INF-1400: Objektorientert programmering

Assignment 3: Mayhem

Håkon Silseth

April 10, 2023

1 Introduction

This assignment is to create a clone of the game Mayhem in python using the pygame module. The Mayhem clone is a game consisting of two players controlling their ship. Each player has 4 controls, rotate left, rotate right, accelerate and shoot. The players can also refuel by landing on a landing pad, and they have to avoid obstacles along the way.

2 Technical Background

The implementation is not very technical and therefore does not warrant a long technical background, but there is still some that should be explained.

2.1 Sprites

The pygame module has a built in class called Sprite, which makes the Mayhem clone implementation much simpler. The sprite base class need to have a surface and a rect. The surface is what we see on screen, usually an image, but can also just be a surface. The rect is a rectangle that defines the bounds of the sprite. What makes sprites so great to work with is that we can create sprite groups and instead of dealing with separate sprites we mostly can just deal with the sprite group. We can draw and update the group and all the sprites in the group will be drawn and updated. Collisions are handled very easily using sprites, there are methods that check if two 'rects' overlap and return a boolean statement based on that.

3 Design & Implementation

As per the technical requirements of the assignment, the implementation follows the basic principles of object oriented programming (OOP). Figure 1 shows the structure of the code (note that although it is not shown in figure 1, all classes except Game inherit from pygame.sprite.Sprite).

