University of Tromsø

INF-1400-Object Oriented Programming

Mandatory assignment 3

Helge Hoff & Øystein Tveito

Department of Computer Science

April 11, 2014

Contents

1 Introduction

For this project a clone of the arcade game Mayhem will be implemented. There is two authors on this project, so the workload will be shared between.

1.1 Technical Background

1.1.1 Mayhem

A classic arcade game with two (or more) spaceships fight each other.

2 Design

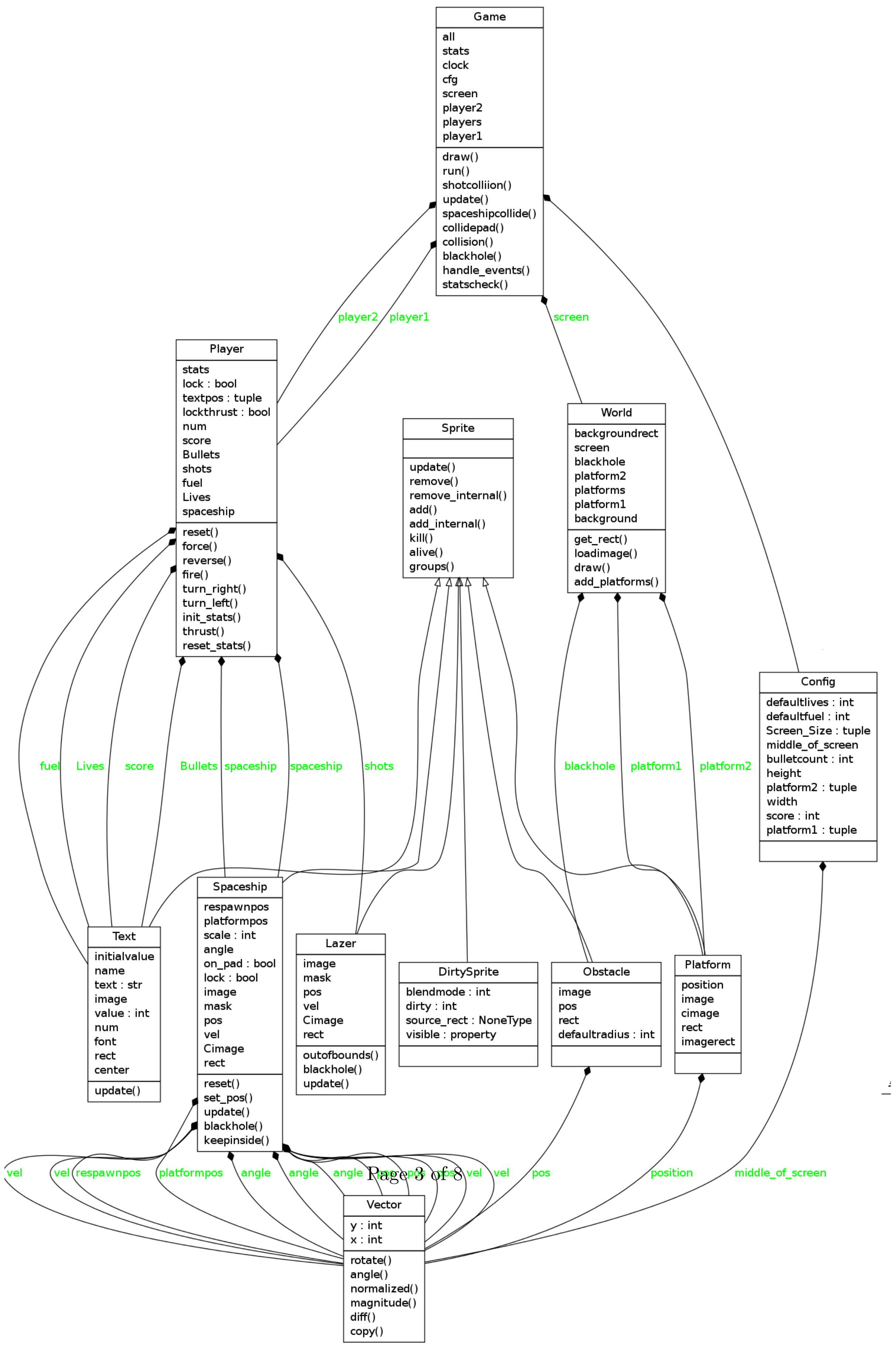
This game is a two player game sharing one screen and keyboard. Each player have a set of keys associated with him, making him able to navigate and shoot his spaceship. Each spaceship starts at a spawn point which is a platform not affected by gravity. On the platform, the player can refuel and restock bullets. The platform a player spawn on is private, and can not be used by other players.

