CMP 327: Introduction to Video Game Design and Development

Dr. Arun K. Kulshreshth

Fall 2020

Lecture 10: Steering and Flocking

*Based on Mat Buckland's Book and Craig Reynolds Paper

Autonomous Character/Agent

- AI Characters
- Represent a character in a story or game and have some ability to improvise their actions.
- Contrast to
  - A character in an animated film whose actions are scripted
  - An "Avatar"/character/player controller by a human being

Autonomous Agent Movement

- The movement of an autonomous agent can be broken down into three layers:

1. Action Selection
2. Steering
3. Locomotion

Action Selection

- This is the part of the agent's behavior responsible for
  - Choosing its goals
  - Deciding what plan to follow
- Examples:
  - "go here" and "do A, B, and then C.“
  - “Go to player position” and then “attack”

Steering

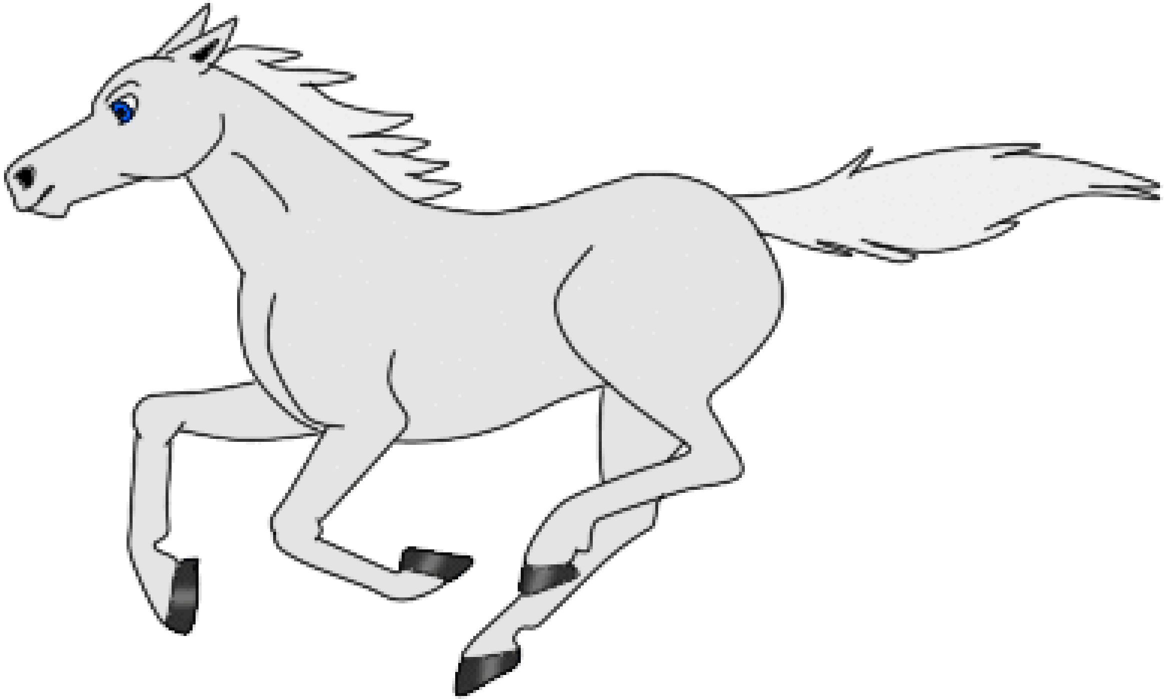
- This layer is responsible for calculating the desired trajectories required to satisfy the goals and plans set by the action selection layer.
- Steering behaviors are the implementation of this layer.
- They produce a steering force that describes where an agent should move and how fast it should travel to get there.

Locomotion

- The bottom layer, locomotion, represents the more mechanical aspects of an agent's movement.
- It is the how of traveling from A to B.
- For example, if you had implemented the mechanics of a camel, a tank, and a goldfish
  - a command for them to travel north
  - they would all use different mechanical processes to create motion even though their intent (to move north) is identical.

Locomotion of a Tank

Locomotion of a Horse



Locomotion of a Goldfish

Locomotion

- By separating this layer from the steering layer, it's possible to utilize, with little modification, the same steering behaviors for completely different types of locomotion.

