VICTORIA UNIVERSITY WELLINGTON
TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

# 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, Marc Gluyas, ECS;**

# Our inspiration

- **National Geographic flight of the starlings**

  **https://www.youtube.com/watch?v=V4f_1_r80RY**

- **A murmuration of starlings**

  **https://www.youtube.com/watch?v=eakKfY5aHmY&t=75s**

https://bio.kuleuven.be/ento/ferrante/images/Flock-of-Starlings.jpg

# 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.

# Implementing Boids

# 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.

# 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

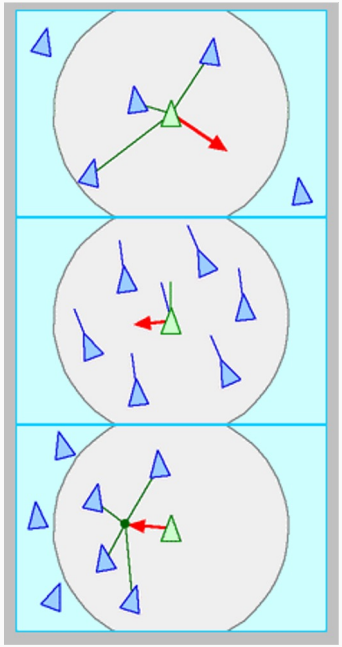
Every rule has several parameters, which you should make controllable with an ImGui interface. Most parameters fall into one of two categories:

- 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:

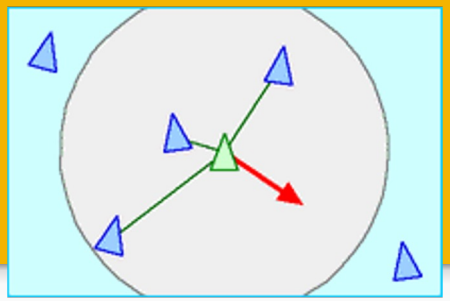
```
total_acceleration = avoidance_force * avoidance_weight
                   + cohesion_force  * cohesion_weight
                   + alignment_force * alignment_weight
```

  - In Zohar's lecture 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.

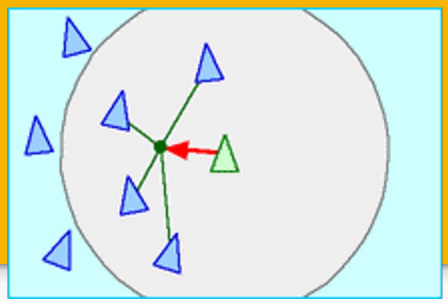
```
avoidance = vec3(0)
foreach other_boid in boids:
  if boid is other_boid: continue // be careful not to copy value when comparing ptrs in C++

  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.

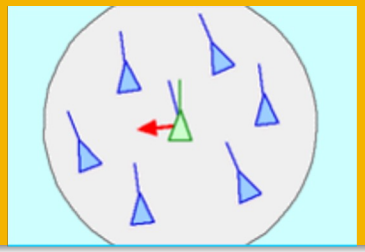
```
average_position = vec3(0)
neighbours = 1

// 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.

