Assignment 2

Christer H. Opdahl

20. March 2018

Introduction:

In this assignment I am going to create a version of the artificial life simulator, first made of Craig Reynolds in 1986. The program will simulate the flocking behaviour of birds. The name "Boids" will be referenced often and refers to a bird-like-object.

Requirements:

To make this program simulate life like flocking behaviour 3 rules needs to be implemented.

1. Boids steer towards the average position of local flock mates; alignment
2. Boids attempt to avoid crashing into other boids; separation
3. Boids steer towards the average heading of local flock mates: cohesion

To make this simulation even more lifelike, predators need to be implemented as well. The predators called "Hoiks" follow some simple rules:

1. Hoiks seek toward the closest boid and tries to catch it.
2. If the Hoik touches the boid, the boid gets eaten and will disappear from the screen.

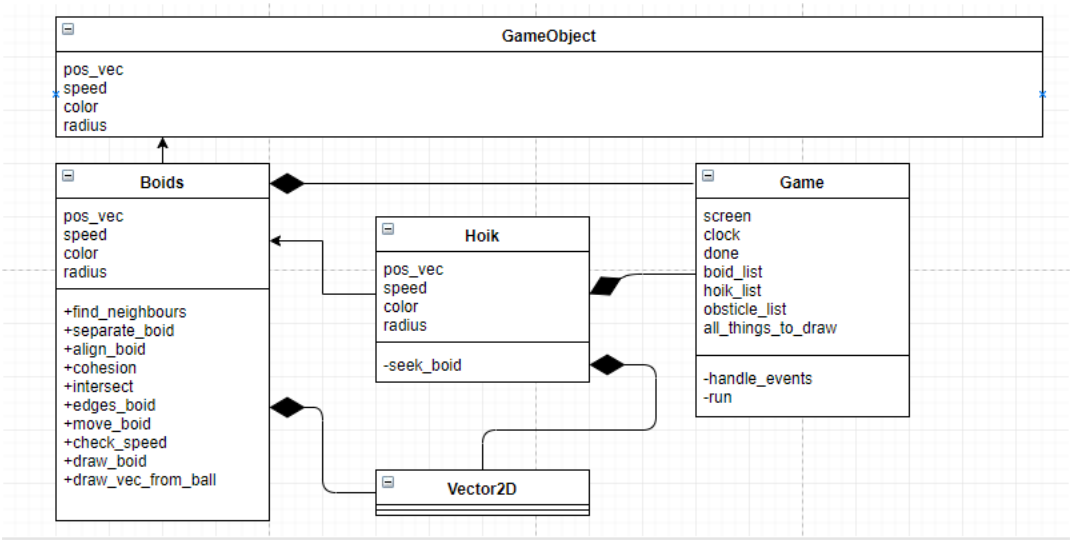
Boids will also avoid obstacles, and hoiks.

A self-made exception is to be implemented.

Technical background:

- Inheritance: allows us to create a general class first, and then give a more specialized class the same attributes as the general class. This makes the code more readable and the developer does not have to write the same code over and over again.
- Polymorphism: by using inheritance, we can use a method written in the parent class on the child class. For example, in this assignment I have a hoik class that inherits the boid class. This makes it possible to draw both obstacles/classes when I call on the draw function written in the boid class. This is called polymorphism.
- Exceptions: where we might expect to get errors, we can use exceptions to handle these errors without causing a crash in the program.

Design:



Classes:
All the code is written in classes, the game loop is a method in the game class.

The GameObject class is a superclass, and the boids and hoiks class both inherits the attributes in it.

The hoiks class inherits from the boids class which makes it possible for me to use polymorphism when moving, checking speed, checking edges, and drawing the boids and hoiks. Hoiks have one method which is specific to the class, the rest of the methods called upon is inherited from the boids class.

Obstacles:
If you click the mouse button an obstacle will appear on the mouse cursors location. The obstacles are just instances of the boid class but has no speed. The boids and hoiks will run away from the obstacles.

Vector2D:
I used the Vector2D class from the pre-code, all the objects x and y speed and position area vectors. This made me have less variables in GameObject class and it was easier to write.

