1 Introduction

In this assignment, we create a simple simulator, specifically a clone of the classic flock simulator boids.

2 Background

For this assignment, Python 3.4[2] was used. We have also used Pygame 3.4, which allows us to use the Pygame library for easier making games and simulations.

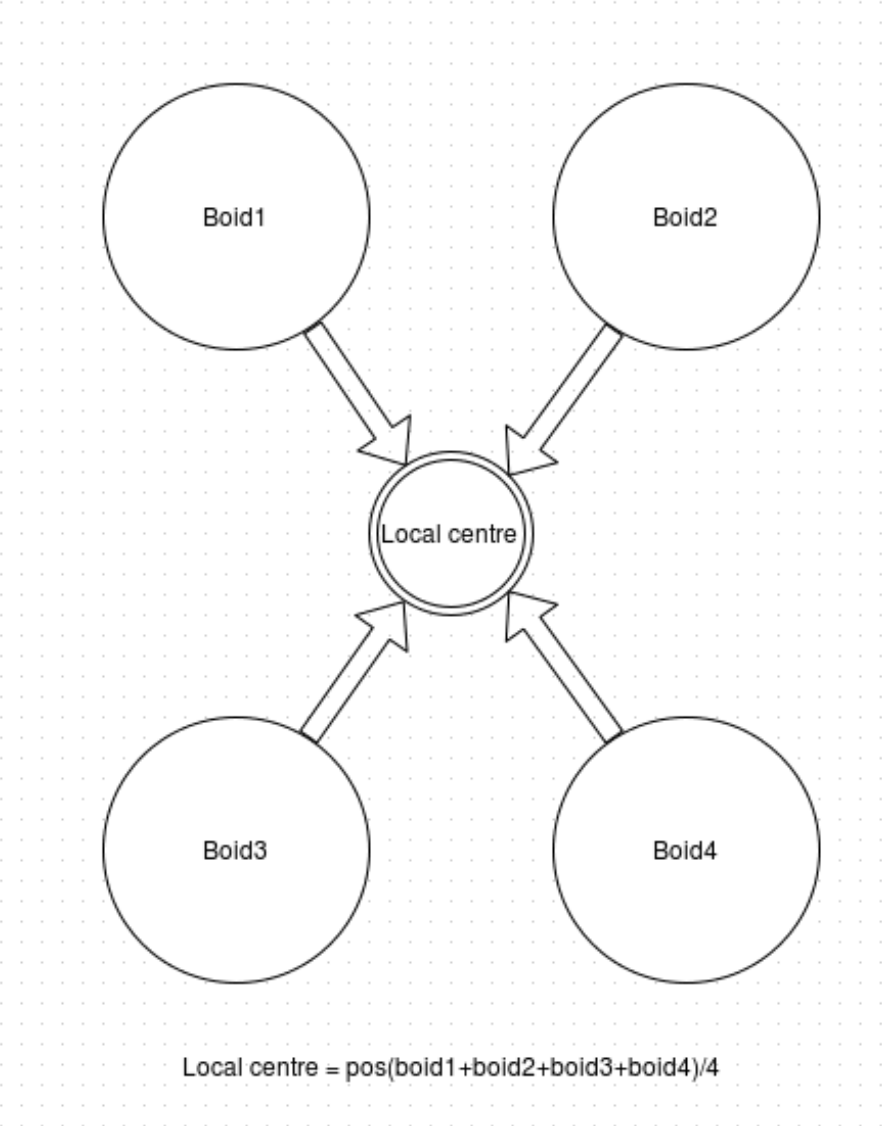
3 Design

In this task, we were required to make use of inheritance. I've used inheritance in all of the classes, except **Canvas**. All the other classes inherit from the pygame library “pygame.sprite.Sprite”, As we can see in Figure1. The inheritance allows them to make use of the entire library.

In Figure 1 we see that the module consists of four classes, **Canvas**, **Boids**, **Hoiks** and **Obstacles**. **Canvas** is the class that is responsible for running and updating all the other classes as a whole. The most important method in **Canvas**, is the '*update*' method, which tells us what will be done for each frame passed. **Canvas** also has a method called '*handle_events*', which makes it possible for us to quit the simulation, and press '1' to add more boids to the screen. The last method in **Canvas**, '*run*', is an infinite loop that continuously runs the other methods.