In the middle of the screen there is a black hole. The black hole has gravity, pulling every player towards it. It also pulls laser bullets, because, as we know, not even light can escape the awesome power of a black hole. This is done instead of having gravity pointing downwards. The scene is set in space, so there is no up and down. When getting close to the black hole, an object will start to spin clockwise towards around the hole, while pulling it towards the center.

3 Implementation

The game is written in python, with pygame as it's most significant building block. All classes are split into separate files for a better overview of the game. All visible objects are children of the pygame class Sprite.

Figure 1: Class diagram



3.1 World

The World class contains the screen on which all of the games objects are to be drawn. It also contains the background image represented as a pygame surface type, the black hole, and the two platforms where the spaceships are spawned. The platforms is of type pygame sprite object as well as the black hole represented in the middle of the screen where the spaceships disappear into when drawing to close.

3.2 Text

The text is as the requirements specified of type pygame sprite, and each status parameter of the spaceships is one objects which is put in a pygame sprite group. the text object has an update method which renders the text for later to be drawn. The different status attribute of the spaceships can be altered by changing the different objects values, value is an attribute of the text class.

3.3 Collision

3.3.1 Spaceships

Collision between the two spaceships is handled by using the pygame sprite mask collision method which has two sprites as input. Then the two spaceship sprites are sent in as arguments to the mask collide function. Then if collision they each looses lives.

3.3.2 Bullets

The bullets for the specific spaceship is in its own group which facilitates the collision detection. Here the mask collide method is also used, and to check every bullet from one spaceship towards the other spaceship is done by getting a list of sprites that the sprite group contains. Then iterating through it and checking every bullet up against the spaceship. To get a list of the sprites in a group is simply done by setting a list equal to the specific group. Every time a bullets position is updated a method inside the Lazer class called ”outofbounds” checks if the bullet has wandered off the screen and if so it is removed from the specific bullet list. The same goes for when the bullet gets in the black hole.

3.3.3 On pad detection

The pads ”docking stations” are sprite object. Collision is detected with ”sprite collide mask”. To prevent one spaceship to fuel up on the other spaceships pad this is taken into account so spaceship one can only ”collide” with the pads its associated with.

3.4 Player

The Player class hold all the parameters which is individual to a player. It has a spaceship, status and bullets. Also all the methods related to steering the spaceship is implemented in the player class. These methods alter the state of the specific spaceship and its bullets.

3.5 Game

3.5.1 Rules

Each spaceship has a set of parameters which defines their state. These are, number of lives, score, fuel left, and bullets left. When one spaceship runs out of lives, it resets all it's stats and respawnes to the pad. However when a spaceship runs out of fuel, the thrust gets locked leading to no movement. Bullets and fuel gets refilled when a collision is detected to the right pad.

3.6 Spaceship

The spaceship is a container for the image, position and velocity of the spaceship shown on screen. It also has methods for updating the position of itself. The class always keep the original image of itself as an extra variable to keep it from being destroyed with the transforms added to the image.

3.7 Bullets

A bullet is a pygame sprite containing a pygame surface as image (ellipse). The bullet has the the velocity of the spaceships angel on release, also it gets rotatet according to the spaceship upon release. Also the release point of the bullet(drawn first time) is equal to the spaships centerposition when key to trigger shot is pressed

