Nature Inspired Computing

K.Dimopoulos

Complex Systems

What is a complex system?

A complex system is typically defined as a system that is “more than the sum of its parts.” While the individual elements of the system may be incredibly simple and easily understood, the behavior of the system as a whole can be highly complex, intelligent, and difficult to predict.

3 Key principles of complex systems

1. **Simple units with short-range relationships.**
   - This is what we've been building all along: vehicles that have a limited perception of their environment.
2. **Simple units operate in parallel.**
   - This is what we need to simulate in code. For every cycle through P5 `draw()` loop, each unit will decide how to move
3. **System as a whole exhibits emergent phenomena.**
   - Out of the interactions between these simple units emerges complex behavior, patterns, and intelligence. Here we're talking about the result we are hoping for in our sketches. Yes, we know this happens in nature (ant colonies, termites, migration patterns, earthquakes, snowflakes, etc.), but can we achieve the same result in our Processing sketches?

3 more key principles of complex systems

1. **Non-linearity.**
   - This aspect of complex systems is often casually referred to as “the butterfly effect,” coined by mathematician and meteorologist Edward Norton Lorenz, a pioneer in the study of chaos theory. A small change in initial conditions can have a massive effect on the outcome.
2. **Competition and cooperation.**
   - One of the things that often makes a complex system tick is the presence of both competition and cooperation between the elements. Competition and cooperation are found in living complex systems, but not in non-living complex systems like the weather.
3. **Feedback.**
   - Complex systems often include a feedback loop where the the output of the system is fed back into the system to influence its behavior in a positive or negative direction.

Group Behaviours

Anti - crowd behaviour

Let’s see how a group behaviour can crop out of an individual behaviour. Assume we have a group of vehicles, randomly distributed in space:

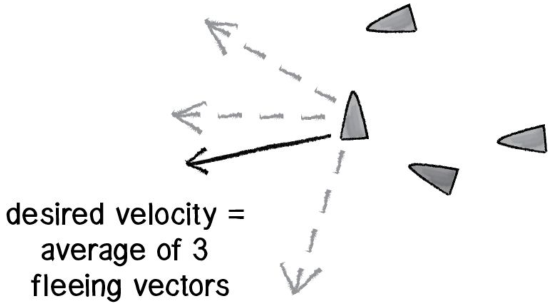
- Use the Vehicles we had in the previous example.
- In the Vehicles, use PBC.
- Create an ArrayList of vehicles
- Populate it with 100 vehicles, each at a random location in space.
- In the `draw()` step and display each vehicle.
- Do not use the `seek()`

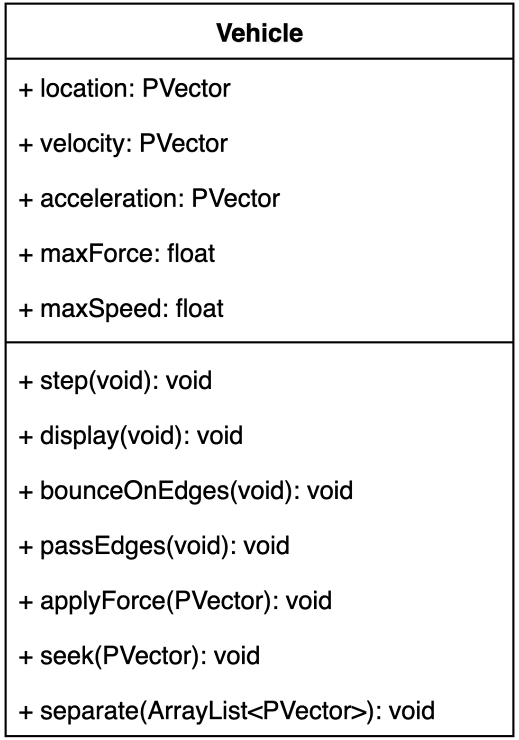
Group behavior: Separation

Let’s begin with separation, a behavior of the vehicle that commands, “Avoid colliding with your neighbors!”

The method will require to know all the vehicles of the simulation. It should:

- Find all vehicles within a specified distance (limited perception) and not zero distance.
- For each found vehicle calculated a steering force away from it
- Calculate the average steering force from all close vehicles
- Apply that steering force
- Do not use the `seek()` method





Vehicle
+ location: PVector
+ velocity: PVector
+ acceleration: PVector
+ maxForce: float
+ maxSpeed: float
+ step(void): void
+ display(void): void
+ bounceOnEdges(void): void
+ passEdges(void): void
+ applyForce(PVector): void
+ seek(PVector): void
+ separate(ArrayList<PVector>): void

Cohesion?