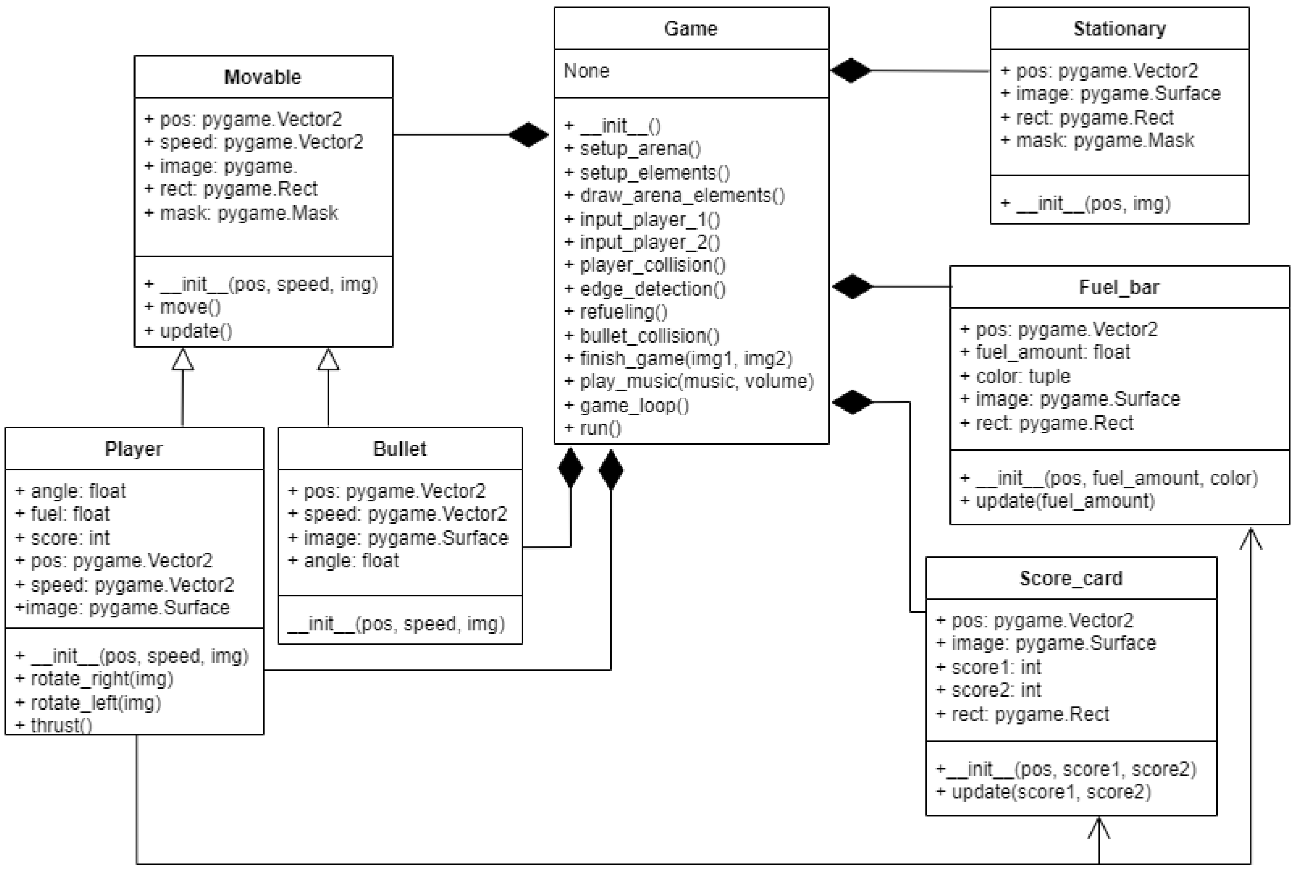


Figure 1: UML diagram of the Mayhem clone

The implementation is made up of one parent class for the movable objects, which lays the groundwork for how these moving objects behave. Then the classes describing the player and bullet objects inherit from this parent class, these two classes then have their own properties defined such that they will behave in the desired manner. For the stationary objects there are three separate classes, first we have Stationary which describes all objects that are stationary and a part of the game, such as the obstacles. All stationary objects are the same, only their appearance is different* so they can be described by the same class. Then there are two other classes whose purpose is to display information on the screen, these are the Fuel_bar and Score_card classes. They are different enough from the Movable class and each other that they warrant their own classes. Finally we have the Game class which puts everything together, the mechanics of the game and the relations between the classes, and instantiating the other classes, all of this happens here, as well as the main game loop.
*Technically they are different in the way that they interact with other classes, but this is handled in the Game class.

3.1 Detailed explanation

3.1.1 Movable

This parent class, as mentioned, lays the groundwork for all moving objects. It initializes an object with a position, speed, and its image, and also defines its 'rect' and 'mask'. It also has two other methods: move() which updates its position and speed, and update() which calls on the move() function.

3.1.2 Player

This class inherits from Movable, and is the blueprint for the player objects in the game. The player is initialized with a given amount of fuel which is equal to the max amount it can have, an angle of 0 as well as a score of 0. This class contains 3 methods, rotate_right() and rotate_left() are essentially the same, but as the names imply they rotate the player in the opposite direction. These classes will rotate the player in their direction by a given amount and return the rotated image and rect. Finally this class also has a method called thrust(), this method first checks if the players fuel is above 0 and if so it will create a thrust vector, rotate it in place such that it points in the same direction as the player, and then add it to the players speed vector, also it will reduce the amount of fuel the player has. These three methods correspond to 3 of the 4 actions a player can do, the last one (shooting) is implemented later in the Game class, this is because it needs to access the Bullet class and the Bullet groups.

3.1.3 Bullet

This class is very straight forward, it is initialized with an angle and a speed that points in that direction, other than that it inherits everything from the Movable class.

3.1.4 Stationary

This is a basic class, used for the landing pads and the obstacles. This class will initialize an object at a given position with a given image, at it defines the objects rect and sprite. There is nothing more to this class, all of the mechanics of how they integrate into the game is handled later on.

3.1.5 Fuel_bar

This class is to represent the fuel amount of a player. The class initializes a rectangular pygame.Surface, which is just a rectangle, and its height is defined by the fuel amount the player this object is associated to has left. It also draws an outline around the fuel bar, which is just a png image. This class also has an update() method which redraws the fuel bar, but takes in the fuel amount as an argument, such that the fuel bar is updated so its height represents the amount of fuel the player has left.

3.1.6 Score_card

This class is to represent the score of each player. An object of this class is initialized with a background image and the scores of the players. This class has an update() method which takes in the scores of each player, this method just draws the score on top of the image again, but with the current score, so if the score has changed it will be updated. Actually inputting the score is handled in the Game class.

3.1.7 Game

This is where the magic happens, this class utilizes the previously described classes to actually create the game. The init() method for this functions sets up all the in game elements such as the players, the obstacles, score card etc. as well as their sprite groups. It does this by calling on two other methods (setup_elements() and setup_arena()). These three methods don't really warrant any explanation, they just instantiate the other classes and places them on the screen. To describe how the game is set up let's skip to the run() method, this method creates a while loop in which will repeatedly call upon another method called game_loop() until we exit the game. game_loop() just calls on the other methods in this class in the right order, to explain the implementation, it's best to just go through this method and explain all the methods along the way.