4 Discussion

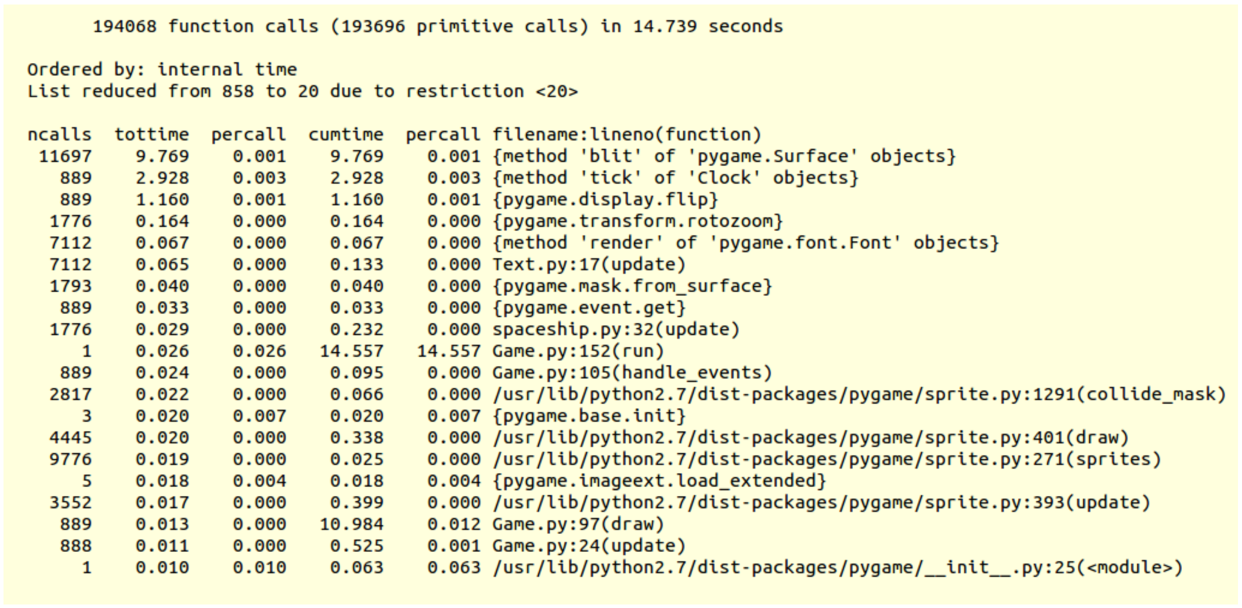
This section starts with a list of requirements and an explanation on whether or not this implementation has successfully implemented this:

- Two spaceships with four controls: rotate left, rotate right, thrust, fire.
  - All the controls are implemented with the addition of backwards thrust. The backwards thrust is added to make it easier to avoid the black hole, and in the same time be able to shoot against your opponent.
- Minimum one obstacle in the game world. This can be as simple as a single rectangle in the middle of the screen.
  - A black hole has been added as an obstacle in the middle of the world.
- Spaceship can crash with walls/obstacles/other spaceship.
  - The spaceships will be absorbed by the black hole. In the event of a crash with an other spaceship, the spaceship with the least amount of health will be destroyed.
- Gravity acts on spaceships (the original has no gravity acting on the bullets, but you can choose what works best).
  - There is no gravity pulling downwards because this is in space. The black hole stands for the gravity, pulling everything towards it and in a spiral.
- 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.

  - A score is implemented and shown on screen. The score increases when a shot from that ship hits the opponent. On destruction the score will be reset to zero.

- 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.
  - Each spaceship has its own fuelling pad that recharges its fuel and bullets. The pad is unique to the spaceship, so a player can only land on his own pad.
- Scrolling window, as seen on the video, is NOT a requirement.
  - Not implemented.
- The implementation must consist of a minimum of two files. One of these shall be a config.py file containing global configuration constants, such as screen size, amount of gravity, amount of starting fuel.
  - The config file is implemented and holds some variables that govern screen size and some initial values.
- The main loop must have timing so that the game is playable on different computers.
  - The game loop is controlled by the pygame clock. This is set to 60 ticks each second, so as long as the computer can handle 60 FPS (any fairly modern computer can) the game will feel the same. If the computer is to slow for this, it will be slower. This is done to not make the spaceship jump long distances for each frame.
- The game shall be started using Python's if __name__ == '__main__': idiom. Inside the if test, a single line shall be instantiate the game object. All other code, except the game configuration constants, shall be inside the classes. This will simplify profiling and documentation generation.
  - The first exception from this rule is that the main function is in a file of its own. To make the game run, pygame need to be initiated in every file expecting to have anything with pygame. For this reason, the first line in main is initialization of pygame. The second exception is that the game class do not start the game by it self. This is done on purpose, so the caller can initiate the object, and chose when to start it. Giving more control to the user is a informed choice, and the authors stand by it.
- All visible objects shall subclass the pygame.sprite.Sprite class. The sprites shall be put into groups using pygame.sprite.Group. Then updating and drawing shall be performed using Group.update and Group.draw.
  - Every visible object is a subclass of sprite, and every object is updated and drawn using the Sprite functionality.
- All modules (files), classes and methods shall contain docstrings. If you are working in a team, the module docstring at the top of the file shall contain the name of both authors. When you are done programming, html documentation shall be generated using pydoc -w command.
  - This requirement is met.