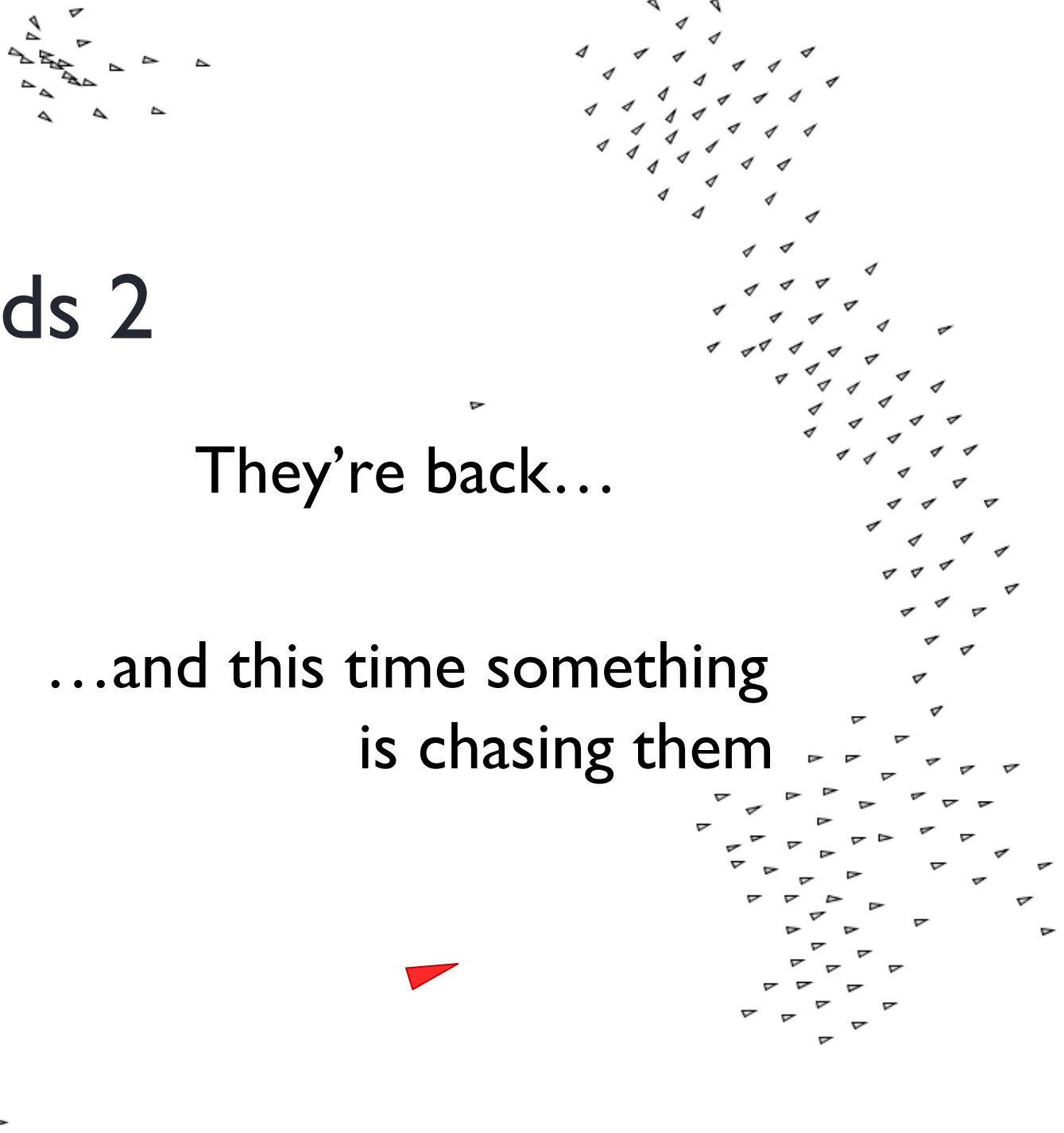
The size (in each dimension) of the AABB is given by the difference of its two bounds, which we use as the interval size.

```
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`.

# Boids 2



They’re back…

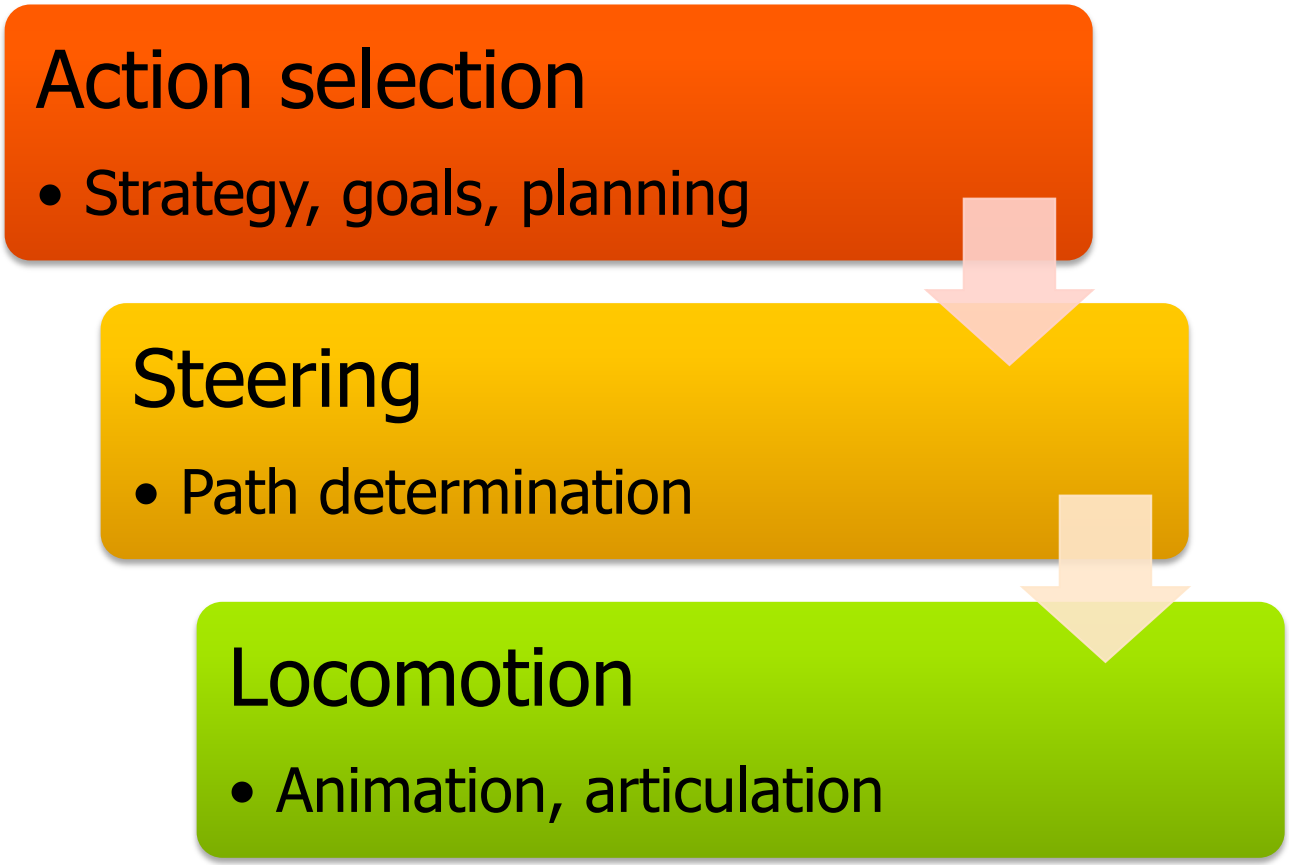
…and this time something is chasing them

- Goal seeking
- Goal avoidance
- Predator behaviour
- Collision detection

# Autonomous agents

- Behaviour levels

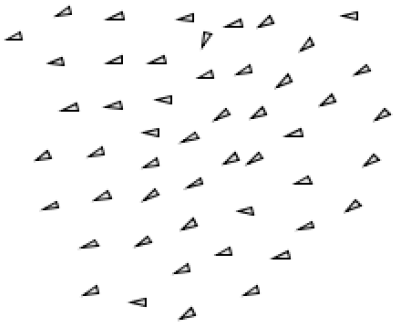
http://www.red3d.com/cwr/steer/gdc99/

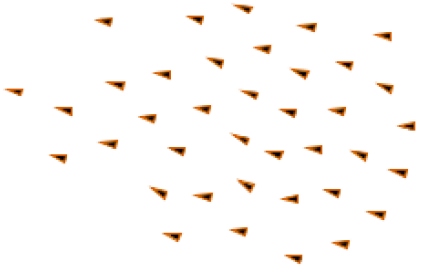


- Action selection: Keep near the flock, don't hit flockmates, align, hunt
- Locomotion: A bird flipping its wings.
- Our boids algorithm is all about the steering part (determine the speed vector).
- Example: A cowboy retrieving a wondering cow

# Boids of a feather flock together

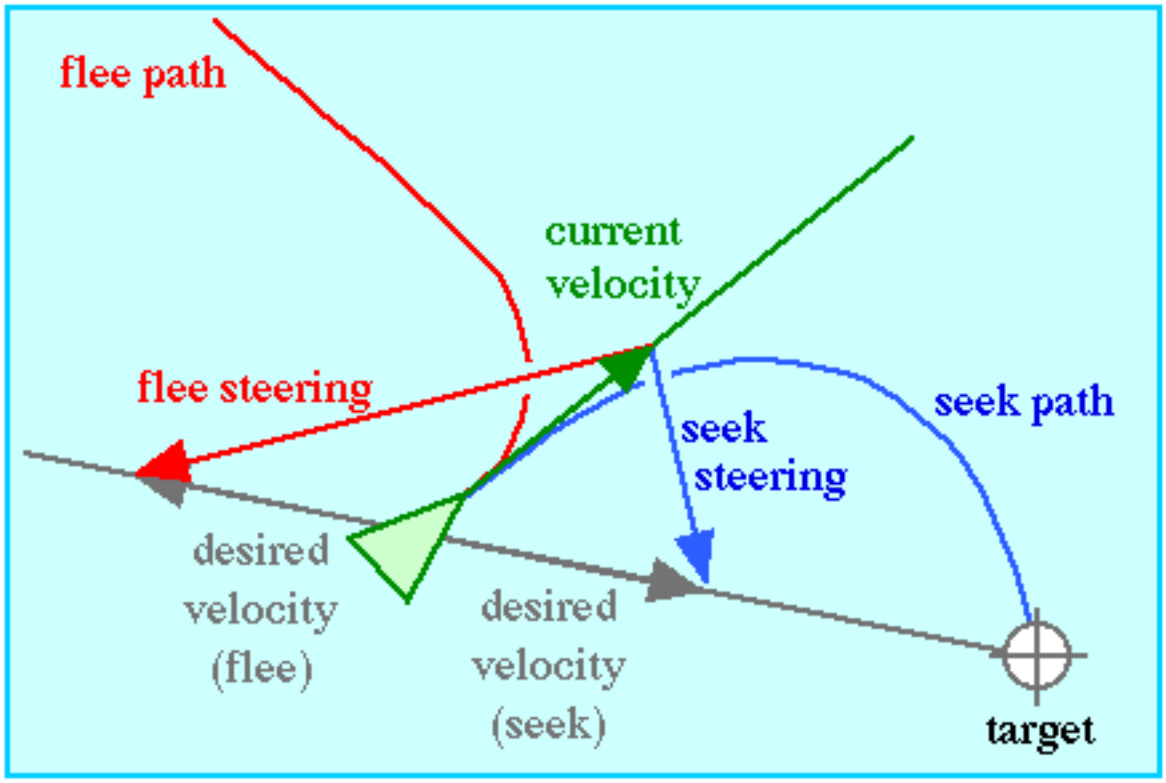
- Can have multiple flocks
  - *Avoid* all boids
  - *Cohere* and *align* only with flock mates





# Target seeking & target avoidance

- We can add a force that steers boids towards or away from a target



- Adding desired velocity instead of the seek velocity would result in orbital movement
- Apply force only if the target is within range
- Slow down desired velocity close to the target