We can rewrite separate() to work in the opposite fashion (“cohesion”). If a vehicle is beyond a certain distance, steer towards that vehicle. This will keep the group together. (Note that in a moment, we’re going to look at what happens when we have both cohesion and separation in the same simulation.)

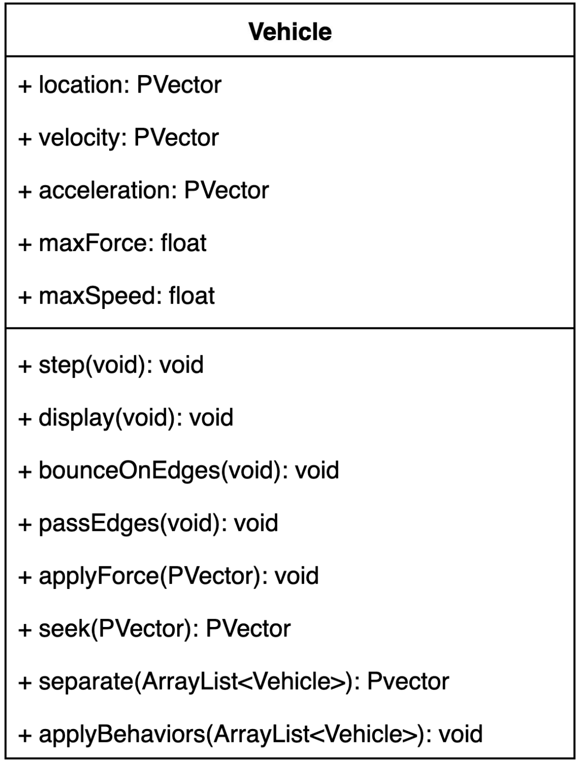
Combining Behaviours

Combining steering behaviors: Seek and separate

Let’s consider the case where all vehicles have two desires:

- Seek the mouse location.
- Separate from any vehicles that are too close.

We could combine the two behaviours in a single method `applyBehaviors()` but to do so it is better to change them into returning the steering functions so that we can apply them in the `applyBehaviors()` method. This way we can scale them and define how much each behaviour carries weight



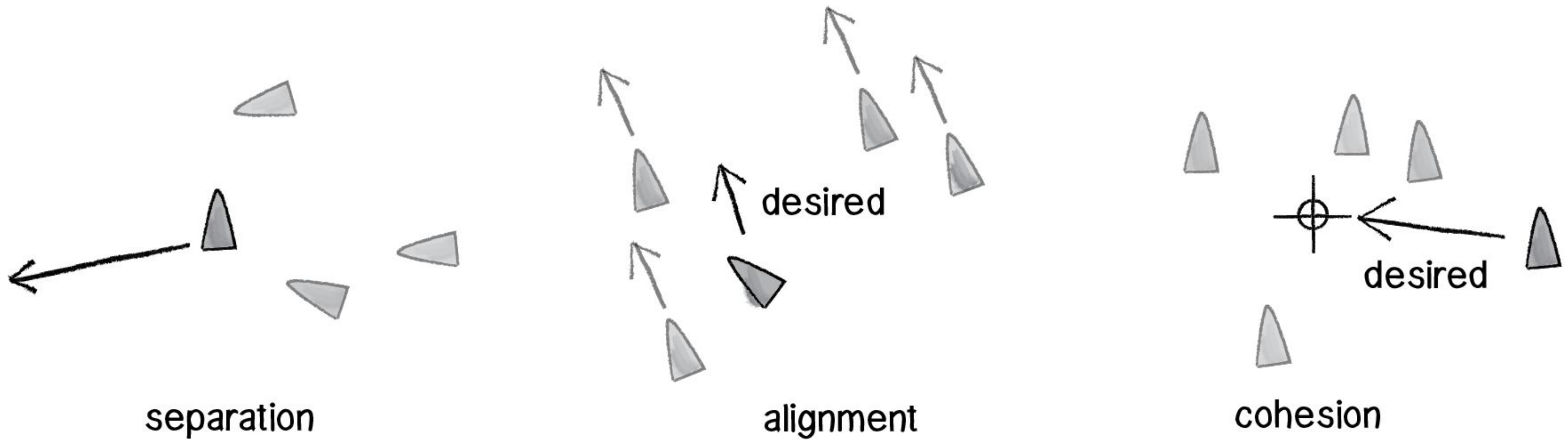
Flocking

Achieving flocking behaviour

1. We will use the steering force formula (steer = desired - velocity) to implement the rules of flocking.
2. These steering forces will be group behaviors and require each vehicle to look at all the other vehicles.
3. We will combine and weight multiple forces.
4. The result will be a complex system—intelligent group behavior will emerge from the simple rules of flocking without the presence of a centralized system or leader.

