Assignment 3: Mayhem report - INF-1400

Sigurd Uhre and Martin H. Evenseth

April 2021

1 Introduction

This assignment is divided into to parts, part 1 and part 2. Part 1 is to implement a clone of the classic amiga game Mayhem. Part two is to answer five theoretical questions. This report is made for part 1. The assignment is solved using object orientated programming principles, using classes and methods. The gameplay requirements are as follows:

•Two spaceships with four controls: rotate left, rotate right, thrust and fire.

•Minimum one obstacle in the game world. This can be as simple as a single rectangle in the middle of the screen.

•Spaceship can crash with walls/obstacles/other spaceship.

•Gravity acts on spaceships (the original has no gravity acting on the bullets, but you can choose what works best).

•Each player has a score that is displayed on the screen. A player's score increases when he shoots down the opponent. A player's score decreases if he crashes.

•Each spaceship has a limited amount of fuel. To refuel, it must land on one of two landing pads. Alternatively, you can put a "fuel barrel"at a random position that is collected by the first spaceship reaching it.

2 Background

The assignment is solved using the pygame library in python. From the pygame library the sprite class is used as a parent class for all classes using inheritance. The group method from the sprite class has been a focus, were updating and drawing in the script are performed using group methods.

3 Design

In this design, our objects fit together as shown in the UML-diagram below. We have based our design on the usage of classes for the different components of the game. All classes inherit from the "parent class" sprites. Sprite are 2D animation or images overlaid into a screen.

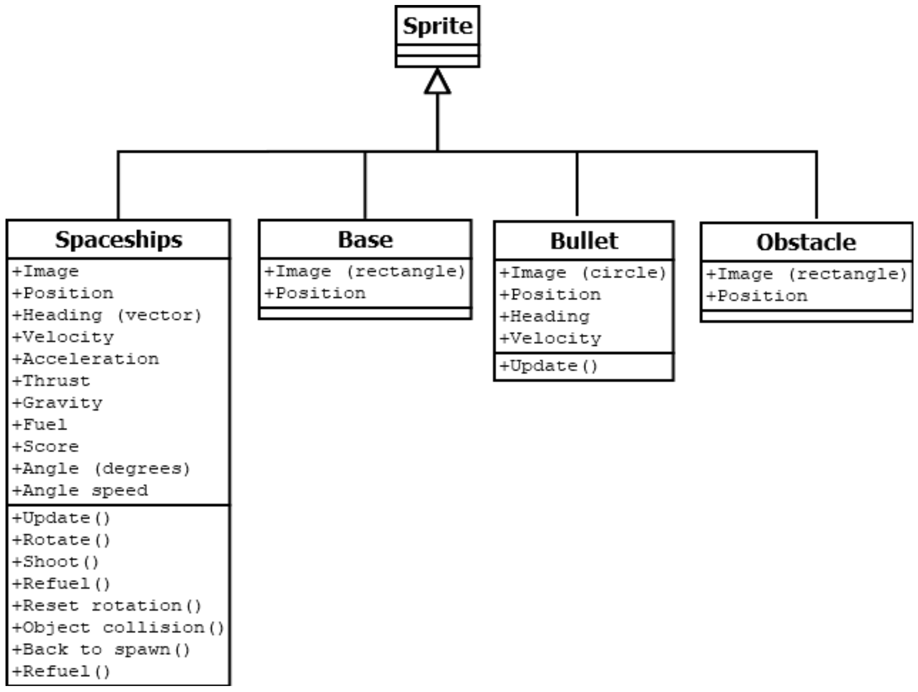


Figure 1: UML diagram for classes

The game code include four classes: spaceship, base, bullet and obstacles. The sprites sub classes have been put into groups with the usage of sprite.Group method. The updating and drawing have been preformed with the usage of Group.update and Group.draw. Inside each class-box (illustrated in figure 1) the attributes of the class is lined up first, followed by the classes methods (if any) in the lower box. Variables such as screen size, colors, spaceship size, bullet size and images is organized separately from the main file in config.py.

4 Implementation

As one of the requirements, all classes inherits from the Sprite class. For the implementation of the spaceship we have used an PNG image of a spaceship to visualize the object. The spaceship class take in two parameters, image and position. The parameters initiated inside the init function is shown in figure 1. The first method we made was rotate. Here we have used pygame's transform.rotozoom to be able to rotate the spaceship. Another success criteria was to always rotate from the original image, else the image got blurry and the game crashed.

The second method we made was the object collision. Here we handle all collision between objects. It works such that when one object position is the same as another objects position, they crash. When the spaceship "collides" with another object, it is sent back to the base/spawn point. Back to spawn is the third method. This method resets variables such as velocity, acceleration, angle, and fuel. It also subtracts an amount from the score of the player. The fourth method is a simple method only for re-setting the rotation. It is used inside the back to spawn method.