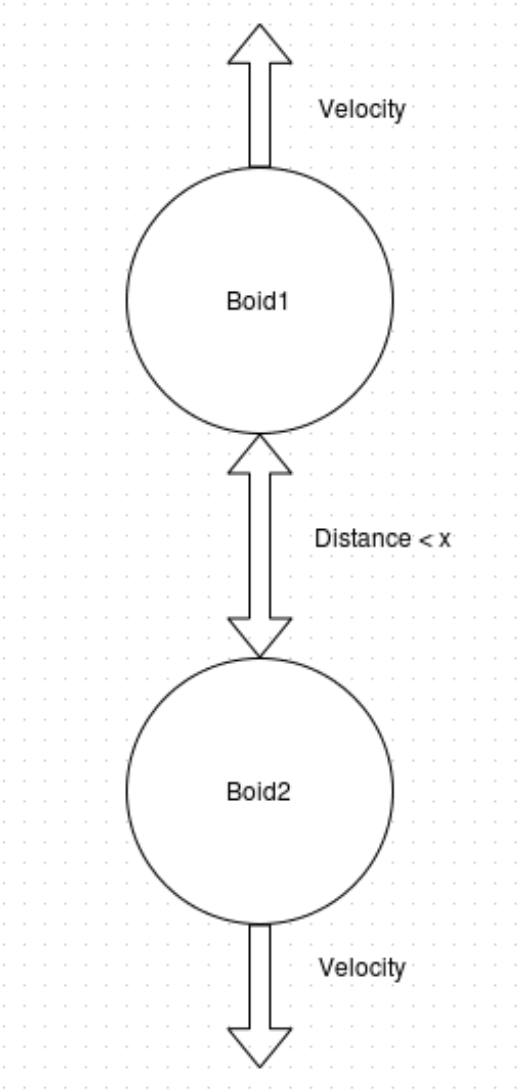
Moving on to class **Boids**, where the most complex methods of the code are to be found. Its '*update*' method controls the vector movement of the boids by adding the return vectors from all the rules (there are five vectors in total, of which only the three fundamental ones are being used).

The method '*first_rule*' creates a local centre for each boid, based on neighboring boid positions. The return value from this rule gives each boid a local centre to move towards.



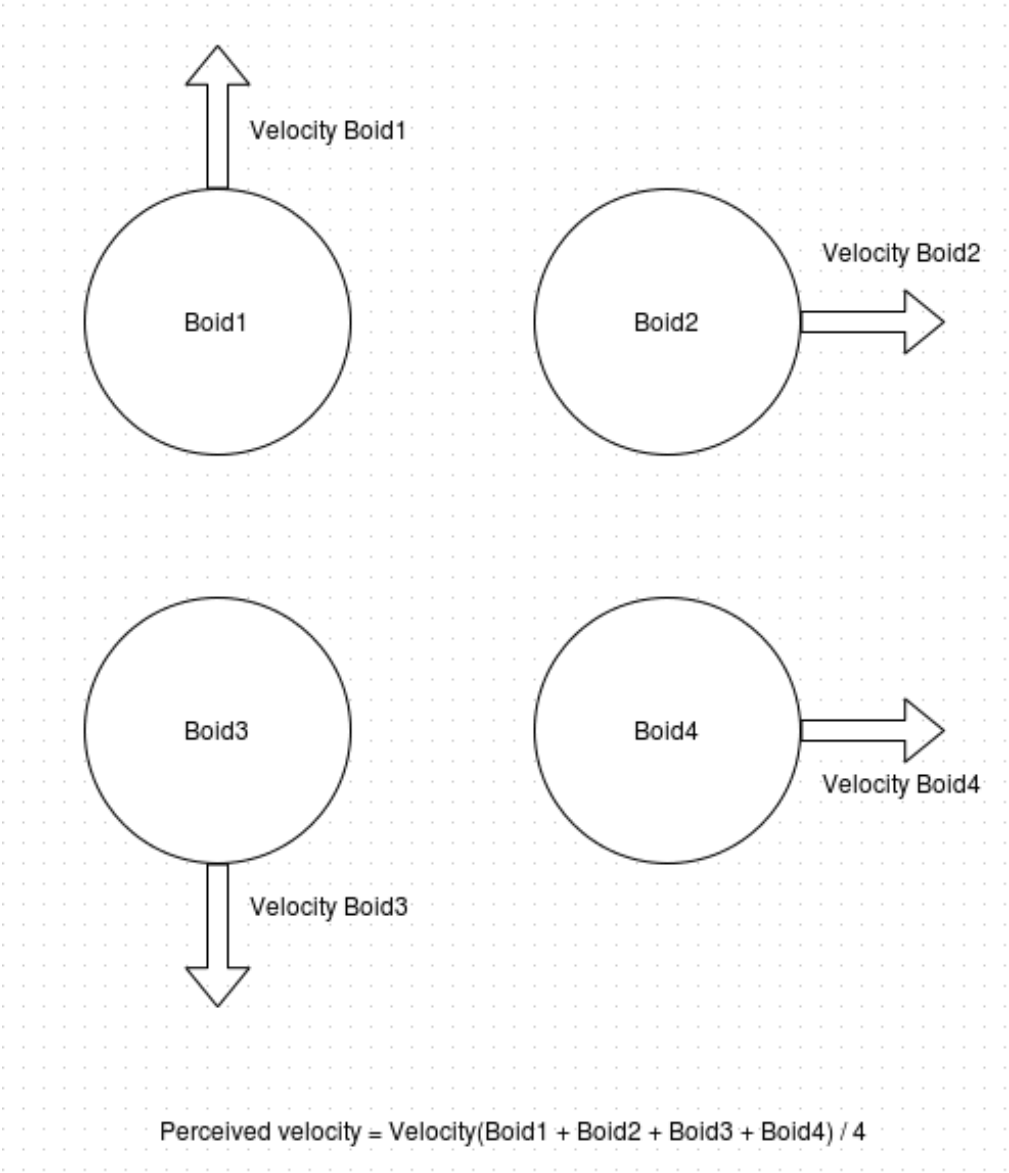
Above is an illustration of '*first_rule'*

