Lecture 14-15:
Boids continued

CGRA 354 : Computer Graphics Programming

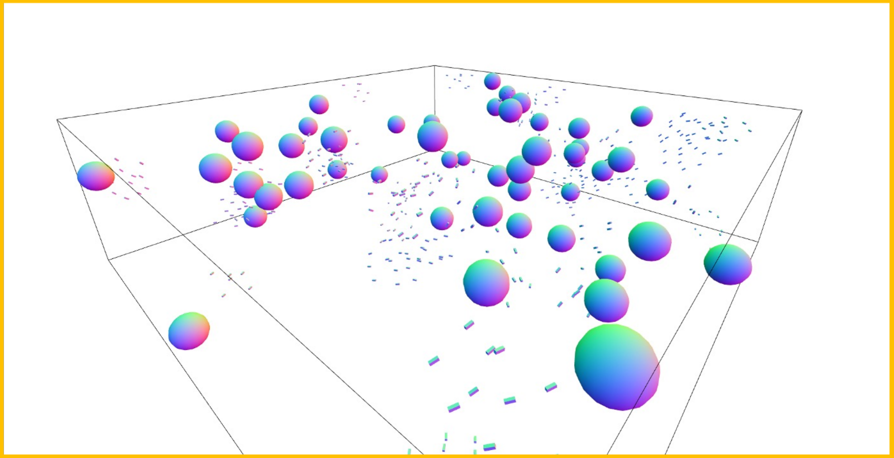
Instructor: Alex Doronin
Cotton Level 3, Office 330
alex.doronin@vuw.ac.nz

With slides from: Holly Rushmeier, Yale; Steve Marschner, Cornell; Taehyun Rhee, CMIC; Zohar Levi, ECS;

What is Boids?

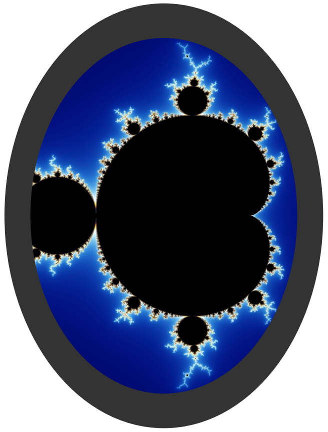
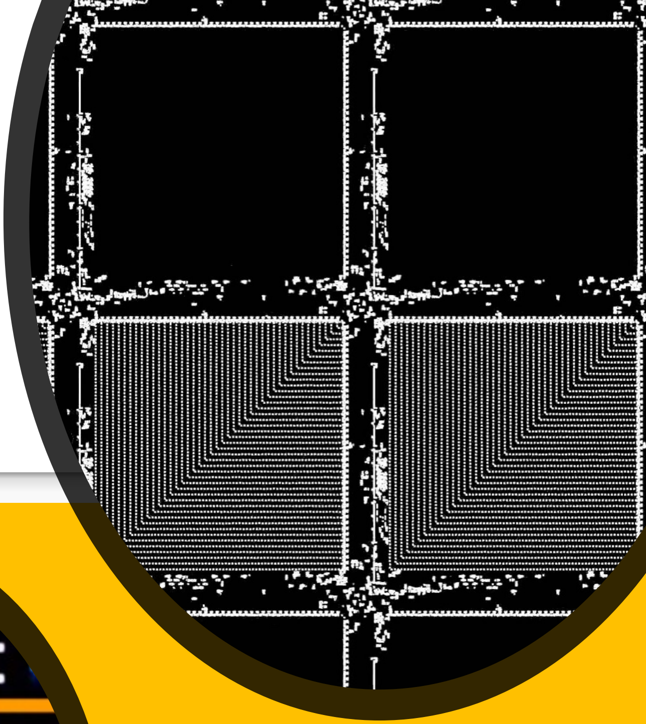
Boids is a classic flocking simulation algorithm created by Craig Reynolds in 1986. It is notable for its simplicity, using a set of only 4 basic rules to generate interesting flock behaviours.

Because of its simplicity, it can easily be adapted to suit many applications, and has been used as a basis for swarm AI in video games and films.

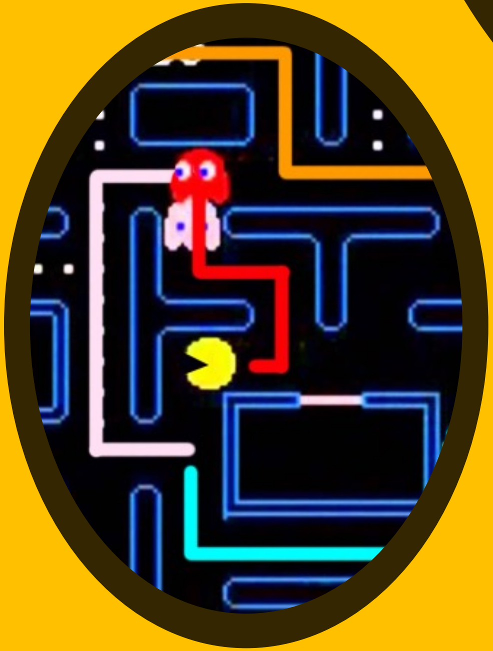


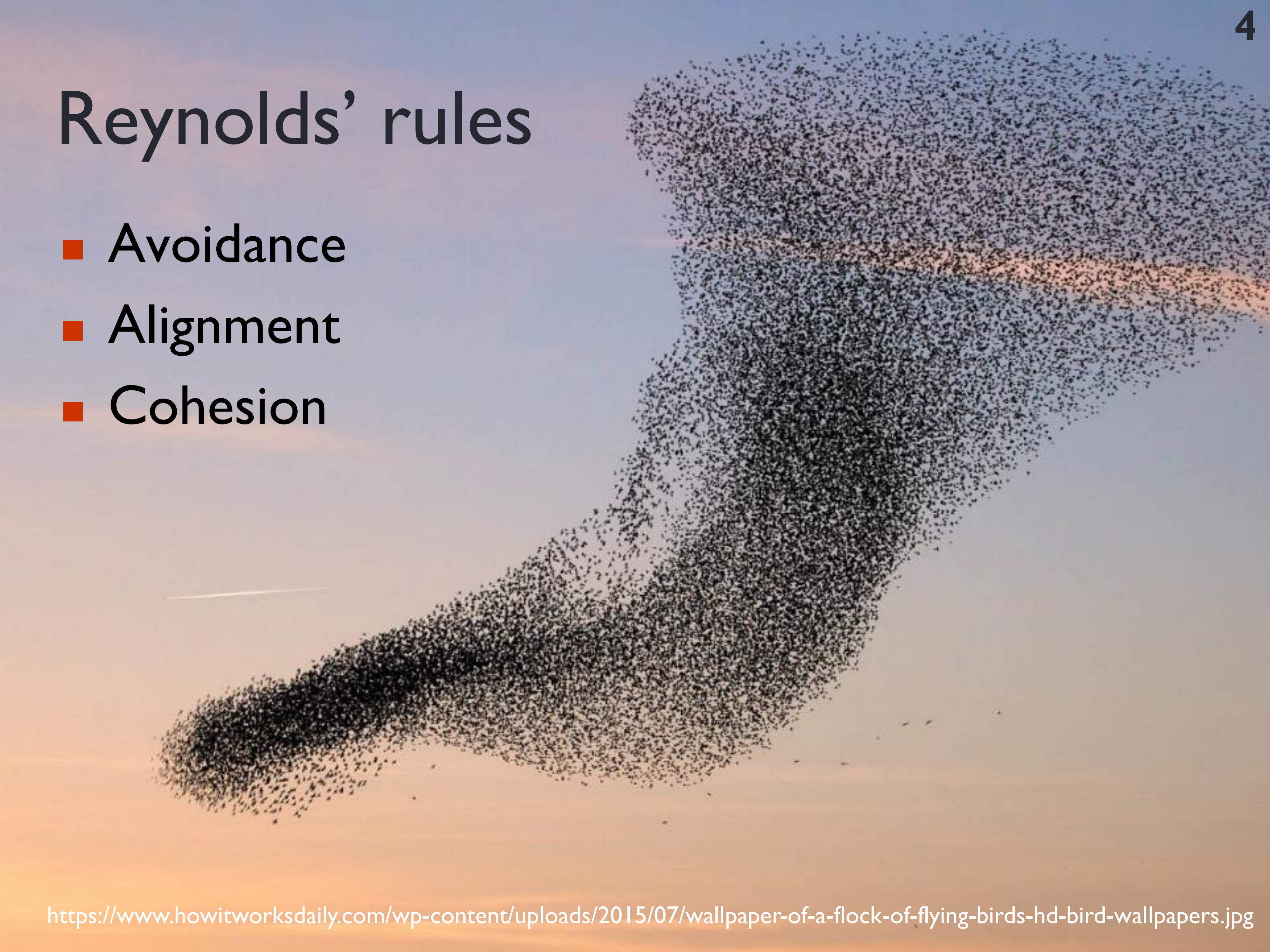
Web-Based Boids Simulation (runs in browser)