The fifth method is refuel. This method also resets some variable and fills the fuel tank. The last method for the spaceship is the update method, which is a premade method from the Sprite class. Inside the update method we convert angle in degrees to coordinates in the unit circe (heading), we set the max thrust, and we combine thrust, heading and gravity to update the position of the spaceship. We also included the call of the rotation function and the bullet class if a given key was pressed, due to that this made the rotation smooth for gameplay. Gravity is also handled here. Gravity is activated on the spaceship as soon as it leaves the base (spaceship y-position is lower than a certain value). The last function in the update method for the spaceship is the call of the refuel method if the spaceship is positioned in the base.

The base is visually a rectangle, implemented using pygame's "surface" and "draw.rect" at a given position on the display.

The bullets are visually a circle, implemented using pygame's "surface" and "draw.circle" at a given position on the display. The bullet class take in two parameteres, position and heading. Inside the init function a bullet velocity is created. The class has only one method named update. Here the position of the bullet is updated using heading and bullet velocity. Also the bullets are removed when they leave the display or collides with an obstacle (same position as obstacles).

The last class, obstacle, take in one parameter, position. It is visually a rectangle, implemented using pygame's "surface" and "draw.rect" at a given position on the display

All objects are implemented calling the classes and putting them inside sprite groups. The sprite group is then drawn and updated inside the game loop. Also the thrust is added to the spaceship inside the game loop when pressing a certain key, while at the same time removing fuel when thrusting.

5 Evaluation

In this assignment, all the requirements for the game play are fulfilled with the inclusion of rotation, thrust and shooting. Obstacles are successfully implemented and destroys both spaceships and bullets if there are any interactions between them. The gravity is implemented and adjusted to the strength of the thrust to make the flying experience some how manageable.

The gravity is not implemented to work on the bullets. Each player have a certain amount of fuel and score displayed above the starting position on the screen. The players score increases the longer they stay "in air" and is given a jump in score if the other player is hit. Likewise, the players lose score if they are hit by bullets or crash into objects, making it rewarding to stay alive and avoid crashing. The players also have an limited amount of fuel in their spaceship. The fuel tank is reduced whenever thrust is used to maneuver the map. The spawn point for each player is also the fuel-refill point, here each player can land on both stations to refill their tanks.

6 Discussion

The game is fully playable and all the in-game requirements are successfully implemented. However, there are several aspects of the implementation that we consider to be less optimized. One of them is the method we used to "hard-code" the hit box of the obstacles to fit the position of the displayed obstacles. This means that we are unable to change the size and position of the obstacles, without also having to manually change the hit box of the obstacles as well.

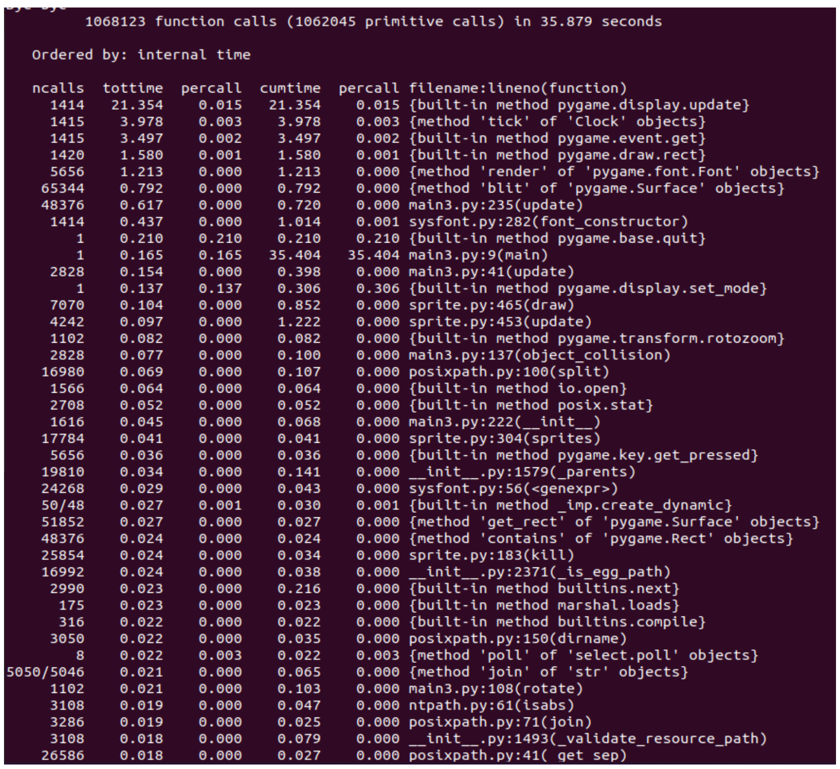
If we wanted to continue developing this game further, this design may not be fully optimal. This collision detection is located in the spaceship class, which means that if we were to add a new "flying object" class we had to do the same bit of code all over again for it to collide with the objects.