The method '*second_rule*' has as a goal to make the boids avoid each other, by iterating through the list of boids, and telling each one of them: 'if you get x pixels close to another boid, change direction'.



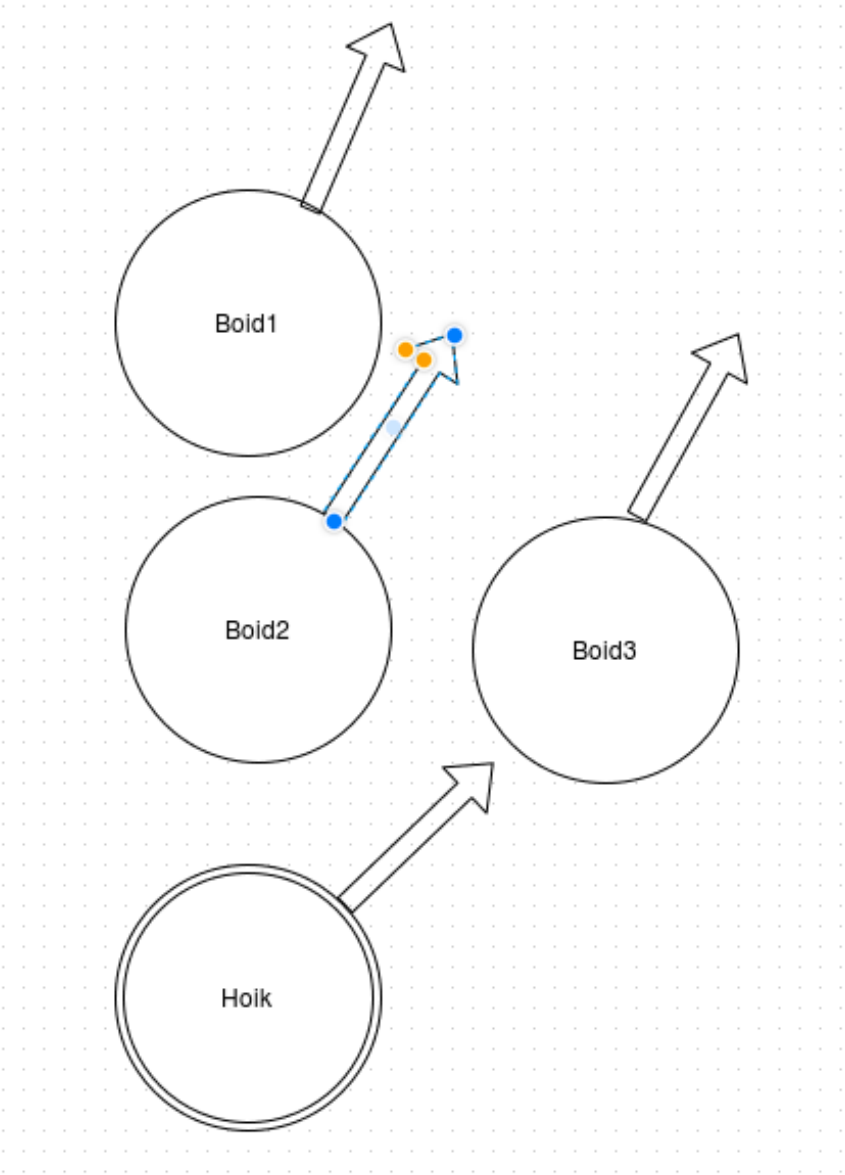
Above is a visualization of '*second_rule'*