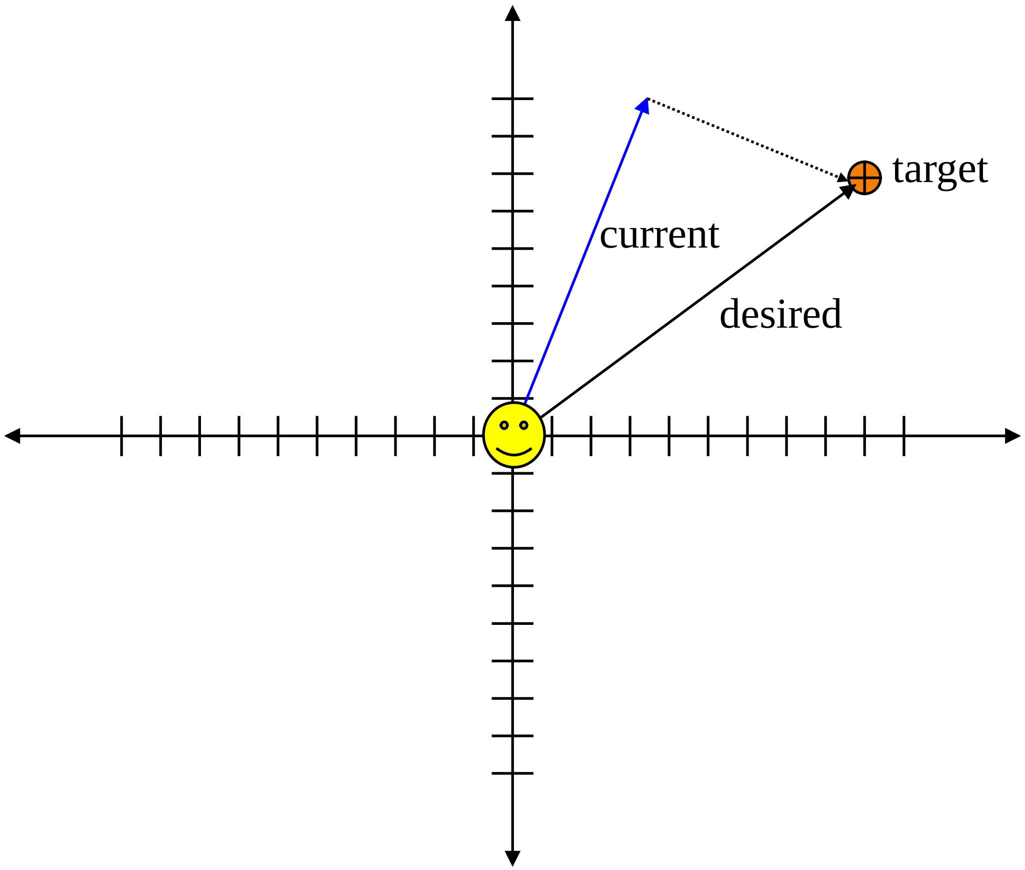
Steering

- Simple Waypoint Steering
  - Seek, Pursuit, Arrive
- Evasive Steering
  - Flee, Evade, Hide
- Advanced Waypoint Steering
  - Interpose, Offset pursuit
  - Collision Avoidance
  - Path Following
  - Wander

Seek

- Direct our AI/player toward a target position
- It will keep moving towards the target position
- If a character continues to seek, it will eventually pass through the target, and then turn back to approach again.

Seek



Move Target Using Mouse Clicks

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MoveTarget : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            UpdateTargetPosition();
        }
    }

