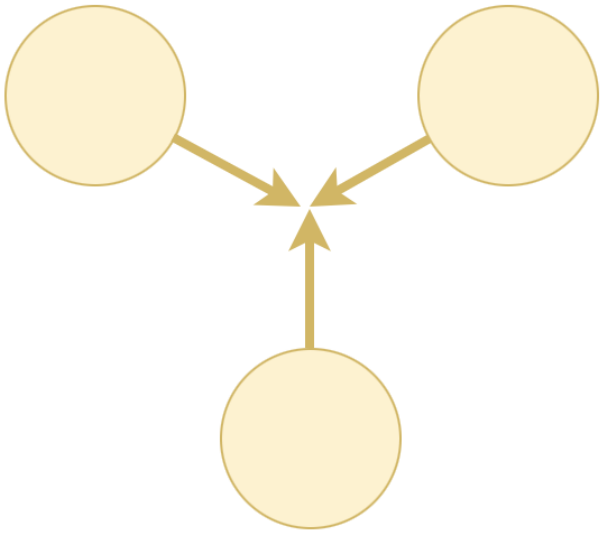
Flocking Simulation

Our flocks simulation is based on the classic algorithm Craig Reynolds showed in his 1987 SIGGRAPH paper, "Flocks, Herds, and Schools: A Distributed Behavioral Model.
The basic behavior of units is based on three principles.

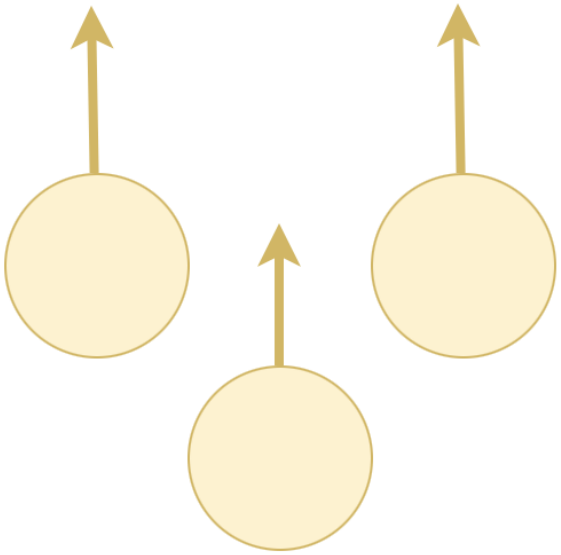
Cohesion

Each unit tends to take a position in the middle between the others



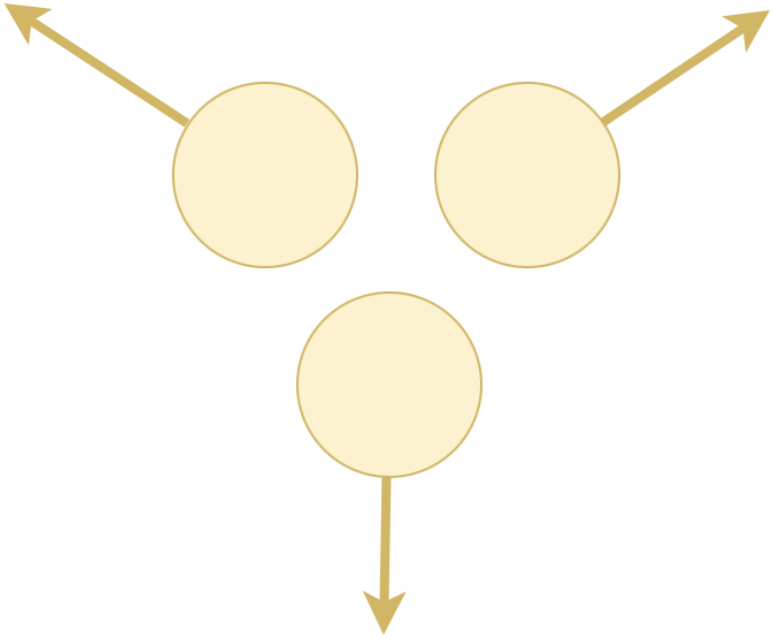
Alignment

Each unit tends to fly in the average direction that its neighbors are flying



Separation

Each unit tends not to collide with neighboring units



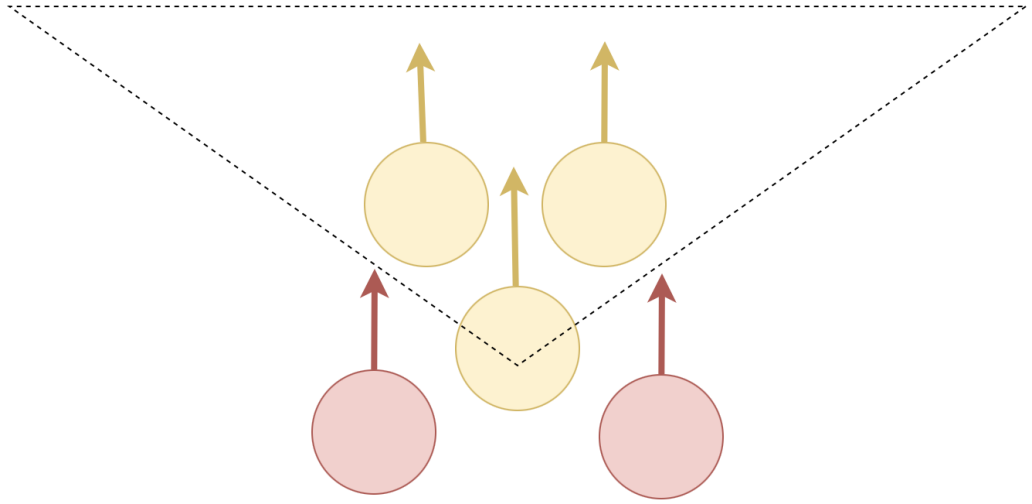
Continuously tracking each of these directions and changing the velocity accordingly gives us a simulation in which the units do not collide, fly in the same direction and stay in groups.

