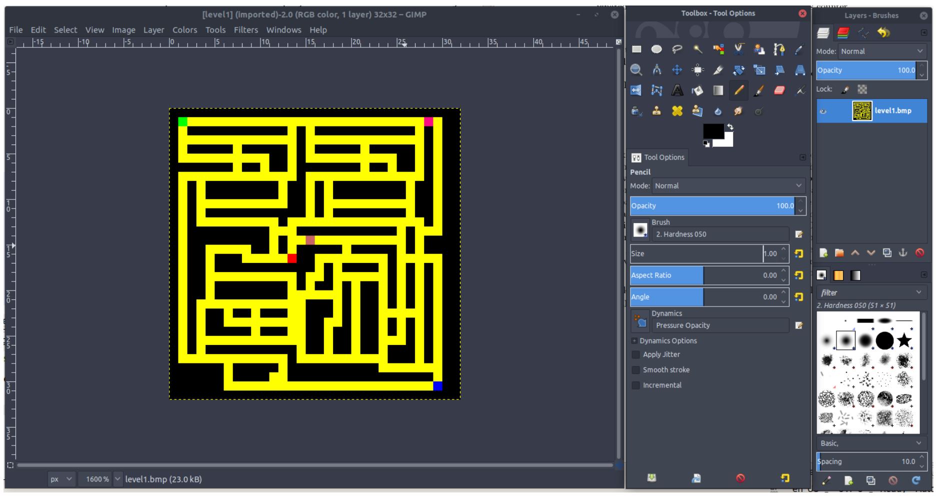


Figure 2.28: An exemplar of a level being edited in the GIMP software

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)
Heart	(200, 200, 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],
 [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, 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, 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, 0, 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, 0, 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, 0, 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, 0, 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, 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, 0, 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, 0, 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, 0, 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, 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, 0, 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, 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, 0, 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, 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, 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, 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, 0, 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, 0, 1, 1, 1, 1, 1, 0, 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, 0, 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, 1, 1, 1, 1, 1, 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, 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, 0, 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, 0, 1, 1, 1, 1, 1, 0, 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, 0, 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, 0, 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, 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, 0, 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, 0, 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, 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, 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
Heart (not included in this level)	7

2.4.3 Interpreting

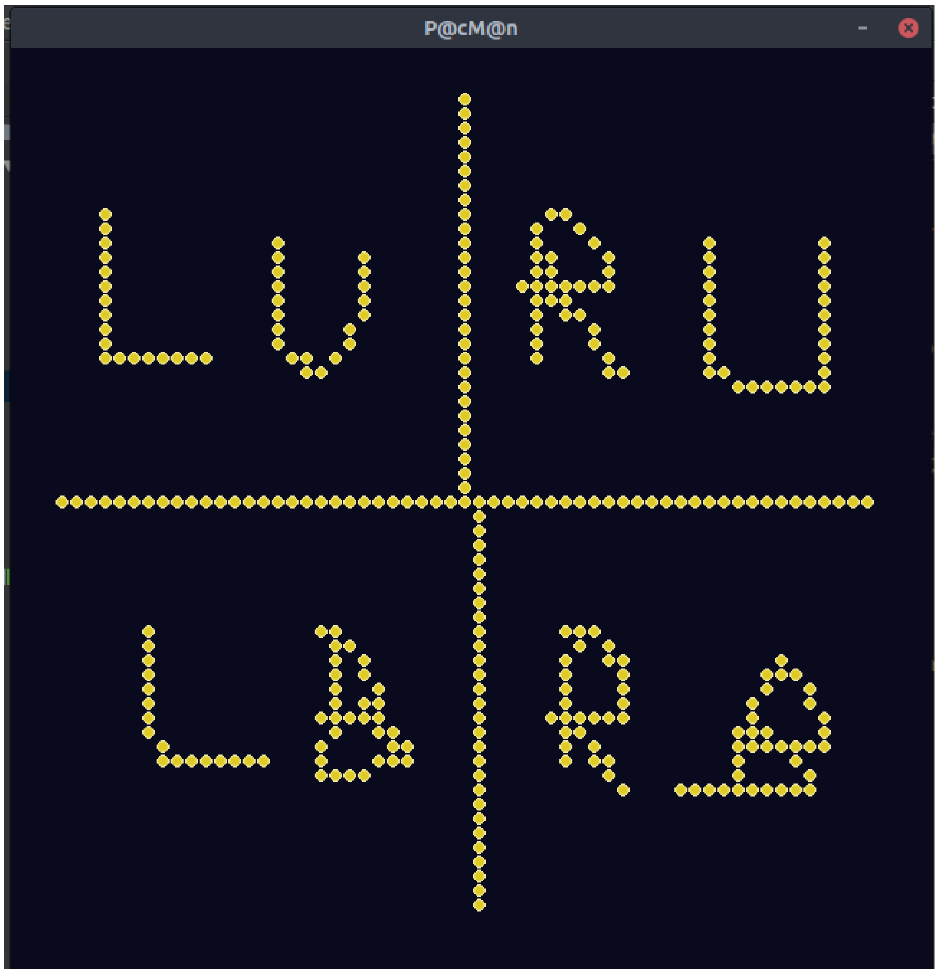


Figure 2.29: First tests of interpretation of level files. Note that coins are formed into letters LU, RU, LB, RB as in words Left, Right, Upper, Bottom to load the tiles in the right orientation. Old 64x64 grid is used.

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 2.28 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 or as an addition to them.

Self-adding levels instruction is available as an appendix in Section 7 7.

2.4.6 Progressing to the next level

When player eats all coins, they advance to the next level, fulfilling Requirement 1j and eventually 1o. 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

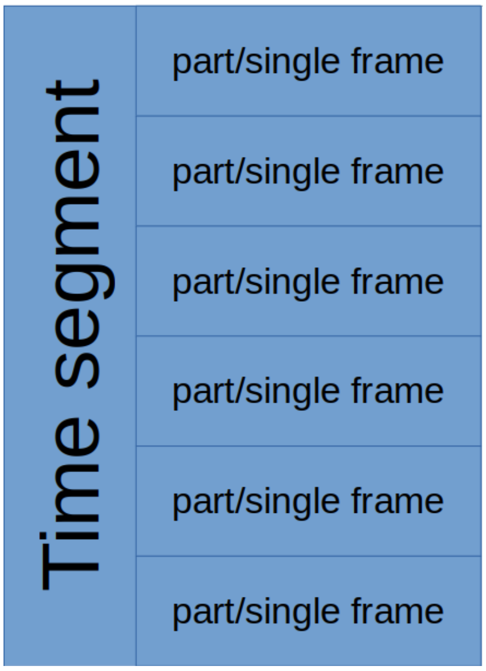


Figure 2.30: Scheme depicting relation between a time segment and parts

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 2.30 for reference.

2.5.2 Player movement

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

```
# ——— 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 left key pressed and tile to the left is not a wall:
  player.move('left', player_part)
elif right key pressed and tile to the right is not a wall:
  player.move('right', player_part)
elif up key pressed and tile above is not a wall:
  player.move('up', player_part)
elif down key pressed and tile below is not a wall:
  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.

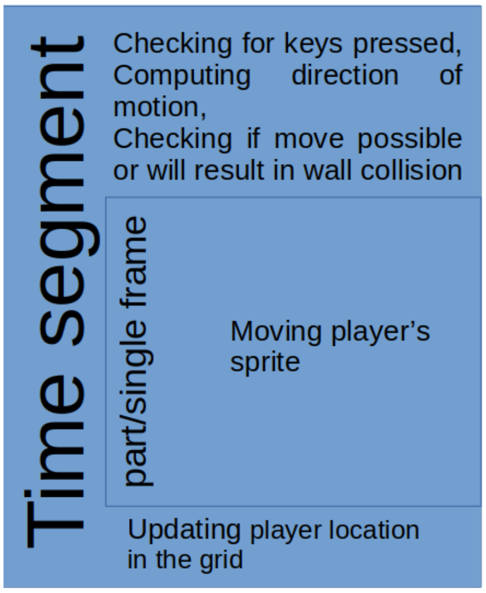

Time segment
Checking for keys pressed, Computing direction of motion, Checking if move possible or will result in wall collision
part/single frame
Moving player's sprite
Updating player location in the grid

Figure 2.31: 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 Requirement 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. This would both fulfill Requirements 1e and 2a 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:

```
def graphbuilder():
  global level, thegraph
  thegraph = [[[] for i in range(32)] for j in range(32)]
  for i in range(32):
    for j in range(32):
      if level[i][j] != wall:
        location = [i, j]
        if location[0] != 0:
          if level[location[0] - 1][location[1]] !=   wall:
            thegraph[location[0]][location[1]] += [[location[0]
 - 1, location[1]]]
        if location[0] != 31:
          if level[location[0] + 1][location[1]] != wall:
            thegraph[location[0]][location[1]] += [[location[0]
 + 1, location[1]]]
        if location[1] != 0:
          if level[location[0]][location[1] - 1] != wall:
            thegraph[location[0]][location[1]] += [[location[0],
   location[1] - 1]]
        if location[1] != 31:
          if level[location[0]][location[1] + 1] != wall:
            thegraph[location[0]][location[1]] += [[location[0],
   location[1] + 1]]

```

Notably, this algorithm neither adds wall as neighbors of other tiles, nor produces neighbors for wall tiles. This successfully prevents ghosts from moving onto wall tiles, therefore fulfilling Requirement 1d.

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.

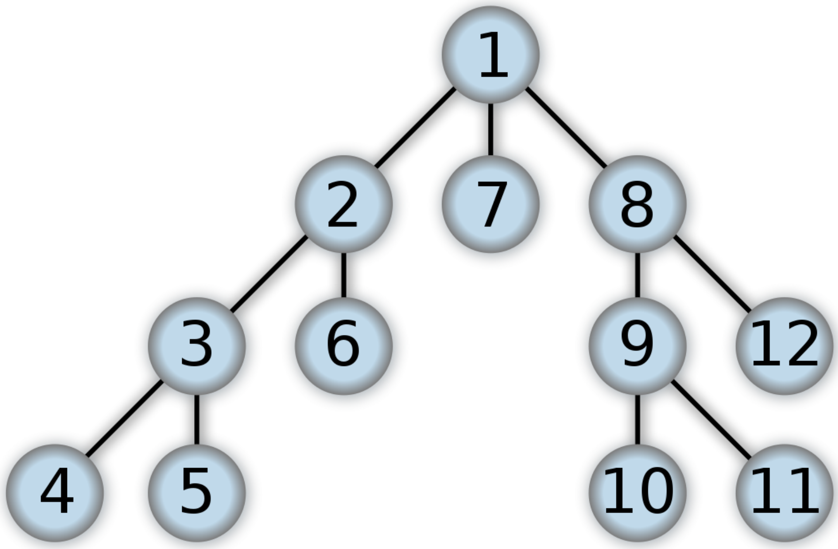


Figure 2.32: Diagram showing order of nodes visits starting from node 1 using DFS algorithm Source: wikipedia.org

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.

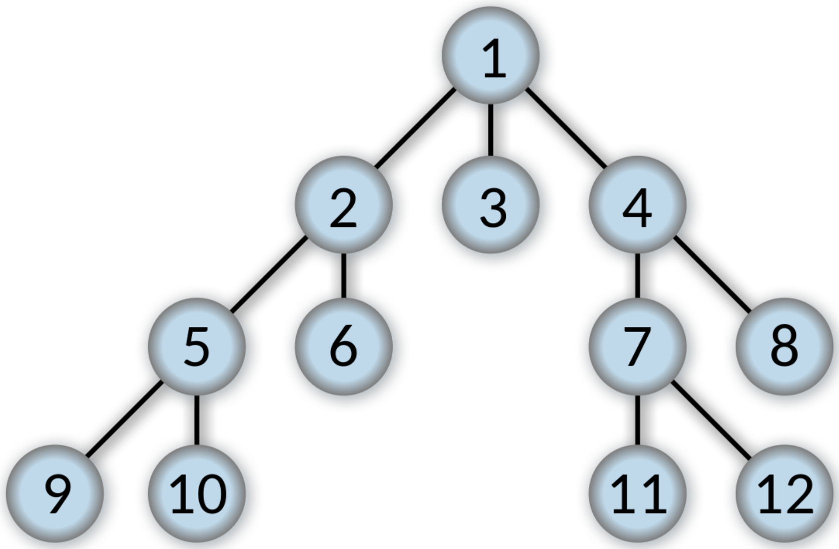


Figure 2.33: Diagram showing order of nodes visits starting from node 1 using BFS algorithm Source: wikipedia.org

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 led 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.

```python
def find_next_move(start, end, forbidden):
  visited = []
  queue = []
  neighbours = thegraph[start[0]][start[1]].copy()
  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:
        queue.append(node)

```

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:

```python
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)

```

Differences 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 2.34: Popular meme. Source: pinterest.com

Now enlightened by a piece of digital artwork originating from Star Wars (see Figure 2.34), 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. I put a big green tick mark by the Requirement 1h.

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.

```python
def find_nearest_not_wall(point):
    if point[0] > 31:
        point[0] = 31
    if point[0] < 0:
        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)

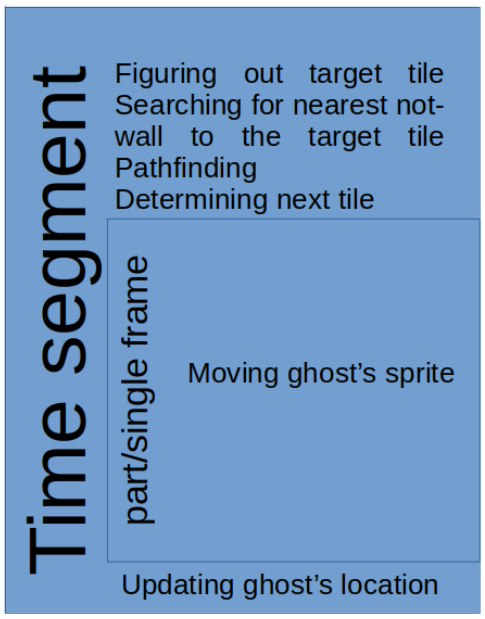


Figure 2.35: Diagram showing how tasks are executed either once per time segment or once per part (for a ghost)
See Figure 2.31 for the same for player.

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.

```
for eatable in eatables:
  if eatable.location == player.location:
    if eatable == heart:
      lifes += 1
    if eatable == coin:
     coins_eaten += 1
    eatable.kill()

```

For every eatable is checked for occupying the same tile as player.
Then if it is a heart, player gains a life, as per Requirement 1n.
if it was a coin, coins eaten counter increases, as per Requirement 1l.
Eatable sprite is killed. This fulfills Requirement 1k.

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. (As per Requirement 1f)
If player has no more lives the game ends. (Requirement 1p)
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)

```

This process is often referred to as reseting within this document as well as in the corresponding code. This function fulfills the Requirement 1g.

2.6 Types of message screens

Messages present throughout the game are displayed in a way that prevents any game event, from occurring unless the message is dismissed by pressing space.

It's done through occupying the main thread with checking for spacebar being pressed.

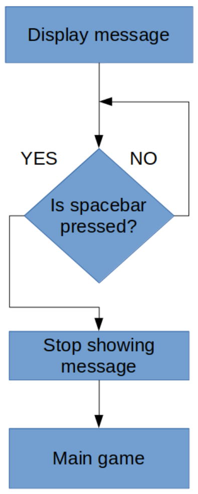
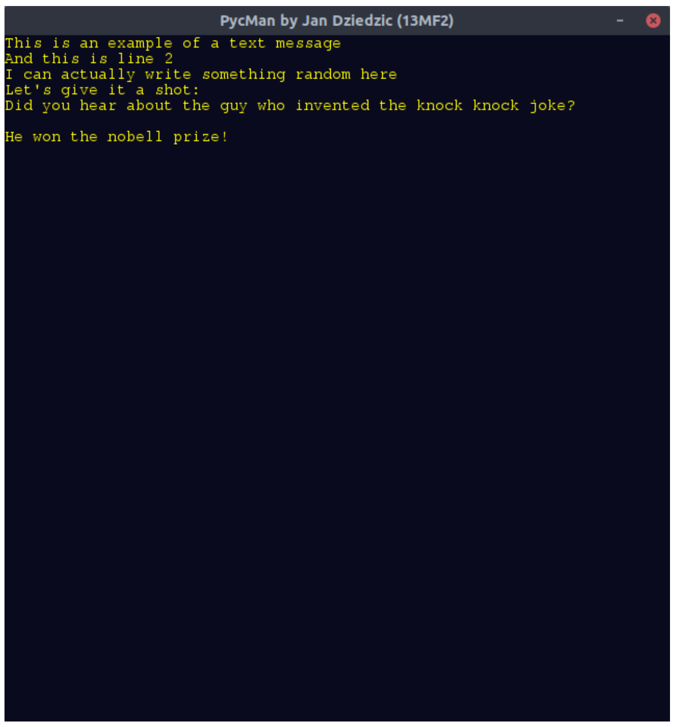


Figure 2.36: Diagram showing how game is halted until message is dismissed using spacebar

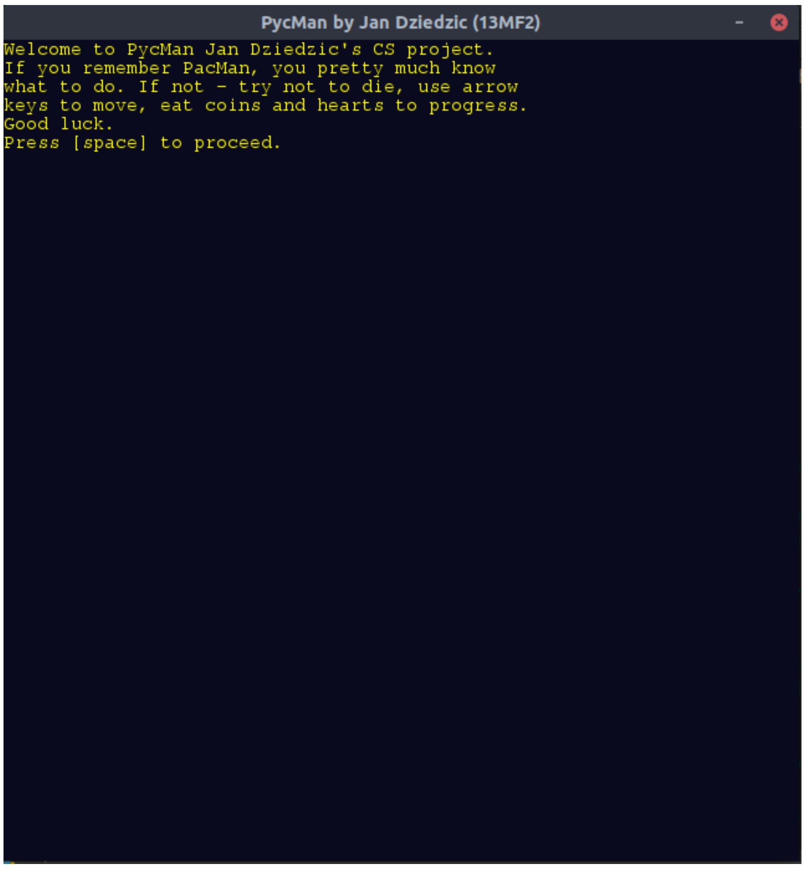


```
This is an example of a text message
And this is line 2
I can actually write something random here
Let's give it a shot:
Did you hear about the guy who invented the knock knock joke?