    void UpdateTargetPosition()
    {
        // Don’t forget to add a plane on the ground in the Unity Scene
        if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition), out RaycastHit hit, 100))
        {
            transform.position = hit.point;
        }
    }
}
```

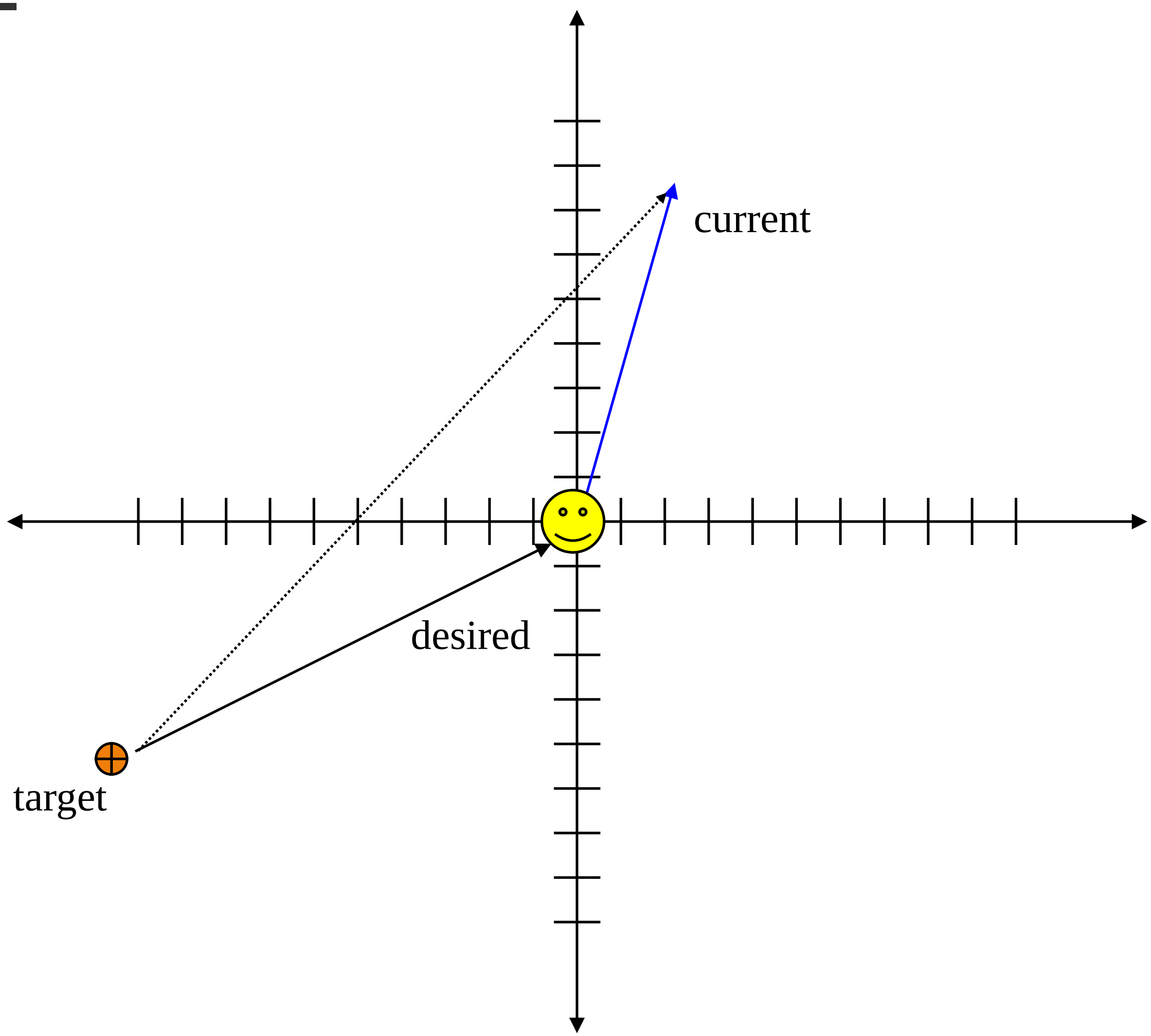
Unity 3D Implementation Details

```
public class Seek : MonoBehaviour {

    public Transform target;
    public float moveSpeed = 6.0f;
    private float minDistance = 0.2f;

    // Update is called once per frame
    void Update () {
                //Call to the fucntion every frame
                SeekTarget();
   }

   void SeekTarget()
    {
        //Subtracting two vectors by each will result in the desired direction
        Vector3 dir = target.position - transform.position;

        //Simple check to see if we continue seeking the target or if we are already at our desired distance from target
        if (dir.magnitude > minDistance)
        {
            Vector3 moveVector = dir.normalized * moveSpeed * Time.deltaTime;

            //If the case check true, we continue moving towards our target
            transform.position += moveVector;
        }
    }
}
```

Flee

- Run away from a given position
- Opposite of seek

Flee



Unity 3D Implementation Details

```
public class Seek : MonoBehaviour {

        public Transform target;
        public float moveSpeed = 6.0f;
        private float maxDistance = 15.0f;

// Update is called once per frame
void Update () {
       //Call to the fucntion every frame
       FleeTarget();
   }

void FleeTarget()
   {
       //Subtracting two vectors by each will result in the desired direction
       Vector3 dir = target.position - transform.position;

       //Simple check to see if we continue seeking the target or if we are already at our desired distance from target
       if (dir.magnitude < maxDistance)
       {
          Vector3 moveVector = dir.normalized * moveSpeed * Time.deltaTime;