Implementation:

Alignment:
To get the boids to seek towards other boids heading, all of the boids that is close to each other is added to a separate list using the find_neighbour method in the boid-class. The list of boids that is close to each other is iterated over and every speed vector is added together and divided by the length of the list.

Now we have the average speed of the list, all that is needed is to normalise the x vector and y vector and add it to the existing speed in x and y axes.

To normalise the speed vector x and y I used this formula:

$$\hat{\mathbf{V}} = \frac{\mathbf{v}}{|\mathbf{V}|}$$

$$|\mathbf{V}| = \sqrt{(x*x)+(y*y}$$

The normalized vector is then multiplied by a global variable MAX_SPEED, and then divided by a magnitude. The magnitude makes it possible to weigh the different rules up against each other. For example, the align method can be more important than the separation method and cohesion and vice versa.

Cohesion:
Cohesion makes the boid seek towards the middle of the group. This is done by gathering all the close boids into one separate list using the find_neighbour method. The separate list is iterated over, and we then find the average of the boids position.

However, if we just find the average of all the boids position, the boids would seek the top right corner and just turn around in circles. This problem was solved with this equation:

Desired (x, y) = target (x, y) – location (x, y)

When desired (x, y) was averaged and normalized, the boids had a much more natural flocking pattern and did not get stuck in the top left corner. By using this formula, we get the average directional vector of the group. The way the method gets called, every boid in the list will seek towards the average directional vector instead of the average position of the group.

Separate:
To separate the boids I first find the boids which are close to each other and append them into a new list of boids using the find_neighbour method. The new list gets iterated over, and much like the cohesion function the desired (x, y) formula is used to calculate the directional vector between a “self” boid and a close boid.

What we really have now is a vector needed to get from one boid to the other, but the opposite is needed. When normalizing the vector, it gets multiplied by -1 turning the vector around the other way. When we apply the force the boids stay clear of each other.

Hoiks:
As mentioned before the hoiks class inherits the boid class. The hoik class, has one method which is special for the class; seek_boid. The seek boid method finds the nearest boid and starts following it. Some of the code is borrowed from stack overflow, reference below [1].

To prevent the hoiks from colliding with each other, separate is used.
Collision detection with boids is handled in the game loop, after the 3 rules are applied.
The intersect method, makes the boids want to run away from boids.

Obstacles:
The obstacles are boids without any speed. To make the boids fear the obstacles the intersect method is used. The method calculates where the boid is relative to the obstacle and increments the speed accordingly.

Exception:
One of the requirements of this assignment was to create and use a self-made exception. In this assignment I really only needed the zero-division exception, but a self-made exception called lonely_boid is implemented and used when finding close objects in find_neighbours.

Discussion:

Bugs:
When big flocks of boids move together, the 3 flocking rules gets more priority over the intersect method which makes the boid flee from the hoiks. This makes the fleeing method(intersect) sometimes not work. Weighing the intersect method with distance would fix this.

Sometimes the intersect method gets more priority then the method that keeps it within the screen, I added a safeguard for it so the boid appears on the other side of the screen if this happens. There are better ways to solve this problem, but It made it interesting to see whether the boid decided to run of the screen to safety or if it would obey the rules and most likely get eaten.

Things I could have done better:
The weighing of the rules could have been less hard coded, sliders for the user to set the weights himself would be a lot better solution then hardcoding it in. Could also use the distance between objects as weights.

The self-made lonely boid exception is not really needed seeing that there are no errors when there are no objects left, but it was a requirement and I have done it.

The big O notation of the object handling (boids, hoiks and obstacles) in this case would be O(n), worst case O(n^2) when dealing with big flocks. I found that 75 boids was the most amount of boids my computer could handle without any problems. This might vary from computer to computer.

Conclusion:

The flocking rules, hoik rules, objects and exception is implemented.

Lessons learned:

- Writing a game/simulator in only classes and methods is a much more efficient way of coding.
- Weighing of different methods in simulations, is really important.

References:

[1] https://stackoverflow.com/questions/35167334/boids-predator-and-obstacle-behavior

Links used:
https://en.wikipedia.org/wiki/Boids
http://www.kfish.org/boids/pseudocode.html