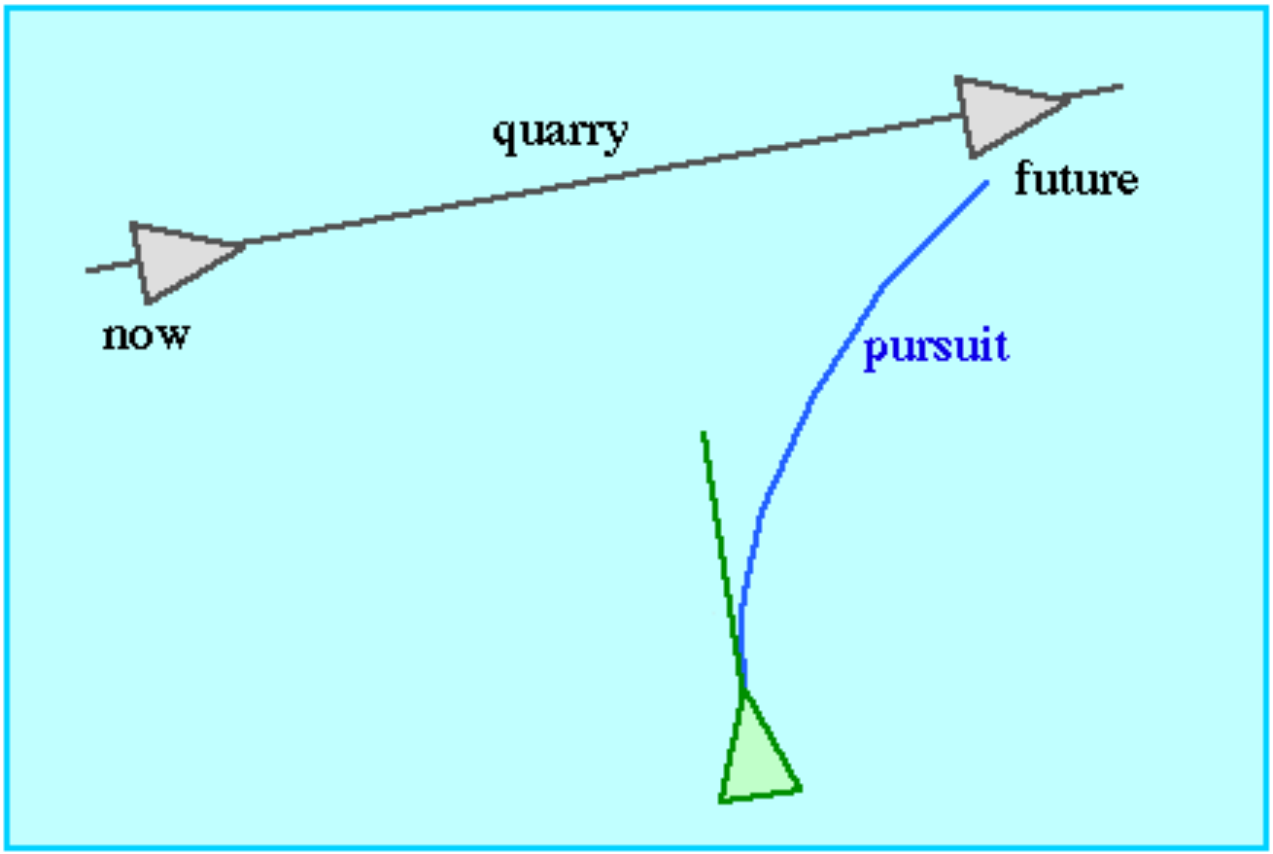
# Predators

- Predator is a type of boid
- Does not flock
- *Seeks* its prey boid
- Other boids *fl*ee from the predator
- Flocking behavior confuses the predator
- Focuses on a single boid
- Must be faster

**https://www.youtube.com/watch?v=V-mCuFYfJdI**

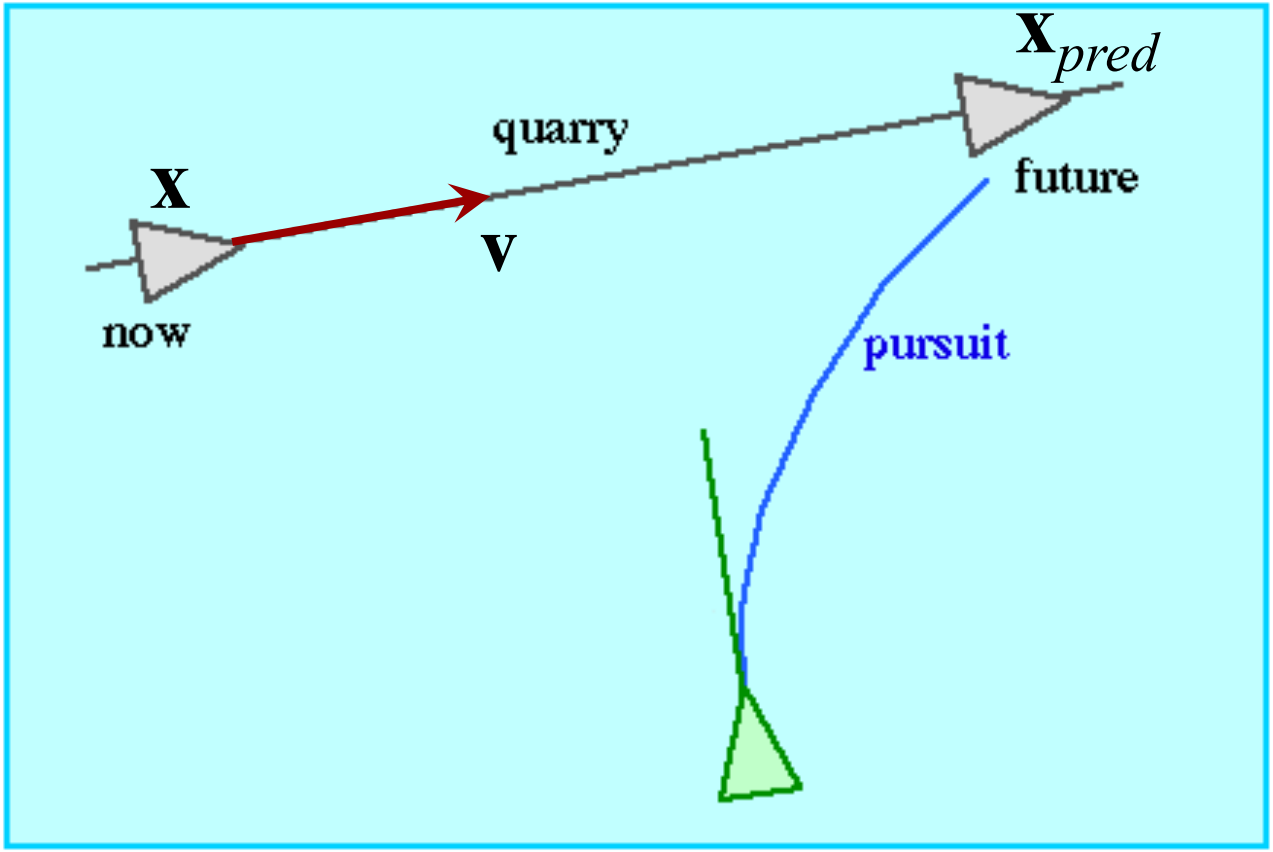
# Predictive pursuit

- Predict where the prey will be in the future
- *Seek* that predicted position



# How do I make a prediction?

- Prey position **x**, prey velocity **v**
- Predicted position, $\mathbf{x}_{pred} = \mathbf{x} + (\mathbf{v} \times T)$

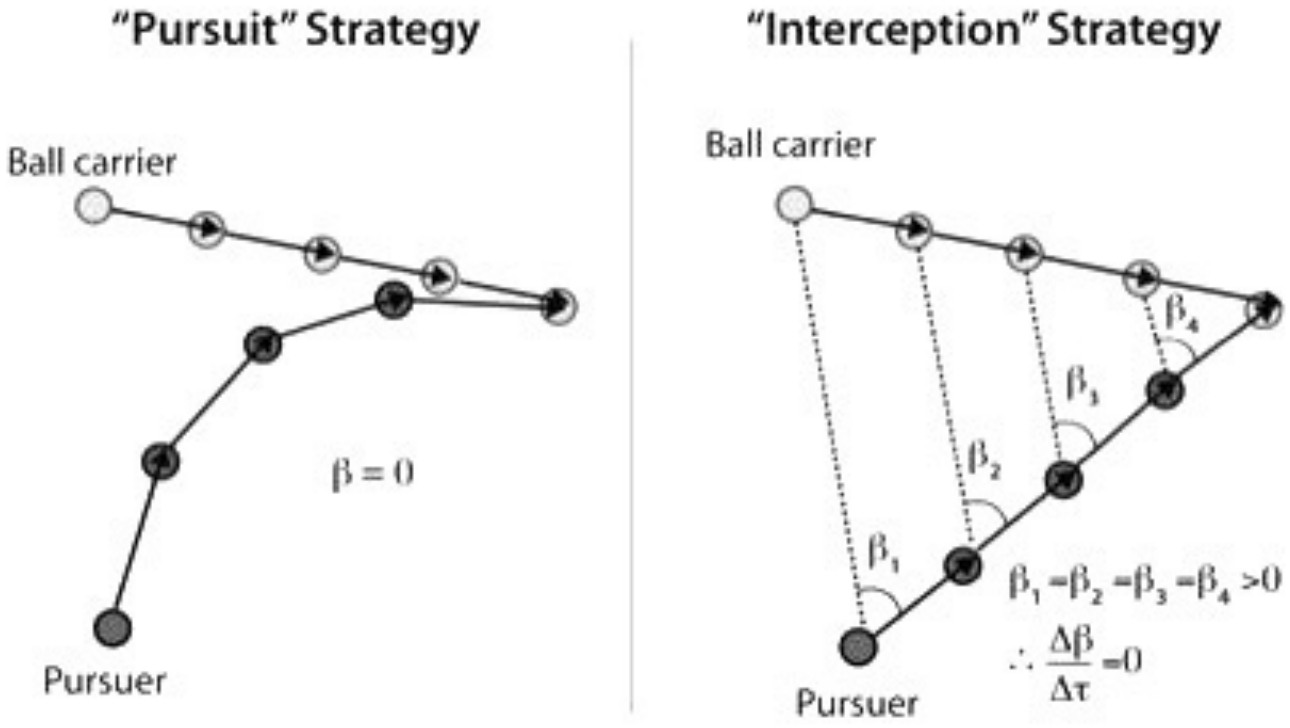


# How to choose T

- Ideally: The time it would take to catch the prey
- Assumed to be constant (better than nothing)
- Bigger when further away, smaller when close. Proportional to the distance: $T=cD$

# Interception strategy

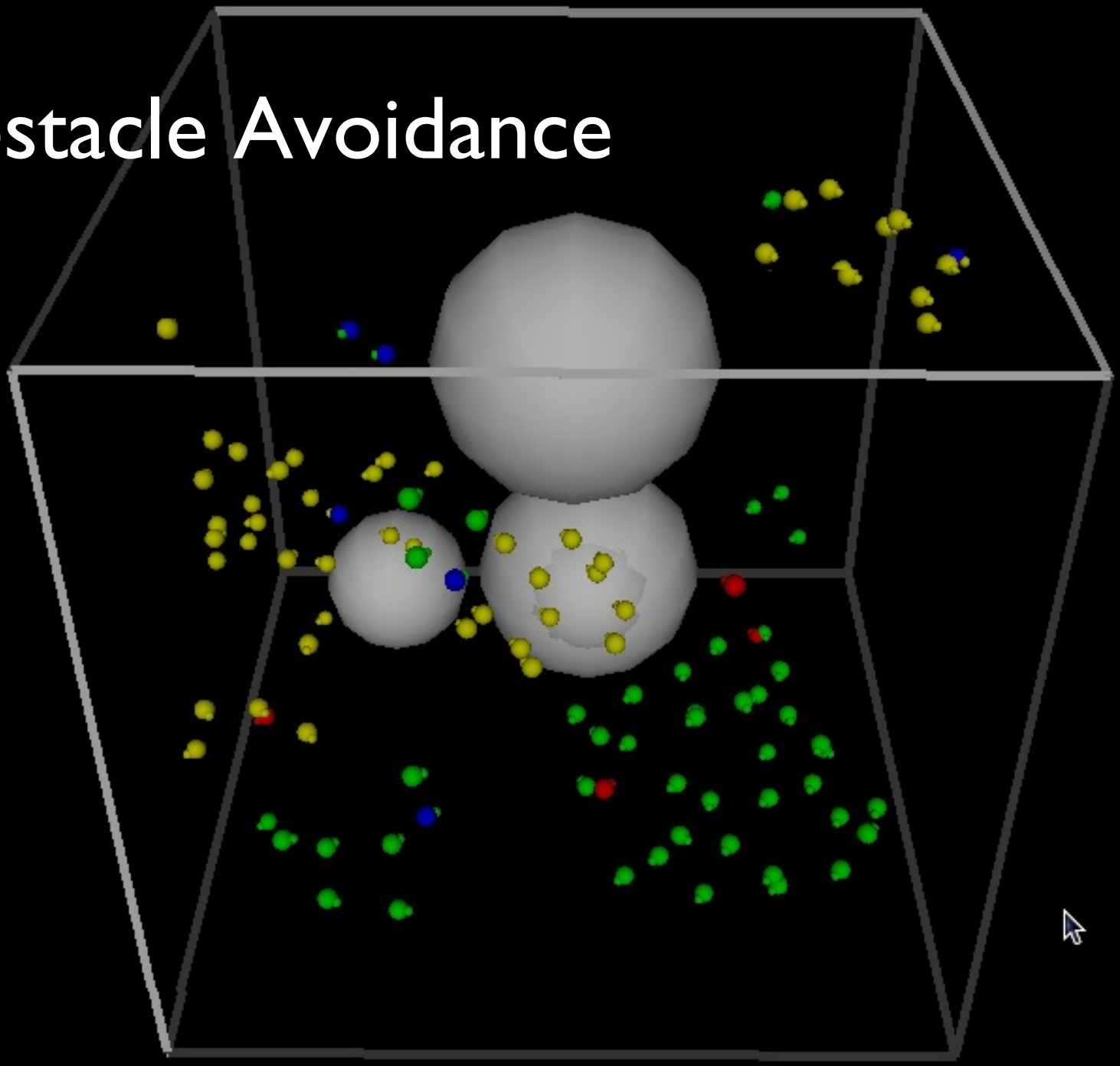
- The interception strategy used by several real predators and professional sportspeople



# Interception strategy

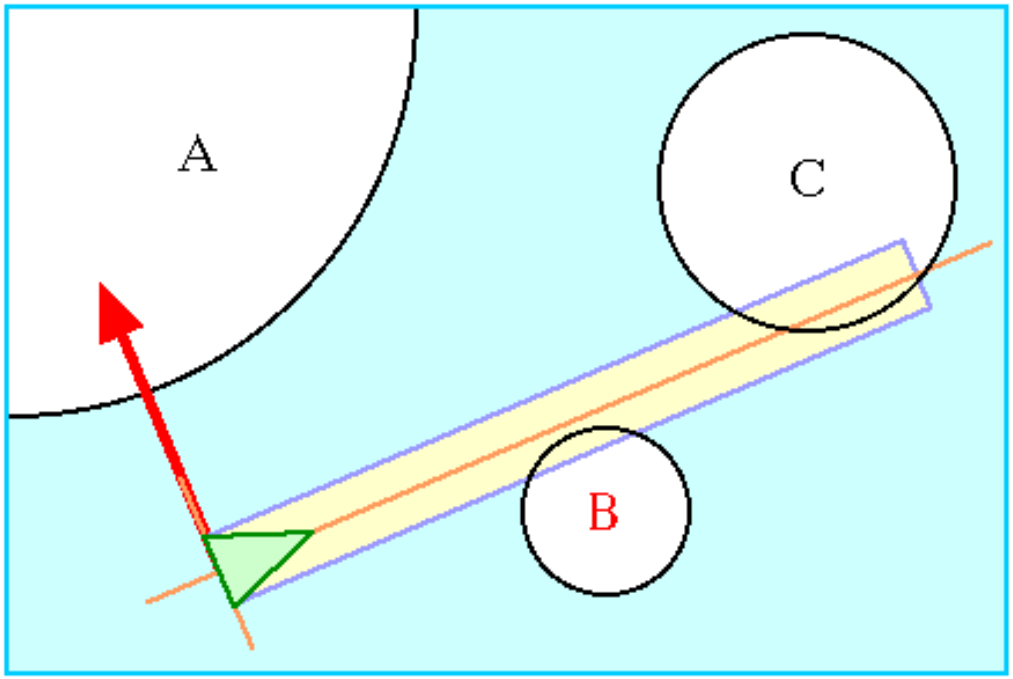
- Keeping a constant angle
- Examples:
  - Intercepting a ball-carrier in football https://www.youtube.com/watch?v=yE2_opbWmTA
  - Catching a ball in baseball or cricket