          //If the case check true, we continue moving towards our target
          transform.position -= moveVector;
       }
   }
}
```

Arrive

- Arrive allows you to seek and come to a slower arrival (based on deceleration parameter)

Arrive

- Arrive allows you to seek and come to a slower arrival (based on deceleration parameter)

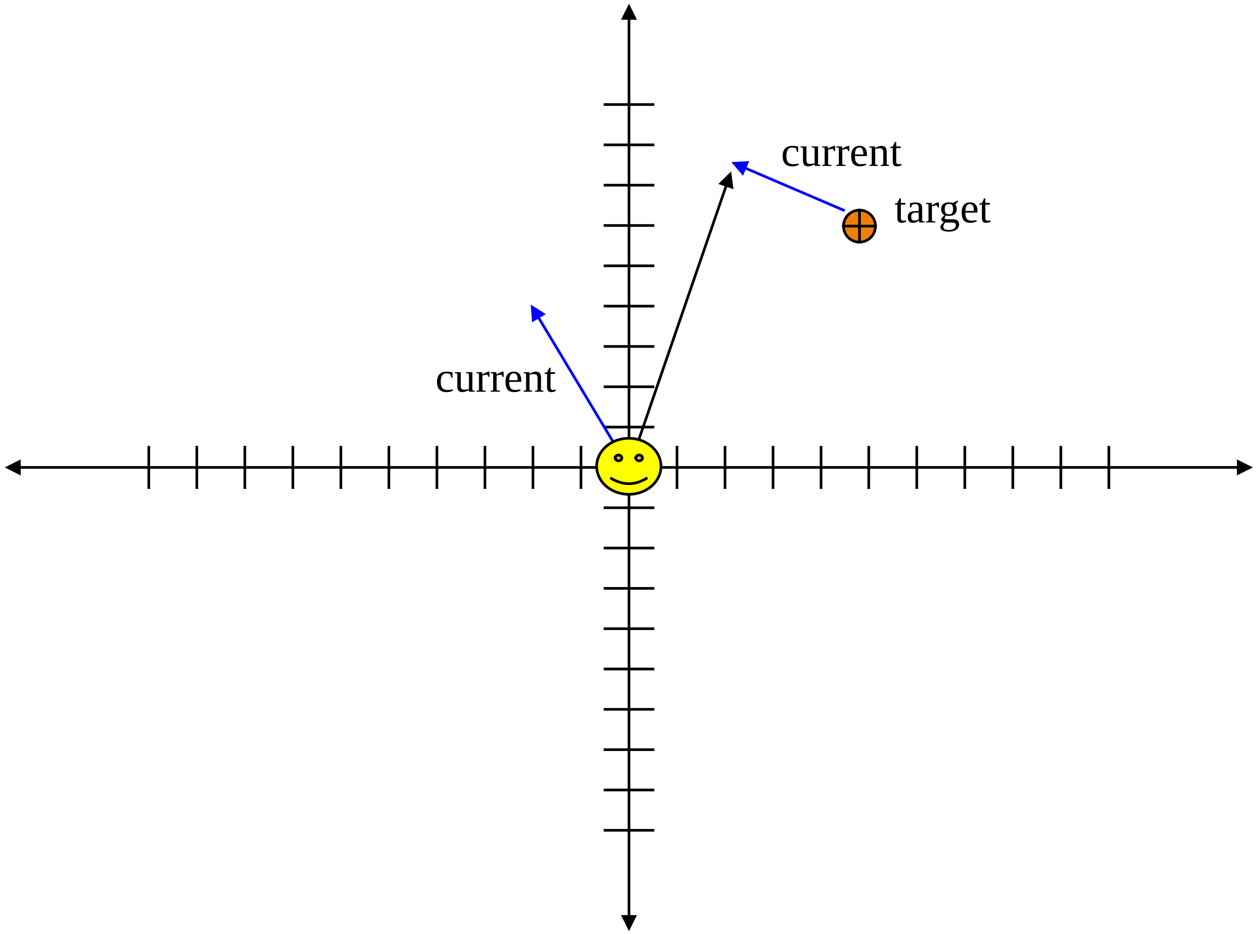
- Just adjust speed multiplier (magnitude) of DesiredVelocity vector to slow down

- Base on distance to target
- For Example
  - **Fast** if distance > 10
  - **Normal** if distance < 10 but > 4
  - **Slow** if distance < 4

Pursuit

- Pursuit is similar to seek except that the target is another moving character.
- Effective pursuit requires a prediction of the target's future position.
- Don't just aim for the target position
- Aim for where you *think* the target will be

Pursuit



Pursuit

- The motion of the target could be unpredictable
- For simplicity
    - Assume that the target is moving linearly for a short duration of time
- Seek position:
    - target->Pos() + target->Velocity *LookAheadTime
- LookAheadTime
    - DistanceToTarget/(yourSpeed + targetSpeed)

Offset Pursuit

- Steering a path which passes near, but not directly into a moving target.
- You stay at a certain offset distance from your target.
- Example:
  - a spacecraft doing a “fly-by”
  - Flying near without colliding with the target

Evade

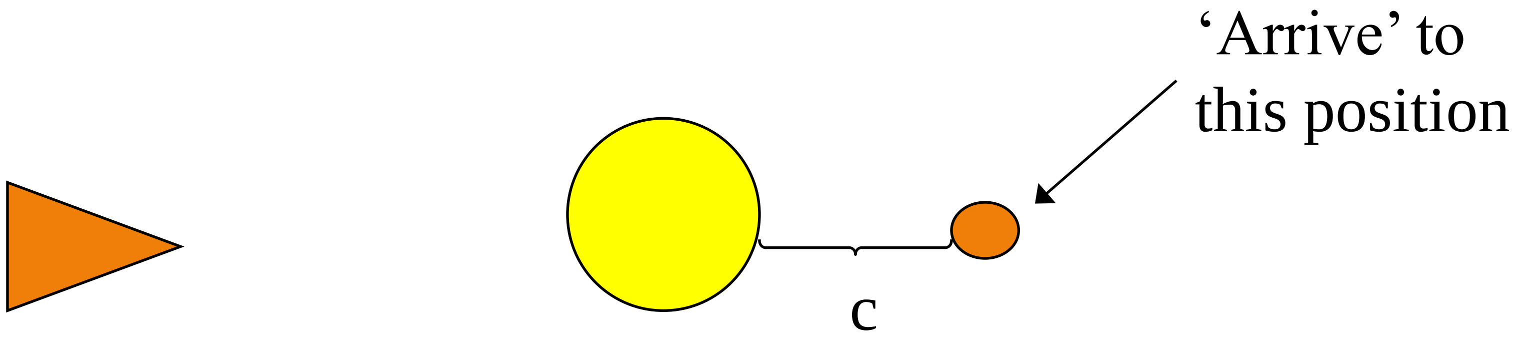
- Opposite of pursuit:
- Evader flees from the estimated future position.

Evade

- The motion of the pursuer (target) could be unpredictable
- For simplicity
  - Assume that the target is moving linearly for a short duration of time
- Seek position:
  - target->Pos() + target->Velocity *LookAheadTime
- LookAheadTime
  - DistanceToTarget/(yourSpeed + targetSpeed)

Hide

- Position yourself such that an obstacle is always between you and an agent
- Constant (c) for distance from obstacle



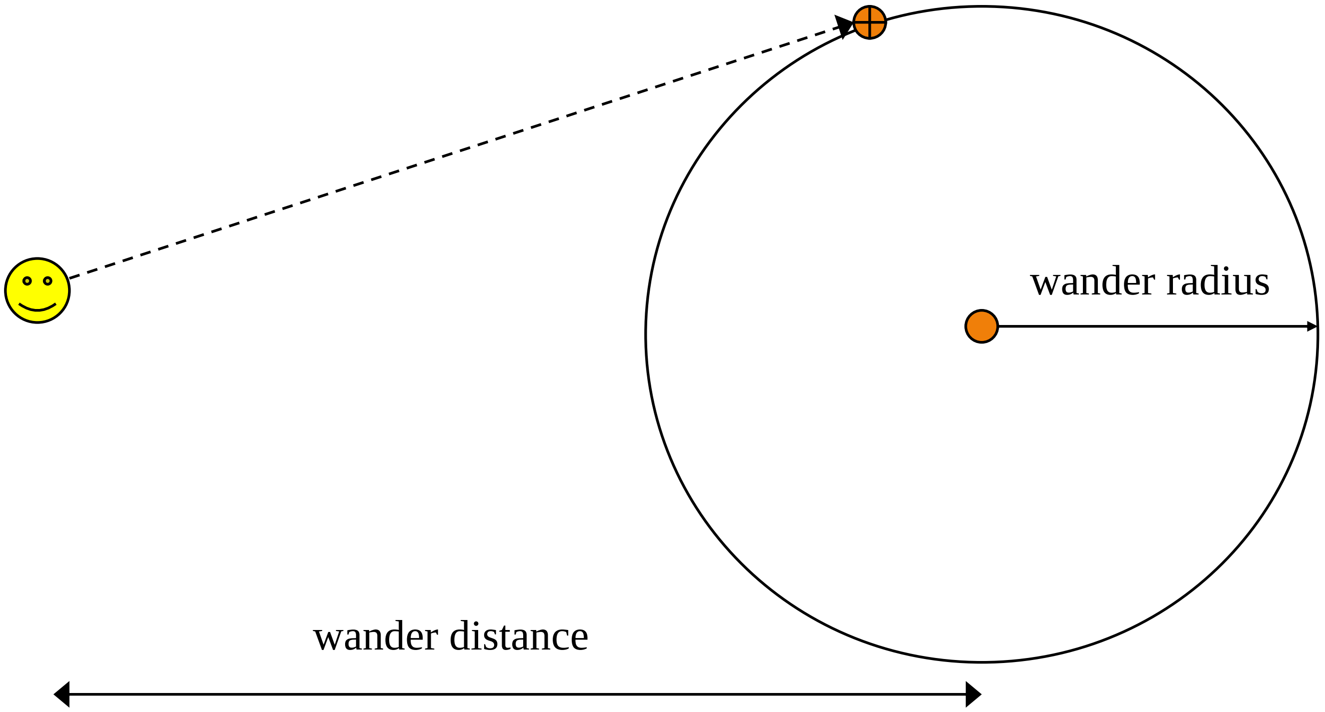
Hide

- For Each Obstacle
  - Calculate a hiding position for that obstacle
  - Calculate distance to the hiding position
- Is Hiding Position Available?
  - Yes – Call ‘Arrive’ to Closest
  - No – Evade

Wander

- Steering behavior for a ‘random walk’
- Just moving randomly is unconvincing and erratic
- No one out on a walk stops and reverses course frequently

Wander

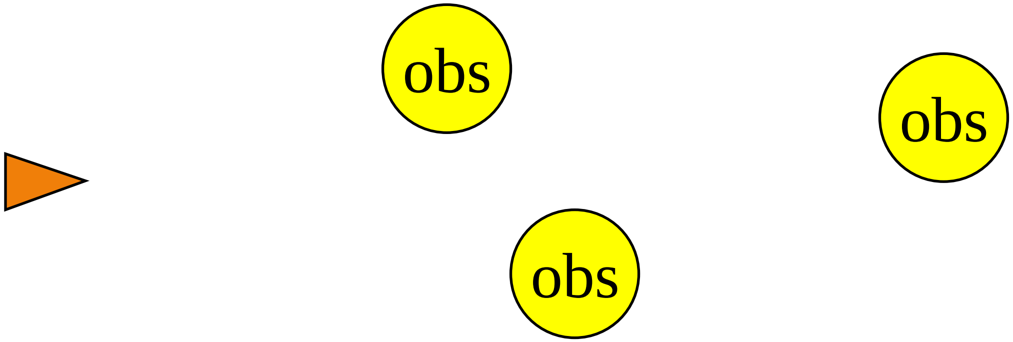


Wander

- Project a circle in front of the agent
- Steer towards a target that is constrained to move along the perimeter of this circle
- Target adjusts in small increments along the perimeter
- Adjust algorithm based on circle radius, target adjustments, circle distance