Double bullets for the left spaceship is another aspect of the implementation that seems to be less optimized and some how problematic for the game play. Spaceship 2 as its also known as, fires twice as many bullets as the the spaceship on the right. We have troubleshooted this problem several times without any success. We drafted one possible reasons for it: The player on the left controls two spaceships on top of each other, making it twice as many bullets. Rotation an thrust is not enough to distinguish these two spaceships, only when holding down the ”shoot” key we can see the differences in the projectiles orientation.

The creation and blitting of the text and score could also be moved into a class or a function to make the code more structured. Also the calling of the object collision method could have/should have been moved to the spaceship update function. To make it a more competitive game we could have included a timer to end the game after a given time, or after a player had reached a given score.

Figure 2 shows a print of the cProfiler running our code in 35 seconds. It's sorted by total time for each execution. We see that the built in method pygame.display.update uses the most time by far with 21 seconds. Method clock.tick comes second with 4 seconds, and pygame.event.get comes third by 3.5 seconds. The render method of pygame.font also comes up high with 1.2 seconds.

Given more time to optimize this code, we would have focus to improve the events of pygame.event.get. This is due to that we are adding thrust and remove fuel inside this execution.



```
         1068123 function calls (1062045 primitive calls) in 35.879 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1414   21.354    0.015   21.354    0.015 {built-in method pygame.display.update}
     1415    3.978    0.003    3.978    0.003 {method 'tick' of 'Clock' objects}
     1415    3.497    0.002    3.497    0.002 {built-in method pygame.event.get}
     1420    1.580    0.001    1.580    0.001 {built-in method pygame.draw.rect}
     5656    1.213    0.000    1.213    0.000 {method 'render' of 'pygame.font.Font' objects}
    65344    0.792    0.000    0.792    0.000 {method 'blit' of 'pygame.Surface' objects}
    48376    0.617    0.000    0.720    0.000 main3.py:235(update)
     1414    0.437    0.000    1.014    0.001 sysfont.py:282(font_constructor)
        1    0.210    0.210    0.210    0.210 {built-in method pygame.base.quit}
        1    0.165    0.165   35.404   35.404 main3.py:9(main)
     2828    0.154    0.000    0.398    0.000 main3.py:41(update)
        1    0.137    0.137    0.306    0.306 {built-in method pygame.display.set_mode}
     7070    0.104    0.000    0.852    0.000 sprite.py:465(draw)
     4242    0.097    0.000    1.222    0.000 sprite.py:453(update)
     1102    0.082    0.000    0.082    0.000 {built-in method pygame.transform.rotozoom}
     2828    0.077    0.000    0.100    0.000 main3.py:137(object_collision)
    16980    0.069    0.000    0.107    0.000 posixpath.py:100(split)
     1566    0.064    0.000    0.064    0.000 {built-in method io.open}
     2708    0.052    0.000    0.052    0.000 {built-in method posix.stat}
     1616    0.045    0.000    0.068    0.000 main3.py:222(__init__)
    17784    0.041    0.000    0.041    0.000 sprite.py:304(sprites)
     5656    0.036    0.000    0.036    0.000 {built-in method pygame.key.get_pressed}
    19810    0.034    0.000    0.141    0.000 __init__.py:1579(_parents)
    24268    0.029    0.000    0.043    0.000 sysfont.py:56(<genexpr>)
    50/48    0.027    0.001    0.030    0.001 {built-in method _imp.create_dynamic}
    51852    0.027    0.000    0.027    0.000 {method 'get_rect' of 'pygame.Surface' objects}
    48376    0.024    0.000    0.024    0.000 {method 'contains' of 'pygame.Rect' objects}
    25854    0.024    0.000    0.034    0.000 sprite.py:183(kill)
    16992    0.024    0.000    0.038    0.000 __init__.py:2371(_is_egg_path)
     2990    0.023    0.000    0.216    0.000 {built-in method builtins.next}
      175    0.023    0.000    0.023    0.000 {built-in method marshal.loads}
      316    0.022    0.000    0.022    0.000 {built-in method builtins.compile}
     3050    0.022    0.000    0.035    0.000 posixpath.py:150(dirname)
        8    0.022    0.003    0.022    0.003 {method 'poll' of 'select.poll' objects}
5050/5046    0.021    0.000    0.065    0.000 {method 'join' of 'str' objects}
     1102    0.021    0.000    0.103    0.000 main3.py:108(rotate)
     3108    0.019    0.000    0.047    0.000 ntpath.py:61(isabs)
     3286    0.019    0.000    0.025    0.000 posixpath.py:71(join)
     3108    0.018    0.000    0.079    0.000 __init__.py:1493(_validate_resource_path)
    26586    0.018    0.000    0.027    0.000 posixpath.py:41(_get_sep)
```

Figure 2: cProfiler

7 Conclusion

Our final implementation fulfills the requirements in a satisfying manner. The code can still be further optimized and debugged regarding some of the issues mentioned above in discussion. In hindsight we should possibly have structured some of the components differently to make it easier to maneuver and edit the code for further implementation.

8 References:

None