The method '*third_rule*' is quite similar to '*first_rule*', except instead of creating a perceived local centre, it creates a perceived velocity.



Shown above is a representation of '*third_rule'*

The method '*fourth_rule*' is supposed to be almost the same as '*second_rule*', the difference being that in this method, the boids don't try to avoid each other, they try to avoid the hoiks.



This illustration is supposed to show how '*fourth_rule'* works.

The method '*avoid_obstacles*' has as purpose for boids to avoid on-screen obstacles. This method is very similar to both '*fourth_rule*' and '*second_rule*'

Moving further on to the class **Hoiks**. It consists of only two methods; '*update*' and '*limit_velocity*' which both respectively do the same for each hoik as the methods with the same name in **Boids** do for each boid.

The last Class represented is **Obstacles**. It does nothing else than just initializing its start-values, since they never change, it doesn't need an '*update*' method.

4 Implementation

For this task, I have chosen to use the built-in sprite library which can be found within the pygame library. This allows the objects to easily communicate with each other. In addition, certain operations such as manipulating lists in any chosen way (add elements, delete elements, collision test). The sprite library is extremely useful in a simulation such as this one.

In figure 1, we see how I've chosen to implement sprites in my simluation, and how each class inherits from the sprite library. Each class that inherits sprites, gains access to the sprite library.

In general, the way inheritance in OOP works, can be compared to how a child inherits attributes from its parents. It will some attributes from each parent, but with the ability to modify them itself. So, if I for example inherited my fathers understanding for mathematics, I will have a pretty good starting point, but I will also have the ability to work with my mathematics understanding, and develop it as time goes by. This example is not a completely accurate representation of how it works in OOP, but it's a good way to gain an understanding of it.

If we take a look at Figure 1, we'll see, as earlier mentioned, that **Boids** and **Hoiks** initialize pretty much identic attributes. I could have saved quite a lot of lines with code by for example making a class called **Bird**, and initialize it with the same attributes we see in **Boids** and **Hoiks**. After doing that, I could have inherited the **Bird** class into the two other classes, and just modified the values after my preference.

