Assignment 2 report - INF-1400

Amund H. Strøm

Mars 10 2022

1 Introduction

In this assignment we were tasked to implement a boids simulation, which is a flocking simulation that follows mainly three different rules to create a life-like appearance. To implement this the focus was to use the principles of Object-Oriented Programming (OOP), this means the implementation must use classes, methods and inheritance.

2 Technical Background

Object-Oriented Programing (OOP) is a technique for structuring the code of a program, the idea is to bundle together related information into classes. To create a class simply give it a name, and inside the class you can add all the different variables you need for this specific class. It is also possible to create methods or behavior, which describes what kind of operations a class can perform.

Inheritance is a concept that comes from OOP, it is used to share different variables and methods to other classes. The idea is that if you know that you are going to create at least two different classes that are almost the same. You can create a parent class which contains the information they have in common and share it to them. The only reason to do this is to not repeat code and make it easier to read for other people.

There are three different **Flocking rules** that need to be implemented to make a boids simulation, that is alignment, cohesion and separation. Alignment makes it so the flock wants to travel in the average direction of each single bird. Cohesion makes it so each single bird wants to be in the center of the flock. Separation makes it so the birds won't collide into each other. With these simple rules we get a life-like appearance

3 Design

In my program there are two parent classes, and four child classes.

Parents

- Moving object is used if an object is supposed to move on the screen. It will rotate the image of an object, separate objects form the same group so they don't collide and avoid collision with obstacles.
- Drawable object is used if an object is supposed to stand still on the screen and be an obstacle for the moving objects

Children

- Boid is the "birds" of the simulation, it will travel the same direction of the flock and travel to the center of the flock
- Hoik is the "predator" of the simulation, it will chase the nearest boid and eat it
- Rect is a rectangular obstacle
- Circle is a circular obstacle

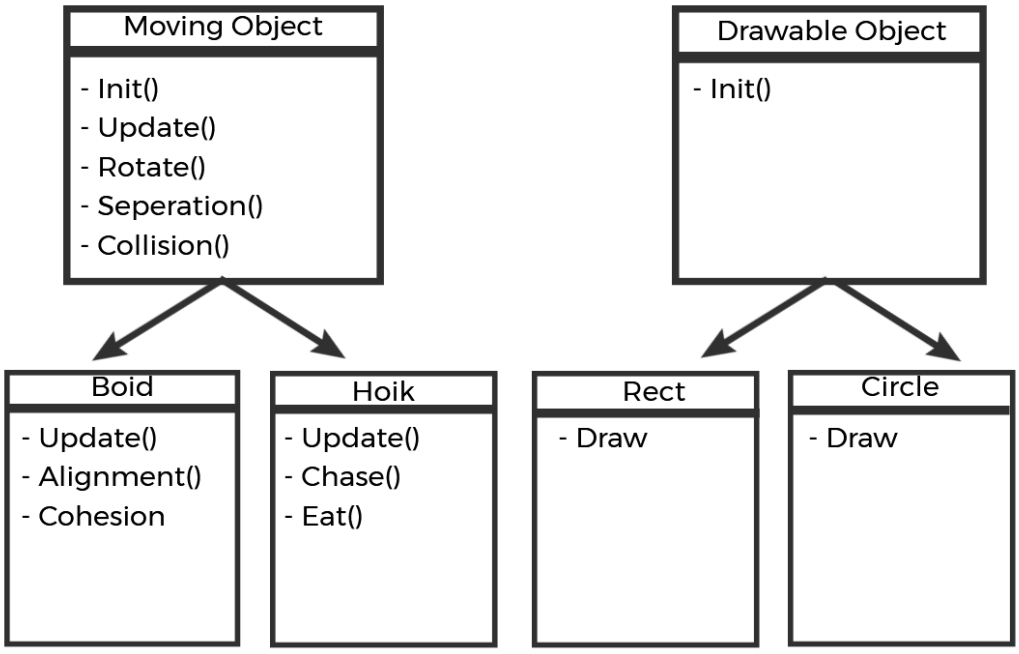

Figure 1: Illustration of class implementation

4 Implementation

There are two parts of the implementation that needs explaining, that is the inheritance of classes, and the different rules for the flock behavior.