Each of the parameters has a weight that determines how much each force affects the simulation, so you can simulate what behavior suits you best - large groups flying together in one direction or smaller groups flying apart when they come into contact with others.
I set up the simulation so that it tends to form multiple groups of units that can fly apart if there are other forces with more weight that can affect them. I think this behavior is more like birds.

View angle and distance

In addition to these forces, it is necessary to understand who will be considered a neighboring unit. You can define all units as neighbors, then you get one big flock. You can use additional constraints, for a more free and variable behavior of each unit.
As restrictions you can use the distance and angle between the direction of movement of the unit and the position of the neighbor.

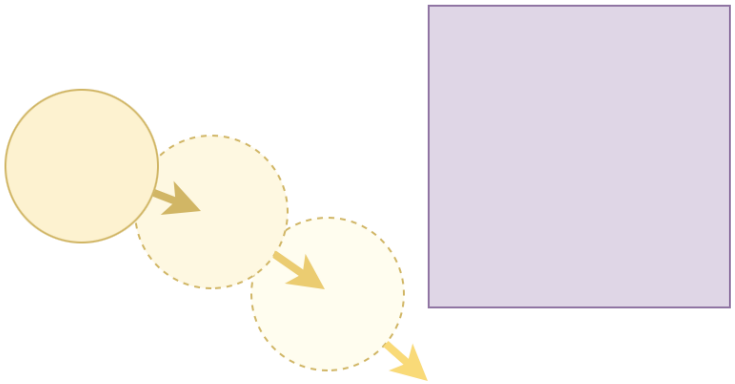
In this way we will make sure that who flies behind will adjust to those in front, which will make our simulation more realistic.



You can also use the view angle to make the flock wide or vice versa, so that the units tend to align in a line.

Obstacle Avoidance

In addition to the behavior of the flock itself, it must interact with its surroundings, and to do this it must avoid colliding with obstacles.



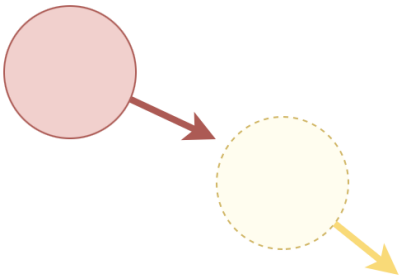
In order to fly around obstacles - we need to find them. One fairly easy way to do this is with RayCast. We throw a ray in the direction of our movement, if it collides with an obstacle, we change the direction of our movement.

Prey and hunter behavior

The special behaviors are additional items. In the future, we may want to create any number of behaviors in the simulation that we want, so it is better to organize them separately.

The prey's behavior implies that when it sees a predator, it should try to avoid it. This is accomplished with separation by analogy to one of the 3 basic principles.

Hunter's behavior, on the other hand, implies hunting for victims and getting as close to them as possible.



Organizing the behavior types into separate entities will allow us to create absolutely any behavior later, depending on our needs, without changing the rest of the simulation.

Current algorithm

The current algorithm assumes the following entities:

1. Boid - a class of unit that implies storage of basic data needed for simulation - position, speed, radius of the unit. As well as the type of its behavior.
2. Behavior - the unit's behavior, storing and encapsulating all the data needed to perform this behavior.
3. Simulation - a class responsible for creating, processing and controlling units.

The current implementation will allow us to extend behaviors to additional requirements, including using the project as a library, without changing its source code, which allows us to have a small class dependency and an easy way to extend the algorithm.
The main project has a FlockingManager class - responsible for things not directly related to the simulation, such as rendering. Our simulation should not be aware of such additional things and should only be responsible for the position of points in space and the interaction of elements, this will allow us to use it in any way we want, without depending on the rendering process or anything else.

Nevertheless, given that this is just a simulation done in a few hours after work during for a week, there are certain problems in it that we haven't had time to solve at the moment.
Let's outline them and consider ways to solve them.

Problems

1. Low speed of finding neighbors. The way of storing units in a vector gives us a high speed of getting a unit by index or iterating through them, but a low speed of searching. In the future I would prefer to use kd-tree to store units, the spatial division will give us a high search speed in a given radius.

2. Not perfect obstacle evasion. Since the algorithm for avoiding obstacles at the moment is extremely simple - the units do not always manage to avoid them, though in most cases do. Rather accurate results would be given by an algorithm for generating quaternions at the moment of program start, which would describe a set of rotations, allowing when multiplied by a velocity of the unit - to get a set of rays, in the direction of movement, but with a deviation from it. Thus, it would be possible to check collision with objects along these directions and choose from them the most suitable one, which did not collide with any object.

How to improve the algorithm

For this algorithm, it is preferable to use ECS instead of OOP, because this will give us the opportunity to get rid of class dependency and define the required set of components with a given behavior and easily change the behavior without any difficulty, simply by replacing the component with an entity.

This structure could be described in the following way:

`MovementComponent`, storing position and speed

`PreyBehaviorComponent`, storing the data needed to simulate the behavior of the victim
`HunterBevaiorComponent` that stores the data needed to simulate the behavior of the predator

and a set of systems that process this data:

`MovementSystem`, which deals with the movement of the position in a given direction on dt
`PreyBehaviorSystem`, which processes the prey's behavior data
`HunterBevaiorComponent`, which processes the hunter's behavior data

This architecture would allow us to add parallelism without any additional effort on our part, to add any behaviors and data we would need in the future, simply by adding a component and system.