  - Dragonfly catching small insects
- Same as making a perfect prediction of T as time of interception

# Obstacle Avoidance



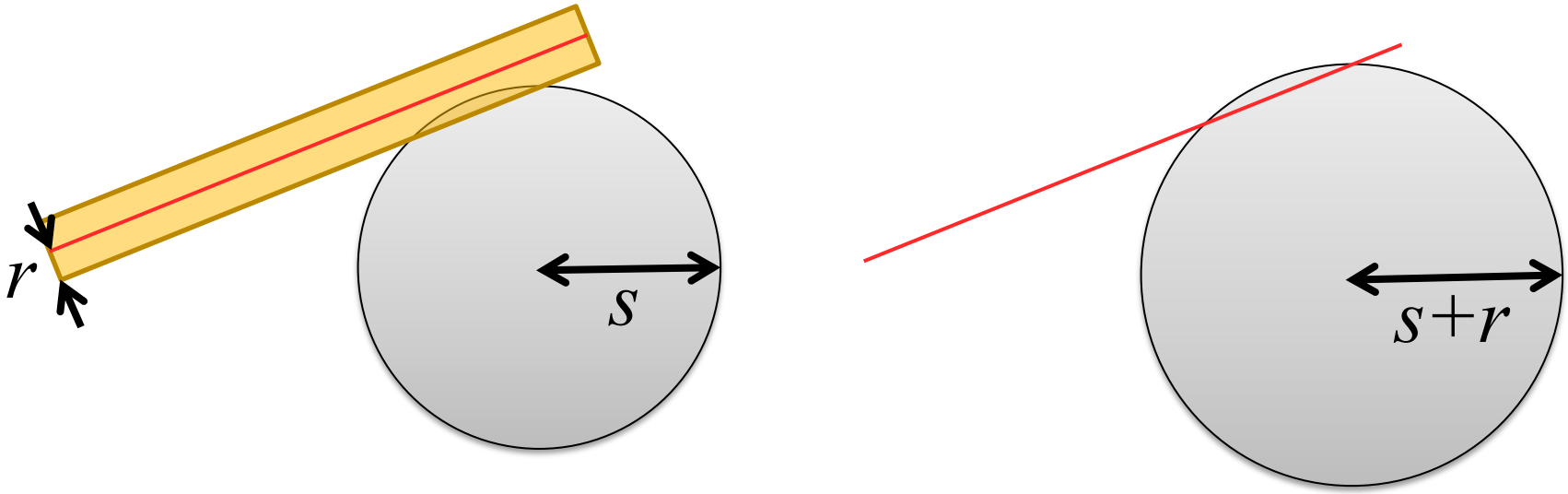
# Avoiding large spheres

- Spheres are easy. Approximate a shape with spheres.
- Project a cylinder forward from the boid
- If it intersects a sphere steer to avoid



# Intersect a cylinder and a sphere?

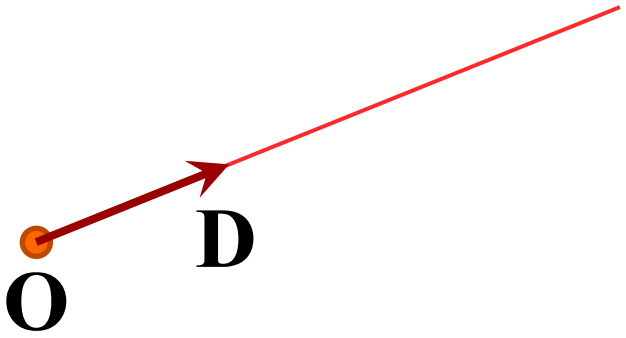
- You want to intersect a cylinder of radius $r$ with a sphere of radius $s$
- Equivalent to intersecting a line with a sphere of radius $R=s+r$



# Intersect a line and a sphere?

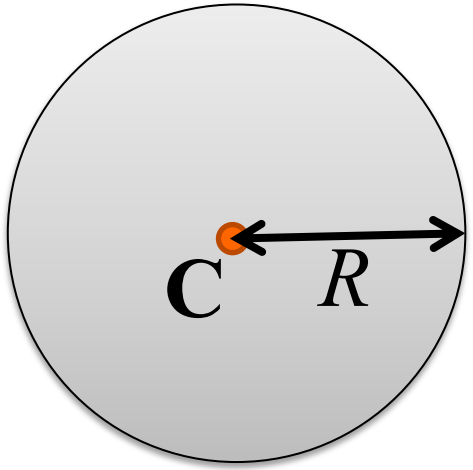
- Equation of a line

$$\mathbf{P}(t) = \mathbf{O} + t\mathbf{D}$$




- Equation of a sphere

$$\left|\mathbf{P} - \mathbf{C}\right| = R$$



# Intersect a line and a sphere

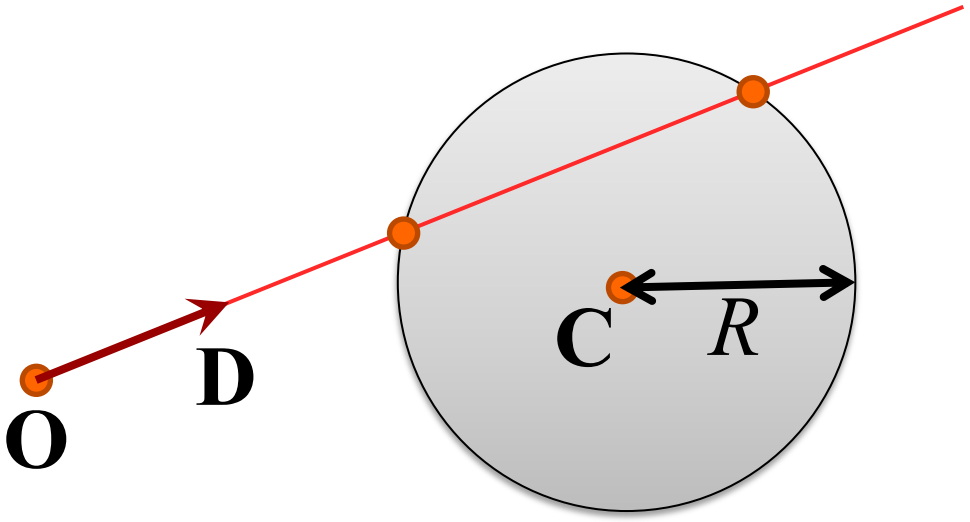
$$|\mathbf{P} - \mathbf{C}| = R$$ ▪ Equation of sphere

$$|\mathbf{P}(t) - \mathbf{C}| = R$$ ▪ Substitute with line equation

$$|\mathbf{O} + t\mathbf{D} - \mathbf{C}| = R$$ ▪ Expand out line equation

We know $R$, $\mathbf{O}$, $\mathbf{D}$, and $\mathbf{C}$

Solve this equation to find $t$



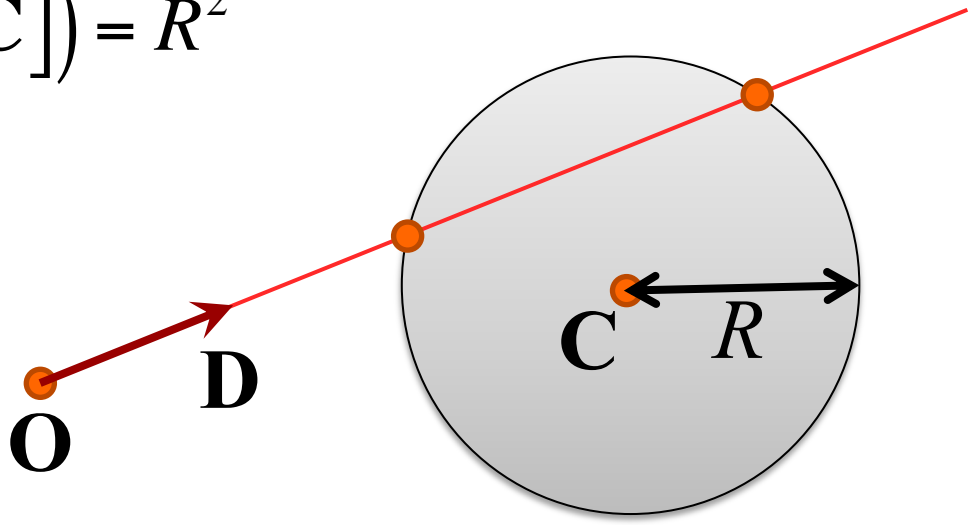
# Intersect a line and a sphere

$$\left|\mathbf{O}+t\mathbf{D}-\mathbf{C}\right| = R$$