4.1 Inheritance and classes

As mentioned, there are 2 parent classes and 4 child classes. It made sense to do so because the assignment stated that both boids (bird) and hoiks (predator) needed to be implemented, and since they both would be moving in similar ways, we bundle together the related information into one parent class. Inside this class there are all the different variables and methods for both child classes. These methods are:

- Update, which moves the object by the given direction and speed, it also moves the object on the other side of the screen if it moves out of bounds
- Rotate, which rotates the given image of the object corresponding to the direction it is moving
- Separation, makes it so the objects of the same group (either boid or hoik) will avoid colliding into each other
- Collision, makes it so the object will avoid colliding with obstacles

The child classes use all the above tools, but what sets them apart is 2 new methods and a personalized version of the update method:

Boid

- Update, calculates the direction and speed for the object using the different boid rules
- Alignment, calculates the average direction of the flock
- Cohesion, calculates the center of the flock

Hoik

- Update, calculates the direction and speed for the object using the different hoik rules
- Chase, calculates the direction of the nearest boid
- Eat, if a boid intersects with the hitbox of the hoik, it will “eat” it and remove it from the screen

The other parent class is used for drawing different obstacles on the screen, and the child classes decides the image of the object. Other than that, there is not much to discuss.

4.2 Rules

In my program there are five different rules that calculates the direction of the object, but only three of them are necessary to create a life-like flocking behavior, that is alignment, cohesion and separation. Since the speed and direction of an object is represented as a 2-dimensional vector, the math involved is based on vectors. The speed and direction are a single vector called velocity, and the position of the object on the screen is also a vector. So, every moving object has 2 vectors that tells the position, and the direction and speed it moves in. This is how the different rules work:

- Alignment calculates the average direction of local flock mates. To do so we sum together the velocity vector of every local mate, then divide that by the number of local mates. We then normalize the vector, so it has a length of 1.
- Cohesion calculates the center of local flock mates. To do so we sum together the position vector of every local mate, then divide that by the number of local mates. We now have the average position, but we need to calculate the direction an object will need to move to reach this position. To do so take the average position minus the current objects position. Then we normalize the vector, so it has a length of 1.
- Separation makes it so objects will avoid colliding into each other. To so we sum together the result of the position of local mates minus the position of the current object, then divide that by the number of local mates. We then normalize the vector so it has a length of 1, and multiply it by minus 1, so it will have the opposite direction.
- Collison makes it so object will avoid colliding with obstacles, this works the exact same way as the separation rule.
- Chase calculates the direction of the nearest prey. To do so we take the difference between the position of local prey and the current object and save the position of the prey that got the smallest difference. We now have the position of the closest prey, so we need to calculate the direction to this position, using the same logic as in cohesion. We take the nearest position minus the current objects position and normalize it, so it has a length of 1.

Since every vector we got from the different rules has a length of 1, we can easily modify the significance of each rule by multiplying it with some number. If it is multiplied with zero, the rule will have no impact, and every number higher than zero will increase the impact. To get the object to change velocity based on these rules, we simply sum together the results of the rules, then add that to the previous velocity of the current object. Then scale the vector to its previous length so it won't spiral out of control.

5 Evaluation

In this solution, all the requirements are fulfilled, but there are two small problems. I have noticed that sometimes boids that move in completely different directions and meet head-on, will collide into each other. I have tried tweaking the significance of the separation rule, but found out that if it is lowered, they will collide more, and if it is increased it looks unnatural. The second problem is that the hitbox of the hoik seems a bit off when it eats the boid, it looks like there is a small difference between the visual representation and the implementation

6 Discussion

In my program each single moving object have a radius to detect local flock mates, this is not a very effective way of doing it and requires a lot of processing power if there are a lot of moving objects. A more elegant approach is to divide the screen into many small squares, like a grid, and use the squares to check where local mates happen to be. But I did not figure out how to do this. Other than that, I don't know if there are more optimalisations.

A small part in the moving object class is taken from stackoverflow, but it seems that the post is taken down. The part is the rotate method and is only 7 lines of code.

7 Conclusion

My program fulfills all the requirements, with some small problems with collision and hitboxes between boids and hoiks, that I don't know how to fix. One other optimalisation is to use a grid instead of a radius to detect local flock mates.

8 Referanser

Pemmaraju, V. (2013, Jan 21). *3 Simple Rules of Flocking Behaviors: Alignment, Cohesion, and Separation.* Obtained from tutsplus.com: https://gamedevelopment.tutsplus.com/tutorials/3-simple-rules-of-flocking-behaviors-alignment-cohesion-and-separation--gamedev-3444

Pygame. (2022, Mars 10). *Pygame.* Obtained from Pygame: https://www.pygame.org/