The rules of flocking

1. **Separation** (also known as “avoidance”): Steer to avoid colliding with your neighbors.
2. **Alignment** (also known as “copy”): Steer in the same direction as your neighbors.
3. **Cohesion** (also known as “center”): Steer towards the center of your neighbors (stay with the group).



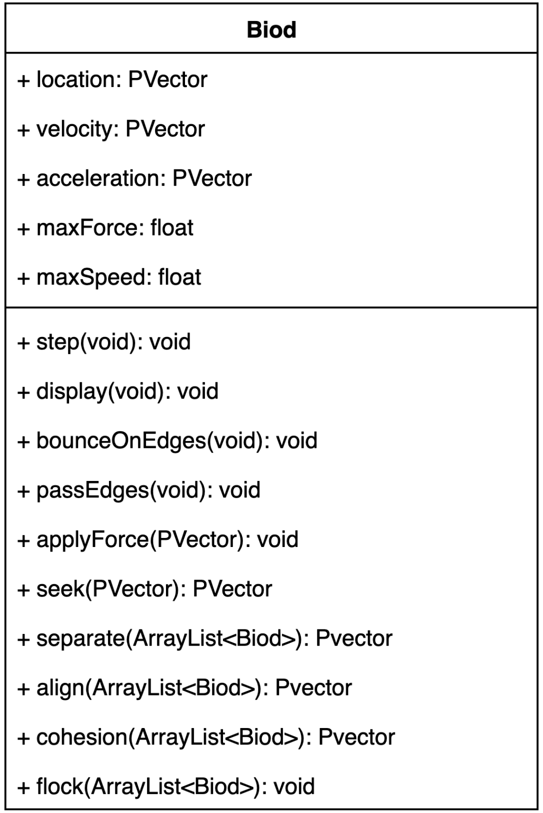
Reimagining the Vehicle as a Biod

Let us rename our Vehicle as Biod.

We need to add two more simple behaviours:

- `align`, and
- `cohesion`

These with `separate` should be enough to help us modify the `applyBehaviour` to `flock`



Biod

+ location: PVector
+ velocity: PVector
+ acceleration: PVector
+ maxForce: float
+ maxSpeed: float

+ step(void): void
+ display(void): void
+ bounceOnEdges(void): void
+ passEdges(void): void
+ applyForce(PVector): void
+ seek(PVector): PVector
+ separate(ArrayList<Biod>): Pvector
+ align(ArrayList<Biod>): Pvector
+ cohesion(ArrayList<Biod>): Pvector
+ flock(ArrayList<Biod>): void

The align behaviour

The boid’s desired velocity is the average velocity of its neighbors.:

- Add up all the velocities of the biods near me and divide by the total of biods near me to calculate the average velocity.
- Change that to have a magnitude of max speed
- Using the desired velocity calculate the steering force and return it