$$\sqrt{\left(\mathbf{O}+t\mathbf{D}-\mathbf{C}\right)\bullet\left(\mathbf{O}+t\mathbf{D}-\mathbf{C}\right)} = R$$

$$\left(\mathbf{O}+t\mathbf{D}-\mathbf{C}\right)\bullet\left(\mathbf{O}+t\mathbf{D}-\mathbf{C}\right) = R^2$$

$$\left(t\mathbf{D}+\left[\mathbf{O}-\mathbf{C}\right]\right)\bullet\left(t\mathbf{D}+\left[\mathbf{O}-\mathbf{C}\right]\right) = R^2$$

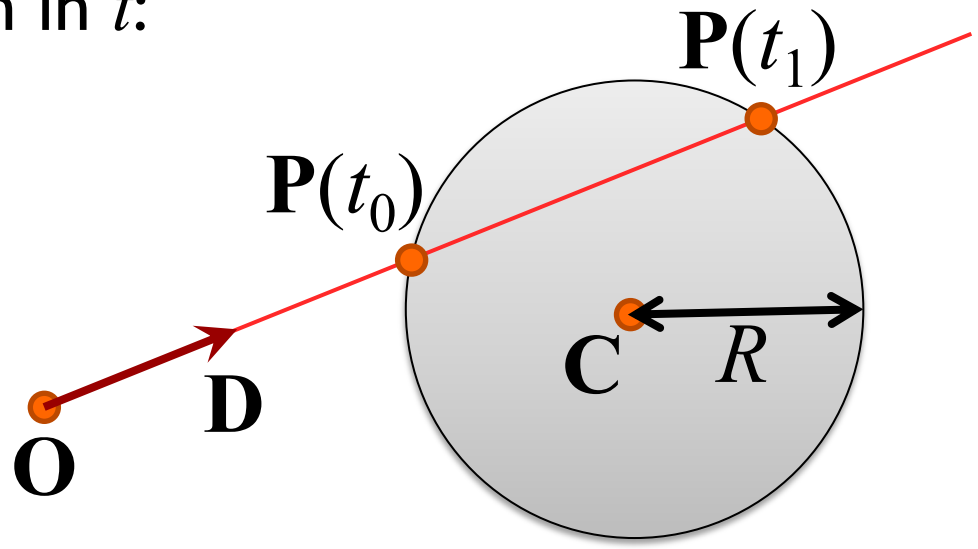


# Intersect a line and a sphere

$$\left(t\mathbf{D}+\left[\mathbf{O}-\mathbf{C}\right]\right)\bullet\left(t\mathbf{D}+\left[\mathbf{O}-\mathbf{C}\right]\right)=R^2$$

$$t^2\left(\mathbf{D}\bullet\mathbf{D}\right)+2t\left(\left[\mathbf{O}-\mathbf{C}\right]\bullet\mathbf{D}\right)+\left(\left[\mathbf{O}-\mathbf{C}\right]\bullet\left[\mathbf{O}-\mathbf{C}\right]\right)=R^2$$

That is a quadratic equation in $t$:
so there are two solutions

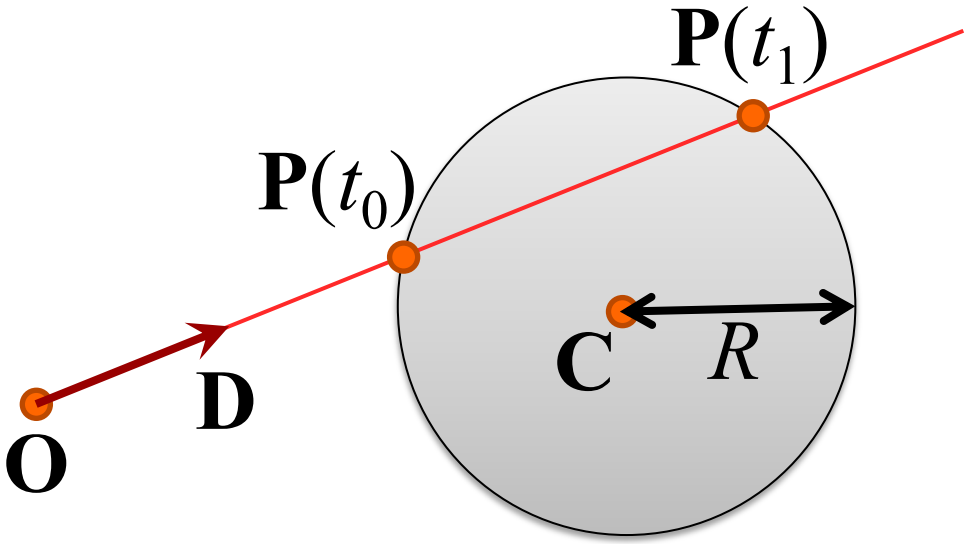


# Solve that quadratic equation

$$t^2(\mathbf{D}\bullet\mathbf{D})+2t([\mathbf{O}-\mathbf{C}]\bullet\mathbf{D})+([\mathbf{O}-\mathbf{C}]\bullet[\mathbf{O}-\mathbf{C}])=R^2$$

$$t^2(\mathbf{D}\bullet\mathbf{D})+t(2[\mathbf{O}-\mathbf{C}]\bullet\mathbf{D})+([\mathbf{O}-\mathbf{C}]\bullet[\mathbf{O}-\mathbf{C}]-R^2)=0$$

$$at^2+bt+c=0$$



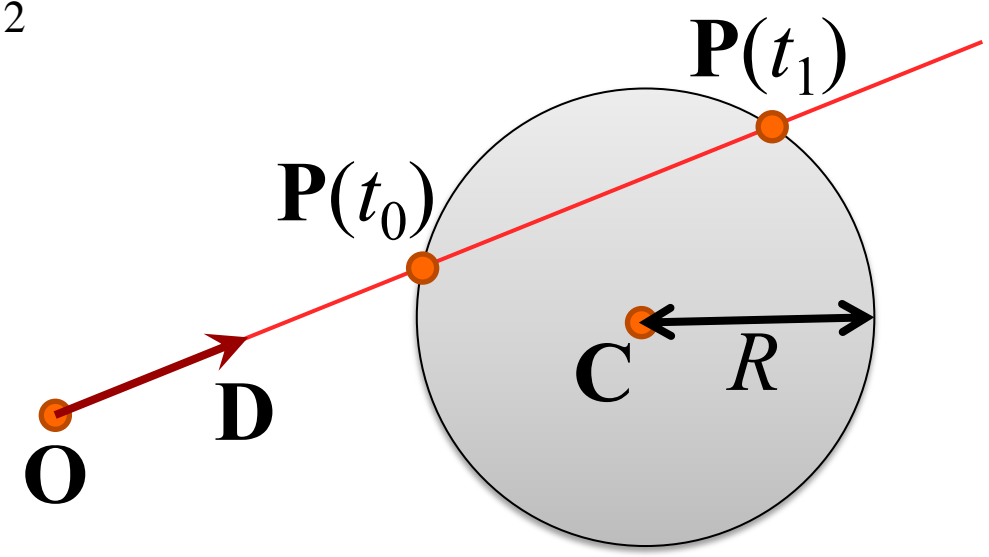