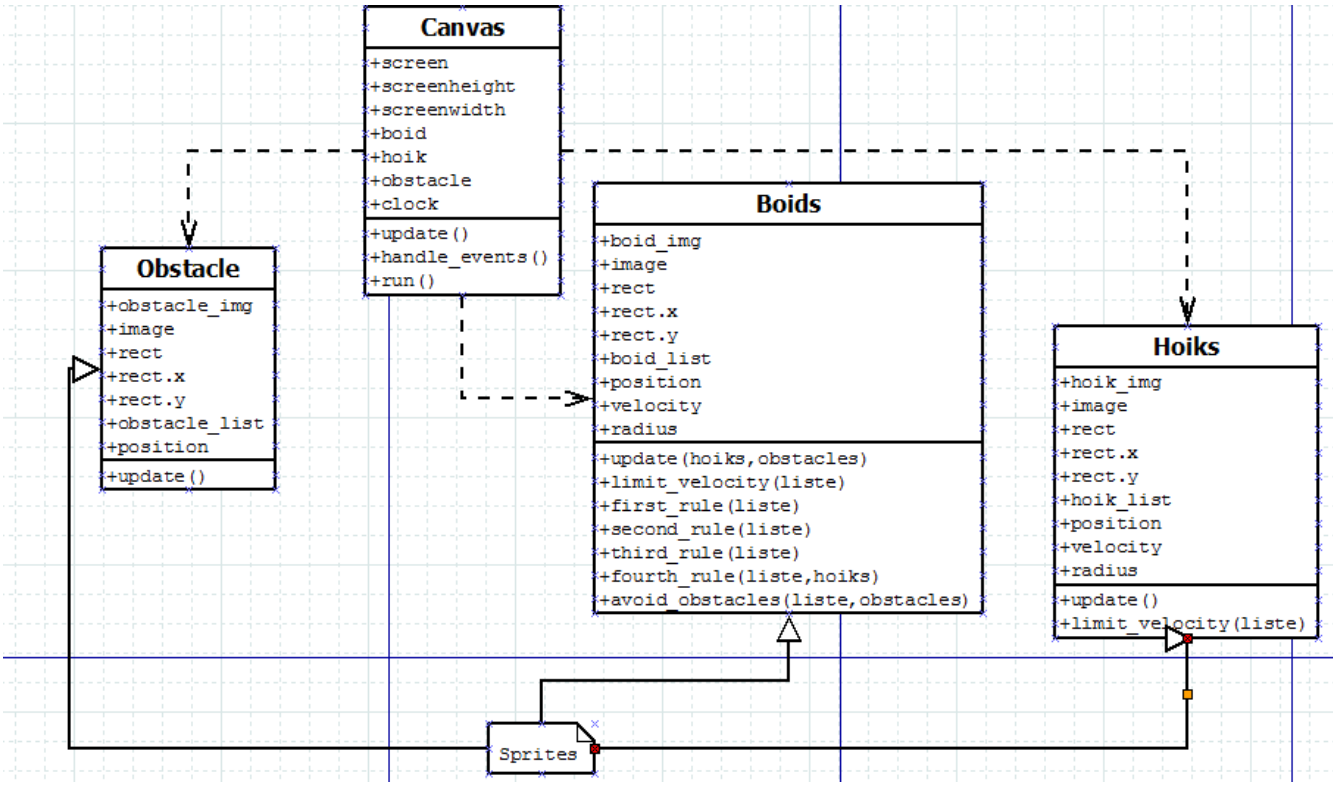


Figure 1

5 Evaluation

In my solution, not all of the requirements are fulfilled. The simulator is implemented in accordance with object-oriented design. Inheritance is used in three classes, which fulfills requirement number 2. The simulator follows the rules described above.
Both **Hoiks** and **Obstacles** are implemented, although they don't serve any particular purpose as of now. Requirement number 5 is fulfilled after my best judgement.

6 Discussion

Looking back at my code as I'm writing this report, I see a lot of things I could have done differently. I should definitely have **Boids** and **Hoiks** inherit from a **Bird** class, since they got the exact same attributes. The way I have done it now, I've basically unnecessarily duplicated many lines of code.

The first three rules applied to **Boids** work pretty well, although I'm struggling to adjust the level of significance of the respective rules in a way that makes them move smoothly.

As mentioned earlier, I have written a '*fourth_rule'* to make boids avoid hoiks. This does not work though, and I'm not sure why, but when i apply that rule, it changes the way the boids behave, but they still don't avoid the hoiks.

The boids don't avoid the obstacles either, although I have written the code for it, which I thought should work, but alas, it doesn't.

Probably the most important mistake I made, was planning. As in I didn't plan at all. I just read through the assignment and started writing based on intuition. Being the rookie I am, this turned out not to be a very good idea. I don't think it's a good idea no matter how experienced you are.

That's the most important thing I've learned after struggling with this assignment as well; from now on I will always use a proper amount of time planning my code before I even start writing a single line.

7 Conclusion

All in all, I'm happy with my performance and the effort I've put into this assignment, even though I realize not all of the requirements are fulfilled. I've gained a massive amount of knowledge during the past couple of weeks, and my understanding of Python keeps expanding. I think this assignment has been a great challenge, and I love being challenged.

References

[1] http://www.kfish.org/boids/pseudocode.html
[2] http://www.en.wikipedia.org/wiki/Boids