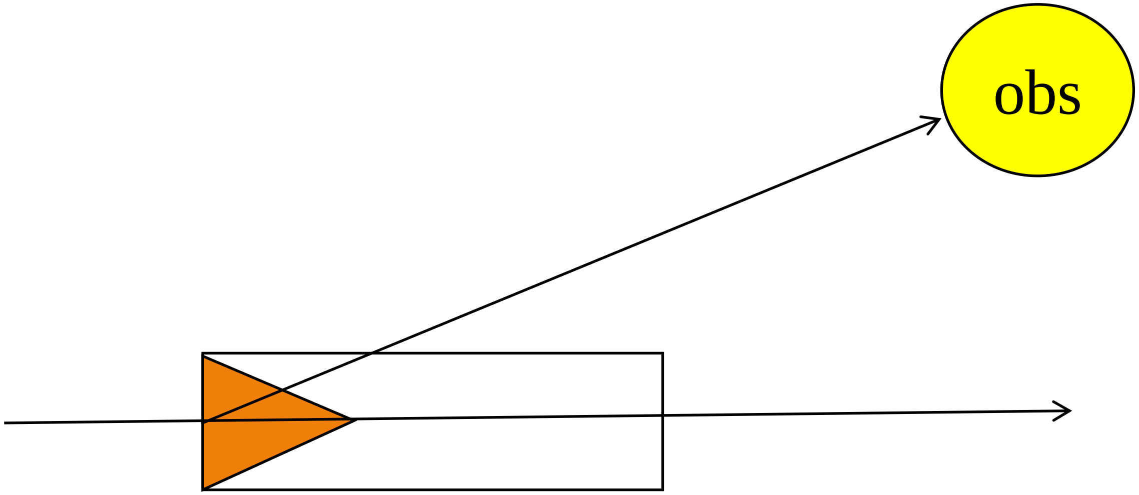
Obstacle Avoidance

- Avoiding collisions on the path



Obstacle Avoidance

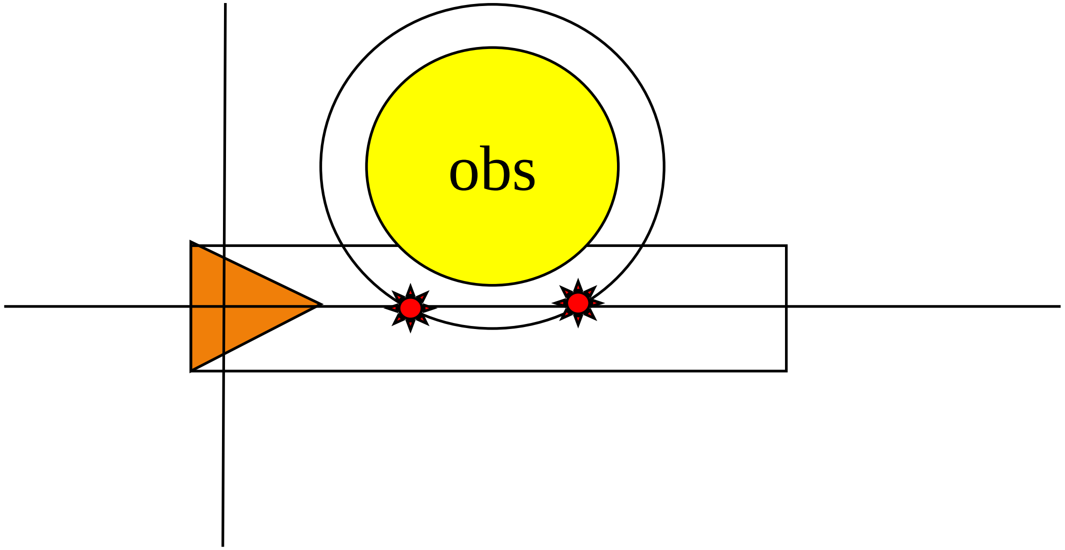
- Dot product of agent vector and vector to obstacle will be positive if obstacle is in front



Detection box
length proportionate to speed

Obstacle Avoidance

- Extend obstacle boundary by ½ width of detection box.
- If the new boundary crosses agent vector,
  - collision occurs at intersection of new boundary and detection box.



Obstacle Avoidance

- Once an obstacle that will cause a collision is detected,
  - ▫ Steer to avoid collision
- Just like driving: slow down and avoid

Path Following

- Used for multi-stage navigation
- Useful for creating bot behaviors such as patrolling
- Very commonly used AI behavior
- Just ‘seek’ each waypoint in turn
  - Can ‘arrive’ at final destination

Path Following Variations

- Wall following
  - To approach wall and then to maintain a certain offset from it.
- Containment:
  - Motion which is restricted to remain within a certain region.

What is a Flock?

- A group of birds or mammals assembled or herded together

Why Model Flocking?

- It looks cool!!
- Difficult to animate using traditional keyframing or other techniques
  - Hard to script the path
  - Hard to handle motion constraints