# Solve that quadratic equation

$$at^2 + bt + c = 0$$

$$a = \mathbf{D} \bullet \mathbf{D}$$

$$b = 2\left[\mathbf{O} - \mathbf{C}\right] \bullet \mathbf{D}$$

$$c = \left[\mathbf{O} - \mathbf{C}\right] \bullet \left[\mathbf{O} - \mathbf{C}\right] - R^2$$



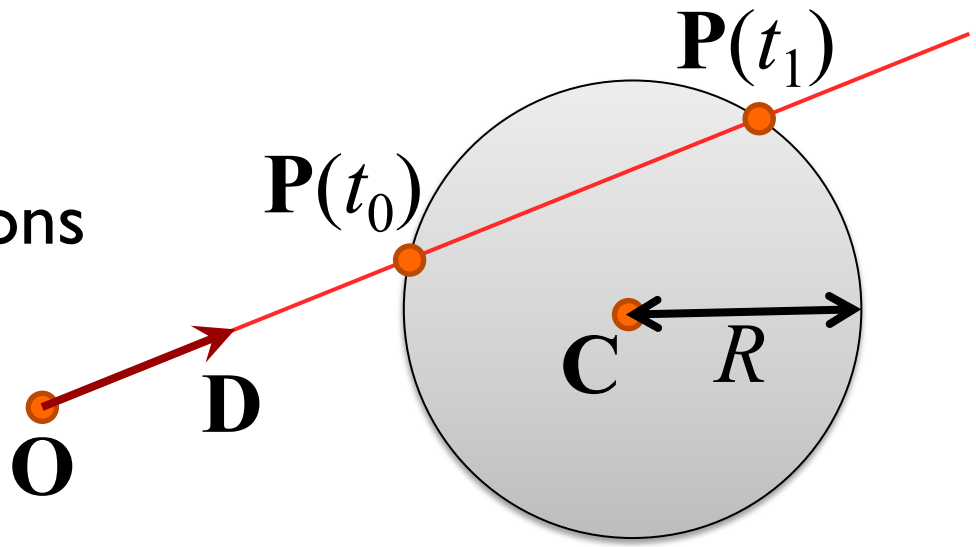
# Solve that quadratic equation

$$at^2 + bt + c = 0$$

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$b^2–4ac$ is the discriminant

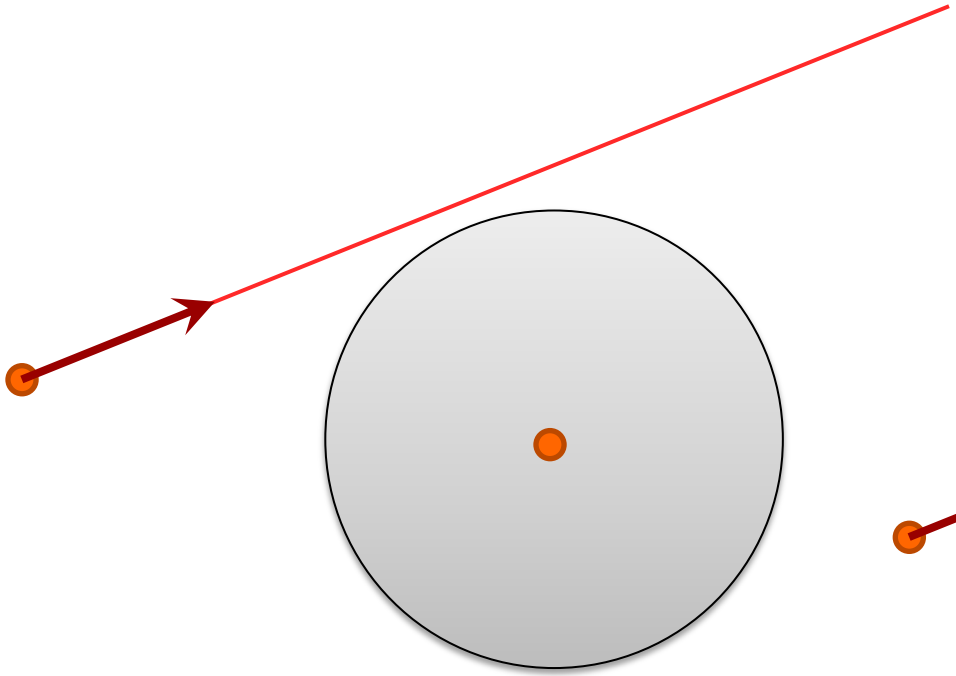
$b^2–4ac < 0$ : no real solutions



# Does the line intersect the sphere?

$b^2 - 4ac < 0$

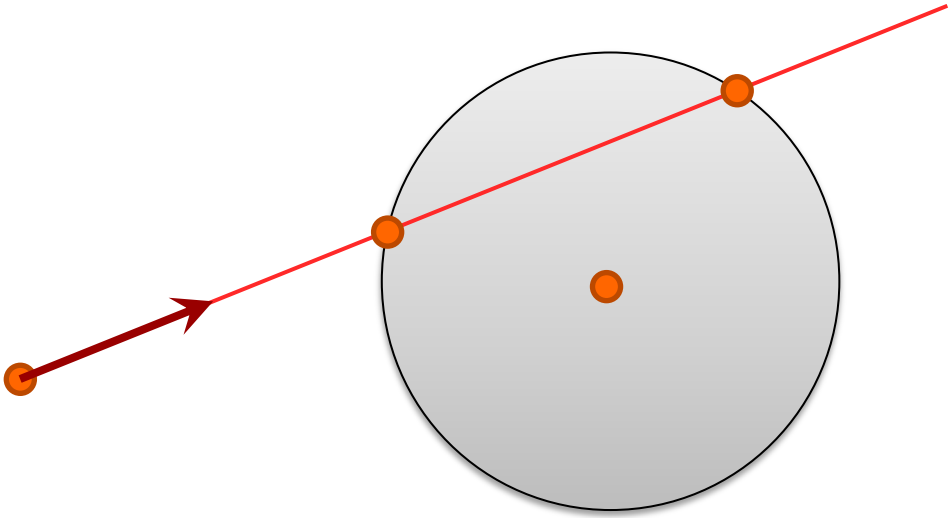
no real solutions



**no intersection**

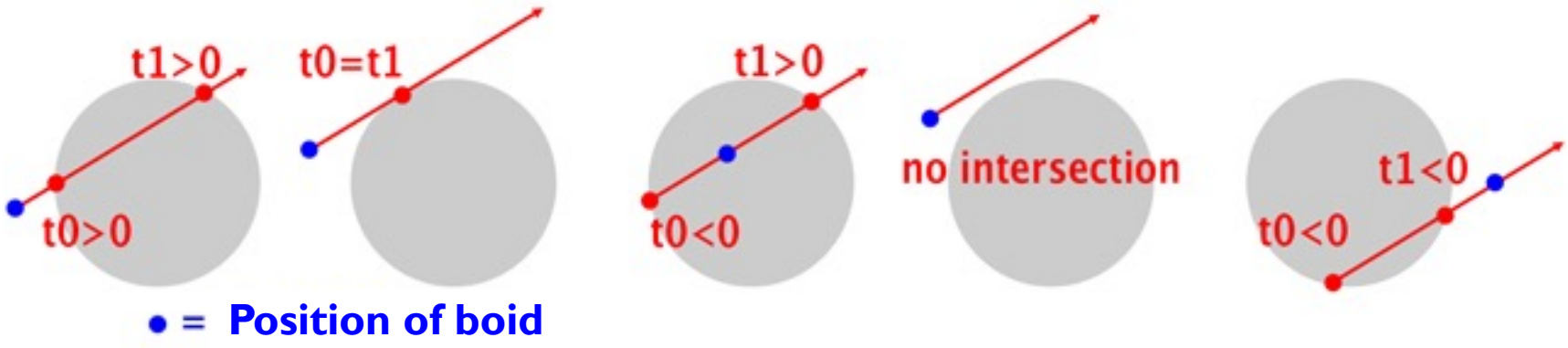
$b^2 - 4ac \geq 0$

two real solutions



**intersection**

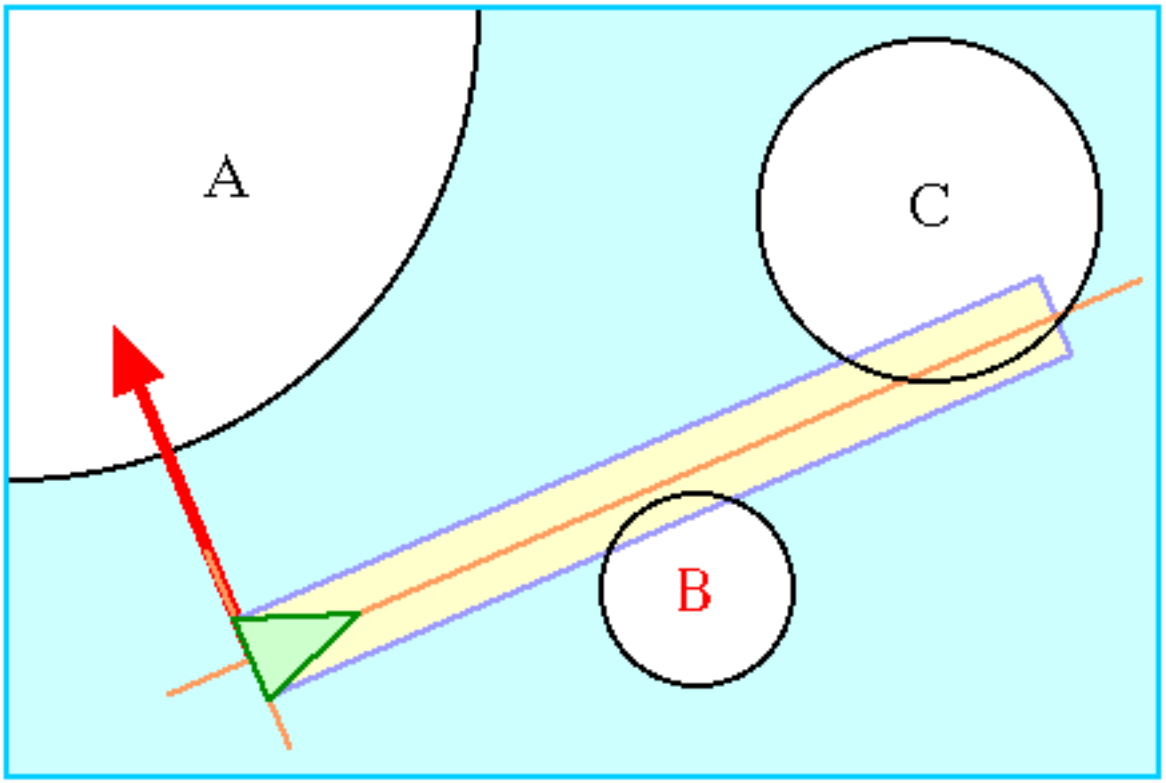
# But wait, there's more to consider…




- Boid is going to hit sphere
- Boid is going to just skim sphere
- Boid is inside sphere
- Boid is going to miss sphere
- Sphere is behind boid

# Which sphere will we hit first?

- We find the smallest positive value of $t$
- Which tells us which sphere is first in our path

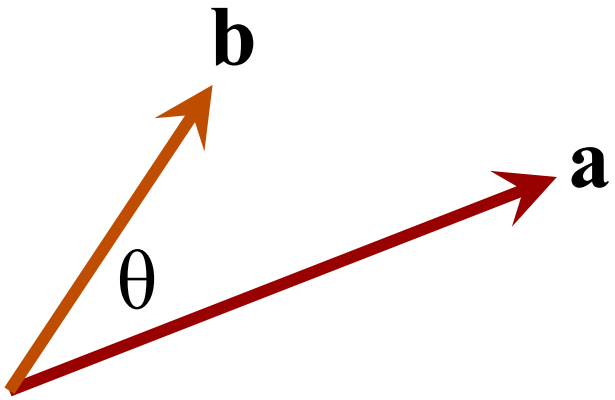


# The dot product is *very* useful

- Definitions

$$\mathbf{a} \bullet \mathbf{b} = x_a x_b + y_a y_b + z_a z_b$$