Emergence

- Boids is a canonical example of an emergent system, where a set of simple rules exhibits complex, occasionally even "intelligent" behaviour.
- Emergence is closely related with chaotic systems and information theory. Other examples of emergent systems include: Rule 30 Cellular Automata, Conway's Game of Life, The Mandelbrot Set, and Pac-Man Ghosts. However, the best examples of emergence are in fact everyday objects, such as birds, ants, fungi, even DNA.





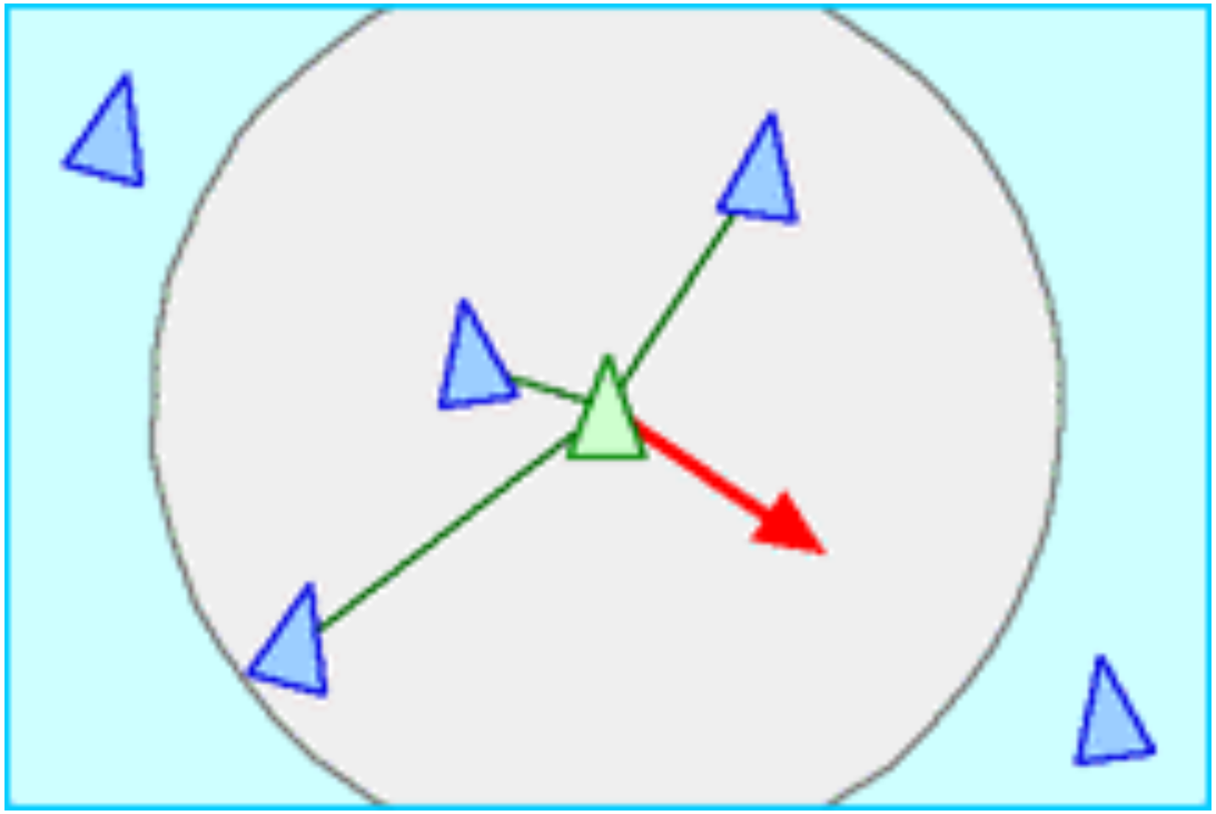
Reynolds' rules

- Avoidance
- Alignment
- Cohesion

https://www.howitworksdaily.com/wp-content/uploads/2015/07/wallpaper-of-a-flock-of-flying-birds-hd-bird-wallpapers.jpg

Avoidance

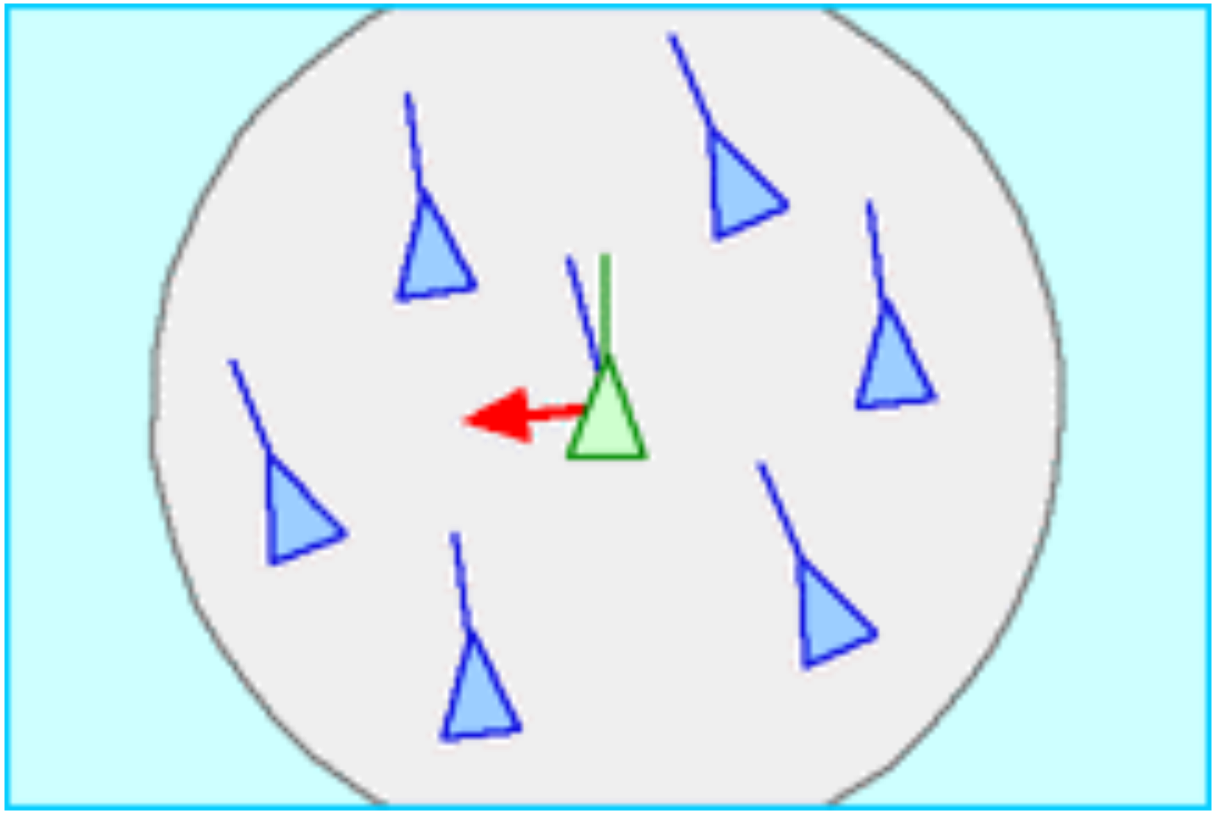
- Boids avoid crashes



- Each boid looks at the flock mates in its neighbourhood and applies a force to push it away from its neighbours