He won the nobell prize!
```

Figure 2.37: Exemplar of the message screen appearing in a window rendered in Ubuntu Gnome graphical environment

2.6.1 Tutorial

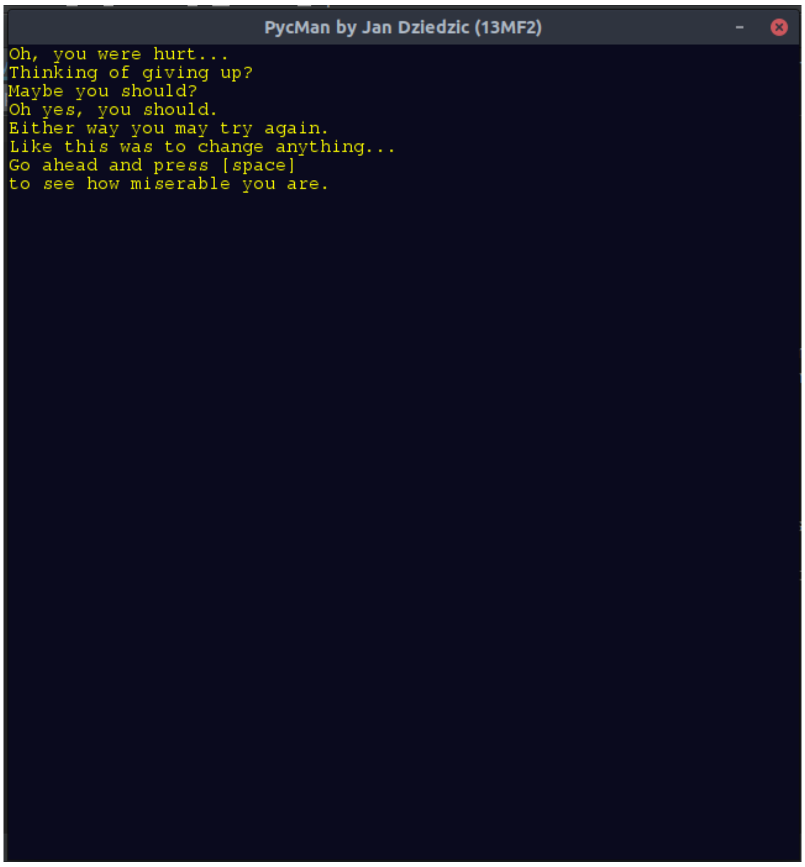


PycMan by Jan Dziedzic (13MF2)

```
Welcome to PycMan Jan Dziedzic's CS project.
If you remember PacMan, you pretty much know
what to do. If not - try not to die, use arrow
keys to move, eat coins and hearts to progress.
Good luck.
Press [space] to proceed.
```

Figure 2.38: Message displayed after starting the game. Shown in a game window rendered in the Ubuntu Gnome graphical environment.

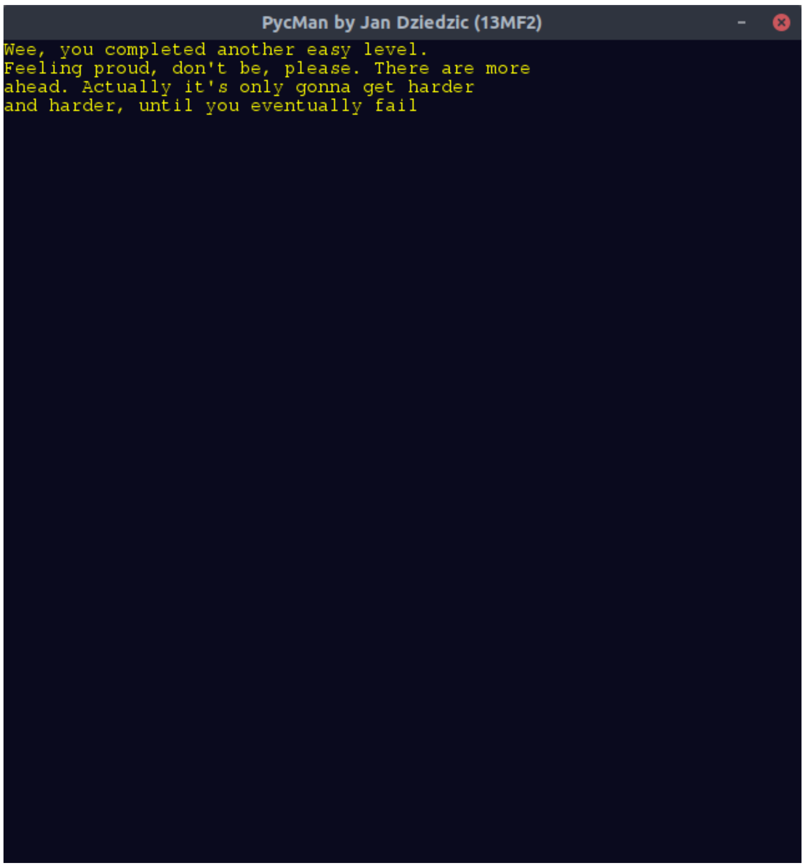
2.6.2 Loss of life



```
Oh, you were hurt...
Thinking of giving up?
Maybe you should?
Oh yes, you should.
Either way you may try again.
Like this was to change anything...
Go ahead and press [space]
to see how miserable you are.
```

Figure 2.39: Message displayed after losing a life. Shown in a game window rendered in the Ubuntu Gnome graphical environment.

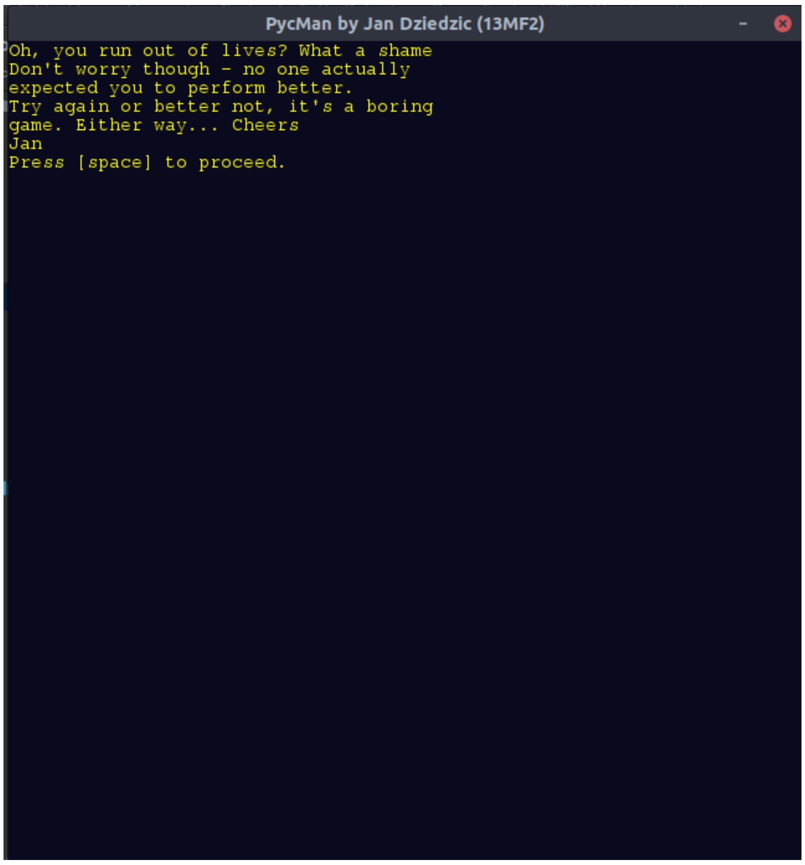
2.6.3 Completing a level



```
Wee, you completed another easy level.
Feeling proud, don't be, please. There are more
ahead. Actually it's only gonna get harder
and harder, until you eventually fail
```

Figure 2.40: Message displayed after completing a level. Shown in a game window rendered in the Ubuntu Gnome graphical environment.

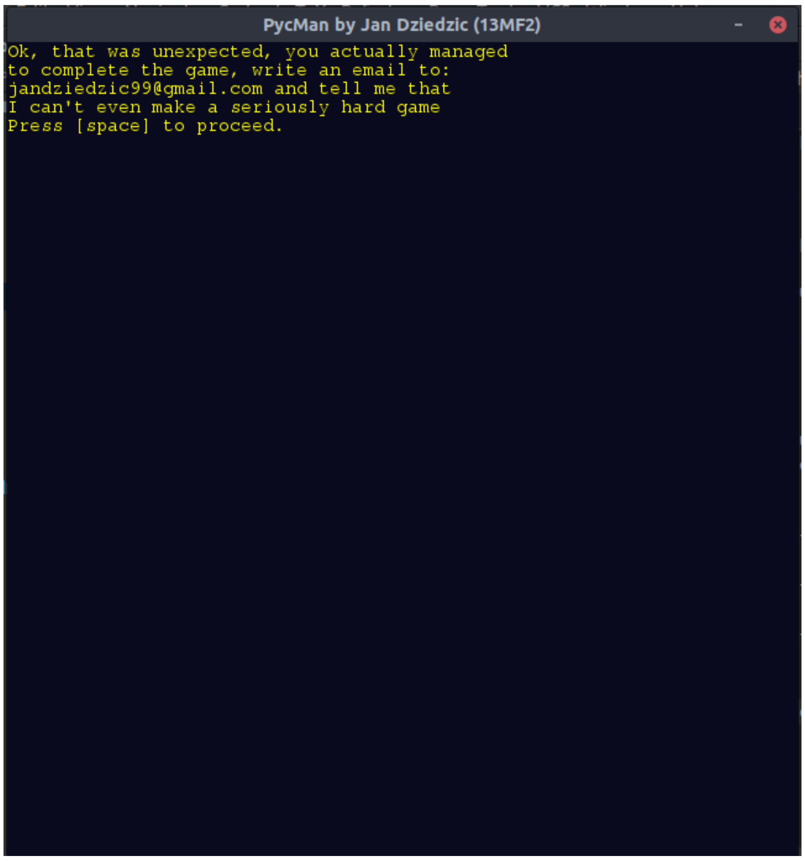
2.6.4 Loosing the game



```
Oh, you run out of lives? What a shame
Don't worry though - no one actually
expected you to perform better.
Try again or better not, it's a boring
game. Either way... Cheers
Jan
Press [space] to proceed.
```

Figure 2.41: Message displayed upon losing the game. Shown in a game window rendered in the Ubuntu Gnome graphical environment.

2.6.5 Completing the entire game



```
Ok, that was unexpected, you actually managed
to complete the game, write an email to:
jandziedzic99@gmail.com and tell me that
I can't even make a seriously hard game
Press [space] to proceed.
```

Figure 2.42: Message displayed after completing the game. Shown in a game window rendered in the Ubuntu Gnome graphical environment.

2.7 Testing

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

2.7.1 zero-testing

This stage was introduced while testing single functions and procedures implemented through the game during the development process. It consisted of mostly running code as-is while writing it using PyCharm. Sometimes I haven't even use debugger in favor of print function to enlist values of variables. This phase commenced until all modules of the game were actually prepared, despite being containing minor bugs that didn't immediately throw an error. At the end of this stage, the game was playable.

2.7.2 α-testing

This was the stage of extensively playing the game to see whether all functions appear to be working correctly. Note the use of word 'appear' - code wasn't actually read at this stage in look for bugs, only user experience was assessed. I have tried different approaches at playing, to make the game algorithms experience different user input. From staying perfectly still to going straight for a ghost, I have played the game checking the following:

1. Getting killed by the ghost approaching from different angles.
2. Eating Eatables from different angles.
3. Trying to pass through numerous walls.
4. Moving the player in different directions repeatedly to check if ghosts get confused and the gameplay remains fluent.
5. Manual counting of coins eaten to see if it matches the counter value.
6. Checking whether all graphics render well, by creating each wall type and each possible sprite.
7. Providing program with unexpected input of unassigned keys, mouse clicks and even connecting a gamepad.

2.7.3 β-testing

This was a stage of testing, during which all major bugs were resolved and the assessment was focused on user experience. Following procedures were performed:

1. Smoothness of animations was checked on different computers, running different OSs.
2. Graphics quality was assessed by an independent body, namely my sister.
3. Actual playing was performed and all levels were played. Each level was actually completed, but never in a single sitting, proving that game indeed can be completed, though it is considerably hard.
4. Different players were used to test the game play.
5. Quality of input controls was assessed.

Chapter 3

Implementation

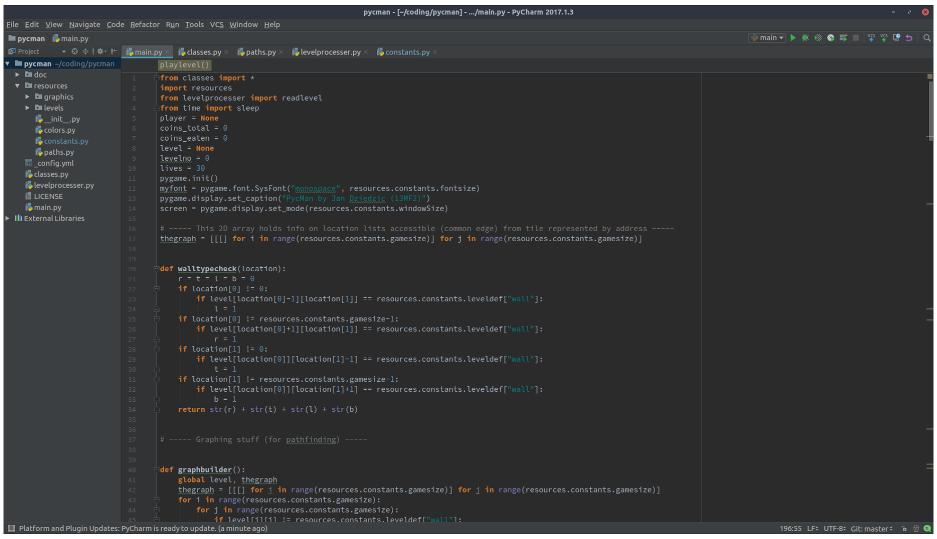
In this section the detailed description of how the game files were created is provided.

3.1 Programming tools

Whole development of the game, as mentioned in Section 1.3, was done using a Ubuntu running computer and legally obtained tools I didn't have to pay for, namely opensource, freeware or free for education software.

3.1.1 PyCharm

I used, in my opinion, a brilliant Python IDE called PyCharm. Developed by a company JetBrains and obtained under Free for Education license for the entirety of the project development. The program has a lot of useful functions, which made my work easier and admittedly contributed to me learning how to code more efficiently.


```python
from classes import *
import resources
from levelprocesser import readlevel
from time import sleep
player = None
coins_total = 0
coins_eaten = 0
level = None
levelno = 0
lives = 30
pygame.init()
myfont = pygame.font.SysFont("monospace", resources.constants.fontsize)
pygame.display.set_caption("PycMan by Jan Dziedzic (13MF2)")
screen = pygame.display.set_mode(resources.constants.windowSize)

# ----- This 2D array holds info on location lists accessible (common edge) from tile represented by address -----
thegraph = [[[] for i in range(resources.constants.gamesize)] for j in range(resources.constants.gamesize)]


def walltypecheck(location):
    r = t = l = b = 0
    if location[0] != 0:
        if level[location[0]-1][location[1]] == resources.constants.leveldef["wall"]:
            l = 1
    if location[0] != resources.constants.gamesize-1:
        if level[location[0]+1][location[1]] == resources.constants.leveldef["wall"]:
            r = 1
    if location[1] != 0:
        if level[location[0]][location[1]-1] == resources.constants.leveldef["wall"]:
            t = 1
    if location[1] != resources.constants.gamesize-1:
        if level[location[0]][location[1]+1] == resources.constants.leveldef["wall"]:
            b = 1
    return str(r) + str(t) + str(l) + str(b)


# ----- Graphing stuff (for pathfinding) -----


def graphbuilder():
    global level, thegraph
    thegraph = [[[] for i in range(resources.constants.gamesize)] for j in range(resources.constants.gamesize)]
    for i in range(resources.constants.gamesize):
        for j in range(resources.constants.gamesize):
            if level[i][j] != resources.constants.leveldef["wall"]:
```

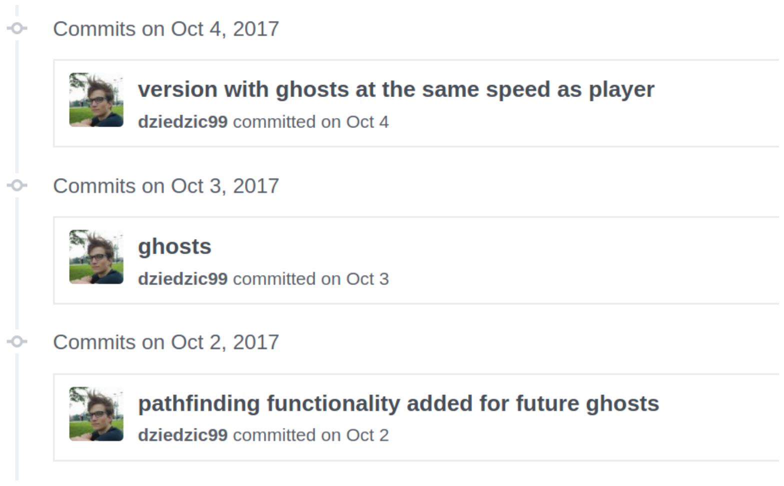
Figure 3.1: Pycharm window rendered in Ubuntu Gnome environment with Pycman project loaded.

3.1.2 GIMP

GIMP software was used to develop graphical resources for the game as well as for this document. It's an open source community-developed tool for graphics manipulation, which sometimes is referred as the free alternative to the Adobe Photoshop, with advanced functions and ability to edit images with Alpha channel, resulting in capacity to make ghost images semitransparent. See Figure 2.23 for reference.

3.1.3 Git

Git is a popular tool for VCS (Version Control System). My previous experience with programming and mostly teamwork made me use it as an obvious choice for keeping my project files. I used free Github for Education account and github.com servers for storing my files. Admittedly it was really useful when I had to revert some advanced changes or look up how something worked earlier, as it worked better. Basically speaking - it made mistakes less painful.


Commits on Oct 4, 2017

version with ghosts at the same speed as player
dziedzic99 committed on Oct 4

Commits on Oct 3, 2017

ghosts
dziedzic99 committed on Oct 3

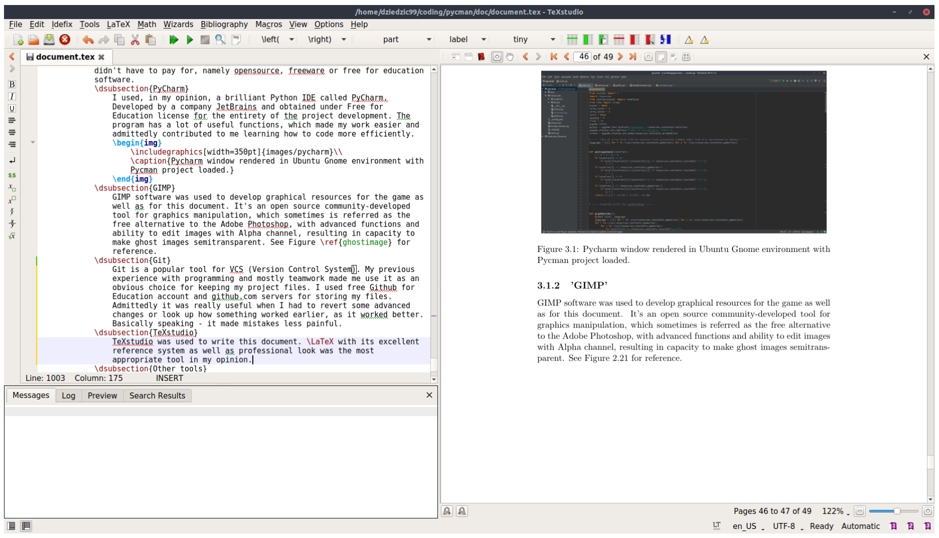
Commits on Oct 2, 2017

pathfinding functionality added for future ghosts
dziedzic99 committed on Oct 2

Figure 3.2: Screenshot of the lists of some of the project commits in the GitHub web interface.

3.1.4 TeXstudio

TeXstudio was used to write this document. LaTeXwith its excellent reference system as well as professional look was the most appropriate tool in my opinion.


/home/dziedzic99/coding/pycman/doc/document.tex - TeXstudio
File Edit Idefix Tools LaTeX Math Wizards Bibliography Macros View Options Help
\left(\right) part label tiny
document.tex
46 of 49
Messages Log Preview Search Results
Pages 46 to 47 of 49 122%
en_US UTF-8 Ready Automatic