Firstly we call on the method draw_arena_elements() what this method does is draw all the elements except the players and bullets on the screen. It does this by using the draw() method built in to the Sprite class that all the classes inherit from, it uses this method on the groups the objects are divided into. Then we call on the player_collision() method. This method iterates through each player in the player sprite group and checks if there has been a collision with another object and performs the corresponding actions. If the player has collided with a landing pad we need to check the y component of the players speed, if its moving downwards then that means that the player has landed on the landing pad, then the y component of its motion will be set to 0 and the x component is set to reduce by 1 percent (that is one percent for each iteration of the loop, which is run through 60 time a second) which will brake the player. If the player is moving upwards however, that must mean that the player has collided with the bottom of the landing pad, which means that it hasn't landed properly, so its speed is multiplied with -1, in other words the player will bounce of the landing pad. If the collision is not with a landing pad but rather an obstacle, the player will bounce off in the same way as if it hits the bottom of a landing pad. Also the player is punished by having its score reduced by 1 (the deduction for crashing can be modified in the config.py file). Finally if the player collides with another player, both players speed is multiplied with -1, bouncing them off each other and both will lose a point.

Then the edge_detection() method is called, this method iterates through the players and checks their position. If a player is 50 pixels out of the screen they have immediately lost the game, this is done by using if statements and if their position is outside the accepted range they get deducted twice the minimum amount of points, which ensure that they get below the minimum score, which results in them losing instantly.

Then bullet_collision() is called, this method iterates through all bullets and checks if they have collided with anything, if they have collided with anything that is not a player, then the bullet gets removed, this also happens if it goes of the screen. The bullets are divided into two bullet groups, one for each player, if a bullet collides with the other player, then the bullet is removed and the player that got shot loses a point and the other one gains a point.

We also need to check for input from the players so the next methods to be called upon are input_player_1() and input_player_2(). These methods check if a certain key has been pressed, and then performs the appropriate action. For example, if the right arrow key has been pressed, it will call on the rotate_right() method for the player_1 object, to rotate player 1.

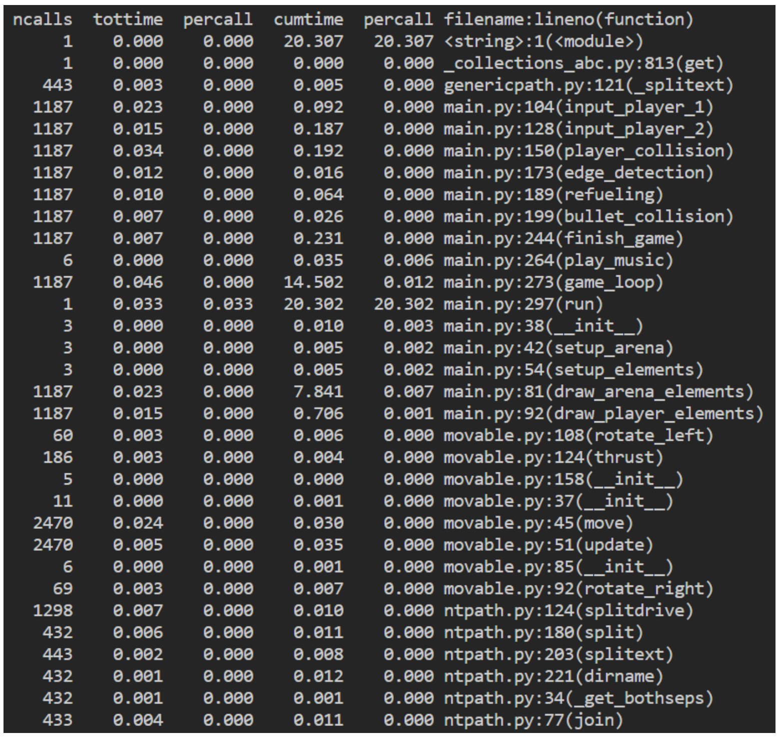
Next up we call on the method called refueling(), what this method does is check for collisions between the players and the landing pads. If there is a collision between a landing pad and a player, we check if the player has more than the max amount of fuel, if not then we increase the players fuel.

Then we call on methods to update the fuel bars, they take in the amount of fuel for each player and updates the height of the fuel bar to be proportional to the amount of fuel the player has.

Then we need to update the score, this method just draws the score of each player at the top of the screen, so if a players score increases this method will update the text that is displayed on screen.
Finally we draw the player elements (players and bullets) these are drawn so late in the loop to ensure that they are displayed above all other elements. Then we also check if a player has a high or low enough score to win/lose the game, and if so display the relevant image on the screen, also, we initialize the game again if the space bar is pressed.