Figure 2: Profiler output



```
      194068 function calls (193696 primitive calls) in 14.739 seconds

   Ordered by: internal time
   List reduced from 858 to 20 due to restriction <20>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    11697    9.769    0.001    9.769    0.001 {method 'blit' of 'pygame.Surface' objects}
      889    2.928    0.003    2.928    0.003 {method 'tick' of 'Clock' objects}
      889    1.160    0.001    1.160    0.001 {pygame.display.flip}
     1776    0.164    0.000    0.164    0.000 {pygame.transform.rotozoom}
     7112    0.067    0.000    0.067    0.000 {method 'render' of 'pygame.font.Font' objects}
     7112    0.065    0.000    0.133    0.000 Text.py:17(update)
     1793    0.040    0.000    0.040    0.000 {pygame.mask.from_surface}
      889    0.033    0.000    0.033    0.000 {pygame.event.get}
     1776    0.029    0.000    0.232    0.000 spaceship.py:32(update)
        1    0.026    0.026   14.557   14.557 Game.py:152(run)
      889    0.024    0.000    0.095    0.000 Game.py:105(handle_events)
     2817    0.022    0.000    0.066    0.000 /usr/lib/python2.7/dist-packages/pygame/sprite.py:1291(collide_mask)
        3    0.020    0.007    0.020    0.007 {pygame.base.init}
     4445    0.020    0.000    0.338    0.000 /usr/lib/python2.7/dist-packages/pygame/sprite.py:401(draw)
     9776    0.019    0.000    0.025    0.000 /usr/lib/python2.7/dist-packages/pygame/sprite.py:271(sprites)
        5    0.018    0.004    0.018    0.004 {pygame.imageext.load_extended}
     3552    0.017    0.000    0.399    0.000 /usr/lib/python2.7/dist-packages/pygame/sprite.py:393(update)
      889    0.013    0.000   10.984    0.012 Game.py:97(draw)
      888    0.011    0.000    0.525    0.001 Game.py:24(update)
        1    0.010    0.010    0.063    0.063 /usr/lib/python2.7/dist-packages/pygame/__init__.py:25(<module>)
```

- The last task is to profile the code using cProfiler. Take a screen shot of the result and include it in the report. Give a short summary of the result and discuss where you would focus to improve the performance of the implementation.

  –

When looking at the table above, showing the total time used in each function (only top 20), it shows that the handling of the graphics is by far the biggest time consumer in the game. Bliting each component to the screen for each tick takes time. One way to improve this time, is for instance to blit the solid objects to the background image once, then only use the newly compiled image as the background for the game. This would be a fast and easy improvement, but there are only three static objects, namely the two pads and the black hole. This would help, but not noticeably.

On second place we have tick(). The fact that this is shown as number two is in itself a victory. This is basically the time we have left after all the operations of the game has been completed for the current frame. The more time the tick method uses, the less time the mechanics of the game are using.

On third place the pygame method flip() is represented. This is the method used to switch between the frame shown on the screen, and the active frame being worked on. This method is only used once per frame, but it is quite heavy because it has to draw pixel for pixel into the screen buffer. There is not to much to be done about this, though it would have been interesting to create an algorithm that used the update function of the screen to only update the parts of the display that actually changed in the new frame. This would potentially increase the performance of the game, because less of the screen would be redrawn at each frame.

For the next in line, we have to go down in time consumption with a factor of 10 from third. This makes everything in fourth place and down quite irrelevant for increasing the performance, because the amount of work needed to improve just a fraction of a millisecond at each frame will not be justified for this type of project. In projects like this, one should always try to improve the parts that uses the most amount of time, and worry about the smaller parts when the earnings from selling the game can justify the time used to fix them.

5 Conclusion

All in all, the two authors are quite pleased with our project. It dos conform with all the requirements, and generally look good and is fun to play. There is always room for improvement, and some performance issues mentioned in the discussion part would be the place to start. Also, the game is an old school ”no winner” game, where it will never stop, and you and a friend can play for ever trying to achieve the highest score feasible.