```
didn't have to pay for, namely opensource, freeware or free for education
software.
\dsubsection{PyCharm}
    I used, in my opinion, a brilliant Python IDE called PyCharm.
    Developed by a company JetBrains and obtained under Free for
    Education license for the entirety of the project development. The
    program has a lot of useful functions, which made my work easier and
    admittedly contributed to me learning how to code more efficiently.
    \begin{img}
        \includegraphics[width=350pt]{images/pycharm}\\
        \caption{Pycharm window rendered in Ubuntu Gnome environment with
        Pycman project loaded.}
    \end{img}
\dsubsection{GIMP}
    GIMP software was used to develop graphical resources for the game as
    well as for this document. It's an open source community-developed
    tool for graphics manipulation, which sometimes is referred as the
    free alternative to the Adobe Photoshop, with advanced functions and
    ability to edit images with Alpha channel, resulting in capacity to
    make ghost images semitransparent. See Figure \ref{ghostimage} for
    reference.
\dsubsection{Git}
    Git is a popular tool for VCS (Version Control System). My previous
    experience with programming and mostly teamwork made me use it as an
    obvious choice for keeping my project files. I used free Github for
    Education account and github.com servers for storing my files.
    Admittedly it was really useful when I had to revert some advanced
    changes or look up how something worked earlier, as it worked better.
    Basically speaking - it made mistakes less painful.
\dsubsection{TeXstudio}
    TeXstudio was used to write this document. \LaTeX with its excellent
    reference system as well as professional look was the most
    appropriate tool in my opinion.
\dsubsection{Other tools}
Line: 1003    Column: 175        INSERT
```

Figure 3.1: Pycharm window rendered in Ubuntu Gnome environment with Pycman project loaded.

3.1.2 'GIMP'

GIMP software was used to develop graphical resources for the game as well as for this document. It's an open source community-developed tool for graphics manipulation, which sometimes is referred as the free alternative to the Adobe Photoshop, with advanced functions and ability to edit images with Alpha channel, resulting in capacity to make ghost images semitransparent. See Figure 2.21 for reference.

Figure 3.3: TeXstudio window rendered in Ubuntu Gnome environment with this document loaded.

3.1.5 Other tools

Minor other programs were used for writing this documentation, mostly for obtaining screenshots and producing flowcharts.

3.2 Interpreter and its modules

3.2.1 Virtual Environment

To keep my computer clean and my main /usr/bin Python interpreter free of heavy, not often used modules as Pygame I have decided to create a separate interpreter instance using Virtualenv tool.

3.2.2 Modules

Modules used by the game are as follows:

- Pygame - for graphics rendering and input control. Origin of the Sprite class.
- Pillow A.K.A. PIL - for loading level files
- time - origin of the function sleep for delaying certain actions.

3.3 Project structure

I have decided to make the project structure as professionally organized as possible. Therefore I have divided program into resource files and Python files, keeping the first separate in a different folder. Documentation and its resources - graphics are also stored separately.

```
pycman
├── classes.py
├── _config.yml
├── doc
│   ├── document.aux
│   ├── document.log
│   ├── document.pdf
│   ├── document.synctex.gz
│   ├── document.tex
│   ├── document.toc
│   ├── images
│   │   ├── 1_1.png
│   │   ├── segment-part-relation.png
│   │   ├── texstudio.png
│   │   ├── trap.jpg
│   │   ├── w0001.png
│   │   ├── w0010.png
│   │   ├── w0011.png
│   │   ├── w1101.png
│   │   ├── w1110.png
│   │   ├── w1111.png
│   │   └── window-in-ubuntu.png
│   ├── template_Report.aux
│   ├── template_Report.log
│   ├── template_Report.pdf
│   └── template_Report.synctex.gz
├── levelprocesser.py
├── LICENSE
├── main.py
└── resources
    ├── colors.py
    ├── constants.py
    ├── graphics
    │   ├── blueghost.png
    │   ├── coin.png
    │   ├── coin.xcf
    │   ├── greenghost.png
    │   ├── pinkghost.png
    │   ├── player.png
    │   ├── player.xcf
    │   ├── redghost.png
    │   ├── redghost.xcf
    │   ├── w0001.gif
    │   ├── w1011.gif
    │   ├── w1100.gif
    │   ├── w1101.gif
    │   ├── w1110.gif
    │   ├── w1111.gif
    │   ├── wall.png
    │   └── wallxcf.xcf
    ├── __init__.py
    ├── levels
    │   ├── 0.bmp
    │   ├── level0.bmp
    │   ├── level1.bmp
    │   ├── test2.png
    │   ├── tutorial0.png
    │   ├── tutorial1.bmp
    │   └── tutorial1.png
    └── paths.py
```

Figure 3.4: Abbreviated directory structure layout

Reason for additional Python files will be provided within this Chapter.

3.4 Classes

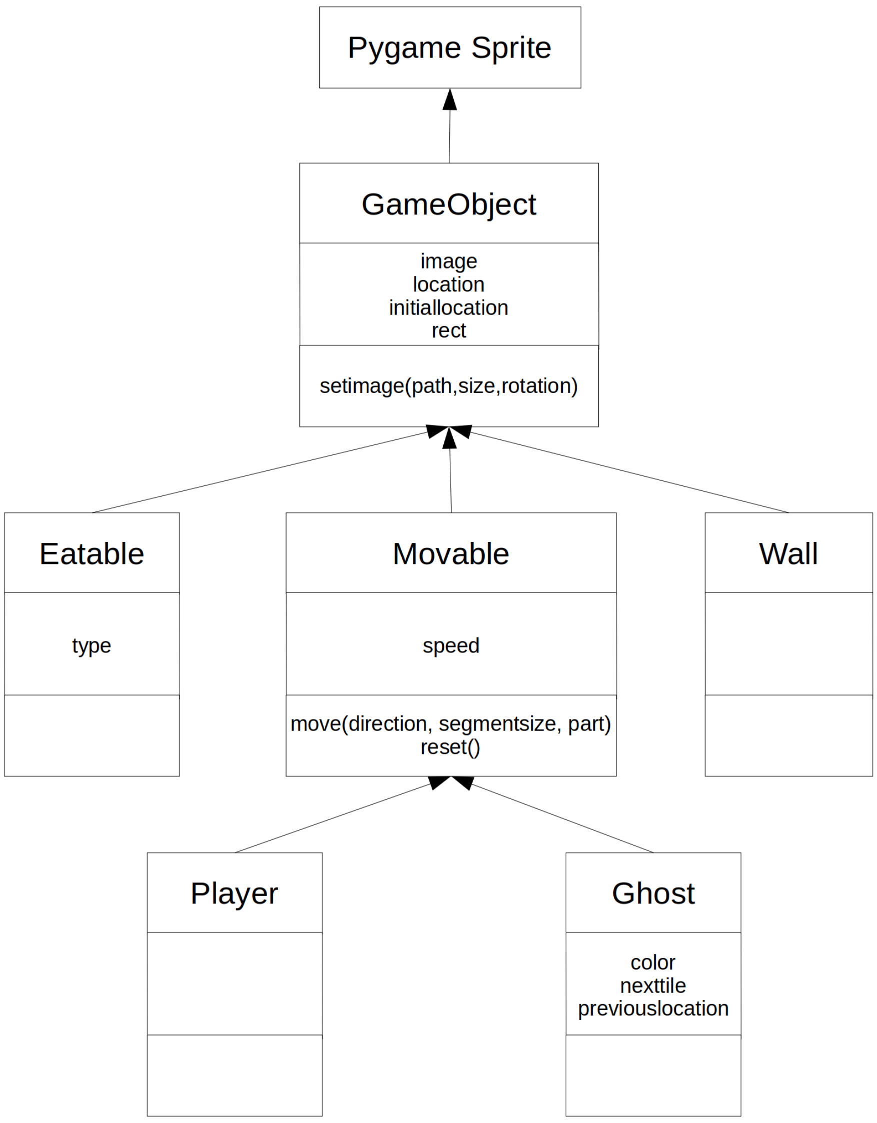


Figure 3.5: Project class diagram

3.4.1 GameObject

As whole game is written around pygame environment I have decided to utilize Pygame Sprite class for all objects (excluding text) rendering on the screen. To facilitate some concepts I assumed in the Design stage I had to add a GameObject class.
Class code:

```python
class GameObject(pygame.sprite.Sprite):
  def __init__(self, location):
    super().__init__()
    self.image = None
    self.location = location
    self.initiallocation = location
    self.rect = None

  def setimage(self, image, size=resources.constants.boxSize,
    rotation=0):
    self.image = pygame.image.load(resources.paths.image[image])
    self.image = pygame.transform.scale(self.image, size)
    self.image = pygame.transform.rotate(self.image, rotation)
    self.rect = self.image.get_rect()
    self.rect.x, self.rect.y = self.location[0] * resources.
    constants.boxSegmentSize, self.location[1] * resources.
    constants.boxSegmentSize

```

The most important part of the class is the 'location' attribute. It is an (integer, integer) tuple storing information of location of the object in a grid, as described in Section 2.2. Location is passed as a constructor argument. It also provides facilities for easy setting the Sprite image.

3.4.2 Eatable

Eatable class is a standard class for Coins and Hearts (see Section 2.3.4 for Coins and Hearts definitions). Its __init__() method sets proper image depending on its type, which it takes as an argument.
Class code:

```python
class Eatable(GameObject):
  def __init__(self, type, location):
    super().__init__(location)
    self.type = type
    if type == "coin":
      self.setimage("coin", resources.constants.eatableSize)
    elif type == "heart":
      self.setimage("heart", resources.constants.eatableSize)

```

3.4.3 Wall

Wall class is fairly simple - during declaration it takes arguments of location in the grid and the type of the Wall. (For reference on types of walls see Section 2.3.1 and Section 2.3.1 for reference on type determination.)
Class code:

```python
class Wall(GameObject):
  def __init__(self, location, type):
    super().__init__(location)
    self.setimage('w'+str(type))

```

3.4.4 Movable

Movable is the most complex class deriving from GameObject. It is a blueprint for both Ghosts and the Player, which in contrary to other GameObjects are not static and may move or reset their position to initial (see Section 2.5.7 for information about Movable respawning). It has an additional attribute of 'speed' which is an (integer, integer) tuple with information in which direction (x, y) is the Movable moving. Naturally it has a 'move' method that processes these information accordingly. This method takes following arguments:

- 'direction' - processed to produce a valid 'speed' attribute,
- 'segmentsize' - number of frames per time segment (See Section 2.5.1 for reference.)
- 'part' - part/frame of the movement (See Sections 2.5.1 and Figure 2.30 for reference.)

Method for resetting is fairly straightforward - object initializes itself back in the initial position, exactly how it started, with the same initialization method.
Class code:

```python
class Movable(GameObject):
  def __init__(self, location):
    super().__init__(location)
    self.image = None
    self.speed = (0, 0)

  def setimage(self, image, size=resources.constants.movableSize
    , rotation=0):
    super().setimage(image, size, rotation)

    def move(self, direction, segmentsize, part=resources.
    constants.playerTimeSegmentSize - 1):
    if direction == 'up' or direction == (0, 1):
      self.speed = (0, -1)
    elif direction == 'down'or direction == (0, -1):
      self.speed = (0, 1)
    elif direction == 'left' or direction == (-1, 0):
      self.speed = (-1, 0)
    elif direction == 'right' or direction == (1, 0):
      self.speed = (1, 0)
    part2 = (part+1) / segmentsize
    self.rect.x, self.rect.y = (self.location[0] + self.speed[0]
     * resources.constants.speedFactor * part2) * resources.
    constants.boxSegmentSize, (self.location[1] + self.speed[1] *
     resources.constants.speedFactor * part2) * resources.
    constants.boxSegmentSize
    if part == segmentsize - 1:
      self.location = self.location[0] + self.speed[0] *
    resources.constants.speedFactor, self.location[1] + self.
    speed[1] * resources.constants.speedFactor

  def reset(self):
    self.__init__(self.initiallocation)

```

3.4.5 Player

Descriptively, Player class is the template for the Player's Sprite. I have created a separate class for object that by definition should only exist in one instance for simplicity. If all Movables are to be moved using same routines - so should be the player. It also matches my objective to learn how to program using the 'Object Oriented Programming' paradigm. Class constructor takes argument of location, as per standard Movable and sets Sprite's image as the yellow ball with 'mouth'. Notably this class only overrides Movable.move method in order to account for Player's rotation (see Requirement 1c). The rotation is determined using speed attribute and stored as a degree value in 'rotation' integer variable local to overridden 'move' method. Then, within the same method, parent's parent's (Movable's parent is GameObject) 'setimage' method is invoked with an extra argument, the rotation. This example nicely shows how inheritance works in practice, a great advantage of OOP, which I learned this way.
Class code:

```
class Player(Movable):
  def __init__(self, location):
    super().__init__(location)
    self.setimage('player', resources.constants.playerSize)

    def move(self, direction, part=resources.constants.
    playerTimeSegmentSize - 1, partsize=resources.constants.
    ghostTimeSegmentSize):
      rotation = 0
      if self.speed == (1, 0):
        rotation = 0
      elif self.speed == (-1, 0):
        rotation = 180
      elif self.speed == (0, -1):
        rotation = 90
      elif self.speed == (0, 1):
        rotation = 270
      super().setimage('player', resources.constants.playerSize, rotation)
      super().move(direction, segmentsize=resources.constants.playerTimeSegmentSize, part=part)

```

3.4.6 Ghost

This class is used by all ghost objects. It derives from Movable with some minor differences. During initialization, constructor accepts parameters of grid location and ghost color (as a string). This class has a 'previouslocation' (note it is not 'initiallocation') parameter - this stores grid location of the tile the ghost previously occupied. It is a significant factor for determining ghost's path - as per Requirement 1i - ghosts cannot reverse. This basically means that whenever path of the ghost to its target tile is determined, 'previouslocation' tile is excluded from potential beginnings of such path, which completely satisfies the Requirement. Move method has been overriden to facilitate 'previouslocation' change. So was 'reset' method to allow initialization with the same color. Class code:

```
class Ghost(Movable):
    def __init__(self, location, color):
    super().__init__(location)
    self.color = color
    self.nexttile = None
    self.setimage(color+'_ghost')
    self.previouslocation = self.location

    def move(self, direction, part=resources.constants.ghostTimeSegmentSize - 1, partsize=resources.constants.ghostTimeSegmentSize):
    self.previouslocation = self.location
    super().move(direction, partsize, part)

  def reset(self):
    self.__init__(self.initiallocation, self.color)

```

3.5 Resources

3.5.1 Constants

By now reader of this documentation should notice that Section 3.4 is full of 'resources.constants.NAME' statements. When writing the game I have discovered that I input many hardcoded values in both class definitions and the core program code, all of these were values that could change, worse, I have changed them multiple times, throughout development. When I had to change the size of board - namely substituting 64 with 32 in a few dozens of lines I figured that I need to write all of those constants in a single place.

I added new file to the project - constants.py.
Here is its abbreviated content:

```
eatableSize = [20, 20]
movableSize = [20, 20]
playerSize = [20, 20]
speedFactor = 1
windowSize = [640, 660]
playerTimeSegmentSize = 10
ghostTimeSegmentSize = 12
tileWidth = 20
tileSize = [tileWidth, tileWidth]
fps = 60
fontsize = 15
gamesize = 32
scatterSize = 2
totallevels = 5

leveldef = {
  'nothing': 0,
  'wall': 1,
  'coin': 2,
  'player': 3,
  'red_ghost': 4,
  'green_ghost': 5,
  'blue_ghost': 6,
  'heart': 7
}

editordef = {
  'nothing': (255, 255, 255),
  'wall': (0, 0, 0),
  'coin': (255, 255, 0),
  'heart': (200, 200, 255),
  'player': (255, 0, 0),
  'red_ghost': (200, 100, 100),
  'green_ghost': (0, 255, 0),
  'blue_ghost': (0, 0, 255)
}

```

It consists of numerical constants used throughout the game. This basically serves as settings - if I realize that player's Sprite appears to small to comfortably navigate it, I can change its size in a single place without worrying that somewhere in the code I left one value unchanged and, for example player will be able to move through walls. Names of these constants are very descriptive so that core code's readability doesn't suffer from using this method. I even think that it's more readable with constants used rather than hardcoded values.

To use constants within any part of the code following import statement must be used.