$$\mathbf{a} \bullet \mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos\theta$$

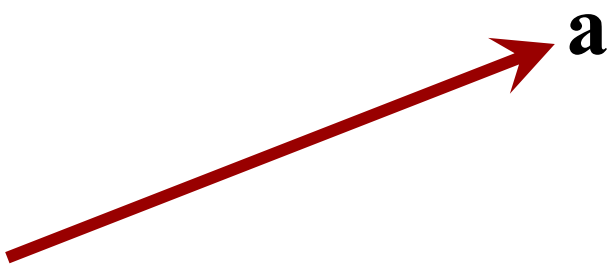


# Dot product gives vector length

$$\mathbf{a} \bullet \mathbf{a} = |\mathbf{a}||\mathbf{a}|\cos 0^\circ$$

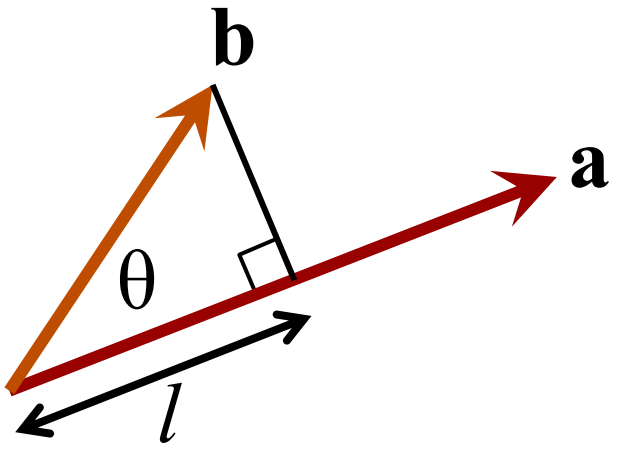
$$= |\mathbf{a}|^2$$

$$= x_a^2 + y_a^2 + z_a^2$$



# Dot product gives projection length

$l$ is the length of the projection of vector $\mathbf{b}$ onto vector $\mathbf{a}$



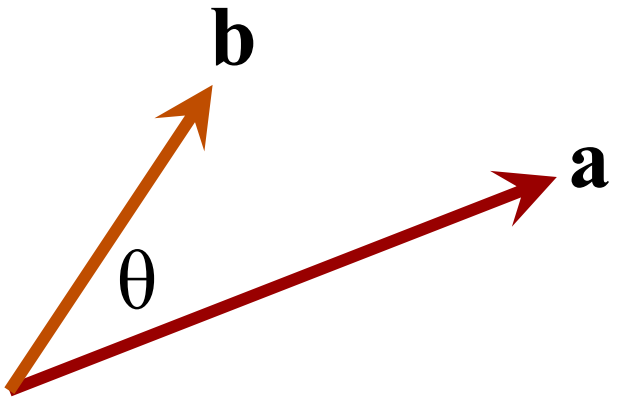

$$l = |\mathbf{b}|\cos\theta$$

$$\mathbf{a} \bullet \mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos\theta$$

$$\text{so } l = \frac{\mathbf{a} \bullet \mathbf{b}}{|\mathbf{a}|}$$

# Dot product gives angle

$$\mathbf{a} \bullet \mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos\theta$$



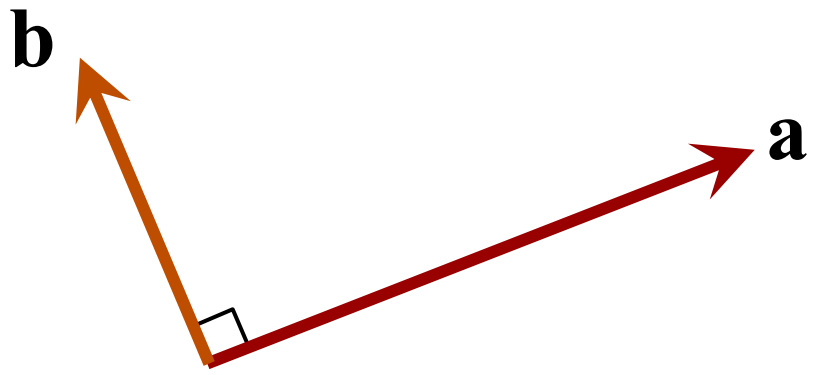

$$\text{so} \quad \cos\theta = \frac{\mathbf{a} \bullet \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}$$

$$\text{so} \qquad \theta = \cos^{-1}\left(\frac{\mathbf{a} \bullet \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}\right)$$

# Are vectors at right angles?

- If the vectors are at right angles then

$$\mathbf{a} \bullet \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos 90^\circ$$