Alignment

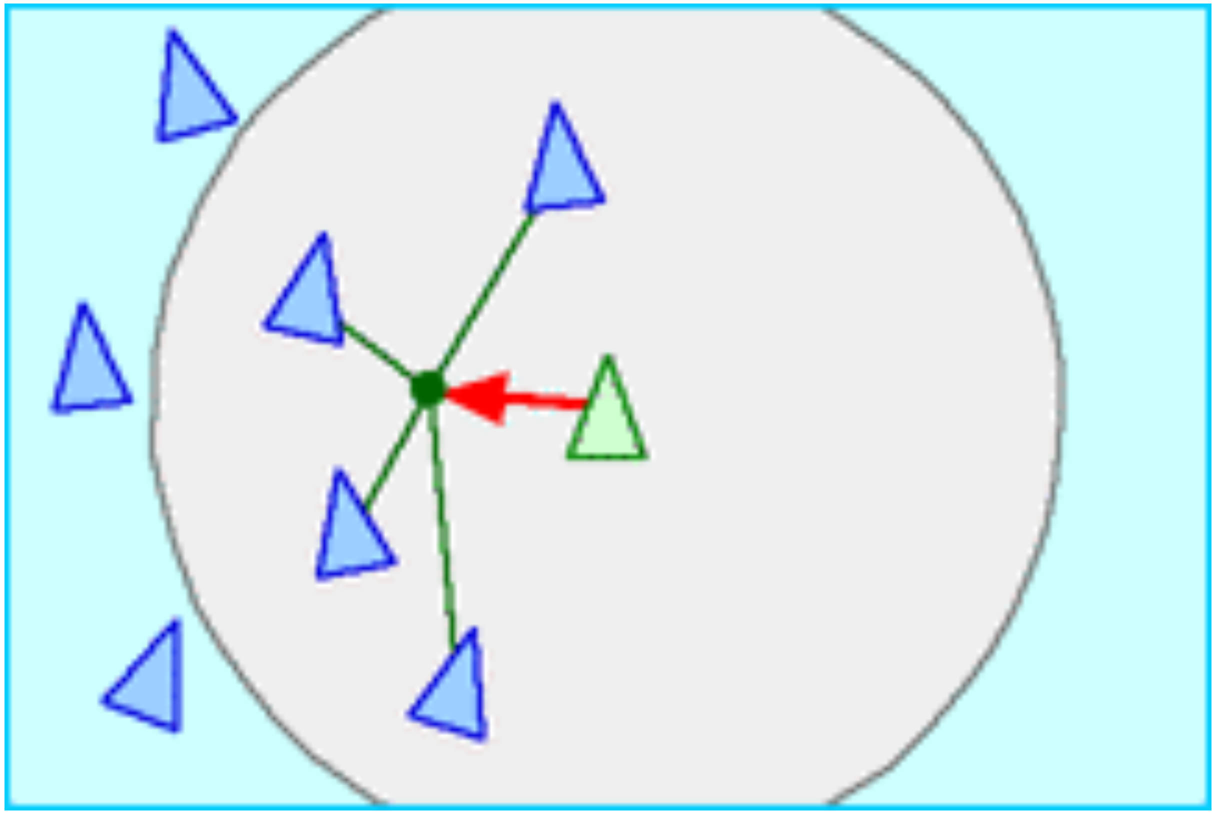
- Boids want to fly in the same direction



- Each boid looks at the flock mates in its neighborhood and applies a force to line it up with the average direction of its neighbours

Cohesion

- Boids want to be near their flock mates



- Each boid looks at the flock mates in its neighborhood and applies a force to move towards the average position of its neighbours

Position, Velocity, Acceleration

Simulation is a major subfield of Computer Graphics. For most of you, this will be your first time programming a 3D simulation.

Boids is **not** a physics simulation - it's much simpler. But it uses notions of position, velocity, and acceleration, which you will need to be familiar with.

- Position tells us where the Boid is in the scene.
- Velocity tells us how far the position will move in 1 second.
- Acceleration (aka Force) tells us how much the velocity will change in 1 second.

We can update an object in the simulation by adding Velocity to Position, and adding Acceleration to Velocity. This will simulate 1 second of real-world time.

Vector and Scalar Quantities

For implementing spatial or physical algorithms such as Boids, we're often interested in quantities in both vector (3 components) and scalar (a single number) forms. It's important to have a strong grasp on the terminology.

Displacement is a vector quantity, which measures a difference in position. In other words, the displacement from A to B is B-A. *Distance* is a scalar, and is the magnitude of the displacement.

Velocity is a vector quantity, which measures a difference in position over time (displacement over time). *Speed* is a scalar, and is the magnitude of the velocity.

Direction is a non-specific term, but may be used for the normalized form of a vector quantity. The direction from A to B is (B-A) / |B-A|; its normalized displacement.

Timestep

Computers are much faster than 1 frame per second, so we want to simulate a much shorter time interval than 1 entire second.

The Timestep tells us how many fractions of a second we want to simulate. The precise value of this will depend on the speed of your computer, but we need to account for it when we update our simulation. We can easily do this by multiplying the Boid's changes in position and velocity by it:

position = position + timestep*velocity
velocity = velocity + timestep*acceleration

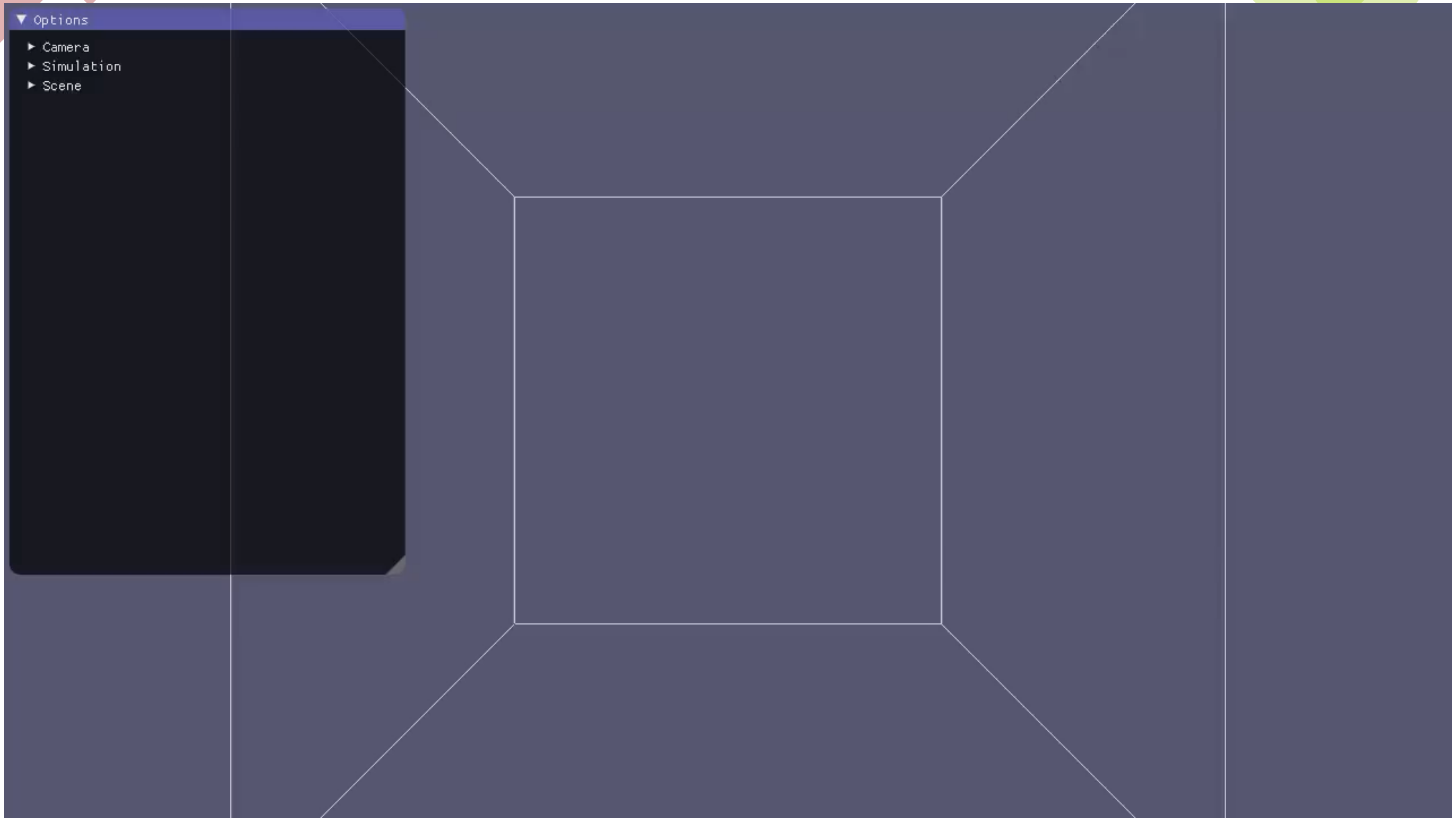
This is more or less what your Boid::update implementation will look like.