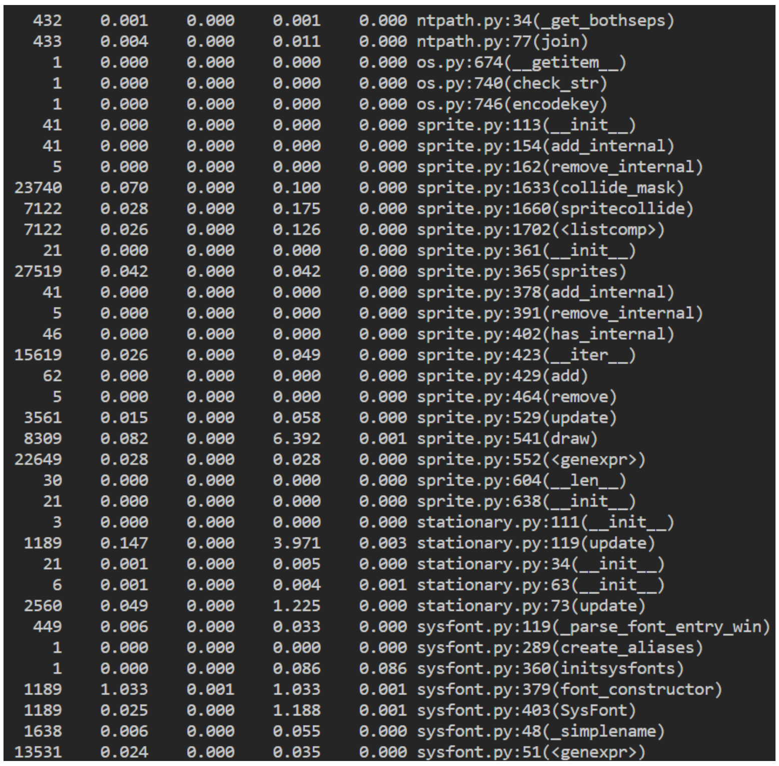
4 Discussion & Results

The implementation works well, and does what it is supposed to, but there are of course some bugs. The main issue is the way the collisions with the landing pads are handled. The players vertical speed is set to zero so that they can't move further into the landing pad. The problem with this is that if the thrust is applied so that the player accelerates towards the landing pad, then they will go 'inside' the landing pad. They can get out again, and it only happens when the player accelerates into the landing pad, but its still an issue that needs resolving. Other than this there are only minor things that could be modified, but not really any that cause a problem with the game.



```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000   20.307   20.307 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 _collections_abc.py:813(get)
   443    0.003    0.000    0.005    0.000 genericpath.py:121(_splitext)
  1187    0.023    0.000    0.092    0.000 main.py:104(input_player_1)
  1187    0.015    0.000    0.187    0.000 main.py:128(input_player_2)
  1187    0.034    0.000    0.192    0.000 main.py:150(player_collision)
  1187    0.012    0.000    0.016    0.000 main.py:173(edge_detection)
  1187    0.010    0.000    0.064    0.000 main.py:189(refueling)
  1187    0.007    0.000    0.026    0.000 main.py:199(bullet_collision)
  1187    0.007    0.000    0.231    0.000 main.py:244(finish_game)
     6    0.000    0.000    0.035    0.006 main.py:264(play_music)
  1187    0.046    0.000   14.502    0.012 main.py:273(game_loop)
     1    0.033    0.033   20.302   20.302 main.py:297(run)
     3    0.000    0.000    0.010    0.003 main.py:38(__init__)
     3    0.000    0.000    0.005    0.002 main.py:42(setup_arena)
     3    0.000    0.000    0.005    0.002 main.py:54(setup_elements)
  1187    0.023    0.000    7.841    0.007 main.py:81(draw_arena_elements)
  1187    0.015    0.000    0.706    0.001 main.py:92(draw_player_elements)
    60    0.003    0.000    0.006    0.000 movable.py:108(rotate_left)
   186    0.003    0.000    0.004    0.000 movable.py:124(thrust)
     5    0.000    0.000    0.000    0.000 movable.py:158(__init__)
    11    0.000    0.000    0.001    0.000 movable.py:37(__init__)
  2470    0.024    0.000    0.030    0.000 movable.py:45(move)
  2470    0.005    0.000    0.035    0.000 movable.py:51(update)
     6    0.000    0.000    0.001    0.000 movable.py:85(__init__)
    69    0.003    0.000    0.007    0.000 movable.py:92(rotate_right)
  1298    0.007    0.000    0.010    0.000 ntpath.py:124(splitdrive)
   432    0.006    0.000    0.011    0.000 ntpath.py:180(split)
   443    0.002    0.000    0.008    0.000 ntpath.py:203(splitext)
   432    0.001    0.000    0.012    0.000 ntpath.py:221(dirname)
   432    0.001    0.000    0.001    0.000 ntpath.py:34(_get_bothseps)
   433    0.004    0.000    0.011    0.000 ntpath.py:77(join)
```