```
import resources

```

To invoke a constant following code must be used: (with an example of an 'fps' constant)

```
resources.constants.fps

```

Here is a list of most notable constants used in this project:

- eatableSize - x, y size of an image of object of class Eatable (in pixels)
- movableSize - x, y size of an image of object of class Movable (in pixels)
- playerSize - x, y size of an image of object of class Player (in pixels)
- speedFactor - integer value that can be used to tune Movables' speed on low performance platforms
- windowSize - size of game window in pixels (note, this is not the size of the board)
- playerTimeSegmentSize - number of frames/parts per Player's Time Segment (see Section 2.5.1 for reference) This value can be used to tune Player's speed.
- ghostTimeSegmentSize - number of frames/parts per Ghosts' Time Segment (see Section 2.5.1 for reference) This value can be used to tune Ghosts' speed.
- tileWidth - this is a single integer value representing width of a single tile (see Section 2.2 for reference on tiles) in pixels.
- tileSize - this is an iterable size of the tile in x,y format (unit is pixels). For square tiles it is equal to [tileWidth, tileWidth]
- fps - desired value of Frames Per Second displaying on screen. Can be modified together with speedFactor to run the game on slower machines.

- fontsize - determines size of the font used in the game in points
- gamesize - number of tiles per boards edge. Note that the game has a total of $gamesize^2$ tiles.
- scatterSize - As described in Section 2.5.3 - two of the ghosts don't chase the Player directly, they follow an optimal path to tiles shifted by a specific vector from the Player's location. This constant represents an absolute value of one axis component of this vector. Namely, one ghost's target is shifted by (scatterSize, scatterSize) and the other's by (-scatterSize, -scatterSize)
- totallevels - number of levels available
- editordef - dictionary that defines what type of tile is represented by a particular color (see Section 2.4.1 for reference on how levels are stored) in level files. Colors are represented by (Red, Green, Blue) tuples.
- leveldef - dictionary that defines what type of tile is represented by a particular integer (see Section 2.4.2 for reference on how level files are processed) in core game program.

Past code example:

```
level = [[resources.constants.leveldef['nothing'] for i in range
    (64)] for j in range(64)]

```

New code example:

```
level = [[resources.constants.leveldef['nothing'] for i in range
    (resources.constants.gamesize)] for j in range(resources.
    constants.gamesize)]

```

Notably the code is longer, but the convenience and reliability of this solution justifies having to type extra characters.

3.5.2 Paths

As number of resources grew and grew I figured out that hardcoding file paths was as bad as values. Upon developing walls graphics I have noted that including a path to them in every bit of code would be long, inefficient and prone to errors, even typos. I have decided to aggregate all of my path strings into one file, 'paths.py'. Nested under directory 'resources' this file contains path strings to every single resource used through the game. Here is an abbreviated version presented:

```python
resources = "./resources/"
graphics = resources + "graphics/"
levels = resources + "levels/"

image = {
  "coin": graphics + "coin.png",
  "w0001": graphics + "w0001.gif",
  "w0010": graphics + "w0010.gif",
  "w0011": graphics + "w0011.gif",
  "w0100": graphics + "w0100.gif"
}

```

Presented example shows a part of the 'image' dictionary which resolves filenames to complete paths.

3.6 Level processing

3.6.1 Reading bitmap files

As mentioned in Section 2.4.1 - levels are stored as bitmaps. To decode these to format that is easily processed (see Section 2.4.2) the following file has been written.

```python
from PIL import Image
import resources


def readlevel(file):
  level = [[resources.constants.leveldef['nothing'] for i in
    range(resources.constants.gamesize)] for j in range(resources
    .constants.gamesize)]
  im = Image.open(resources.paths.levels + file)
  im.load()
  for col in range(0, resources.constants.gamesize):
    for row in range(0, resources.constants.gamesize):
      pixel = im.getpixel((col, row))
      if pixel == resources.constants.editordef["coin"]:
        level[col][row] = resources.constants.leveldef['coin']
      elif pixel == resources.constants.editordef["wall"]:
        level[col][row] = resources.constants.leveldef['wall']
      elif pixel == resources.constants.editordef["player"]:
        level[col][row] = resources.constants.leveldef['player']
      elif pixel == resources.constants.editordef["red_ghost"]:
        level[col][row] = resources.constants.leveldef['
    red_ghost']
      elif pixel == resources.constants.editordef["green_ghost"
    ]:
        level[col][row] = resources.constants.leveldef['
    green_ghost']
      elif pixel == resources.constants.editordef["blue_ghost"]:
        level[col][row] = resources.constants.leveldef['
    blue_ghost']
  return level

```

As seen, this code uses Pillow A.K.A. 'PIL' Python module to process bitmaps. Resources are also imported to utilize leveldef and editordef dictionaries. Input of the readlevel function is a path of a 32px x 32px bitmap, the output is a 32 x 32 array of integers representing tile types at particular positions on the board.

3.6.2 Checking wall type

As walls are displayed as different images (which is extensively described in Section 2.3.1), following function is available to determine wall type.

```
def walltypecheck(location):
  r = t = l = b = 0
  if location[0] != 0:
    if level[location[0]-1][location[1]] == resources.constants.
   leveldef["wall"]:
      l = 1
  if location[0] != resources.constants.gamesize-1:
    if level[location[0]+1][location[1]] == resources.constants.
   leveldef["wall"]:
      r = 1
  if location[1] != 0:
    if level[location[0]][location[1]-1] == resources.constants.
   leveldef["wall"]:
      t = 1
  if location[1] != resources.constants.gamesize-1:
    if level[location[0]][location[1]+1] == resources.constants.
   leveldef["wall"]:
      b = 1
  return str(r) + str(t) + str(l) + str(b)

```

Function walltypecheck takes an input of grid location in (x,y) format and outputs a string of 0s and 1s corresponding with wall name (See Section 2.3.1 for wall names). Note usage of r,t,l,b variables which correspond with this scheme:
separately.

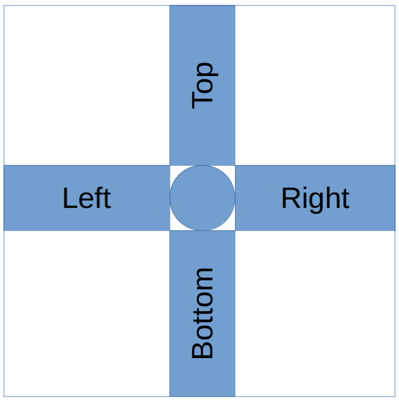


Figure 3.6: Scheme showing which parts of wall are added according to r,t,l,b values

3.6.3 Adding Sprites according to tile types

The following script is used to add Sprites to their respective groups. See section 3.6.4 for reference.

```python
def loadlevel(file):
  global level, coins_total, player
  # level-loading procedure with input of standard 32x32 bitmap level file
  level = readlevel(file)
  for col in range(0, resources.constants.gamesize):
    for row in range(0, resources.constants.gamesize):
      box = level[col][row]
      if box != resources.constants.leveldef["wall"] and box != resources.constants.leveldef['nothing']:
        eatable = Eatable(0, (col, row))
        eatables_list.add(eatable)
        coins_total += 1
      if box == resources.constants.leveldef["wall"]:
        wall = Wall((col, row), walltypecheck((col, row)))
        walls_list.add(wall)
      elif box == resources.constants.leveldef["player"]:
        player = Player((col, row))
      players_list.add(player)
      elif box == resources.constants.leveldef["red_ghost"]:
        ghost = Ghost((col, row), 'red')
        ghosts_list.add(ghost)
      elif box == resources.constants.leveldef["blue_ghost"]:
        ghost = Ghost((col, row), 'blue')
        ghosts_list.add(ghost)
      elif box == resources.constants.leveldef["green_ghost"]:
        ghost = Ghost((col, row), 'green')
        ghosts_list.add(ghost)
```

3.6.4 Sprite Groups

Within the main program file the following groups are globally declared.

```
eatables_list = pygame.sprite.Group()
walls_list = pygame.sprite.Group()
players_list = pygame.sprite.Group()
ghosts_list = pygame.sprite.Group()
```

Each of them stores Sprites of respective type - a very useful feature of pygame.sprite allowing manipulating multiple objects at once.

3.7 Ghost pathfinding algorithm

This section covers the most algorithmically complex part of this project - ghost pathfinding system. As seen in section 2.5.3 the concept is based on BFS graph search. But first there must be a graph to begin with. It's important to note that the graph is stored in 'thegraph' global variable - pretty descriptive, as this is the only graph implementation used in this project.

3.7.1 Graph structure

Formally - the graph is a list of lists of lists of lists of integers - notably complex structure. It's easier to imagine it though as a two-dimensional matrix of lists of tile locations - that sounds like something comprehensible by a human mind. Namely - first two iterable arguments are (x,y) coordinates of the tile we want to query for neighboring tiles. If we were to

```
print(thegraph[5][5])

```

we would see a list of points in [x,y] format, all tiles that a Movable could move to from tile (5,5).

3.7.2 Graph building

```
def graphbuilder():
  global level, thegraph
  thegraph = [[[] for i in range(resources.constants.gamesize)]
    for j in range(resources.constants.gamesize)]
  for i in range(resources.constants.gamesize):
    for j in range(resources.constants.gamesize):
    # iterating through all tiles in the grid
      if level[i][j] != resources.constants.leveldef["wall"]:
      # taking advantage of the fact that Movables can move on
    all tiles that are not initially marked as walls
        location = [i, j]
        # processing location in more suitable format
        # and now, iterating through all possible neighbours of the processed tile (with checking if it isn't by any chance a border tile)
        if location[0] != 0:
          if level[location[0] - 1][location[1]] != resources.constants.leveldef["wall"]:
            thegraph[location[0]][location[1]] += [[location[0] - 1, location[1]]]
        if location[0] != resources.constants.gamesize-1:
          if level[location[0] + 1][location[1]] != resources.constants.leveldef["wall"]:
            thegraph[location[0]][location[1]] += [[location[0] + 1, location[1]]]
        if location[1] != 0:
          if level[location[0]][location[1] - 1] != resources.constants.leveldef["wall"]:
            thegraph[location[0]][location[1]] += [[location[0], location[1] - 1]]
        if location[1] != resources.constants.gamesize-1:
          if level[location[0]][location[1] + 1] != resources.constants.leveldef["wall"]:
            thegraph[location[0]][location[1]] += [[location[0], location[1] + 1]]

```

As seen for every possible neighboring tile the algorithm checks whether it is a wall and if it's not - it adds it to the list of all tiles a Movable may move to. As walls do not move, appear nor disappear this structure remains static throughout the level and must only be processed once upon level loading.

3.7.3 Choosing a target tile for each ghost

As described in Section 3.5.1, resources.constants.scatterSize governs to how distant tile should green and blue ghost move. But when we add let's say a vector (2,2) to player's location, there is no guarantee that this tile is not a wall. If it was a wall it wouldn't appear on any list of neighbors of surrounding tile - ghost wouldn't be able to move there. Worse, there is no guarantee that this tile is even within the boundaries of the game board. I have discovered it during first tests of ghost chasing - game crashed as internal algorithms tried to fetch neighbors of tile which coordinates were greater than board size - this resulted in indexError. Therefore a need for the following function appeared:

```
def find_nearest_not_wall(point):
  if point[0] > resources.constants.gamesize -1:
    point[0] = resources.constants.gamesize-1
  if point[0] < 0:
    point[0] = 0
  if point[1] > resources.constants.gamesize -1:
    point[1] = resources.constants.gamesize-1
  if point[1] < 0:
    point[1] = 0
```

```
10   if level[point[0]][point[1]] != resources.constants.leveldef['
       wall']:
11     return point
12   radius = 1
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] != resources.constants.leveldef['wall']:
18           return [i, j]
19       except IndexError:
20         pass
21     radius += 1
22
```

With a slightly deceiving name - find_nearest_not_wall - not only finds the coordinates of the nearest tile that isn't marked as a wall but it finds the nearest tile that isn't marked as a wall **that actually exists**. Within lines 2-9 the location is being confined to the boundaries of the board - then condition in line 10 checks whether the tile in question is a wall - if it isn't - that solves the problem. Else - the actual search begins. With growing radius neighboring tiles are checked for not being a wall. As this algorithm may eventually lead to checking type of a tile outside the board, a try/catch routine is used. By filtering cases of IndexError, nonexistent tiles aren't checked. I found this solution cleaner that limiting the borders of search, checking whether radius doesn't extend beyond the board, after all it's only 3 very clean lines. The function eventually returns a nearest existing point not being a wall.

3.7.4 Actual pathfinding

After giving it a thought, the ghosts don't really require a complete path to operate. They just need to know where next to move. One single tile location or even a direction. I also figured that running searches from each tile of ghost's neighbors is inefficient. Out of pure reason... What seems faster? Finding one needle in a haystack three times or finding one of three needles in a haystack once? I would go for the second one. Here are two diagrams illustrating the concept:

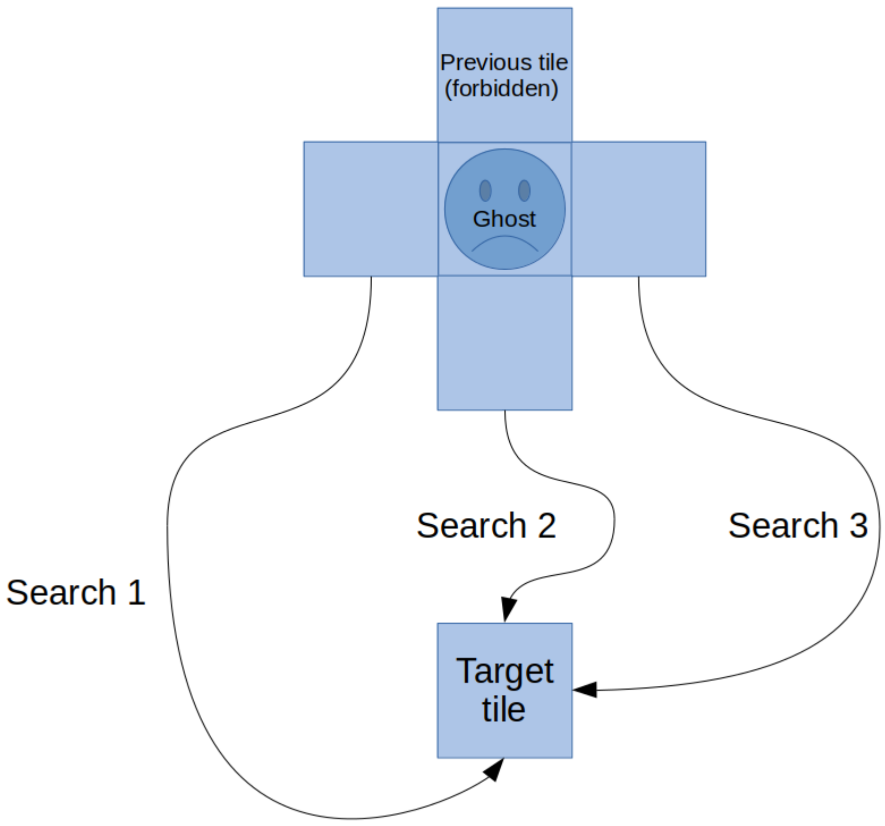


Figure 3.7: Inefficient method of running three searches.

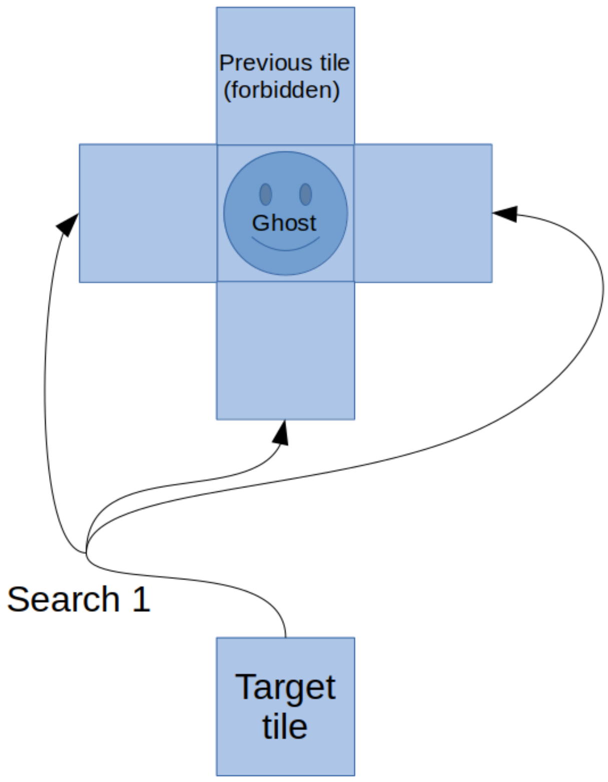


Figure 3.8: Efficient method of running one search.

To keep things simple, my algorithm for pathfinding accepts three arguments: start (ghost position), end(target tile position) and forbidden (tile the ghost moved from - ghost reversing prevention as per requirement 1i). The initial version of the code had the following form.

```
def find_next_move(start, end):
  visited = []
  queue = []
  end = find_nearest_not_wall(end)
  # as start is guaranteed not to be wall, both ends are now
    places a Movable may move to
  neighbours = thegraph[start[0]][start[1]]
  queue.append(end)
  visited.append(end)
  for point in queue:
    if point in neighbours:
      return point
    for node in thegraph[point[0]][point[1]]:
      if node not in visited:
        visited.append(node)
        queue.append(node)
```

While this algorithm actually produced solutions in the initial testing stage, it was slow and I realized it didn't comply with Requirement 1i. To address these issues, changes were made and now the code for the main pathfinding function is as follows:

```
def find_next_move(start, end, forbidden):
  visited = []
  # storing visited nodes
  queue = []
  # storing nodes to visit
  end = find_nearest_not_wall(end)
  # as start is guaranteed not to be wall, both ends are now places a Movable may move to
  neighbours = thegraph[start[0]][start[1]].copy()
  # preventing corrupting thegraph when excluding forbidden tile
  try:
    # try as forbidden tile may be any tile after reset
    # particularly it may not be in neighbors
    neighbours.remove([forbidden[0], forbidden[1]])
    # removing forbidden tile from neighbors
  except Exception:
    pass
  queue.append([end[0], end[1]])
  # start search from the end (target tile)
  visited.append(end)
  # mark end as a visited node
  for point in queue:
    if point not in visited:
      visited.append(point)
      # mark current point as visited
    if point in neighbours:
      return point
      # if solution is find return next tile for the ghost
    for node in thegraph[point[0]][point[1]]:
      if node not in visited and node != [forbidden[0], forbidden[1]]:
        queue.append(node)
        # add all unvisited neighbors of current node to the queue
```

Being a rather clean BFS implementation this code returns only the location of one of ghost's neighbors - the tile it has to move to next. It was rewritten from scratch and well commented. Forbidden tile was added and therefore Requirement 1i fulfilled.

3.8 Designing resources

Admittedly, my computer graphics skills are quite limited, so are manual. I considered designing graphics on paper, scanning them and importing into game. Following my first sketches I have decided to rather use GIMP instead of paper and pencils. TODO: add mock drawing
As per Requirement 1a, I needed to produce designs similar to those in original PacMan.

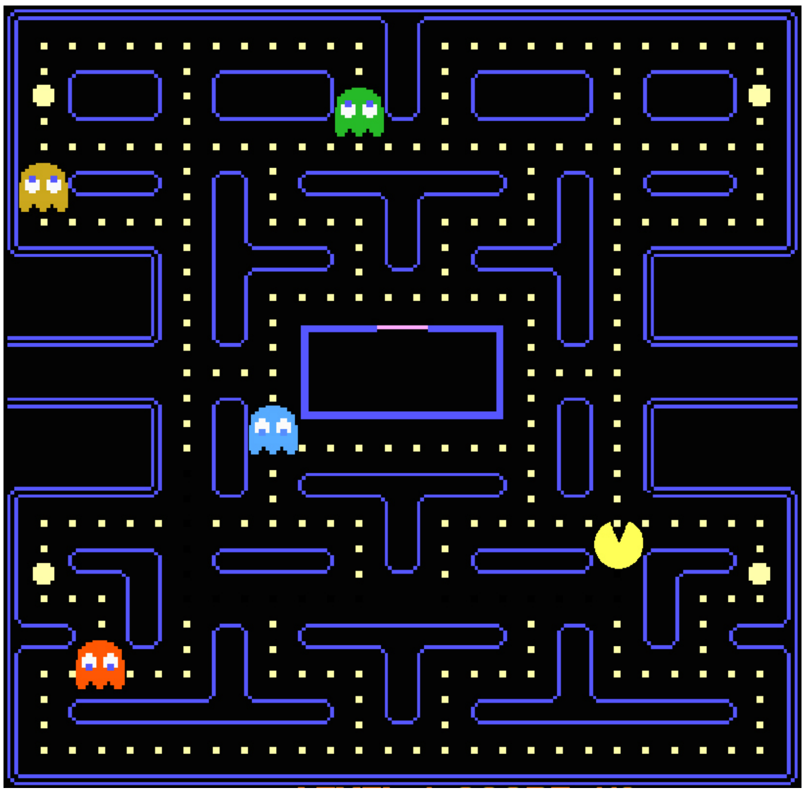


Figure 3.9: Original PacMan graphics. Source: www.todayifoundout.com

3.8.1 Walls

These were rather simple. While original PacMan walls were build with a double blue framework, I went for a single framework. There is a clearly justified reason for that - my game was initially bigger and if I were to have double framework walls, player might have become confused, whether some space is accessible or not. After all the game board is a maze. I started with the simplest walls - 1010 and 0101. See pictures in the Design Section 2.3.1. Then I generated a simple 1100 arc. Rest of the arcs were just rotated versions of it. Walls with three 1s and one 0 in their code, resembling letter

T, were just merges of two of those double 1s arcs and wall 1111 was a merger of them all. GIMP offered tools for easy rotation and cop-pasting, which led to the graphics being produced in no time. I had to figure out how to make them transparent so that the game background could be seen from underneath. Again, GIMP offered function "Background to Alpha" which solved the problem instantly. And last, the format - it had to support transparency, be quite compact and readable by Pygame. I turned to GIF as it fitted these requirements perfectly.

3.8.2 Ghosts

In my opinion PycMan's greatest advantage is its simplicity - ghosts were created using GIMP's spray tool while pretty randomly stroking laptop's touchpad. This gave them their unique shape. To produce three ghosts of the same shape and different colors I used color manipulation tools available within GIMP to alter RGB channels in the original picture. Again these were exported as GIFs to preserve the partial transparency. For ghost graphic in red see Figure 2.23.

3.8.3 Player

The under-appreciated GIMP's ability to create circles of a single color and its unprecedented selection tool were used to create player's graphic of a yellow ball with 'mouth', fulfilling Requirement 1b. Again this was exported as GIF to preserve the Alpha channel.

3.9 Core program code

3.9.1 Playing a single level

To initialize the game, the core program calls function playlevel(), which takes no arguments as it utilizes global variables for reference on which level should be played and on lives remaining.

```python
def playlevel():
  global coins_eaten, coins_total, lives, levelno
  # marking variables as global
  coins_total = coins_eaten = 0
  # resetting coin counters
  prepare_level(resources.paths.levelorder[levelno])
  # level loading
  game_on = True
  # game starter
  clock = pygame.time.Clock()
  # clock init
  eatables_list.update()
  walls_list.update()
  players_list.update()
  ghosts_list.update()
  # updating Sprite Groups
  player_time_segment = resources.constants.
    playerTimeSegmentSize
  ghost_time_segment = resources.constants.ghostTimeSegmentSize
  # forcing pathfinding to update

```

By this part all initialization is done and the program starts main game loop.

Main loop

Following code initializes the main while loop. Its execution finishes when some part of the program sets game_on flag to False.

```
# ——— Main game loop ———
while game_on:

```

Handling completing the level

The following code executes when all coins are eaten. It switches the game to next level and prevents next loop execution by setting the game_on flag to False. This fulfills Requirement 1o

```
# ——— Level - winning handling ———
if coins_eaten == coins_total:
  levelno += 1
  game_on = False
  # ——— stop the level if all coins are eaten ———

```

Handling user input and movement part 1

The following snippet executes only once per Player's Time Segment. It checks for user input, whether the game window was closed using operating system infrastructure and processes player's surroundings to determine to which tiles user can move (not to move onto wall tile as per Requirement 1d).

```
if player_time_segment == resources.constants.
    playerTimeSegmentSize:
  # ——— Handling user input pt1 ————
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      game_on = False

  # ——— checking pressed keys ———
  keys = pygame.key.get_pressed()
  # ——— wall collision check ———
  surroundings = walltypecheck(player.location)

```

Handling ghost movement part 1

This code, executed once per Ghosts' Time Segment assigns new target tiles for each ghost using pathfinding function described in Section 3.7.4, implementing changes to target tiles form Section 2.5.3, fulfilling Requirement 1h and 1e.

```
# ----- ghost movement handling pt 1 -----
if ghost_time_segment == resources.constants.
    ghostTimeSegmentSize:
  # if it's time to update paths update them
  for ghost in ghosts_list.sprites():
    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] + resources.constants.scatterSize,
      player.location[1] + resources.constants.scatterSize],
      ghost.previouslocation)
    elif ghost.color == 'green':
      ghost.nexttile = find_next_move(ghost.location,
      [player.location[0] - resources.constants.scatterSize,
      player.location[1] - resources.constants.scatterSize],
      ghost.previouslocation)

```

Updating 'part' counters

As Movable 'move' method requires supplying it with with Time Segment's Part, the following codes produces such number for both Ghosts (note that Ghosts share the same Time Segments)and the Player. For reference on Parts and Time Segments see Section 2.5.1.

```
# ----- handling some frames/move -----
player_part = resources.constants.playerTimeSegmentSize -
    player_time_segment
ghost_part = resources.constants.ghostTimeSegmentSize -
    ghost_time_segment

```

Handling input part 2

As actual user input occurs every Player's Time Segment, rather than every Part, the following code executes every part, to actually move the Player's sprite.

```
# ------ input handling pt 2 / player movement handling ------
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)

```

Handling ghost movement part 2

Same as with player, actual Sprites moving must occur every Part, rather than every Time Segment. This loop iterates through Ghosts and moves each of them.

```
# ------ ghost movement handling pt 2 ------
for ghost in ghosts_list.sprites():
  try:
    ghost.move((ghost.nexttile[0] - ghost.location[0], -ghost.nexttile[1] + ghost.location[1]), ghost_part)
  except Exception:
    # if for some unforseen reason it's impossible to move the ghost just leave it there
    pass

```

Handling player dying

As per Requirement 1f, the game processes player dying by checking whether its current location matches any of the Ghosts' locations. If so, the life is subtracted from lives counter and the board is reseted. If the Player is subsequently in possession of no lives, game_on flag is set to False to prevent next loop execution.

```
# ------ player dying handling ------
for ghost in ghosts_list.sprites():
  if ghost.location == player.location:
    # decrease lives counter
    lives -= 1
    reset()
    # reset the board
    player_time_segment = resources.constants.playerTimeSegmentSize+1
    ghost_time_segment = resources.constants.ghostTimeSegmentSize+1
    # force game to update ghost paths after respawning
if lives <= 0:
  # if the player is dead for good
  game_on = False
  break

```

Drawing

Following code is used to render Sprites on the screen.

```
# ------ Updating sprites position ------
eatables_list.update()
walls_list.update()
players_list.update()
ghosts_list.update()
# ------ Fill screen with background ------
screen.fill(resources.colors.background)
# ------ ...and sprites ------
eatables_list.draw(screen)
walls_list.draw(screen)
players_list.draw(screen)
ghosts_list.draw(screen)

```

Processing and rendering counters

This code handles counters displayed below the board.

```
# ------ Counter info ------
label = myfont.render("Coins eaten " + str(coins_eaten) +" / " +
    str(coins_total) +
  "     Lives remaining " + str(lives) +
  "     Level " + str(levelno+1), 1, (255, 255, 0))
screen.blit(label, (resources.constants.tileWidth / 2, resources
    .constants.gamesize * resources.constants.tileWidth))

```

Refreshing screen

The following code handles screen refreshing using pygame internal methods and supplies new values for player_time_segment and ghost_time_segment variables.

```
# ------ Refresh Screen ------
pygame.display.flip()

# ------ FPS handling ------
player_time_segment -= 1
ghost_time_segment -= 1
clock.tick(resources.constants.fps)

```

Handling eating Eatables

TODO: update for hearts As per Requirement 1k and 1j, the following code checks whether any of the Eatables is affected by Player's presence and if so, it kills it, modifying appropriate counters.

```
# ----- eating handling -----
for eatable in eatables_list.sprites():
  if eatable.location == player.location:
    eatable.kill()
    coins_eaten += 1
    # increasing eaten points counter

```

Initializing new Time Periods

Finally, if either of Time Segments comes to an end, the following code executes to initialize a new one.

```
# ----- initializing new time periods -----
if not player_time_segment:
  player_time_segment = resources.constants.
    playerTimeSegmentSize
if not ghost_time_segment:
  ghost_time_segment = resources.constants.ghostTimeSegmentSize

```

Provided the game_on flag is still set to True, the loop continues.

Development of the code described in the above Section was particularly difficult due to having to simultaneously implement two Time Segment handlers. (See Section 2.5.1 for Time Segment explanation.) My first approach was to implement those within single counter. It worked but there was no way to satisfy Requirement 2f. Another problem I encountered was with the input handling. The player's sprite was jumping around, because player could change moving direction mid-time segment. The obvious choice was to move keypress checking to section that only executes once per time segment, rather than once per frame. My concern was that this will affect overall user input performance, compromising Requirement 3a. My preliminary testing as well as β-testing revealed that with 60fps and player segment size equal to 10, the input rate of 6 key presses per second seemed optimal.

3.9.2 Function switching levels

Function named main() taking no parameters, iterating through levels and displaying appropriate messages. This is the main function that launches the whole game.

```
def main():
  show_message(resources.constants.tutorialStringList)
  while lives > 0 and levelno <= resources.constants.totallevels
    :
    print(levelno)
    playlevel()
    show_message(resources.constants.congratsStringList)
  if lives > 0:
    show_message(resources.constants.realCongratsStringList)
  else:
    show_message(resources.constants.theEndStringList)

```

When the while loop breaks, either Game Over screen is displayed or a congratulations screen, fulfilling Requirements 1r.

3.9.3 Displaying messages

As it's seen in the Section 3.9.2 - the function show_message is used to display small text messages on the screen. Those messages include:

- Tutorial screen
- Congratulations on completing a level
- A Game Over screen saying that the player lost
- Congratulations on completing the whole game

Following is the code for the show_message function. It takes a single argument of a list of lines to display.

```
def show_message(messageboard):
  screen.fill(resources.colors.background)
  for i in range(len(messageboard)):
    label = myfont.render(messageboard[i], 1, (255, 255, 0))
    screen.blit(label, (0, resources.constants.fontsize * i))
  pygame.display.flip()
  waiting = True
  while waiting:
    for event in pygame.event.get():
      if event.type == pygame.KEYDOWN and event.key == 32:
        waiting = False
  sleep(0.05)

```

Chapter 4

Evaluation

The Implementation phase of project development is followed by Testing phase, during which the potentially ready project is checked for errors, functionality, usability and overall performance.

4.1 General testing

General testing involves mostly assessing game's responsiveness to various input, expected and unexpected under different initial conditions. The overall performance of the project satisfies both me and the stakeholder example, Wojciech, who did the β-testing. The game was tested under following conditions.

Input	Response	Comment
Performing mouse operations on the window	No response other than window closing when X button is clicked.	Expected response confirmed.
Pressing keys other than arrow keys during gameplay	No response related to these keys	As expected
Pressing keys other than space bar when message is displayed	No response related to those keys	As expected
Pressing arrow keys repeatedly, very fast	Performance is not compromised, gameplay appears to be normal.	Acceptable response.
Abandoning the game when it is on	The ghosts kill the player repeatedly and eventually the game comes to an end with the proper message displayed.	As expected

4.2 Meeting the requirements

Requirement	Testing method	Result
1a	Game was visually assessed	The colors resemble these of original PacMan.
1b	Player's graphic was visually assessed.	Matches the requirement.
1c	Game was played and player was moved	Player's sprite was rotated accordingly
1d	Player was intentionally, excessively moved in direction of the wall, ghost movement was observed.	Neither of the sprites managed to pass the wall nor appear on the wall tile.
1e	Ghost movement was observed.	Ghosts appear to chase the player.
1f	Player was exposed to contact with the ghosts while stationary, moving away, sideways and towards the ghosts.	In a small number of cases the player was able to move 'through' the ghost when moving directly at it.
1g	The game was played and lives were lost.	Each time the loss of a life made the board reset.
1h	The ghosts were observed when the game was played.	Though ghosts sometimes took the same path, most of the time they were using different paths to surround the player.
1i	The ghosts were observed when the game was played.	Ghosts didn't reverse.
1j & 1k	The game was played and the coins were exposed to the player.	Coins disappeared only upon contact and the appropriate counter increased.
1m & 1n	The game was played and the hearts were exposed to the player.	Hearts disappeared only upon contact and the appropriate counter increased.

1o & 1q & 1s	The game was played to the end.	All levels were passable and upon game completion the appropriate message appeared.
1p	The game was played and the player lost.	The appropriate message appeared when the number of lives fell below 0.
1r	The game was played and all conditions to display all messages were met.	Each time the messages appeared appropriately.
2a	The game was played. Assessors were also shown the original game.	Most of the testers named Pycman ghosts more intelligent
2b & 2c	The game was played.	No ghost house, nor power-ups were found.
2d	The game was played.	No ghosts were harmed during testing.
2e	The game was played.	No behavior indicating any ghost 'modes' were found.
2f	The game was played. The speed of the sprites was observed.	Player was moving notably faster than the ghosts and was able to outrun them.
3a	The arrow Keys were tested during the game.	Game was responsive to various combinations of pressed arrows and the game responded accordingly with no anomalies.
3b	The spacebar was used to dismiss the messages.	The messages disappeared accordingly.
3c	The player was observed when no input was provided.	Player stayed still.

4.3 Changes

As seen above, Requirement 1f was not fully met as abnormal behavior was observed when the game was subject to rather atypical input. This is exactly why such a throughout testing was made. Further debugging indicated that

Player was able to move through the ghost without losing life when following situation took place:

- The player is moving directly towards the ghost.
- When contact occurs, the frame number is divisible by both size of the player's and ghosts' time segment in frames.

Reason to this bug was order of commands in the main loop. The location of both player and ghosts was changed during a single execution of the main loop and only later the locations are checked for a collision. This allows a ghost and the player to 'swap' grid locations in a single frame without having the same location in time other than executions of the code in between the locations change. The solution to this problem was rather simple - the condition to check whether location of the ghost and the player are the same was moved there. the issue was fixed as indicated by further testing.

The updated main loop code is as follows:

```
# ----- Main game loop -----
while game_on:
  # ----- Level - winning handling -----
  if coins_eaten == coins_total:
    levelno += 1
    game_on = False
    # ----- stop the level if all coins are eaten -----
  if player_time_segment == resources.constants.
    playerTimeSegmentSize:
    # ----- Handling user input pt1 -----
    for event in pygame.event.get():
      if event.type == pygame.QUIT:
        game_on = False
        exit(0)
    # ----- checking pressed keys -----
    keys = pygame.key.get_pressed()
    # ----- wall collision check -----
    surroundings = walltypecheck(player.location)

  # ----- ghost movement handling pt 1 -----
  if ghost_time_segment == resources.constants.
    ghostTimeSegmentSize:
    # if it's time to update paths update them
    for ghost in ghosts_list.sprites():
      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] + resources.
    constants.scatterSize,
                                   player.location[1] + resources.
    constants.scatterSize],
                      ghost.previouslocation)
    elif ghost.color == 'green':
      ghost.nexttile = find_next_move(ghost.location,
                      [player.location[0] - resources.constants.scatterSize,
                       player.location[1] - resources.constants.scatterSize],
                      ghost.previouslocation)

# ----- handling some frames/move -----
player_part = resources.constants.playerTimeSegmentSize - player_time_segment
ghost_part = resources.constants.ghostTimeSegmentSize - ghost_time_segment

# ----- input handling pt 2 / player movement handling -----
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)

# ----- player dying handling -----
for ghost in ghosts_list.sprites():
  if ghost.location == player.location:
    # decrease lives counter
    lives -= 1
    reset()
    # reset the board
    player_time_segment = resources.constants.playerTimeSegmentSize + 1
    ghost_time_segment = resources.constants.ghostTimeSegmentSize + 1
    # force game to update ghost paths after respawning

# ----- ghost movement handling pt 2 -----
for ghost in ghosts_list.sprites():
  try:
    ghost.move((ghost.nexttile[0] - ghost.location[0], -ghost.nexttile[1] + ghost.location[1]), ghost_part)
  except Exception:
    # if for some unforseen reason it's impossible to move the ghost just leave it there
    pass

if lives <= 0:
  # if the player is dead for good
  break

# ----- Updating sprites position -----
eatables_list.update()
```

```
walls_list.update()
players_list.update()
ghosts_list.update()
# ------ Fill screen with background ------
screen.fill(resources.colors.background)
# ------ ...and sprites ------
eatables_list.draw(screen)
walls_list.draw(screen)
players_list.draw(screen)
ghosts_list.draw(screen)

# ------ Counter info ------
label = myfont.render("Coins eaten " + str(coins_eaten) + " / " + str(coins_total) +
                      "      Lives remaining " + str(lives) +
                      "      Level " + str(levelno+1), 1, (255, 255, 0))
screen.blit(label, (resources.constants.tileWidth / 2, resources.constants.gamesize * resources.constants.tileWidth))

# ------ Refresh Screen ------
pygame.display.flip()

# ------ FPS handling ------
player_time_segment -= 1
ghost_time_segment -= 1
clock.tick(resources.constants.fps)

# ------ eating handling ------
for eatable in eatables_list.sprites():
    if eatable.location == player.location:
        eatable.kill()
        if eatable.type == 'coin':
            coins_eaten += 1
            # increasing eaten points counter
        elif eatable.type == 'heart':
            lives += 1
            # increasing number of lives