Structure of a Physics Simulation

Physics simulations generally follow a similar structure.

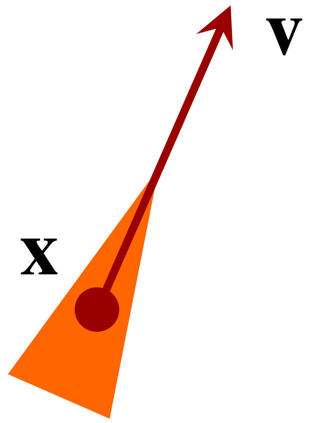
```
initialize_simulation()
loop {
  foreach object in scene: object.calculate_forces()
  foreach object in scene: object.apply_forces()
  render()
}
```

This code structure is provided for you by the Assignment 3 Framework. You only need to implement parts of each step.


▼ Options
▶ Camera
▶ Simulation
▶ Scene

Representing a boid

- A boid has
  - A position $$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$
  - Velocity $$\mathbf{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$



Updating a boid's position

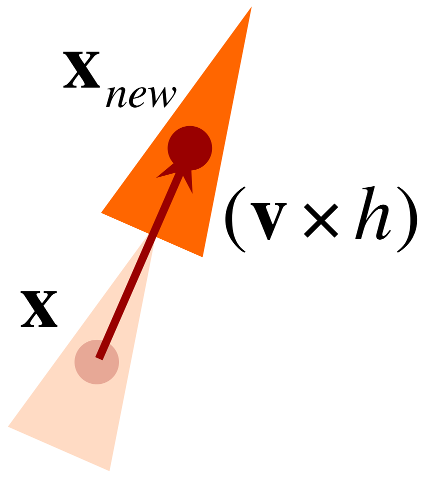
- Add velocity (times the time-step) to position

$$\mathbf{x}_{new} = \mathbf{x} + (\mathbf{v} \times h)$$

$$x_{new} = x + (v_x \times h)$$

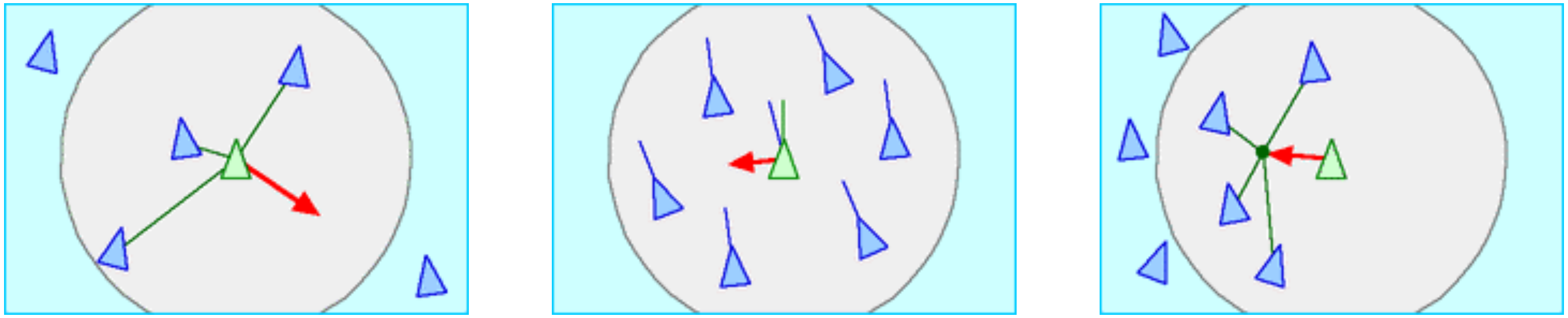
$$y_{new} = y + (v_y \times h)$$

$$z_{new} = z + (v_z \times h)$$



- h is the time-step (scalar)

How do the forces work?



$$\mathbf{x}_{new} = \mathbf{x} + (\mathbf{v} \times h)$$

- Position is updated by velocity

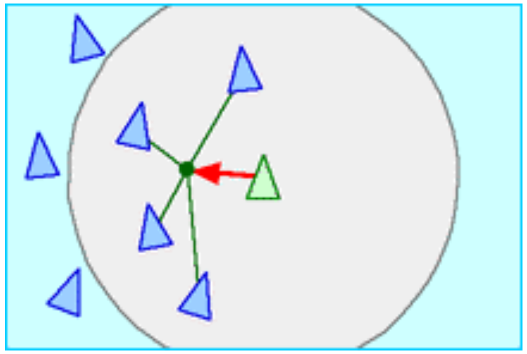
$$\mathbf{v}_{new} = \mathbf{v} + (\mathbf{a} \times h)$$

- Velocity is updated by acceleration

$$\mathbf{a} = \mathbf{f} / m$$

- Acceleration is force/mass

Cohesion



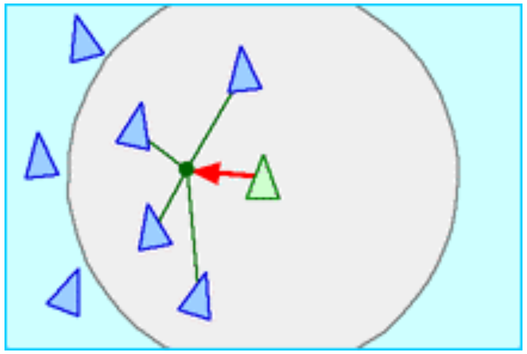
- Find the centroid of the neighbours' positions

$$\mathbf{x}_c = \sum_{j \in N(i)} \mathbf{x}_j / |N(i)|$$

- Create a force that goes from your position to the centroid

$$\mathbf{f}_x = k_x(\mathbf{x}_c - \mathbf{x}_i)$$

Who are the boids in your neighbourhood?