$$= 0$$



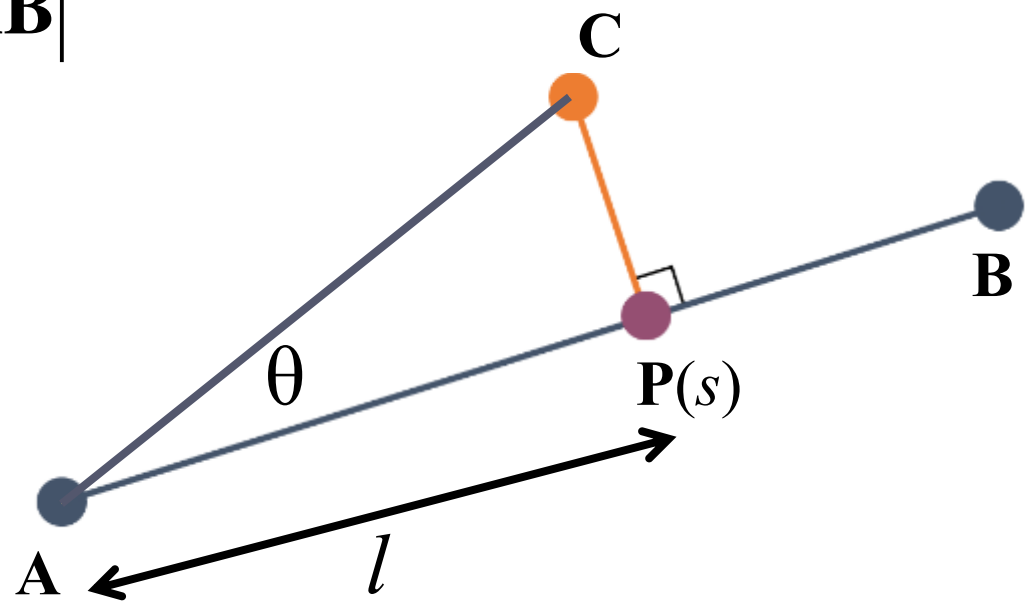
# Closest point on a line

- $\mathbf{P}(s)$ is the point on line $\mathbf{AB}$ that is closest to point $\mathbf{C}$

$$l = |\mathbf{AC}| \cos\theta = \frac{\mathbf{AC} \bullet \mathbf{AB}}{|\mathbf{AB}|}$$

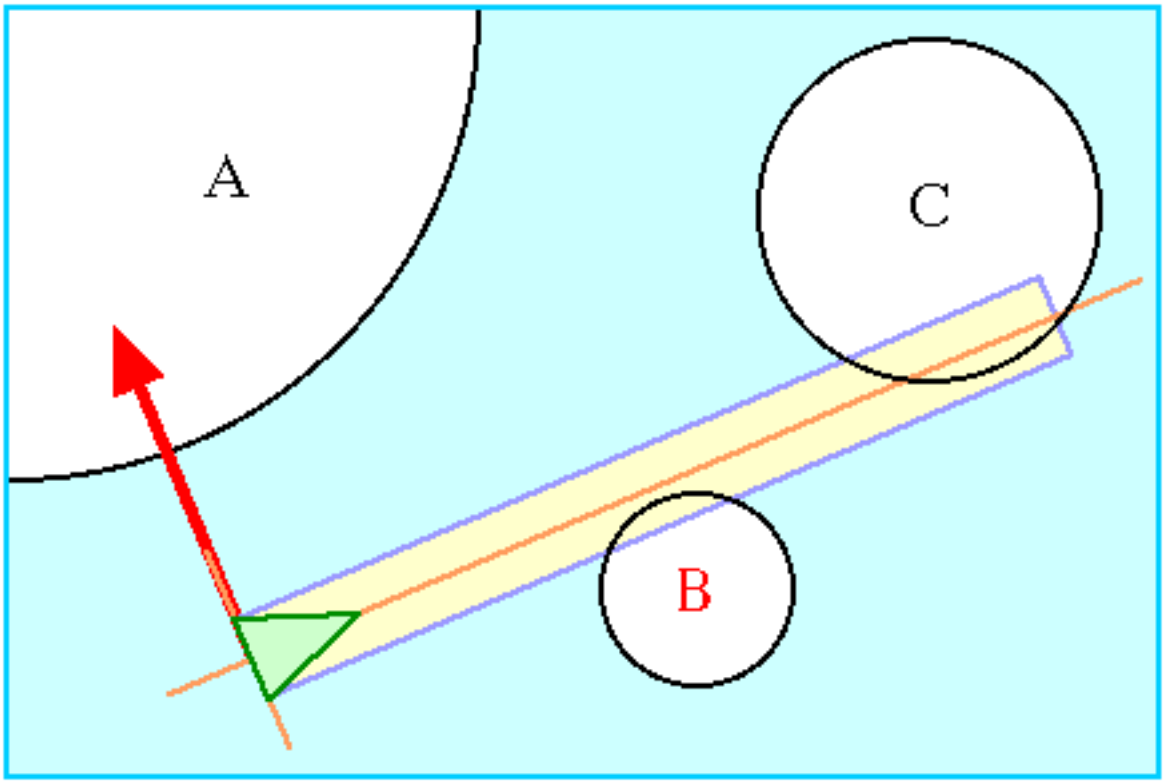
$$s = \frac{l}{|\mathbf{AB}|}$$

$$\mathbf{P}(s) = (1 - s)\mathbf{A} + s\mathbf{B}$$



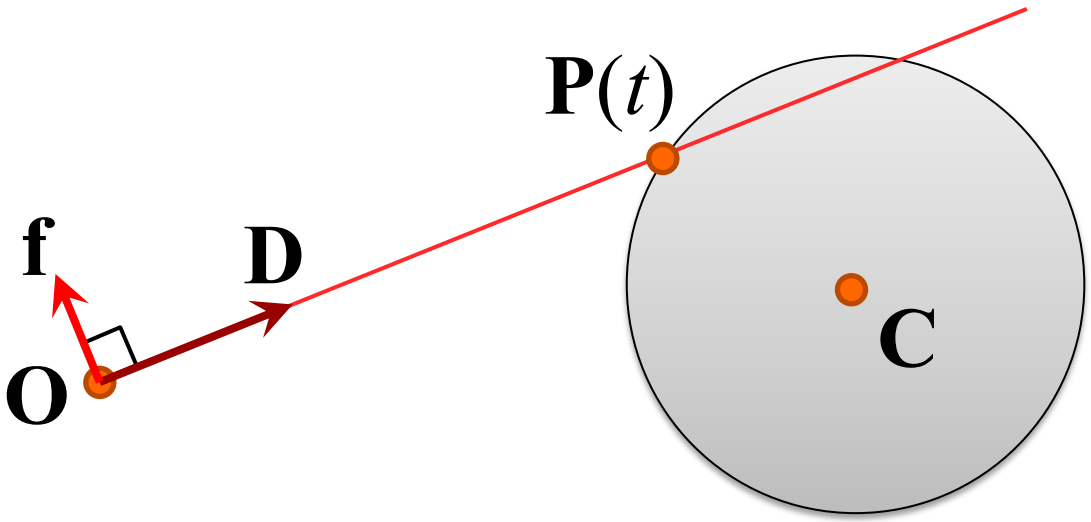
# How do we calculate that force?

- The force must push us away from the sphere that we are going to hit



# What do we know?

- We know **O**, **D**, **C**, $t$, **P**($t$)
- We want **f**

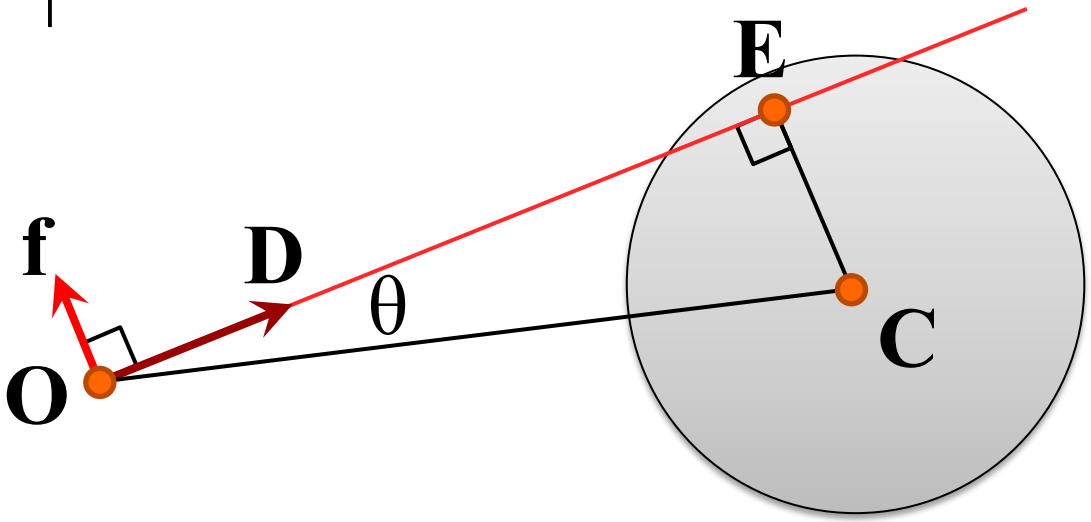


# Getting a vector at right angles to $\mathbf{D}$

$$|\mathbf{OE}| = |\mathbf{OC}|\cos\theta = \frac{\mathbf{OC} \bullet \mathbf{D}}{|\mathbf{D}|}$$

$$\mathbf{E} = \mathbf{O} + |\mathbf{OE}|\frac{\mathbf{D}}{|\mathbf{D}|}$$

$$\mathbf{f} = \frac{\mathbf{CE}}{|\mathbf{CE}|}$$




- Our force $\mathbf{f}$ is a vector that points in the same direction as $\mathbf{CE}$

# Prioritizing forces

- Some forces take priority over others, e.g. avoiding a sphere vs responding to flockmates.
- Have a "force budget"
  - You only apply lower priority forces if there is budget left over from higher priority forces