# ------ initializing new time periods ------
if not player_time_segment:
    player_time_segment = resources.constants.playerTimeSegmentSize
if not ghost_time_segment:
    ghost_time_segment = resources.constants.ghostTimeSegmentSize

```

4.4 Room for future development

The game was tested on a personal laptop with rather mediocre specifications. It performed very well, though when framerate was measured I have identified the performance bottleneck to be the pathfinding algorithm. I have figured that there are at lest two very related ways of fixing its performance. As player chooses the next tile every time Period, the next pathfinding computation can begin immediately following this decision in a separate thread. This leads to obvious improvement of running the game in the total number of four threads. One main thread for displaying the graphics and providing general gameplay as well as handling the input + total of three threads, one per ghost for pathfinding. This would allow to run the game on processors with less computing power but could possibly make the project require more primary memory.

The general structure of the project and its cleanness allows for non-obvious user modifications, adding new ghosts, hunting in different regions would require mere extra 10 lines or even less. The possibilities are unlimited as it is an Open Source project. But for the easiest start - users may create their own levels as a form of customizing the game.

This game may be referred to as 'skinnable' - users may change the graphics as they wish by a matter of a simple files substitution. For example, the player's image can be substituted with the user face and ghosts may be substituted with images of college tuition invoices. Possibilities are endless though limited to 16 million colors bitmaps of size 64x64.

4.5 Final assessment

The game appears to be working as expected, all assumed conditions were objectively met without major issues, no complaints were made other than that the game is hard which was actually intended.

4.6 Discussion

Admittedly the development of such project required some assumptions on how to approach the general problem of actually not wasting 20% worth of A-Levels CS marks as well as learning something useful and presenting a quality work. Initial decisions that I have appreciated when beginning to develop this project include:

- Using LaTeX instead of any popular word processor which broadened my knowledge on how to use this tool which will be an obvious choice later in my astronomy related future.

- Writing a clean code. The development of this project was not continuous, I joined English education system straight into Year 13 without knowing that I would have to make up for all the project work my peers already did over the Year 12 and holidays in between. This plus an exceptional amount of other work made me spread the development into large batches of sleep deprived weekend nights, without clean code I would have to spend extra time getting around undocumented solutions each time I attempted to move the project forward.

- Regular backups - honestly, these saved tens of hours when I corrupted my main drive.
- Doing something I know how to approach - yes, making a game was a challenge but then I knew what tools to use and when I stumbled on a problem I knew where to seek answers.

Things I would have done differently:

- I have said that LaTeXwas a good choice over for example Microsoft Word. Which still doesn't make it the perfect choice for documenting a Python project. There is a module for that (like for almost everything else). Jupyter Notebook would be my choice for writing this document if I were to do it again.
- Spending more time to figure out how to save time. I have spent definitely too much time on writing this documentation and I probably should have planned it better.
- Realizing that giving up on a feature is more painful than implementing it. Hearts were to vanish but then the cleanness of my OOP code made them appear in just a few lines.

Chapter 5

Code appendix

TODO: update these files before final submission

5.1 main.py

```python
from classes import *
import resources
from levelprocesser import readlevel
from time import sleep
player = None
coins_total = 0
coins_eaten = 0
level = None
levelno = 0
lives = 30
pygame.init()
myfont = pygame.font.SysFont("monospace", resources.constants.fontsize)
pygame.display.set_caption("PycMan by Jan Dziedzic (13MF2)")
screen = pygame.display.set_mode(resources.constants.windowSize)

# ------ This 2D array holds info on location lists accessible (common edge) from tile represented by address ------
thegraph = [[[] for i in range(resources.constants.gamesize)] for j in range(resources.constants.gamesize)]


def walltypecheck(location):
  r = t = l = b = 0
  if location[0] != 0:
    if level[location[0]-1][location[1]] == resources.constants.leveldef["wall"]:
      l = 1
  if location[0] != resources.constants.gamesize-1:
    if level[location[0]+1][location[1]] == resources.constants.leveldef["wall"]:
      r = 1
  if location[1] != 0:
    if level[location[0]][location[1]-1] == resources.constants.leveldef["wall"]:
      t = 1
  if location[1] != resources.constants.gamesize-1:
    if level[location[0]][location[1]+1] == resources.constants.leveldef["wall"]:
      b = 1
  return str(r) + str(t) + str(l) + str(b)


# ------ Graphing stuff (for pathfinding) ------


def graphbuilder():
  global level, thegraph
  thegraph = [[[] for i in range(resources.constants.gamesize)] for j in range(resources.constants.gamesize)]
  for i in range(resources.constants.gamesize):
    for j in range(resources.constants.gamesize):
      # iterating through all tiles in the grid
      if level[i][j] != resources.constants.leveldef["wall"]:
        # taking advantage of the fact that Movables can move on all tiles that are not initially marked as walls
        location = [i, j]
        # and now, iterating through all possible neighbours of the processed tile (with checking if it isn't by any chance a border tile)
        if location[0] != 0:
          if level[location[0] - 1][location[1]] != resources.constants.leveldef["wall"]:
            thegraph[location[0]][location[1]] += [[location[0] - 1, location[1]]]
        if location[0] != resources.constants.gamesize-1:
          if level[location[0] + 1][location[1]] != resources.constants.leveldef["wall"]:
            thegraph[location[0]][location[1]] += [[location[0] + 1, location[1]]]
        if location[1] != 0:
          if level[location[0]][location[1] - 1] != resources.constants.leveldef["wall"]:
            thegraph[location[0]][location[1]] += [[location[0], location[1] - 1]]
        if location[1] != resources.constants.gamesize-1:
          if level[location[0]][location[1] + 1] != resources.constants.leveldef["wall"]:
            thegraph[location[0]][location[1]] += [[location[0], location[1] + 1]]


def find_nearest_not_wall(point):
  if point[0] > resources.constants.gamesize-1:
    point[0] = resources.constants.gamesize-1
  if point[0] < 0:
```

```python
    point[0] = 0
  if point[1] > resources.constants.gamesize-1:
    point[1] = resources.constants.gamesize-1
  if point[1] < 0:
    point[1] = 0
  radius = 1
  if level[point[0]][point[1]] != resources.constants.leveldef['wall']:
    return point
  while True:
    for i in range(point[0]-radius, point[0]+radius):
      for j in range(point[1]-radius, point[1]+radius):
        try:
          if level[i][j] != resources.constants.leveldef['wall']:
            return [i, j]
        except IndexError:
          pass
    radius += 1


def find_next_move(start, end, forbidden):
  visited = []
  # storing visited nodes
  queue = []
  # storing nodes to visit
  end = find_nearest_not_wall(end)
  # as start is guaranteed not to be wall, both ends are now places a Movable may move to
  neighbours = thegraph[start[0]][start[1]].copy()
  # preventing corrupting thegraph when excluding forbidden tile
  try:
    # try as forbidden tile may be any tile after reset
    # particularly it may not be in neighbors
    neighbours.remove([forbidden[0], forbidden[1]])
    # removing forbidden tile from neighbors
  except Exception:
    pass
  queue.append([end[0], end[1]])
  # start search from the end (target tile)
  visited.append(end)
  # mark end as a visited node
  for point in queue:
    if point not in visited:
      visited.append(point)
      # mark current point as visited
    if point in neighbours:
      return point
      # if solution is find return next tile for the ghost
    for node in thegraph[point[0]][point[1]]:
      if node not in visited and node != [forbidden[0], forbidden[1]]:
        queue.append(node)
        # add all unvisited neighbors of current node to the queue


# ------ ------ ------


def loadlevel(file):
  global level, coins_total, player
  # level-loading procedure with input of standard 32x32 bitmap level file
  level = readlevel(file)
  for col in range(0, resources.constants.gamesize):
    for row in range(0, resources.constants.gamesize):
      box = level[col][row]
      if box != resources.constants.leveldef["wall"] and box != resources.constants.leveldef['nothing']:
        eatable = Eatable('coin', (col, row))
        eatables_list.add(eatable)
        coins_total += 1
      if box == resources.constants.leveldef["wall"]:
        wall = Wall((col, row), walltypecheck((col, row)))
        walls_list.add(wall)
      elif box == resources.constants.leveldef["player"]:
        player = Player((col, row))
        players_list.add(player)
      elif box == resources.constants.leveldef["red_ghost"]:
        redghost = Ghost((col, row), 'red')
        ghosts_list.add(redghost)
      elif box == resources.constants.leveldef["blue_ghost"]:
        redghost = Ghost((col, row), 'blue')
        ghosts_list.add(redghost)
      elif box == resources.constants.leveldef["green_ghost"]:
        redghost = Ghost((col, row), 'green')
        ghosts_list.add(redghost)


eatables_list = pygame.sprite.Group()
walls_list = pygame.sprite.Group()
players_list = pygame.sprite.Group()
ghosts_list = pygame.sprite.Group()


def prepare_level(levelfile):
  global eatables_list, walls_list, players_list, ghosts_list, player
  eatables_list = pygame.sprite.Group()
  walls_list = pygame.sprite.Group()
  players_list = pygame.sprite.Group()
  ghosts_list = pygame.sprite.Group()
  loadlevel(levelfile)
  graphbuilder()


```

```
def show_message(messageboard):
  screen.fill(resources.colors.background)
  for i in range(len(messageboard)):
    label = myfont.render(messageboard[i], 1, (255, 255, 0))
    screen.blit(label, (0, resources.constants.fontsize * i))
  pygame.display.flip()
  sleep(5)


def show_tutorial():
  show_message(resources.constants.tutorialStringList)


def show_congrats():
  screen.fill(resources.colors.background)
  for i in range(len(resources.constants.congratsStringList)):
    label = myfont.render(resources.constants.congratsStringList
    [i], 1, (255, 255, 0))
    screen.blit(label, (0, resources.constants.fontsize*i))
  pygame.display.flip()
  sleep(5)


def show_real_congrats():
  screen.fill(resources.colors.background)
  for i in range(len(resources.constants.realCongratsStringList)
    ):
    label = myfont.render(resources.constants.
    realCongratsStringList[i], 1, (255, 255, 0))
    screen.blit(label, (0, resources.constants.fontsize*i))
  pygame.display.flip()
  sleep(5)

def show_permanent_death():
  screen.fill(resources.colors.background)
  for i in range(len(resources.constants.theEndStringList)):
    label = myfont.render(resources.constants.theEndStringList[i
    ], 1, (255, 255, 0))
    screen.blit(label, (0, resources.constants.fontsize*i))
  pygame.display.flip()
  sleep(5)


def reset():
  for ghost in ghosts_list.sprites():
    ghost.reset()
  player.reset()


def playlevel():
  global coins_eaten, coins_total, lives, levelno, screen
  # marking variables as global
  coins_total = coins_eaten = 0
  # resetting coin counters
```

```
prepare_level(resources.paths.levelorder[levelno])
# level loading
game_on = True
# game starter
clock = pygame.time.Clock()
# clock init
eatables_list.update()
walls_list.update()
players_list.update()
ghosts_list.update()
# updating Sprite Groups
player_time_segment = resources.constants.playerTimeSegmentSize
ghost_time_segment = resources.constants.ghostTimeSegmentSize
# forcing pathfinding to update
# ——— Main game loop ———
while game_on:
  # ——— Level – winning handling ———
  if coins_eaten == coins_total:
    levelno += 1
    game_on = False
    # ——— stop the level if all coins are eaten ———
  if player_time_segment == resources.constants.playerTimeSegmentSize:
    # ——— Handling user input pt1 ————
    for event in pygame.event.get():
      if event.type == pygame.QUIT:
        game_on = False
        exit(0)
    # ——— checking pressed keys ———
    keys = pygame.key.get_pressed()
    # ——— wall collision check ———
    surroundings = walltypecheck(player.location)
  # ——— ghost movement handling pt 1 ———
  if ghost_time_segment == resources.constants.ghostTimeSegmentSize:
    # if it's time to update paths update them
    for ghost in ghosts_list.sprites():
      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] + resources.constants.scatterSize,
                           player.location[1] + resources.constants.scatterSize],
                          ghost.previouslocation)
      elif ghost.color == 'green':
        ghost.nexttile = find_next_move(ghost.location,
                          [player.location[0] - resources.constants.scatterSize,
                           player.location[1] - resources.constants.scatterSize],
                                    ghost.previouslocation)

    # ------ handling some frames/move ------
    player_part = resources.constants.playerTimeSegmentSize -
player_time_segment
    ghost_part = resources.constants.ghostTimeSegmentSize -
ghost_time_segment

    # ------ input handling pt 2 / player movement handling ------
    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)

    # ------ ghost movement handling pt 2 ------
    for ghost in ghosts_list.sprites():
        try:
            ghost.move((ghost.nexttile[0] - ghost.location[0], -
ghost.nexttile[1] + ghost.location[1]), ghost_part)
        except Exception:
            # if for some unforseen reason it's impossible to move
the ghost just leave it there
            pass
    # ------ player dying handling ------
    for ghost in ghosts_list.sprites():
        if ghost.location == player.location:
            # decrease lives counter
            lives -= 1
            reset()
            # reset the board
            player_time_segment = resources.constants.
playerTimeSegmentSize+1
            ghost_time_segment = resources.constants.
ghostTimeSegmentSize+1
            # force game to update ghost paths after respawning
    if lives <= 0:
        # if the player is dead for good
        break

    # ------ Updating sprites position ------
    eatables_list.update()
    walls_list.update()
    players_list.update()
    ghosts_list.update()
    # ------ Fill screen with background ------
    screen.fill(resources.colors.background)
    # ------ ...and sprites ------
    eatables_list.draw(screen)
    walls_list.draw(screen)
    players_list.draw(screen)
    ghosts_list.draw(screen)

    # ----- Counter info -----
    label = myfont.render("Coins eaten " + str(coins_eaten) +" / " + str(coins_total) +
                "  Lives remaining " + str(lives) +
                "  Level " + str(levelno+1), 1, (255, 255, 0))
    screen.blit(label, (resources.constants.tileWidth / 2, resources.constants.gamesize * resources.constants.tileWidth))

    # ----- Refresh Screen -----
    pygame.display.flip()

    # ----- FPS handling -----
    player_time_segment -= 1
    ghost_time_segment -= 1
    clock.tick(resources.constants.fps)

    # ----- eating handling -----
    for eatable in eatables_list.sprites():
      if eatable.location == player.location:
        eatable.kill()
        coins_eaten += 1
        # increasing eaten points counter

    # ----- initializing new time periods -----
    if not player_time_segment:
      player_time_segment = resources.constants.playerTimeSegmentSize
    if not ghost_time_segment:
      ghost_time_segment = resources.constants.ghostTimeSegmentSize


def main():
  show_message(resources.constants.tutorialStringList)
  while lives > 0 and levelno <= resources.constants.totallevels:
    print(levelno)
    playlevel()
    show_message(resources.constants.congratsStringList)
  if lives > 0:
    show_message(resources.constants.realCongratsStringList)
  else:
    show_message(resources.constants.theEndStringList)

main()

```

5.2 classes.py

```
import pygame
import resources


class GameObject(pygame.sprite.Sprite):
  def __init__(self, location):
    super().__init__()
    self.image = None
    self.location = location
    self.initiallocation = location
    self.rect = None

  def setimage(self, image, size=resources.constants.tileSize,
    rotation=0):
    self.image = pygame.image.load(resources.paths.image[image])
    self.image = pygame.transform.scale(self.image, size)
    self.image = pygame.transform.rotate(self.image, rotation)
    self.rect = self.image.get_rect()
    self.rect.x, self.rect.y = self.location[0] * resources.
    constants.tileWidth, self.location[1] * resources.constants.
    tileWidth


class Eatable(GameObject):
  def __init__(self, type, location):
    super().__init__(location)
    self.type = type
    if type == "coin":
      self.setimage("coin", resources.constants.eatableSize)
    elif type == "heart":
      self.setimage("heart", resources.constants.eatableSize)


class Wall(GameObject):
  def __init__(self, location, type):
    super().__init__(location)
    self.setimage('w'+str(type))


class Movable(GameObject):
  def __init__(self, location):
    super().__init__(location)
    self.image = None
    self.speed = (0, 0)

  def setimage(self, image, size=resources.constants.movableSize
    , rotation=0):
    super().setimage(image, size, rotation)

  def move(self, direction, segmentsize, part=resources.
    constants.playerTimeSegmentSize - 1):
    if direction == 'up' or direction == (0, 1):
      self.speed = (0, -1)
    elif direction == 'down'or direction == (0, -1):
      self.speed = (0, 1)
```

```python
    elif direction == 'left' or direction == (-1, 0):
      self.speed = (-1, 0)
    elif direction == 'right' or direction == (1, 0):
      self.speed = (1, 0)
    part2 = (part+1) / segmentsize
    self.rect.x, self.rect.y = (self.location[0] + self.speed[0] * resources.constants.speedFactor * part2) * resources.constants.tileWidth,\
                   (self.location[1] + self.speed[1] * resources.constants.speedFactor * part2) * resources.constants.tileWidth
    if part == segmentsize - 1:
      self.location = self.location[0] + self.speed[0] * resources.constants.speedFactor,\
              self.location[1] + self.speed[1] * resources.constants.speedFactor

  def reset(self):
    self.__init__(self.initiallocation)


class Player(Movable):
  def __init__(self, location):
    super().__init__(location)
    self.setimage('player', resources.constants.playerSize)

  def move(self, direction, part=resources.constants.playerTimeSegmentSize - 1, partsize=resources.constants.ghostTimeSegmentSize):
    rotation = 0
    if self.speed == (1, 0):
      rotation = 0
    elif self.speed == (-1, 0):
      rotation = 180
    elif self.speed == (0, -1):
      rotation = 90
    elif self.speed == (0, 1):
      rotation = 270
    super().setimage('player', resources.constants.playerSize, rotation)
    super().move(direction, segmentsize=resources.constants.playerTimeSegmentSize, part=part)


class Ghost(Movable):
  def __init__(self, location, color):
    super().__init__(location)
    self.color = color
    self.nexttile = None
    self.setimage(color+'_ghost')
    self.previouslocation = self.location

  def move(self, direction, part=resources.constants.ghostTimeSegmentSize - 1, partsize=resources.constants.
      ghostTimeSegmentSize):
      self.previouslocation = self.location
      super().move(direction, partsize, part)

  def reset(self):
    self.__init__(self.initiallocation, self.color)

```

5.3 levelprocessor.py

```
from PIL import Image
import resources


def readlevel(file):
  # level-loading procedure with input of gamesize^2 bitmap level file
  level = [[resources.constants.leveldef['nothing'] for i in range(resources.constants.gamesize)] for j in range(resources.constants.gamesize)]
  im = Image.open(resources.paths.levels + file)
  im.load()
  for col in range(0, resources.constants.gamesize):
    for row in range(0, resources.constants.gamesize):
      pixel = im.getpixel((col, row))
      if pixel == resources.constants.editordef["coin"]:
        level[col][row] = resources.constants.leveldef['coin']
      elif pixel == resources.constants.editordef["wall"]:
        level[col][row] = resources.constants.leveldef['wall']
      elif pixel == resources.constants.editordef["player"]:
        level[col][row] = resources.constants.leveldef['player']
      elif pixel == resources.constants.editordef["red_ghost"]:
        level[col][row] = resources.constants.leveldef['red_ghost']
      elif pixel == resources.constants.editordef["green_ghost"]:
        level[col][row] = resources.constants.leveldef['green_ghost']
      elif pixel == resources.constants.editordef["blue_ghost"]:
        level[col][row] = resources.constants.leveldef['blue_ghost']
  return level

```

5.4 paths.py