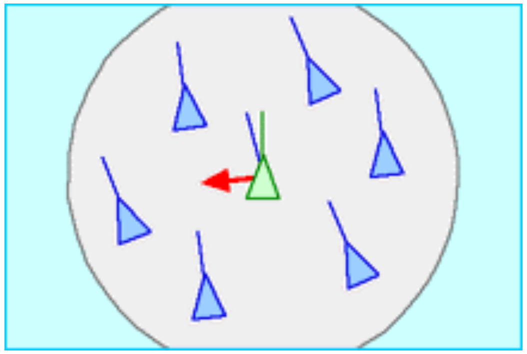


- We specify a distance d that a boid can "see"
- We check all other boids to see if they are within distance d

$$N(i) = \left\{ j : j \neq i \wedge \left| \mathbf{x}_j - \mathbf{x}_i \right| < d \right\}$$

Google: "Sesame Street who are the people in your neighbourhood"

Alignment



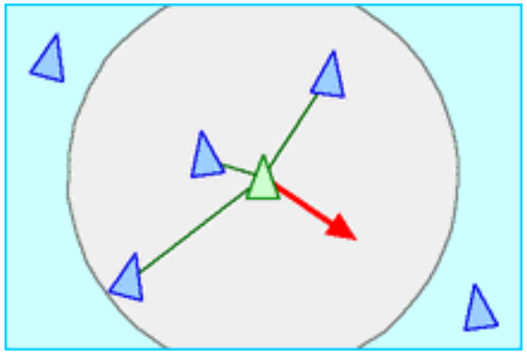
- Find the average of the neighbours' velocities

$$\mathbf{v}_c = \sum_{j \in N(i)} \mathbf{v}_j / |N(i)|$$

- Create a force that adjusts the boid's velocity to be closer to the average speed

$$\mathbf{f}_v = k_v(\mathbf{v}_c - \mathbf{v}_i)$$

Avoidance



$$\mathbf{f}_a = k_a \overbrace{\sum_{\underbrace{j \in N(i)}_{1}}}^{4} \underbrace{\frac{1}{\left|x_i - x_j\right|}}_{3} \underbrace{\frac{(x_i - x_j)}{\left|x_i - x_j\right|}}_{2}$$

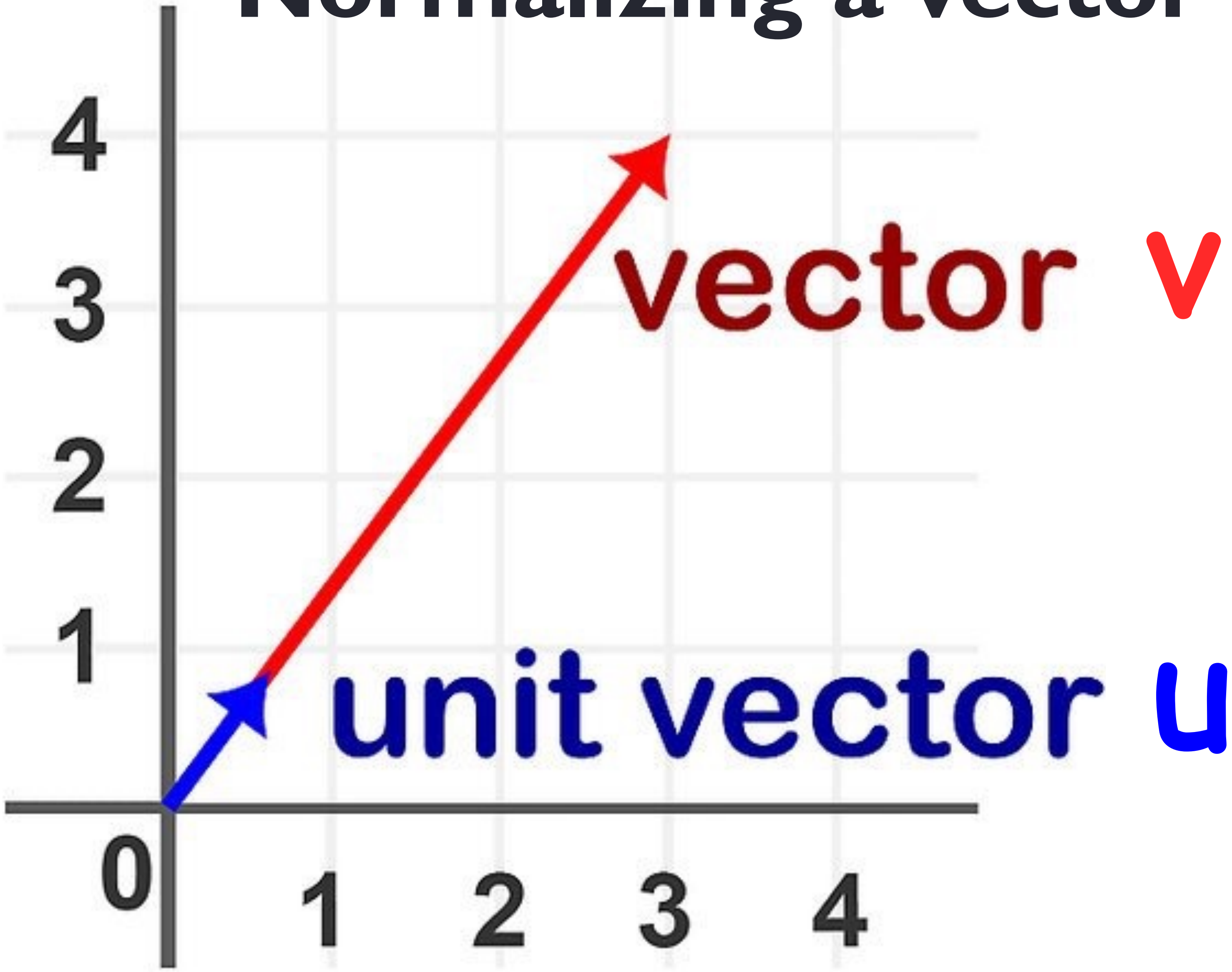
1 For each boid in the neighbourhood create a force that

2 pushes away from the boid,

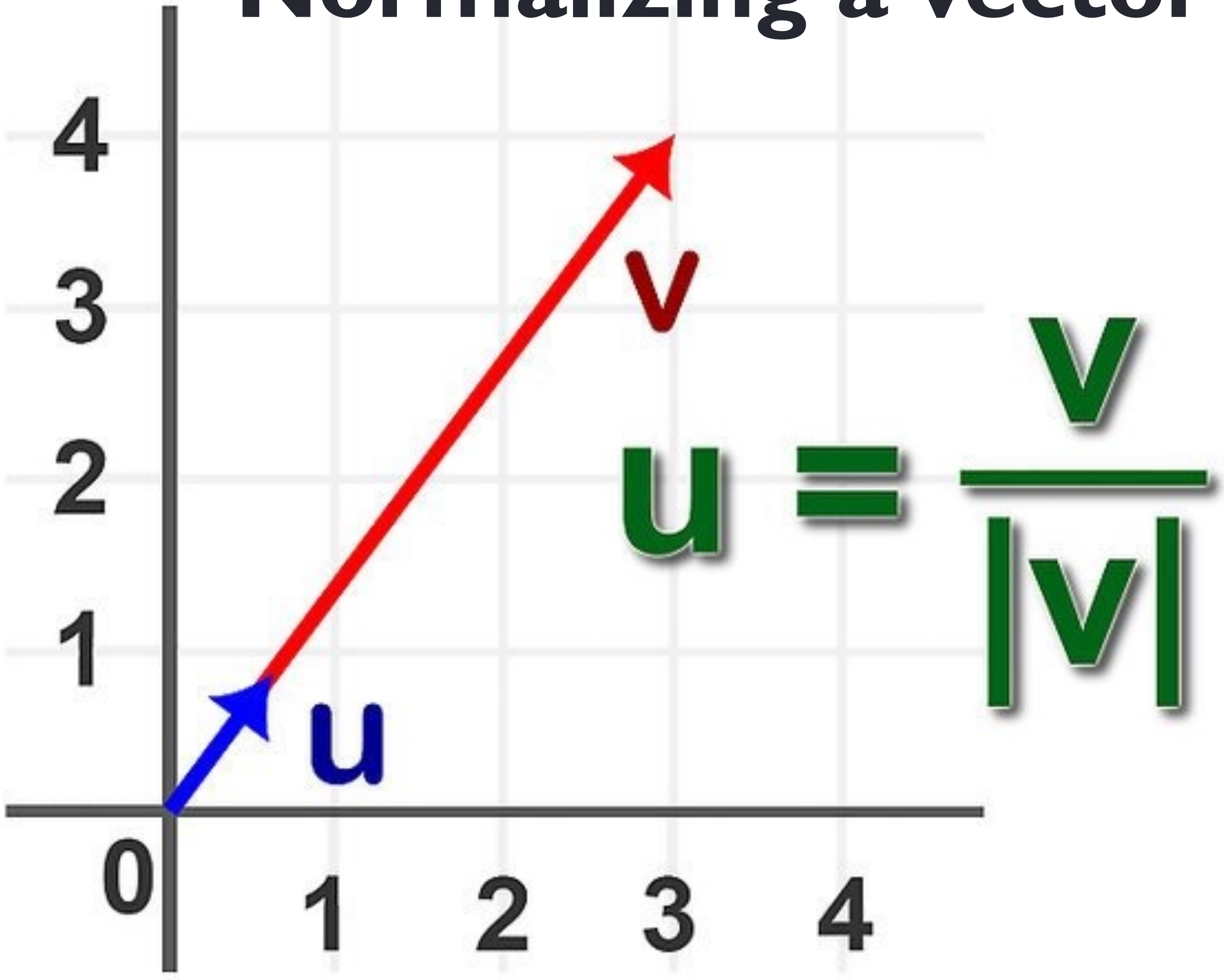
3 weighted by the inverse of the distance,