  - Hard to edit motion

Boids!

- “boids” comes from “bird-oids”
- Similar to particle systems, but have orientation
- Have a geometric shape used for rendering
- Behavior-based motion

Boid motion

- Boids have a local coordinate system
- Flight is accomplished using a dynamic, incremental, and rigid geometrical transformation
  - Not a particle system
- Flight path not specified in advance

Boid motion

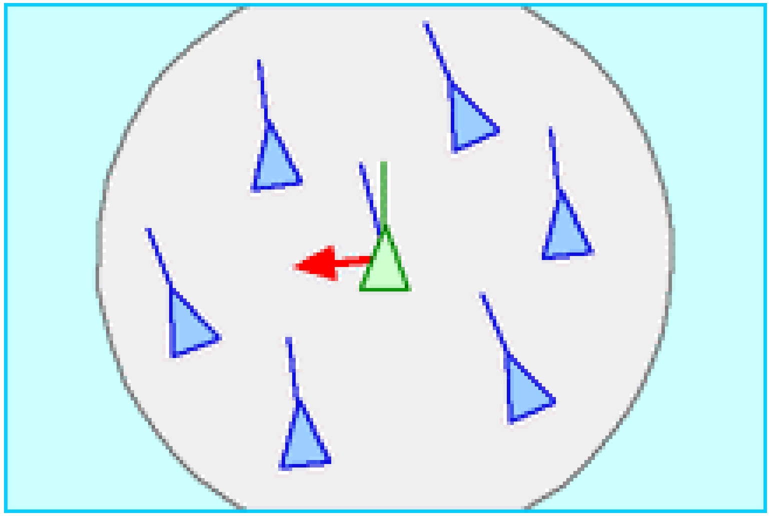
- Rotation about X, Y, and Z axes for pitch, yaw, and roll
- No notion of lift or gravity
  - Except for banking (incline)
- Limits set for maximum speed and maximum acceleration

Flocking Behavior

- Flocking is a group behavior
- Agents can react to other agents
- Combination of
  - Alignment
  - Cohesion
  - Separation

Alignment

- Gives a character the ability to align itself with (that is, head in the same direction and/or speed as) other nearby characters.

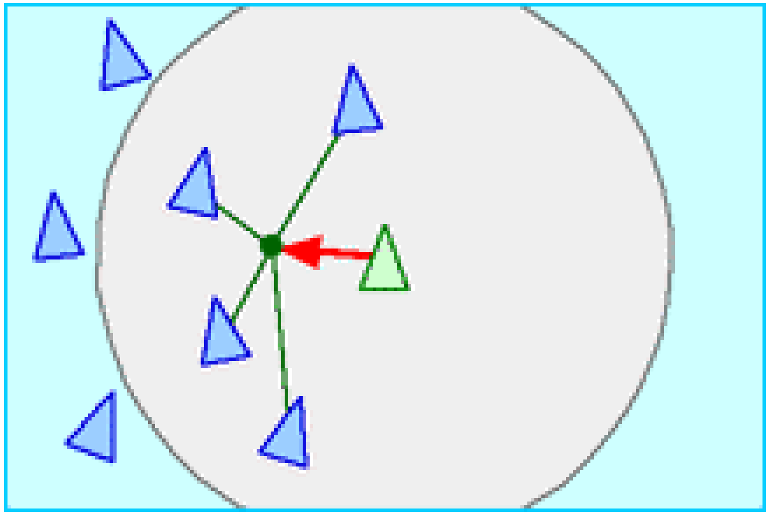


Alignment Vector

- Every agent of the flock will check the direction of all the other agents.
- Calculate the average *direction* of the flock.
- To get this direction you can simply access the forward vector from the transform of the game object.

Cohesion

- Gives a character the ability to cohere with (approach and form a group with) other nearby characters.