```
resources = "./resources/"
graphics = resources + "graphics/"
levels = resources + "levels/"

image = {
  "coin": graphics + "coin.png",
  "w0001": graphics + "w0001.gif",
  "w0010": graphics + "w0010.gif",
  "w0011": graphics + "w0011.gif",
  "w0100": graphics + "w0100.gif",
  "w0101": graphics + "w0101.gif",
  "w0110": graphics + "w0110.gif",
  "w0111": graphics + "w0111.gif",
  "w1000": graphics + "w1000.gif",
  "w1001": graphics + "w1001.gif",
  "w1010": graphics + "w1010.gif",
  "w1011": graphics + "w1011.gif",
  "w1100": graphics + "w1100.gif",
  "w1101": graphics + "w1101.gif",
  "w1110": graphics + "w1110.gif",
  "w1111": graphics + "w1111.gif",
  "player": graphics + "player.png",
  "red_ghost": graphics + "redghost.png",
  "blue_ghost": graphics + "blueghost.png",
  "green_ghost": graphics + "greenghost.png"
}

levelorder = [
  "level0.bmp",
  "level1.bmp",
  "level2.bmp",
  "level3.bmp",
  "level4.bmp",
  "level5.bmp",
  "level6.bmp"
]

```

Chapter 6

Levels appendix

6.1 Level 1

6.1.1 Source image

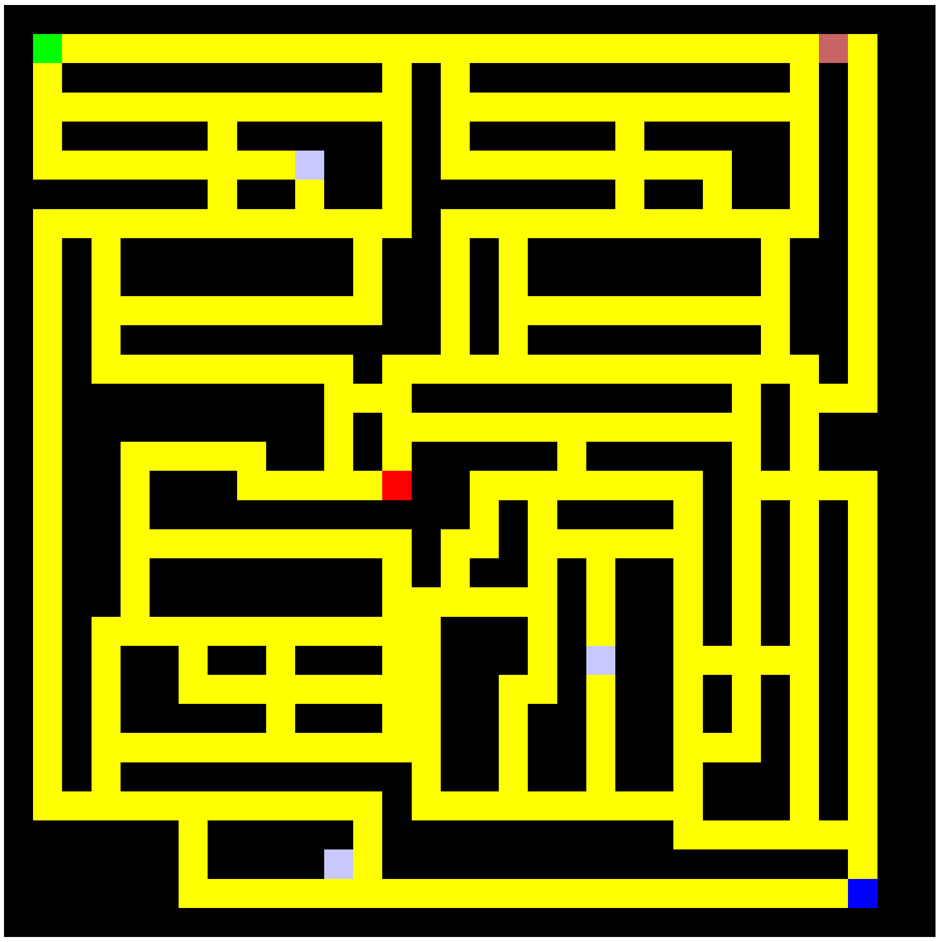


Figure 6.1: Level 1 source image

6.1.2 Rendered level

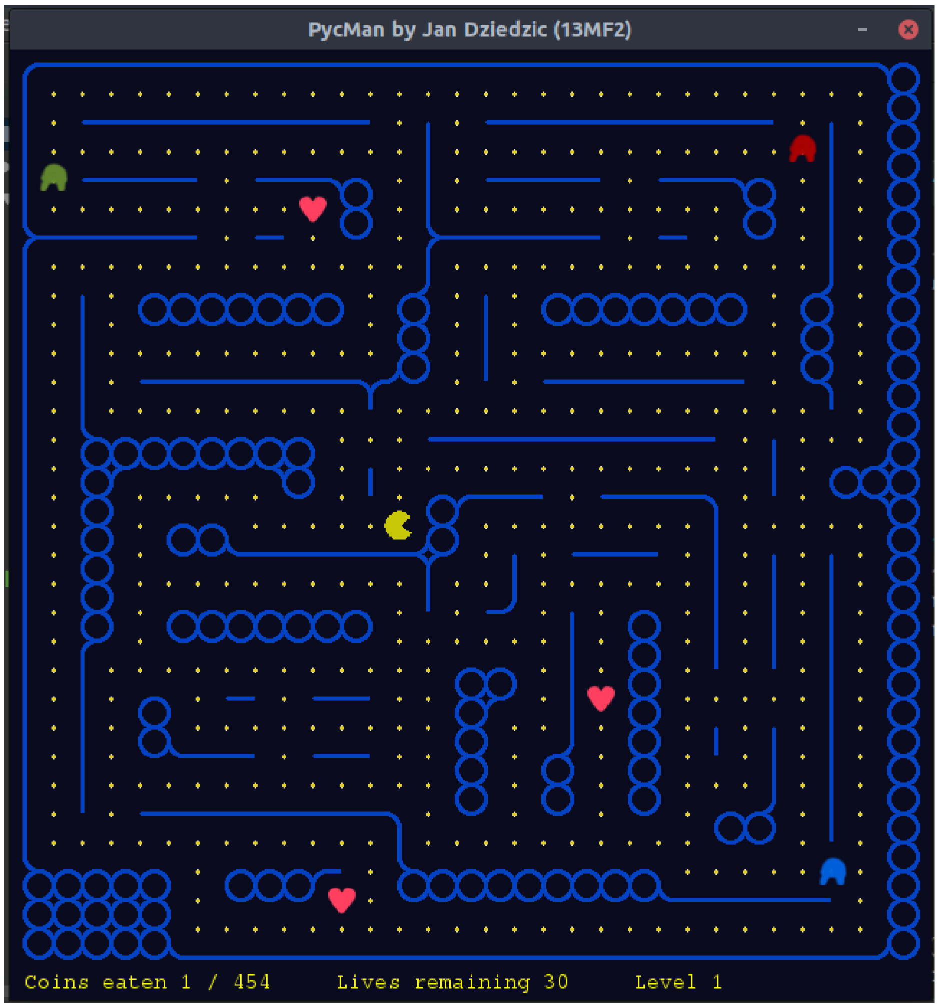


Figure 6.2: Level 1 rendered in game

6.2 Level 2

6.2.1 Source image

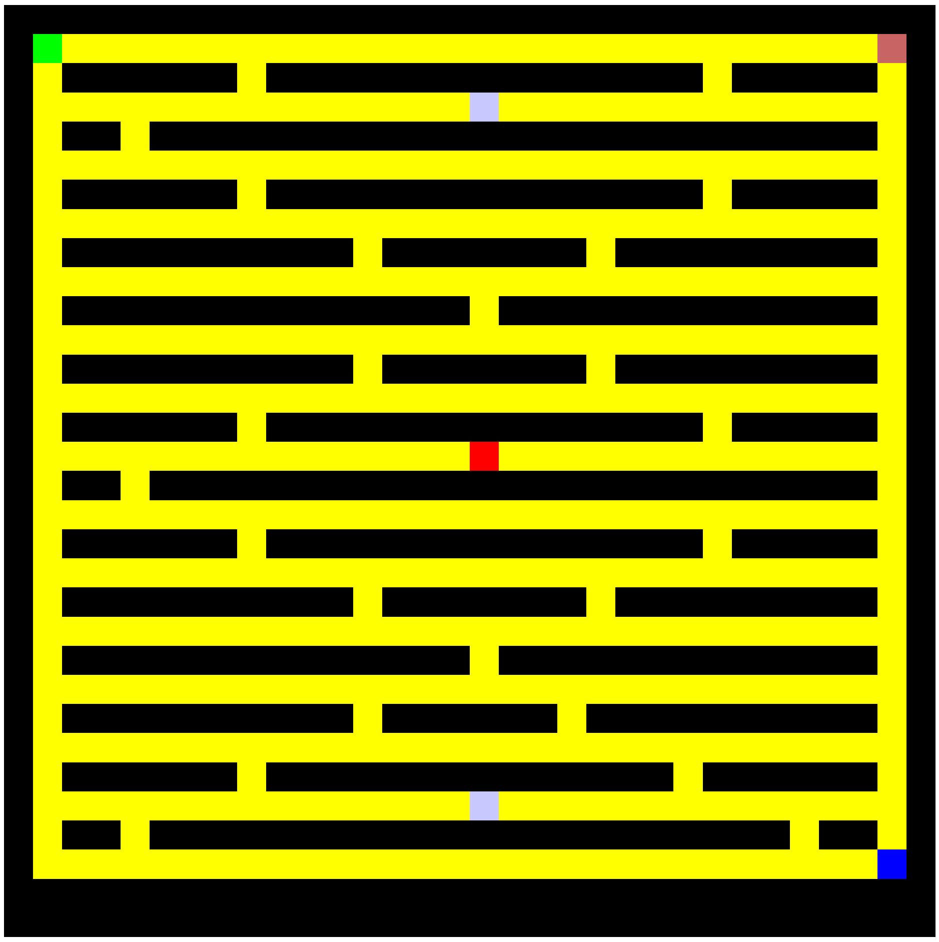


Figure 6.3: Level 2 source image

6.2.2 Rendered level

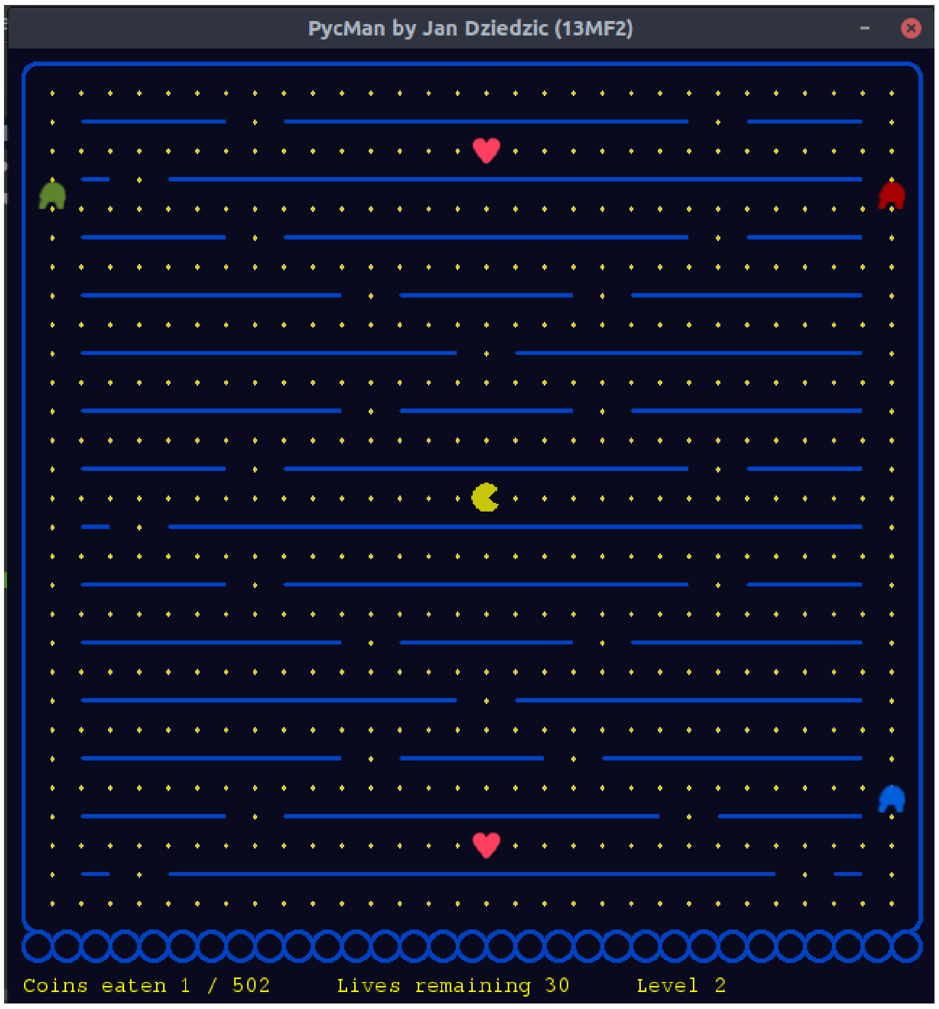


Figure 6.4: Level 2 rendered in game

6.3 Level 3

6.3.1 Source image

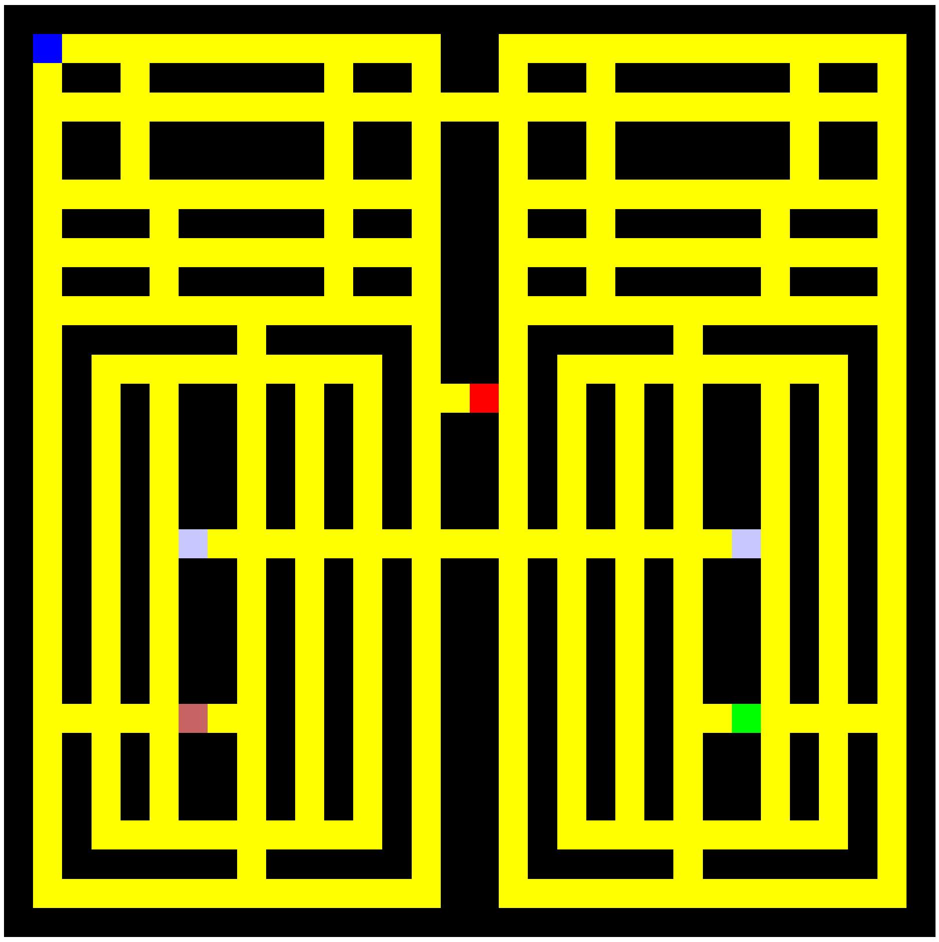


Figure 6.5: Level 3 source image

6.3.2 Rendered level

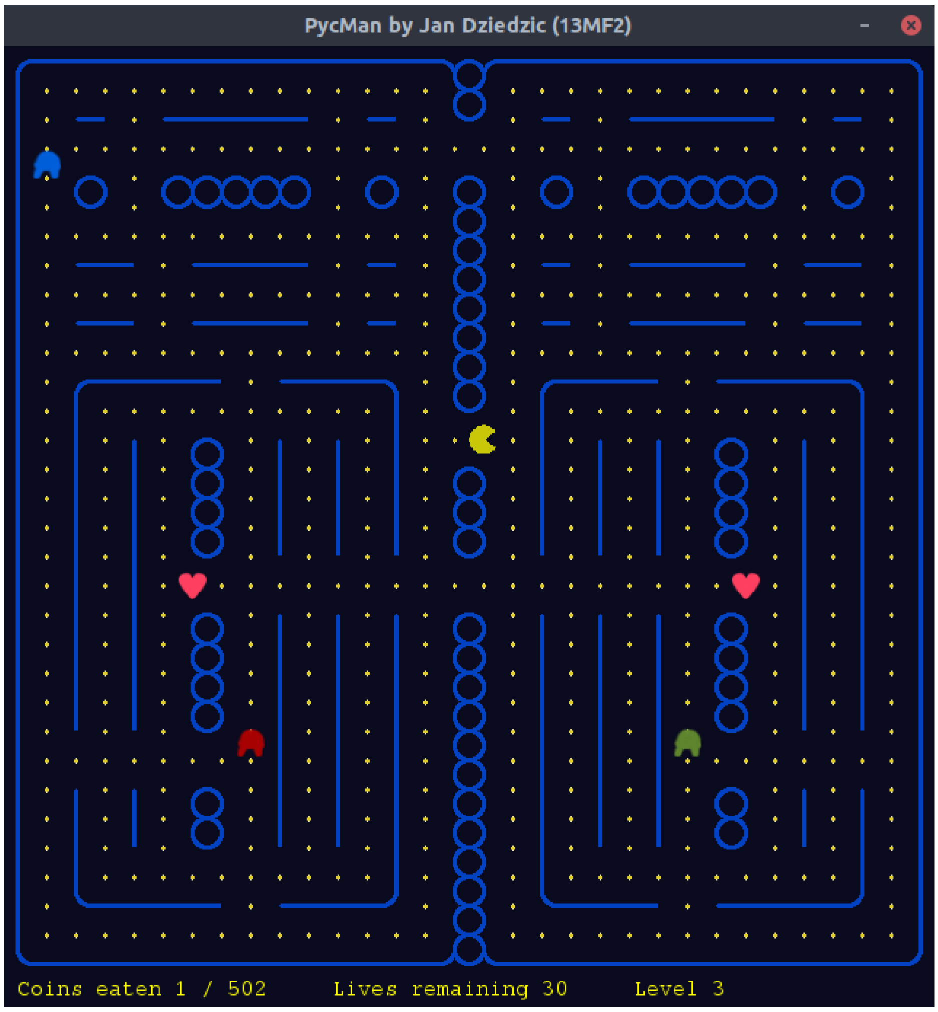


Figure 6.6: Level 3 rendered in game

6.4 Level 4

6.4.1 Source image

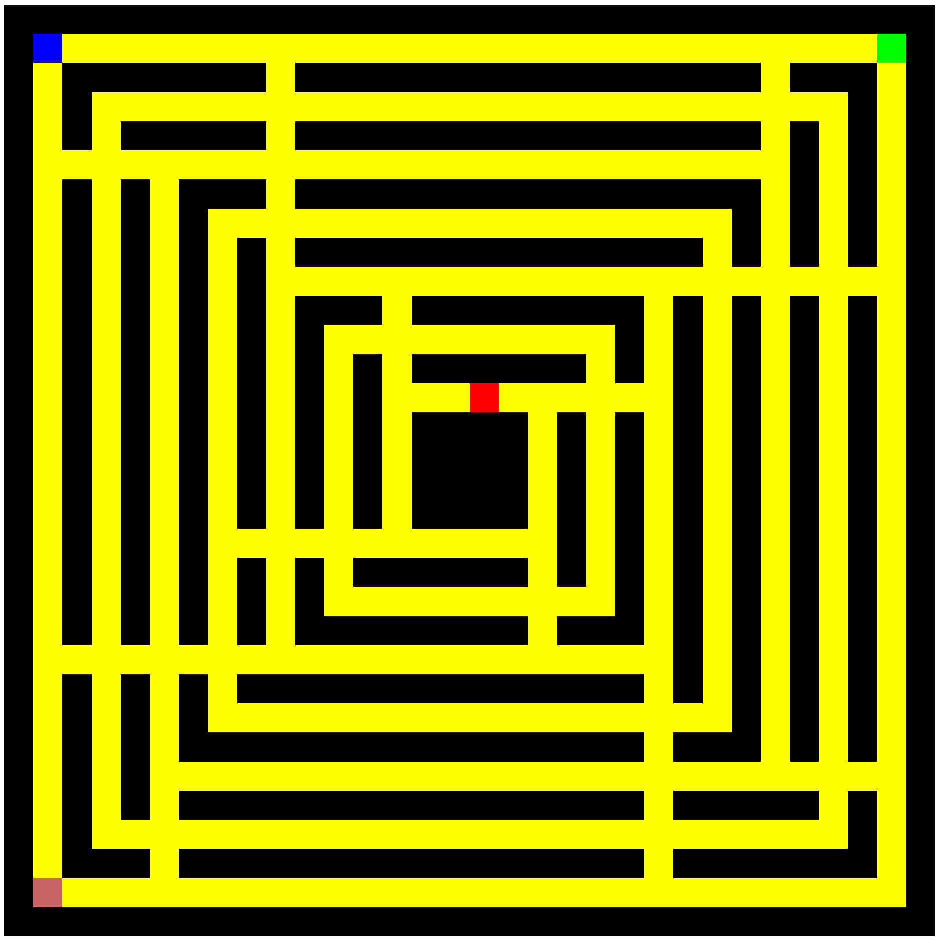


Figure 6.7: Level 4 source image