4 add all these forces together

Normalizing a vector



Normalizing a vector

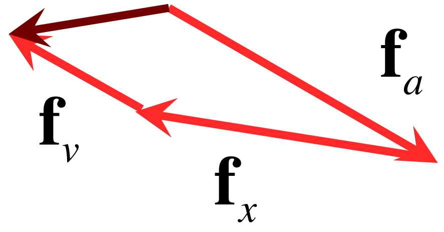


$$u = \frac{v}{|v|}$$

Applying those forces

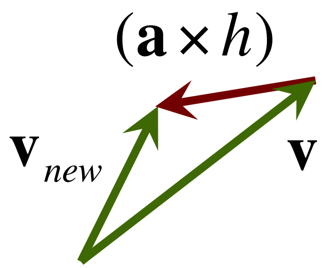
- Add up the forces and divide by the boid's mass

$$\mathbf{a} = (\mathbf{f}_x + \mathbf{f}_v + \mathbf{f}_a) / m$$



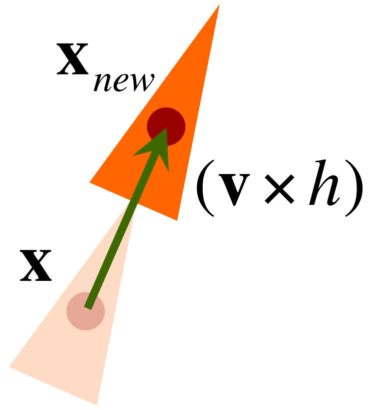
- Update velocity

$$\mathbf{v}_{new} = \mathbf{v} + (\mathbf{a} \times h)$$



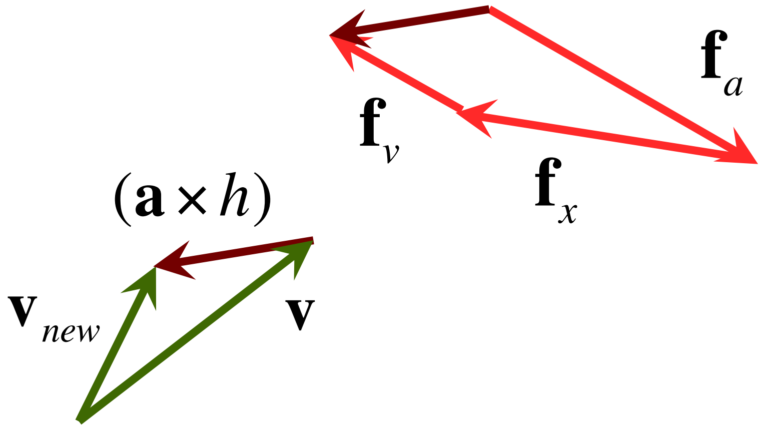
- Update position

$$\mathbf{x}_{new} = \mathbf{x} + (\mathbf{v} \times h)$$

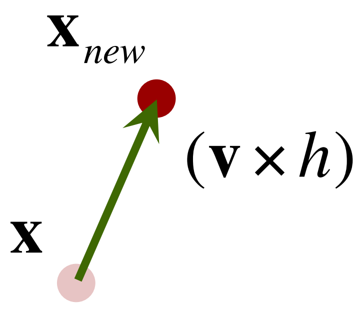


What operations make sense?

- vector = vector + vector
- vector = vector – vector
- vector = scalar × vector
- vector = point – point

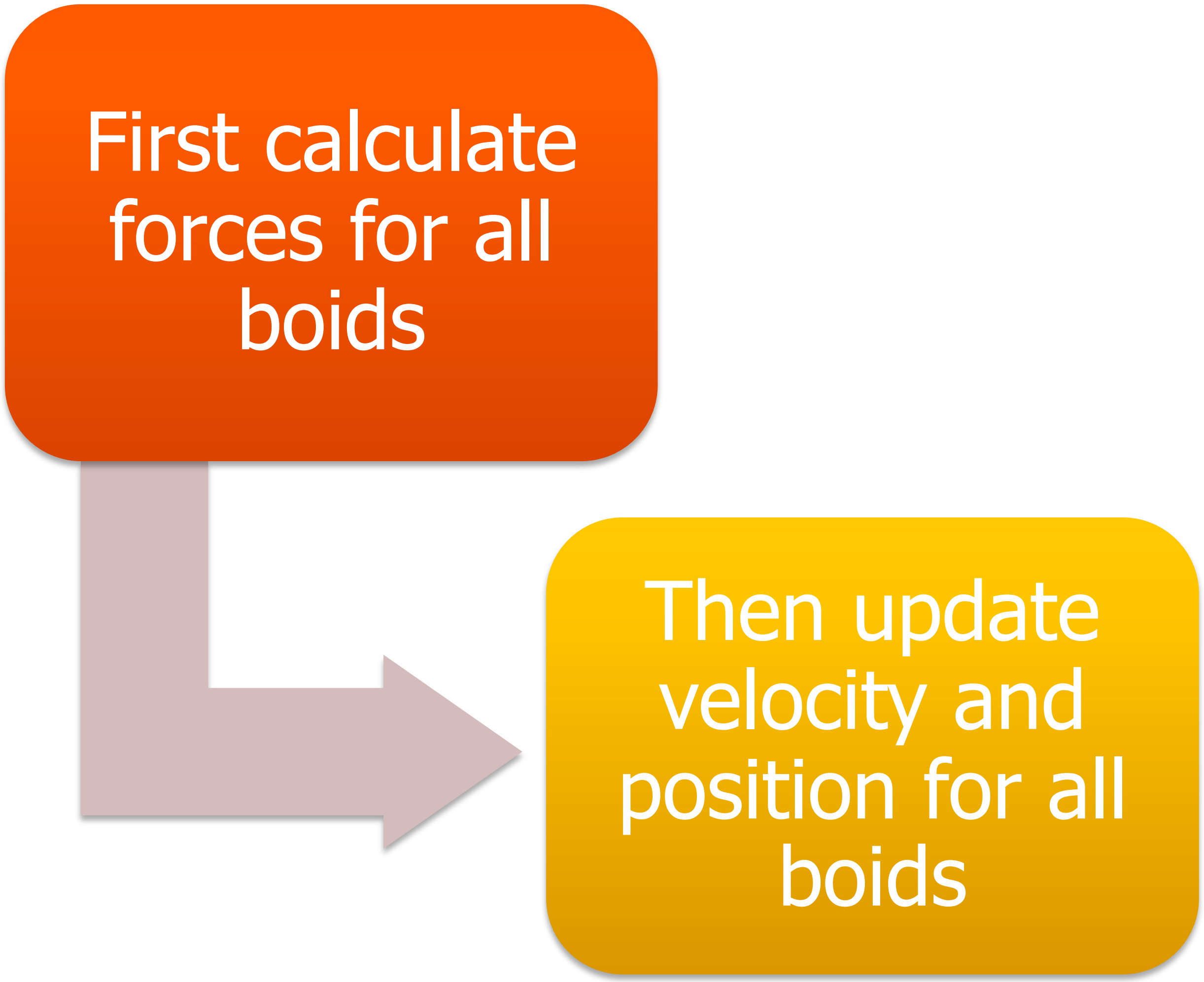


- point = point + vector
- point = point – vector
- point = average of points



$$\mathbf{x}_c = \sum_{j \in N(i)} \mathbf{x}_j / |N(i)|$$

Must update all boids together


First calculate forces for all boids
Then update velocity and position for all boids