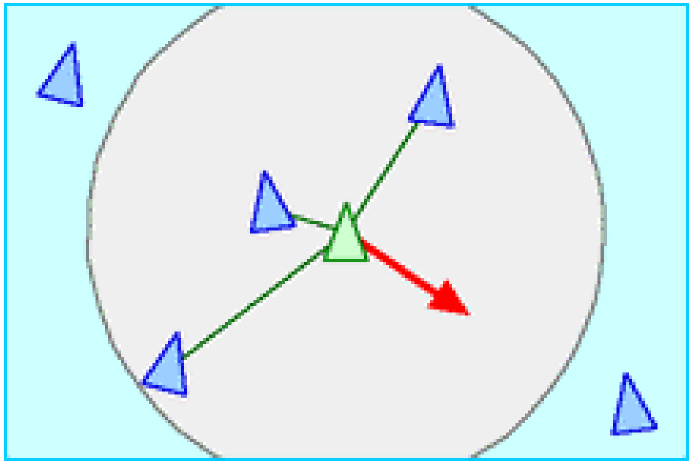


Cohesion Vector

- Vector pointing towards the center of mass of the flock
- Each agent needs to calculate the average position of all the other agents
- Calculate the direction towards the center of the flock from its current location

Separation

- Gives a character the ability to maintain a certain separation distance from others nearby.
- This can be used to prevent characters from crowding together.



Separation Vector

- Separation is the average distance from the current agent to all the other agents
- Average distance vector obtained this way will point towards the other agents
- Need to invert it to get the separation vector

Flocking Vector

- Weightage Average
  - Alignment Weight (W1)
  - Cohesion Weight (W2)
  - Separation Weight (W3)

```
Vector3 flockingVector = (( AlignmentVector.normalized * W1) +
                           ( cohesionVector.normalized * W2) +
                           ( separationVector.normalized * W3) );
```

Summary

- Steering Behaviors
- Flocking/Boids

- Read the paper by Craig Reynolds
  - Available on Moodle

- A book chapter on flocking is also available on Moodle

- Flocking Implementation Help:
  - http://www.lorenzomori.com/unity3d/flocking-behaviour-a-unity3d-ai-experiment/
  - https://github.com/lormori/FlockingDemo