6.4.2 Rendered level

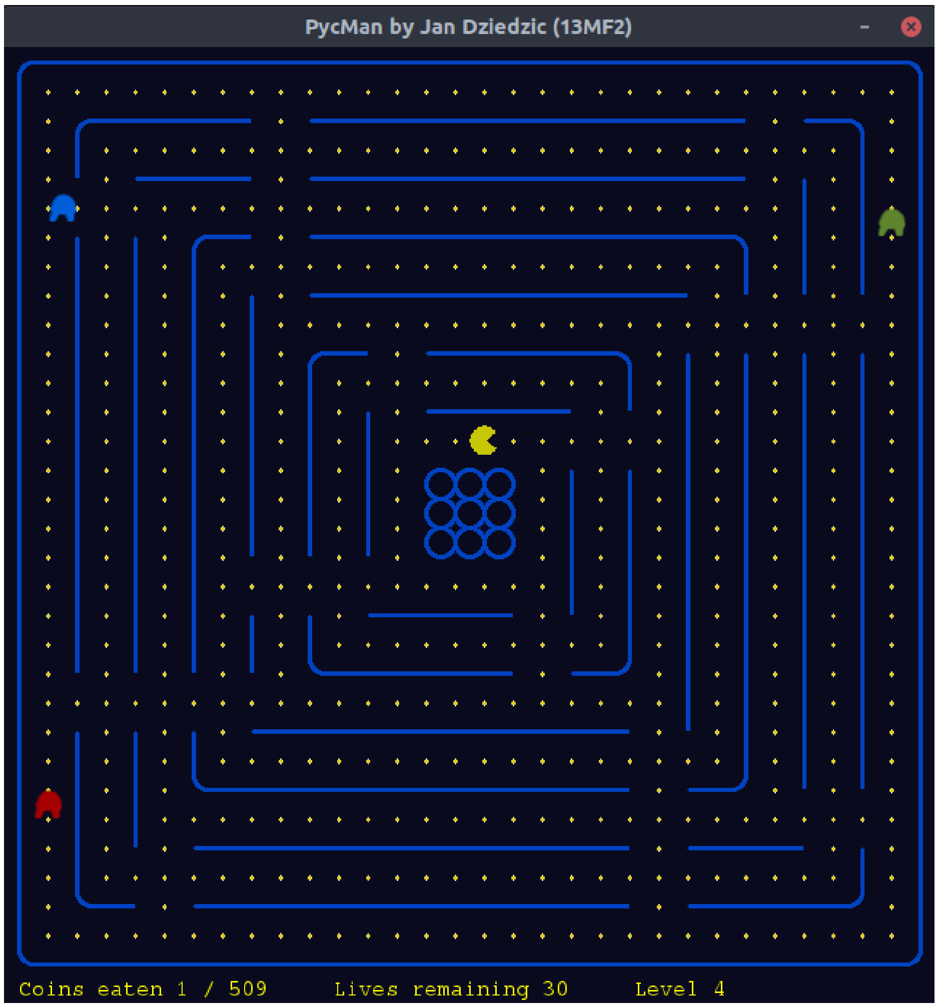


Figure 6.8: Level 4 rendered in game

6.5 Level 5

6.5.1 Source image

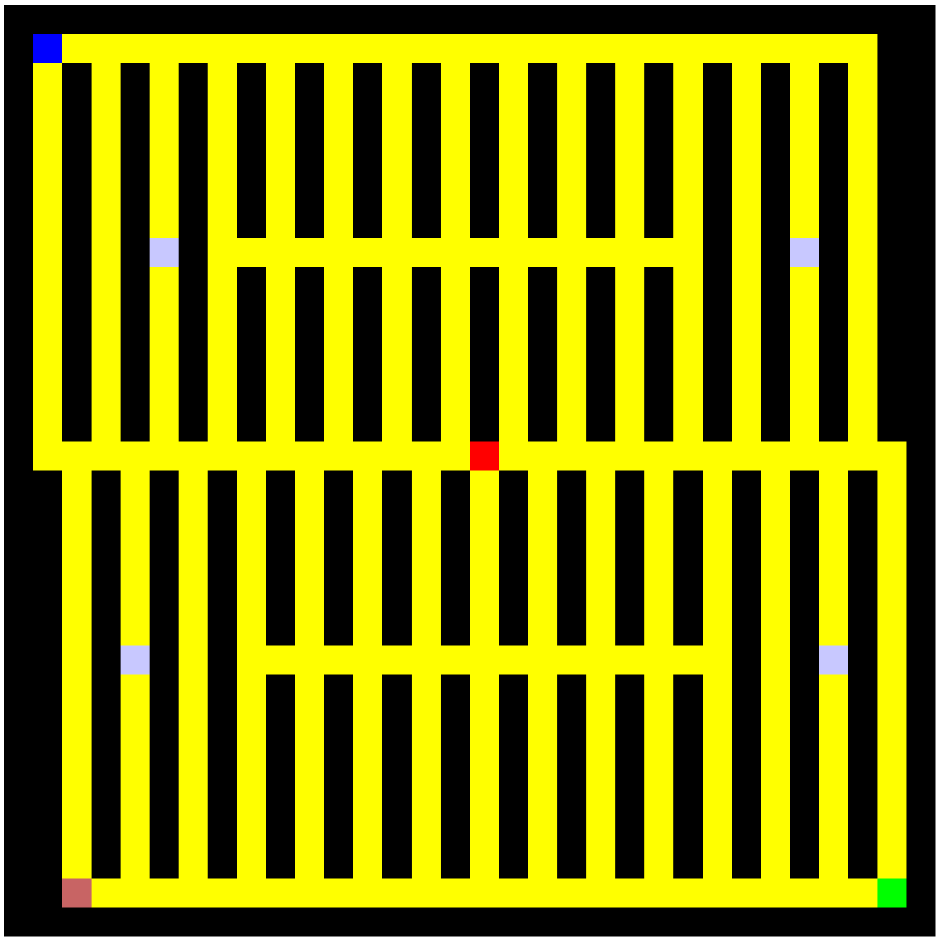


Figure 6.9: Level 5 source image

6.5.2 Rendered level

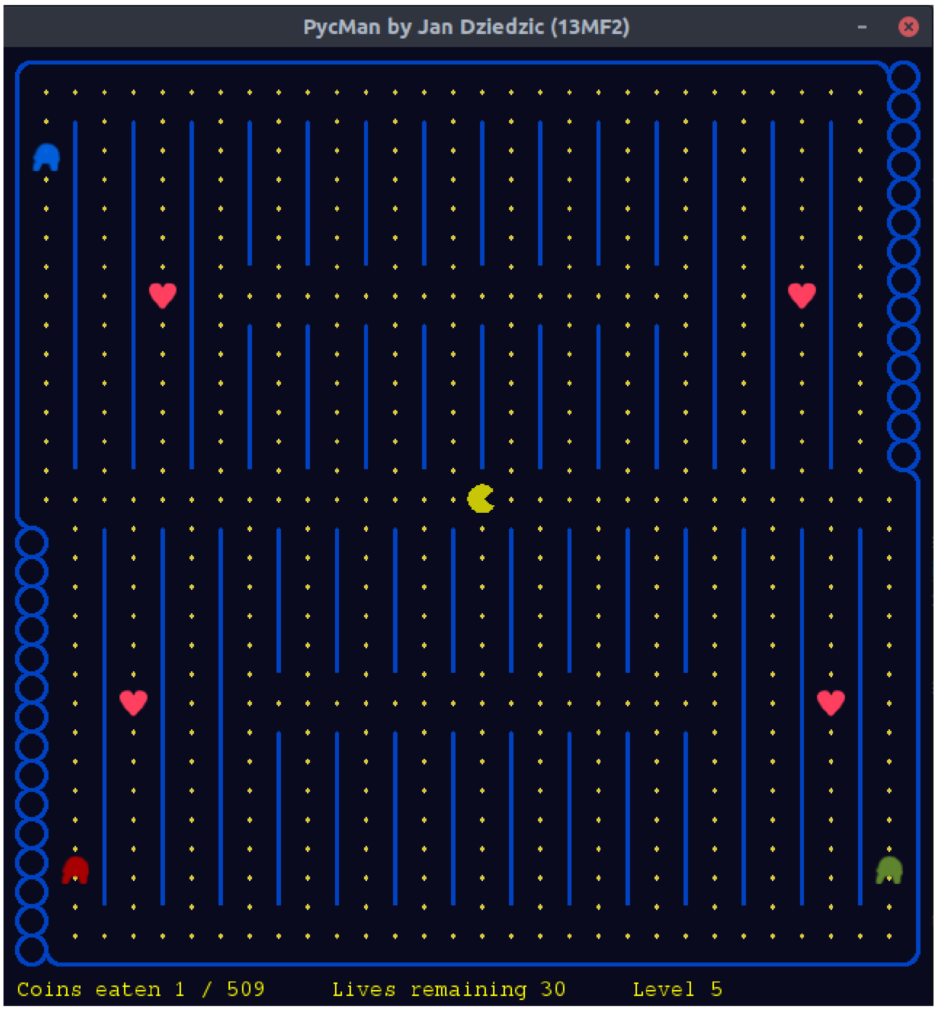


Figure 6.10: Level 5 rendered in game

6.6 Level 6

6.6.1 Source image

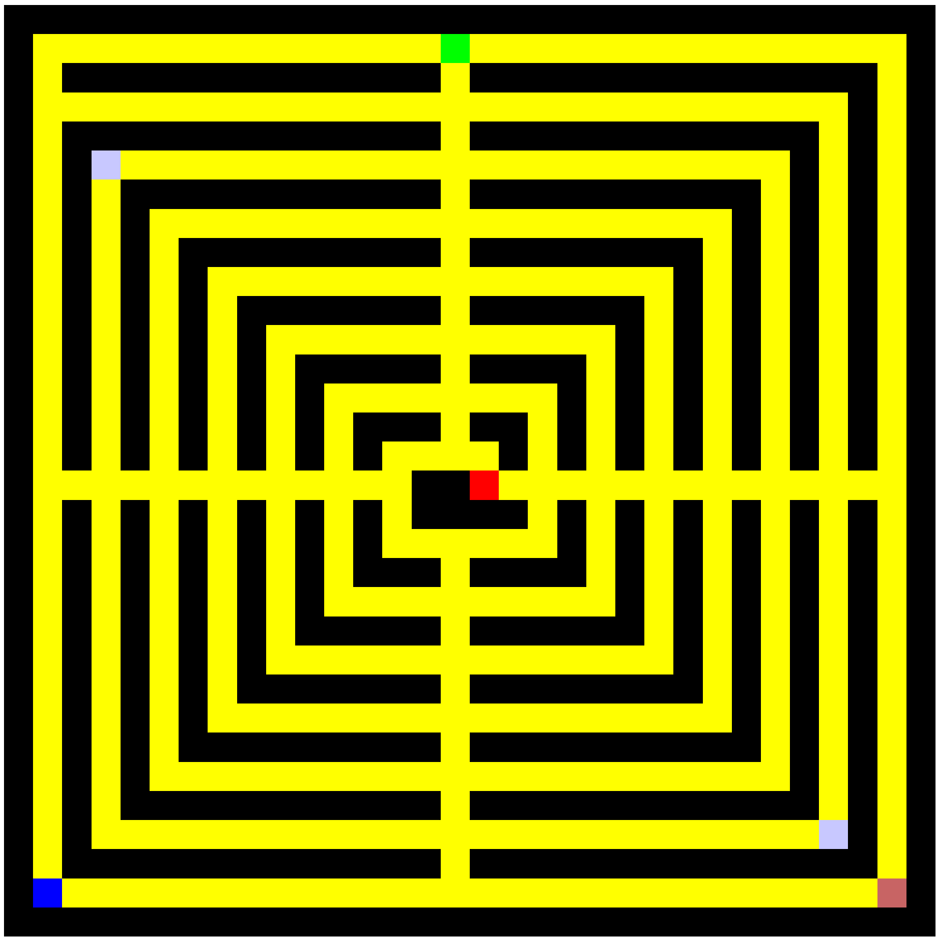


Figure 6.11: Level 6 source image

6.6.2 Rendered level

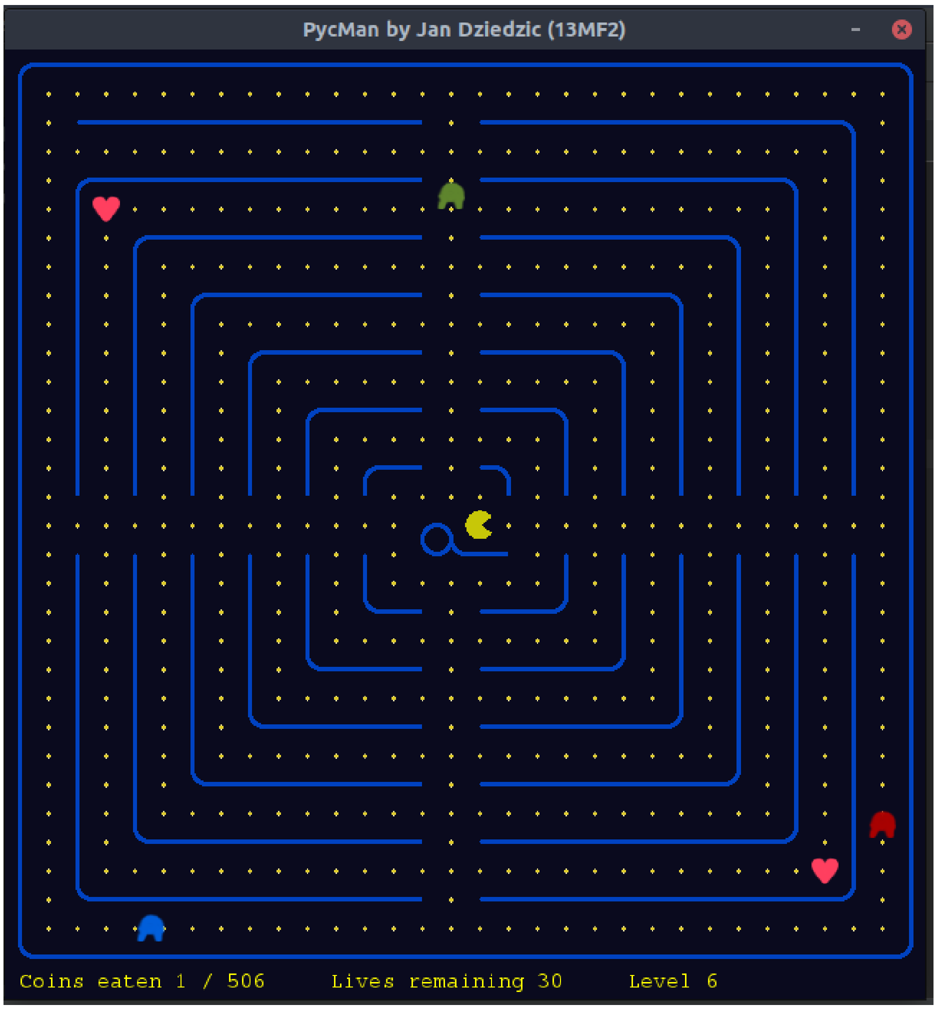


Figure 6.12: Level 6 rendered in game

Chapter 7

Guide to creating new levels

Recommended software: GIMP version 2.3 or higher.
Recommended hardware: screen, keyboard, mouse or a touch/stylus interface.

- Create a 32 x 32 RGB bitmap in .bmp format.
- Fill it with black color (255, 255, 255)
- Use pencil tool with a solid tip of size 1px to draw paths for the Movables, refer to Section 2.4.1 for available colors. It is strongly discouraged not to set border of the image as color black (255, 255, 255) symbolizing walls. While this shouldn't produce an error, this may negatively impact the gameplay.
- Again, use the same tool to place the player, ghosts, coins and hearts according to the above-mentioned specification. Remember that there should be precisely one player tile. You can add as many ghosts of all the three colors as you want. You may also decide not to include some.
- Place the bitmap in the resources/levels directory.
- Update 'totallevels' in recources/constants.py to indicate how many levels are in the folder.
- Add filename of your new level source image to 'levelorder' dictionary in resources/paths.py.
- Play your newly modified game!

Chapter 8

References

1. https://goo.gl/36A62h
2. https://www.pygame.org/docs