What they don't usually tell you

- Need to balance forces carefully (experiment)
  - Careful choice of k_x, k_v, k_a
- Mass is an arbitrary number
  - If all boids weigh the same, can pretend that $m=1$
- Need to limit speed and force
  - Enforce a maximum speed and a maximum force
  - Apply maximum force limit to each force individually

How do I limit a vector?

Limiting a scalar

$$\text{if } f > \max$$

$$\text{then } f_{new} = \max$$

Limiting a vector

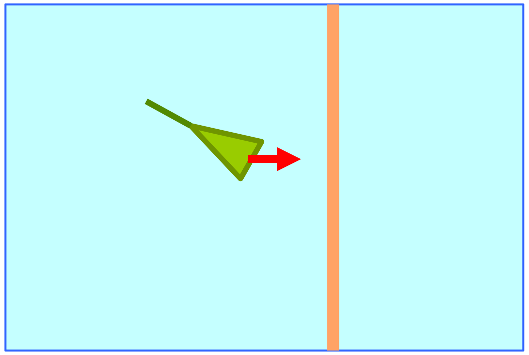
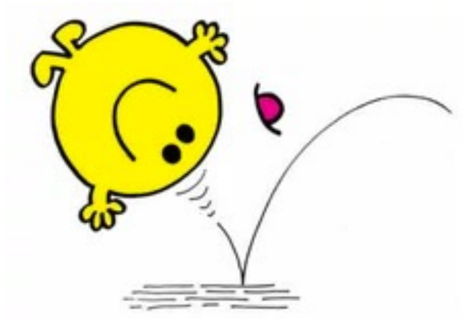
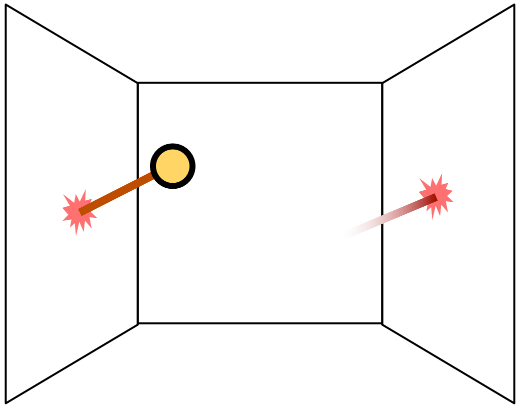
$$\text{if } |\mathbf{f}| > \max$$

$$\text{then } \mathbf{f}_{new} = \max \times \frac{\mathbf{f}}{|\mathbf{f}|}$$

What else haven't you told me?

- You need to stop the boids from flying off into the distance
- Define an axis-aligned box to keep the boids in and then:

Force *or* ***Bounce*** *or* ***Wrap***

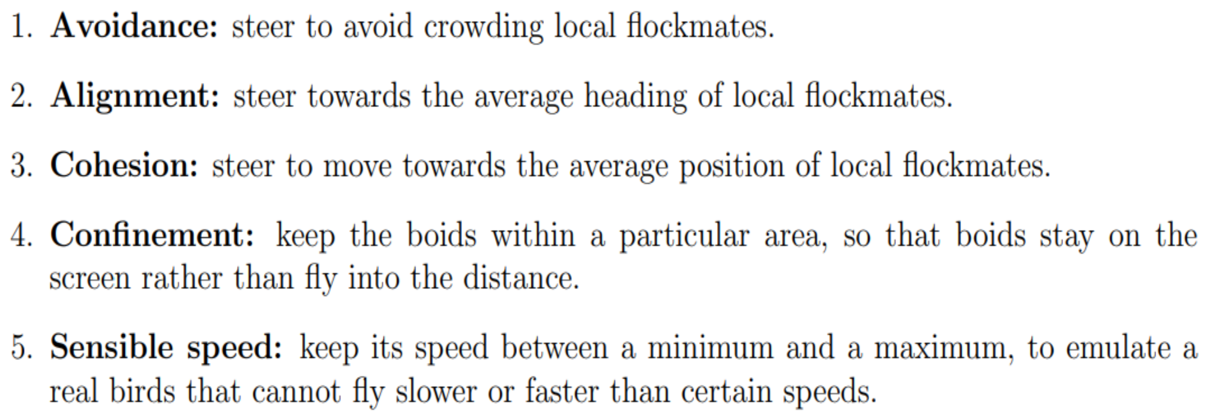




Implementing Boids

Overview

The original Boids algorithm features 3 basic rules. For Core, you need to implement these 3 and 2 additional rules which improve the overall quality of the simulation.

For Completion and Challenge you will need to add additional rules and extend the existing ones.



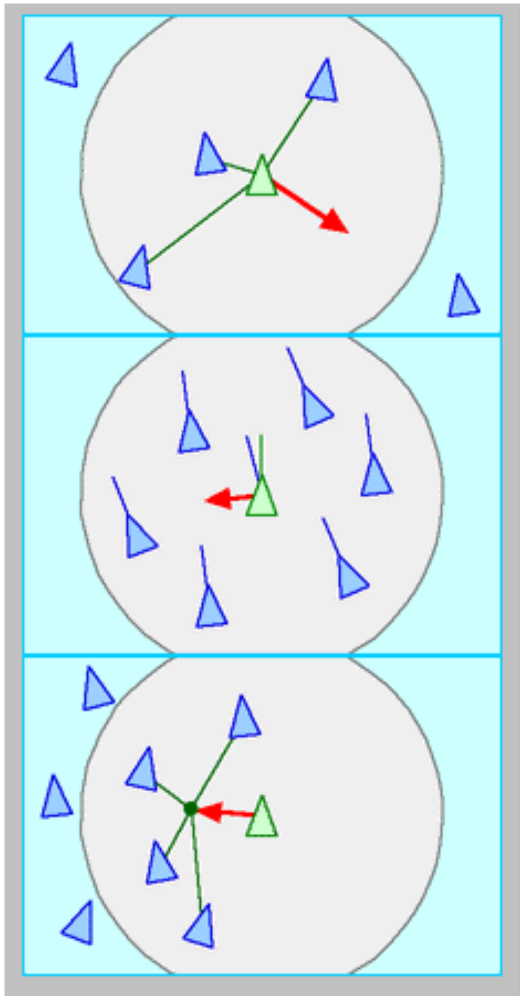
1. **Avoidance:** steer to avoid crowding local flockmates.
2. **Alignment:** steer towards the average heading of local flockmates.
3. **Cohesion:** steer to move towards the average position of local flockmates.
4. **Confinement:** keep the boids within a particular area, so that boids stay on the screen rather than fly into the distance.
5. **Sensible speed:** keep its speed between a minimum and a maximum, to emulate a real birds that cannot fly slower or faster than certain speeds.