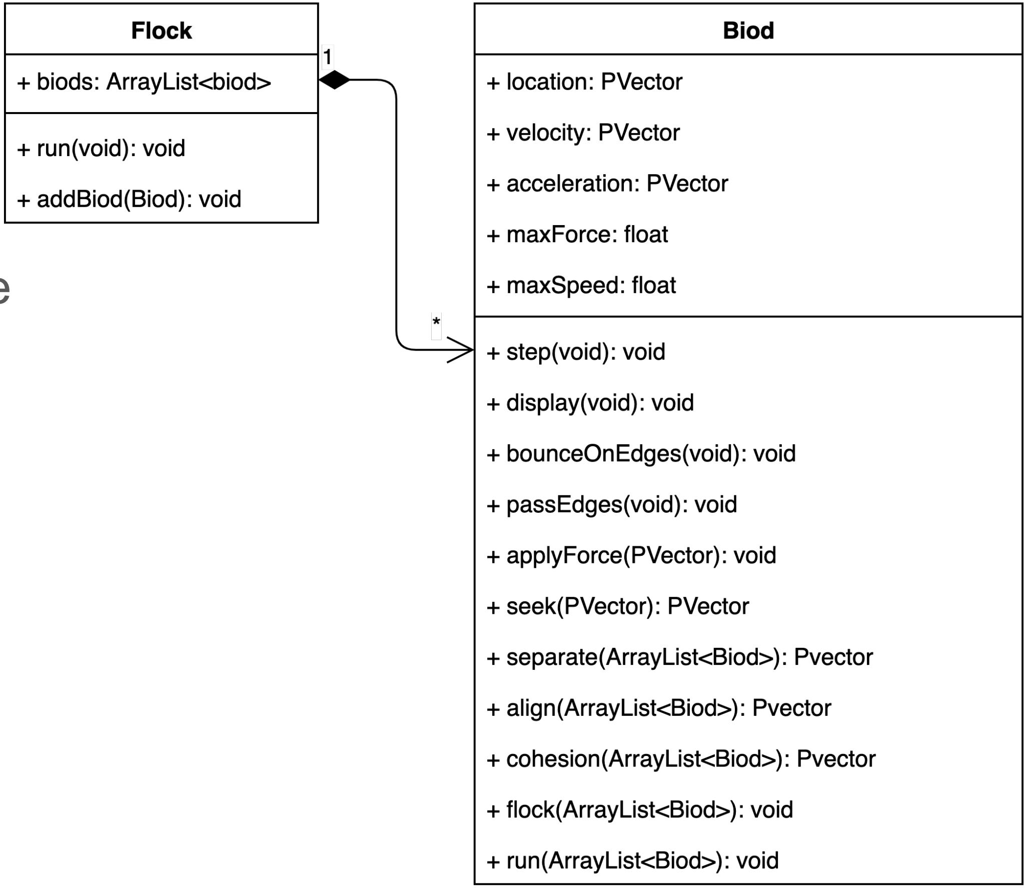
The cohesion behaviour

Here our code is virtually identical to that for alignment—only instead of calculating the average velocity of the boid’s neighbors, we want to calculate the average location of the boid’s neighbors (and use that as a target to seek).

The flock class

Let us create a class `Flock` that will contain all the biods. It will handle the addition of biods.

We will also add a run method to the biod (steps and displays the biod)



Efficiency

The Big O (big-oh) notation

Big O describes the efficiency of an algorithm: how many computational cycles does it require to complete (at worst case)?

- Assume you have an array of 100 ints with only one above 10. The big-O notation is 100 (max checks you need to do)
- If we have 100 biods, how many checks we need to do in order to find for every biod who are those close to it?
- In 1000? In 10000?

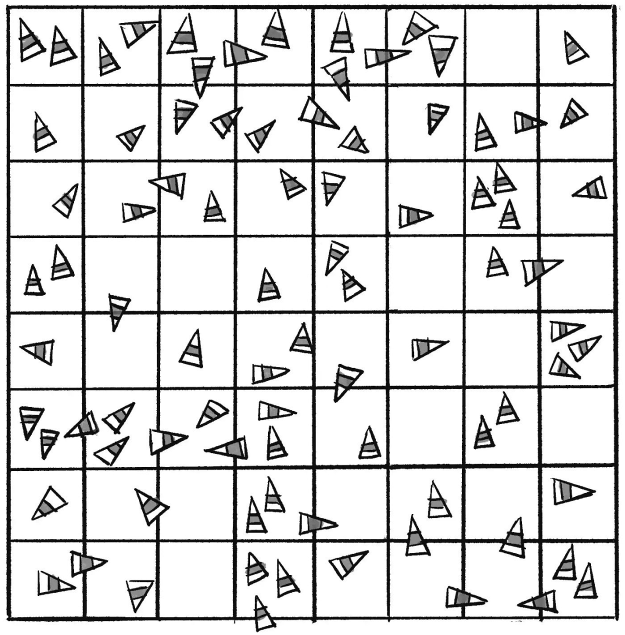
Solving the Big-O problem

What if we could divide the screen into a grid? We would take all 2,000 boids and assign each boid to a cell within that grid.

We would then be able to look at each boid and compare it to its neighbors within that cell at any given moment.

This technique is known as “bin-lattice spatial subdivision”.

Even better, to capture all the border cases we would only need to consider the cells around the central cell that contains the boid of interest, giving rise to just looking at 9 cells.