Figure 2: Cprofiler



```
  432   0.001    0.000    0.001    0.000 ntpath.py:34(_get_bothseps)
  433   0.004    0.000    0.011    0.000 ntpath.py:77(join)
    1   0.000    0.000    0.000    0.000 os.py:674(__getitem__)
    1   0.000    0.000    0.000    0.000 os.py:740(check_str)
    1   0.000    0.000    0.000    0.000 os.py:746(encodekey)
   41   0.000    0.000    0.000    0.000 sprite.py:113(__init__)
   41   0.000    0.000    0.000    0.000 sprite.py:154(add_internal)
    5   0.000    0.000    0.000    0.000 sprite.py:162(remove_internal)
23740   0.070    0.000    0.100    0.000 sprite.py:1633(collide_mask)
 7122   0.028    0.000    0.175    0.000 sprite.py:1660(spritecollide)
 7122   0.026    0.000    0.126    0.000 sprite.py:1702(<listcomp>)
   21   0.000    0.000    0.000    0.000 sprite.py:361(__init__)
27519   0.042    0.000    0.042    0.000 sprite.py:365(sprites)
   41   0.000    0.000    0.000    0.000 sprite.py:378(add_internal)
    5   0.000    0.000    0.000    0.000 sprite.py:391(remove_internal)
   46   0.000    0.000    0.000    0.000 sprite.py:402(has_internal)
15619   0.026    0.000    0.049    0.000 sprite.py:423(__iter__)
   62   0.000    0.000    0.000    0.000 sprite.py:429(add)
    5   0.000    0.000    0.000    0.000 sprite.py:464(remove)
 3561   0.015    0.000    0.058    0.000 sprite.py:529(update)
 8309   0.082    0.000    6.392    0.001 sprite.py:541(draw)
22649   0.028    0.000    0.028    0.000 sprite.py:552(<genexpr>)
   30   0.000    0.000    0.000    0.000 sprite.py:604(__len__)
   21   0.000    0.000    0.000    0.000 sprite.py:638(__init__)
    3   0.000    0.000    0.000    0.000 stationary.py:111(__init__)
 1189   0.147    0.000    3.971    0.003 stationary.py:119(update)
   21   0.001    0.000    0.005    0.000 stationary.py:34(__init__)
    6   0.001    0.000    0.004    0.001 stationary.py:63(__init__)
 2560   0.049    0.000    1.225    0.000 stationary.py:73(update)
  449   0.006    0.000    0.033    0.000 sysfont.py:119(_parse_font_entry_win)
    1   0.000    0.000    0.000    0.000 sysfont.py:289(create_aliases)
    1   0.000    0.000    0.086    0.086 sysfont.py:360(initsysfonts)
 1189   1.033    0.001    1.033    0.001 sysfont.py:379(font_constructor)
 1189   0.025    0.000    1.188    0.001 sysfont.py:403(SysFont)
 1638   0.006    0.000    0.055    0.000 sysfont.py:48(_simplename)
13531   0.024    0.000    0.035    0.000 sysfont.py:51(<genexpr>)
```

Figure 3: Cprofiler

Figures 2 and 3 show the results of using Cprofiler on the code while playing a quick round of the game. From this we can get information on the performance of the game, and what takes time, and if there are any potential bottlenecks that impair the performance of the game. Looking at the results we can see that there are not really any processes that impair the performance of the game, but one thing that stands out is rendering the text. Acquiring the font ant displaying the text seems to take a lot of time compared to the rest of the processes. It does not affect much in this game, but if the game was more intensive and used more text, then this might be a problem. Other than this there is not much else that creates a bottleneck, checking for collisions takes some time, but this is a necessary process and also effect is not to great.

5 Conclusion

To sum up it is safe to conclude that the implementation is a successful clone of the game Mayhem which fulfills all the criteria laid out in the task. While there are a few bugs the game works mostly as intended, and is both playable and enjoyable.

References

[1] Pygame documentation (Retrieved 04.04.2023), from https://www.pygame.org/docs/