Boids Parameters

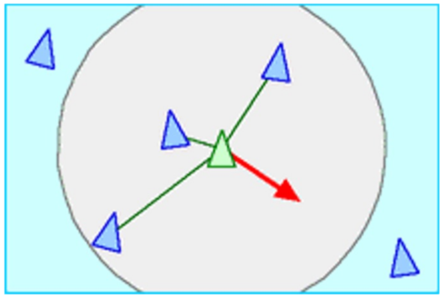
Every rule has several parameters, which you should make controllable with an ImGui interface. Most parameters fall into one of two categories:

- Radius: The farthest distance away that another Boid can have to affect this one.
  - For example, avoidance will have no effect if all other Boids are outside of the avoidance radius.
- Weight: How strong the force of this rule is in comparison to the other forces.
  - When you calculate the final force on each Boid, use a weighted sum:
    ```
    total_acceleration = avoidance_force * avoidance_weight
                       + cohesion_force  * cohesion_weight
                       + alignment_force * alignment_weight
    ```
  - In previous slides these weights are factored into each force calculation as k.

You'll need to have at both a radius and weight parameter for all of the first 3 rules. That's a lot of parameters though, so it's fine to have a single "neighbourhood radius" that we use for the first 3 rules.



Avoidance



Avoidance is the force which stops Boids from being too close to each other.

Each Boid within the avoidance radius contributes to the avoidance force. Boids outside the radius have no effect.

The strength of the force should be inversely proportional to the distance - the closer the boids, the stronger the repulsion. We can do this by dividing the displacement direction by the distance. Equivalently: the displacement divided by its squared magnitude.

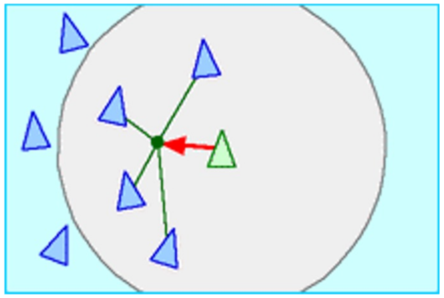
```
avoidance = vec3(0)
foreach other_boid in boids:
  if boid is other_boid: continue // be careful not to copy value when comparing ptrs in C++

  displacement = boid.position - other_boid.position
  distance     = length(displacement)

  if distance > avoidance_radius: continue

  avoidance += displacement / (distance * distance)
```

Cohesion



Cohesion is the force which helps Boids stay together in flocks

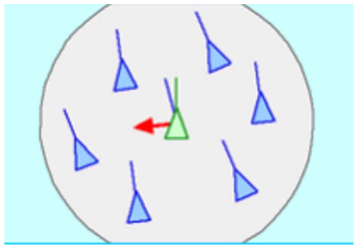
We can implement cohesion by first calculating the average position of other Boids within the cohesion radius. Then, apply the force away from that. The strength cohesion of should be proportional to the distance - the opposite of avoidance. For this, we can simply use the displacement from the centroid.

```
average_position = vec3(0)
neighbours = 1

// inside foreach:
  average_position += other_boid.position
  neighbours += 1

average_position /= neighbours

cohesion = average_neighbour_position - boid.position
```



Alignment

Alignment is the force that keeps the flocks of Boids heading in a similar direction.

The easiest way to implement alignment is exactly the same as cohesion, but using velocity instead of position.

```
average_velocity = 0

// inside foreach:
  average_velocity += other_boid.velocity

average_velocity /= neighbours

cohesion = average_neighbour_position - boid.position
```

Containment

Containment can be implemented either as a force or constraint. The simplest implementation is to wrap the boids' positions to the other side of their bounding box.

We can concisely implement wrapping using the modulo operator if we temporarily map the points to a range with the origin as the lower bound. For an AABB, you can think of this as shifting the box's lower corner to the origin (with all the boids contained in it), performing the modulo, then shifting everything back to its original position.

The size (in each dimension) of the AABB is given by the difference of its two bounds, which we use as the interval size.

```
boid.position = aabb.min + mod(boid.position - aabb.min, aabb.max-aabb.min)
```

This is an element-wise operator: the position's x is wrapped by the interval's x, y wrapped by y, etc..

In the assignment, the bounds are represented by a single `vec3 m_bound_hsize`. This represents *half* the bounding box in each direction. So, the AABB is given by the range from `-m_bound_hsize` to `+m_bound_hsize`.

Sensible Speed

Sensible speed is not a force, but a constraint that we apply every simulation update. It's important to apply it *after* acceleration has been applied to velocity.

We can resize a vector by normalizing it and multiplying it by the desired magnitude.

```
speed    = length(velocity)
speed    = clamp(speed, min_speed, max_speed)
velocity = speed * normalize(velocity)
```

Note that this will fail if the velocity is zero, as the zero vector cannot be normalized.

Use Your Creativity

For this assignment, you do not need to use the algorithms provided verbatim. If your implementation is not working as well as you'd like, you are encouraged to modify the algorithms to solve these issues. For example, you may want to try using non-linear response functions (such as squaring) to make Boids react much more sharply when they are about to touch an object.

The important part is to find a configuration of code and parameters that you feel produces the most interesting bird-like behaviour. Computer Graphics is primarily graded on visual output, and this assignment is no exception.

When you submit your assignment, please make sure to choose default settings that best show off your simulation. We will not have time to find them for you while marking. You're also welcome to include any other interesting configurations in your readme.

Assignment 3 Framework

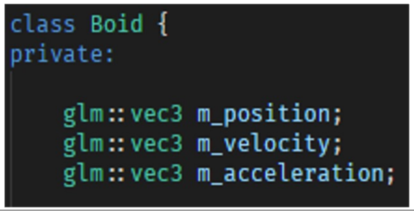
Framework Overview

For Assignment 3 (at least Core and Completion), you do not need to implement any rendering yourself. Two classes are given (`Boid` and `Scene`) which provide this functionality for you.

Almost all of your code will be implemented in the following two files:

- `boid.cpp` (where we implement the boids behaviour)
- `scene.cpp` (where we populate the scene with boids and draw additional GUI elements)

boid.cpp

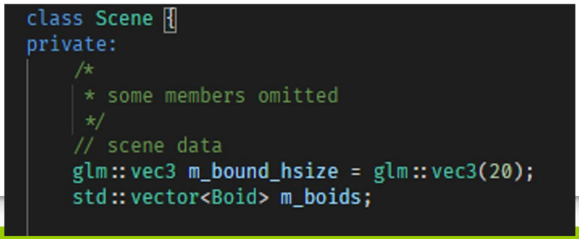


```
class Boid {
private:

    glm::vec3 m_position;
    glm::vec3 m_velocity;
    glm::vec3 m_acceleration;
```

- `Boid::calculateForces(Scene* scene)`
  - This is where the boids rules are implemented.
  - You'll iterate through every other Boid instance contained in the Scene object (`scene->m_boids`), and calculate their contribution to the force on this Boid by applying the 3 original Boids rules.
  - The 4th and 5th Boid rules only take into account the speed and velocity of the current Boid.
  - Store the result by updating the value of `this->m_acceleration` (do **not** modify `m_position` or `m_velocity` in this function)
- `Boid::update(float timestep)`
  - This is where we apply our physics simulation step
  - Update position based on velocity
  - Update velocity based on acceleration
  - Remember to take the timestep into account

scene.cpp



```
class Scene {
private:
    /*
     * some members omitted
     */
    // scene data
    glm::vec3 m_bound_hsize = glm::vec3(20);
    std::vector<Boid> m_boids;
```

- `Scene::loadCore()`
  - This is where you initialize the simulation
  - You'll need to create some number of Boids with random positions and velocities (about 15 should be good to start with, 100-300 once you've got it working) and add them to `this->m_boids`.
  - Make sure they're within the scene bounds.
- `Scene::loadCompletion()`
  - Same as above, but add predators.
- `Scene::loadChallenge()`
  - Same as above, but add obstacles.
  - Spheres are the simplest obstacle to implement.