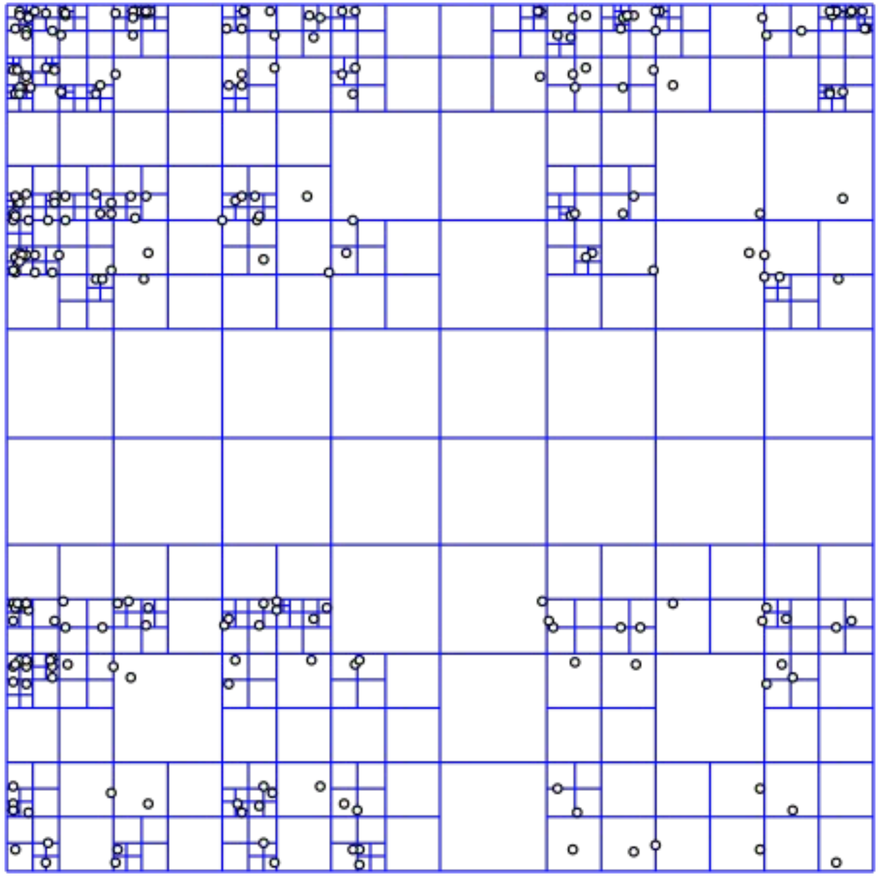
A better division: Quadtrees

A quadtree is a tree data structure in which each internal node has exactly four children. Quadtrees are the two-dimensional analog of octrees and are most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions. The data associated with a leaf cell varies by application, but the leaf cell represents a "unit of interesting spatial information".

The subdivided regions may be square or rectangular, or may have arbitrary shapes. This data structure was named a quadtree by Raphael Finkel and J.L. Bentley in 1974.[1] A similar partitioning is also known as a Q-tree.

Quadtree features

1. They decompose space into adaptable cells.
2. Each cell (or bucket) has a maximum capacity. When maximum capacity is reached, the bucket splits. Every extra item is added to the appropriate child cell
3. The tree directory follows the spatial decomposition of the quadtree.



Quadtree implementation

To implement a quadtree we need the following classes:

- Point: describes a point of interest (position) of an item in the quadtree. It also has a reference to the object that occupies that position
- Rectangle: describes an area in the quadtree. This is a node in the tree, It has a capacity and a flag set to true if it contains children areas
- Circle: describes a circular area (usually around the point we want to investigate.
- Quadtree

How a quadtree works

Originally the quadtree has one area (a single rectangle. Each time a point is added, we examine the capacity of the area, and when that is exceeded, the area is divided to four child rectangular areas. Then the point is attempted to be added to each of the subareas recursively.

The quad tree needs to be rebuild at every frame: For each item in the tree, a point is created and added to the tree.

When we want to find the neighbours of a point (given a range), we first find all areas that intersect the range, adding the other points in a collection.

Even if this operations seem expensive to do at every frame, it still vastly improves efficiency.

Costly operations (and how to avoid them)

1. To find the magnitude requires the square root operation. Square root is very costly. When we can, it is more effective to calculate the square of the magnitude:
   - Use `magSq()` instead of `mag()`

2. Calculating sin and cos is expensive. We can avoid it by creating arrays of lookup tables

3. Avoid making gazillions of unnecessary vector objects like in `draw()` writing:

Costly operations

```
let sinvalues = [];

for (let i = 0; i < 360; i++)
sinvalues.push(sin(radians(i)));



for (Vehicle v of vehicles) {

   Vector mouse = createVector(mouseX,mouseY);

   v.seek(mouse);

}
```