

# Artificial Intelligence

## Lecture 07 – Steering Behaviors

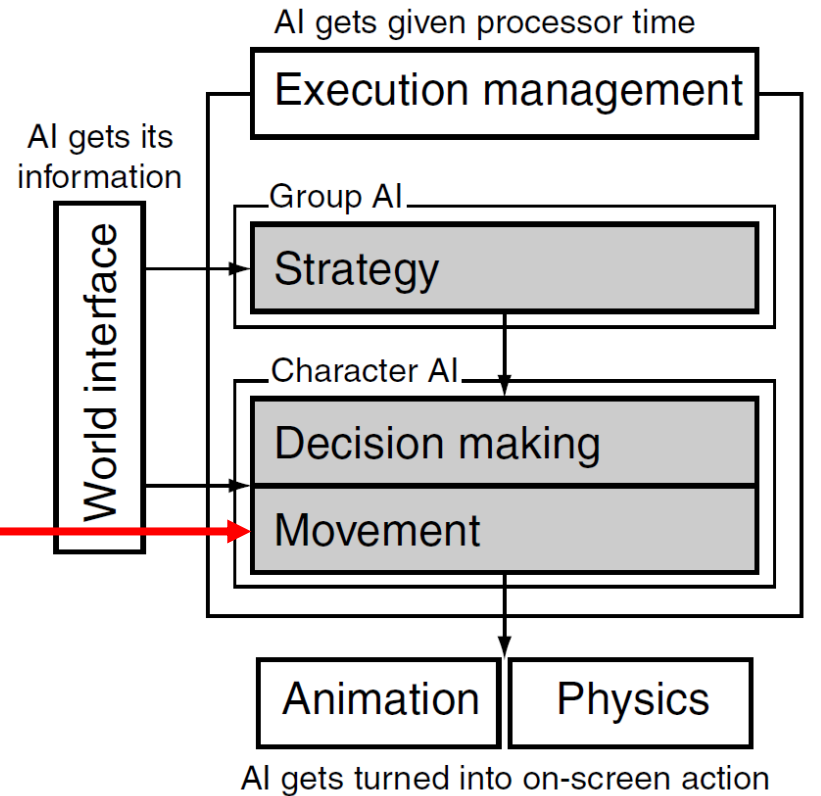
Edirlei Soares de Lima

<edirlei.lima@universidadeeuropeia.pt>

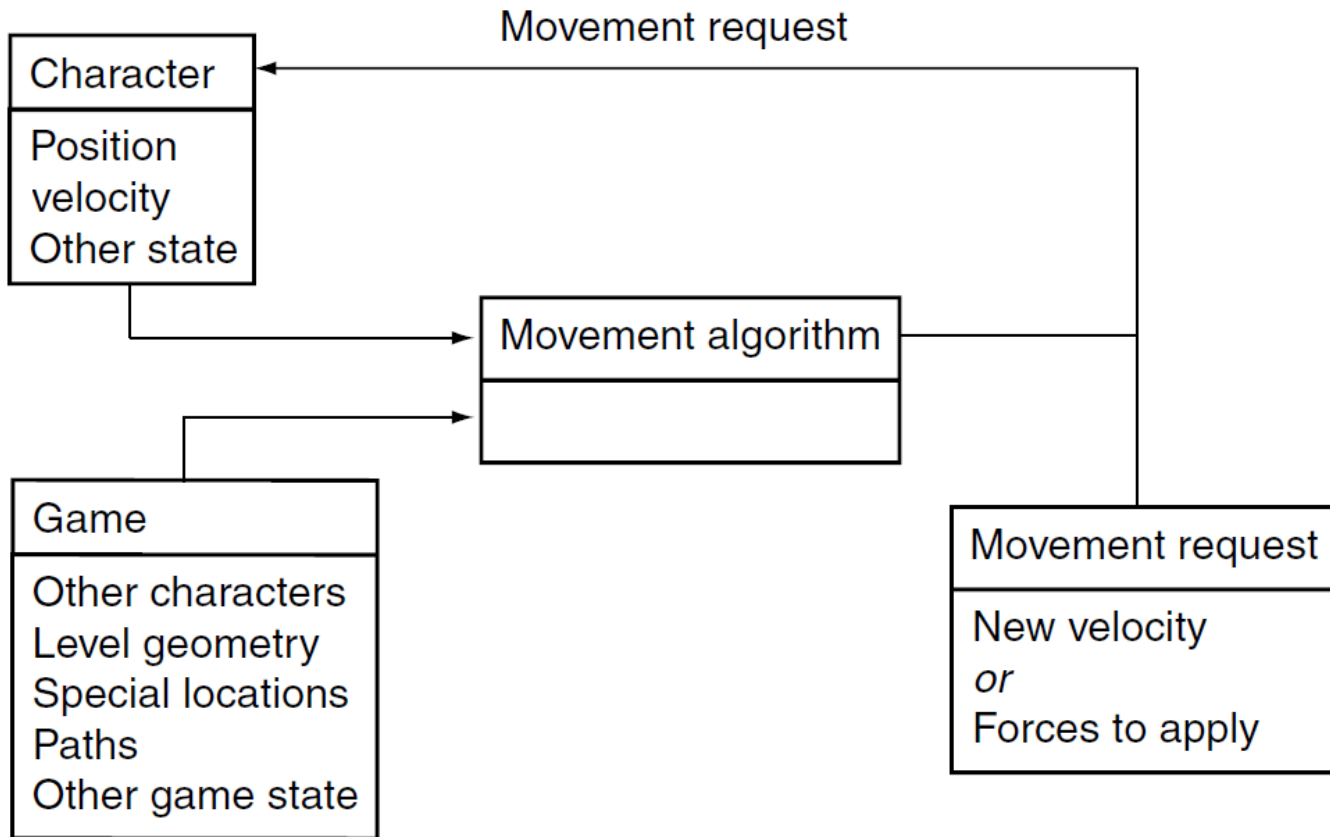


# Game AI – Model

- Pathfinding
- **Steering behaviors**
- Finite state machines
- Automated planning
- Behaviour trees
- Randomness
- Sensor systems
- Machine learning

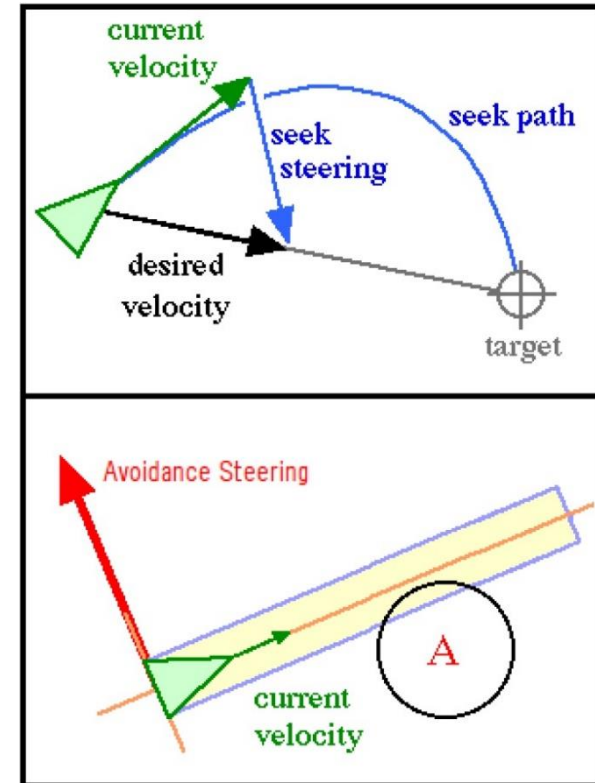


# Basics of Movement Algorithms



# Steering Behaviors

- Steering Behaviors are designed to give autonomous characters the ability to navigate around their world in a life-like and improvisational manner.
- They are not based on complex strategies involving path planning or global calculations, but instead use local information.
- Combinations of steering behaviors can be used to achieve higher level goals.



# Steering Behaviors Data

- A character can be defined by the following kinematic data structure:

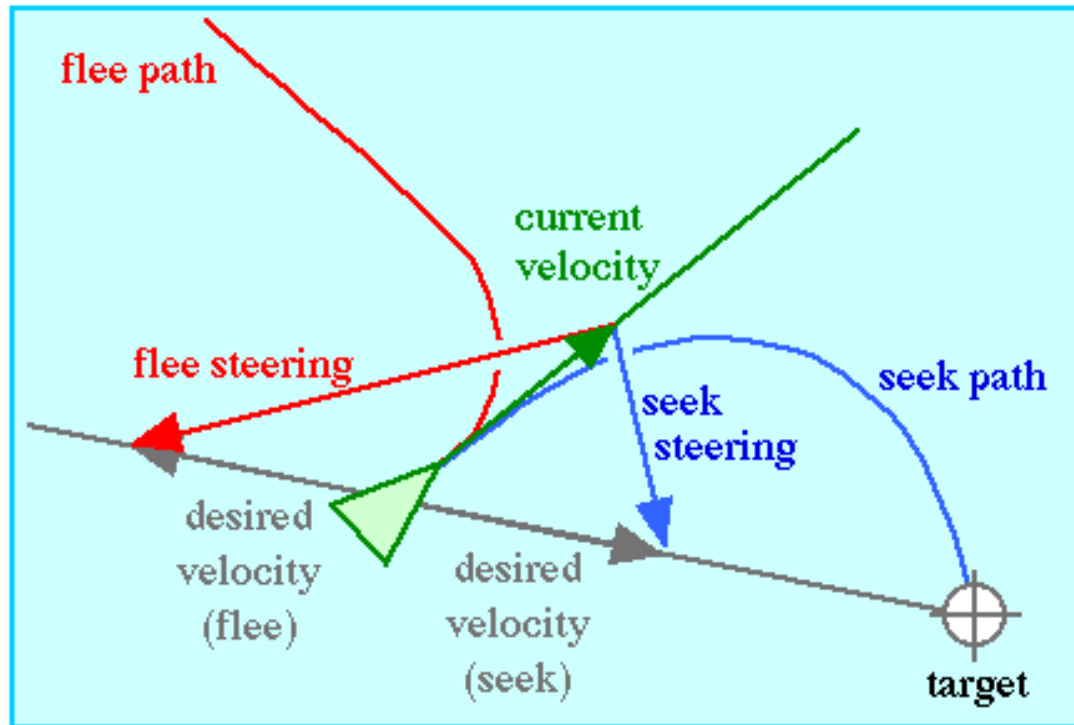
```
struct Kinematic:  
  
    position      # a 2 or 3D vector  
    orientation   # a single floating point value  
    velocity      # another 2 or 3D vector
```

- Steering behaviors operate with these kinematic data. Their output is a set of accelerations that will change the velocities of characters in order to move them around the level:

```
struct SteeringOutput:  
  
    linear        # a 2 or 3D vector  
    angular       # a single floating point value
```

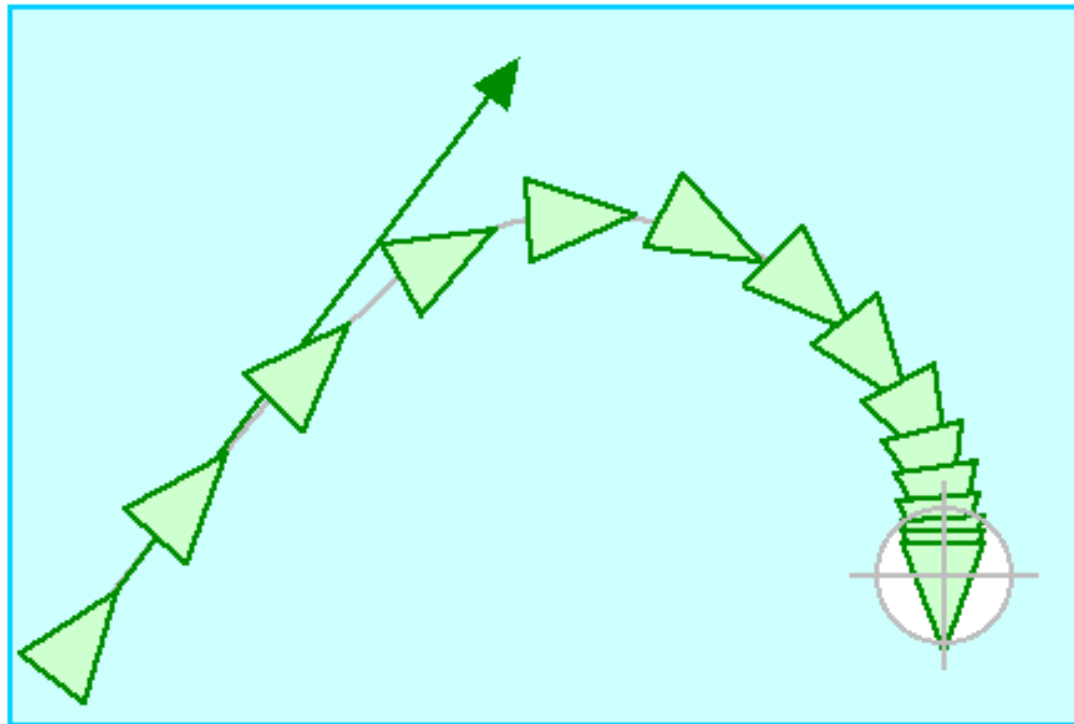
# Steering Behaviors – Overview

- Seek and Flee:



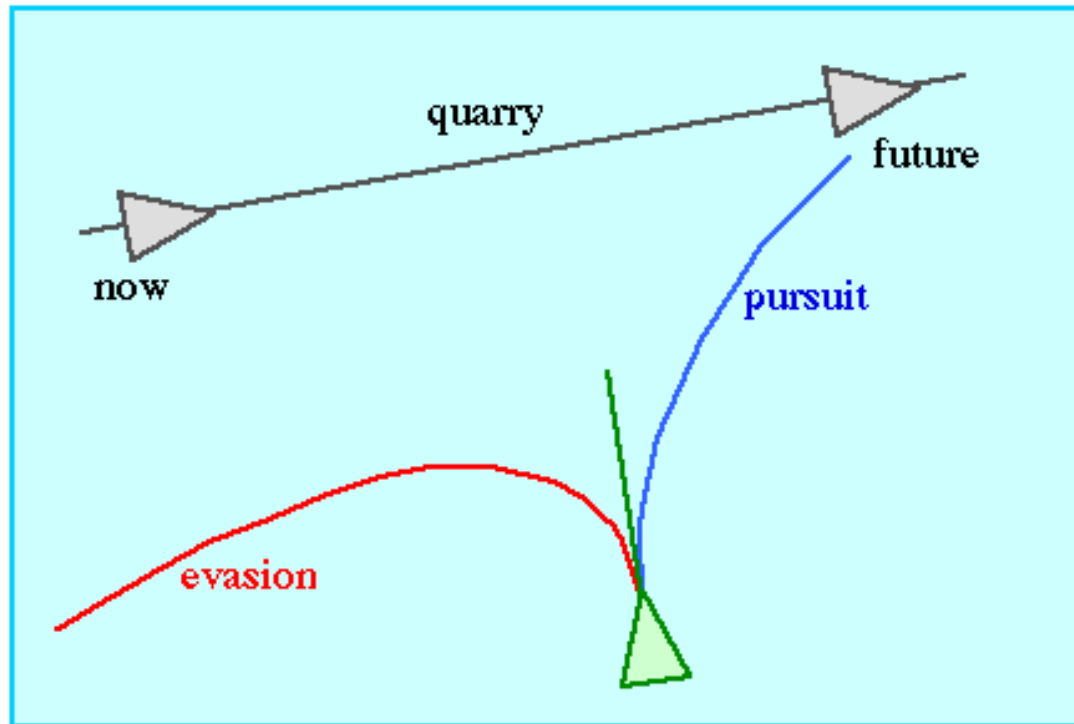
# Steering Behaviors – Overview

- Arrival:



# Steering Behaviors – Overview

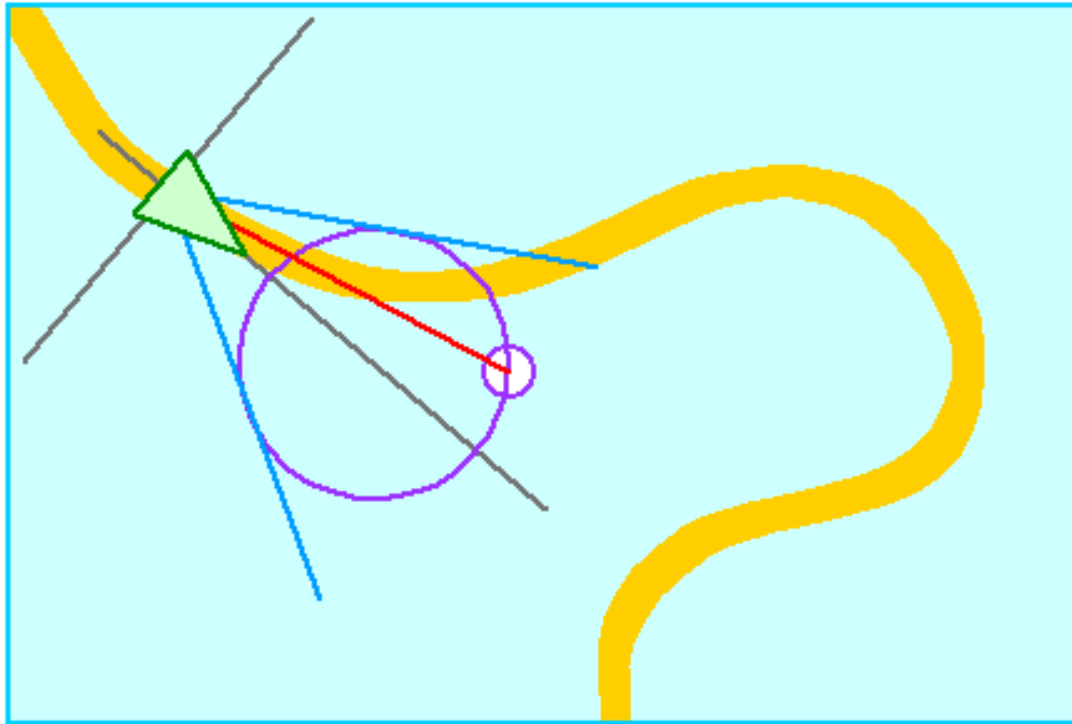
- Pursue and Evade:





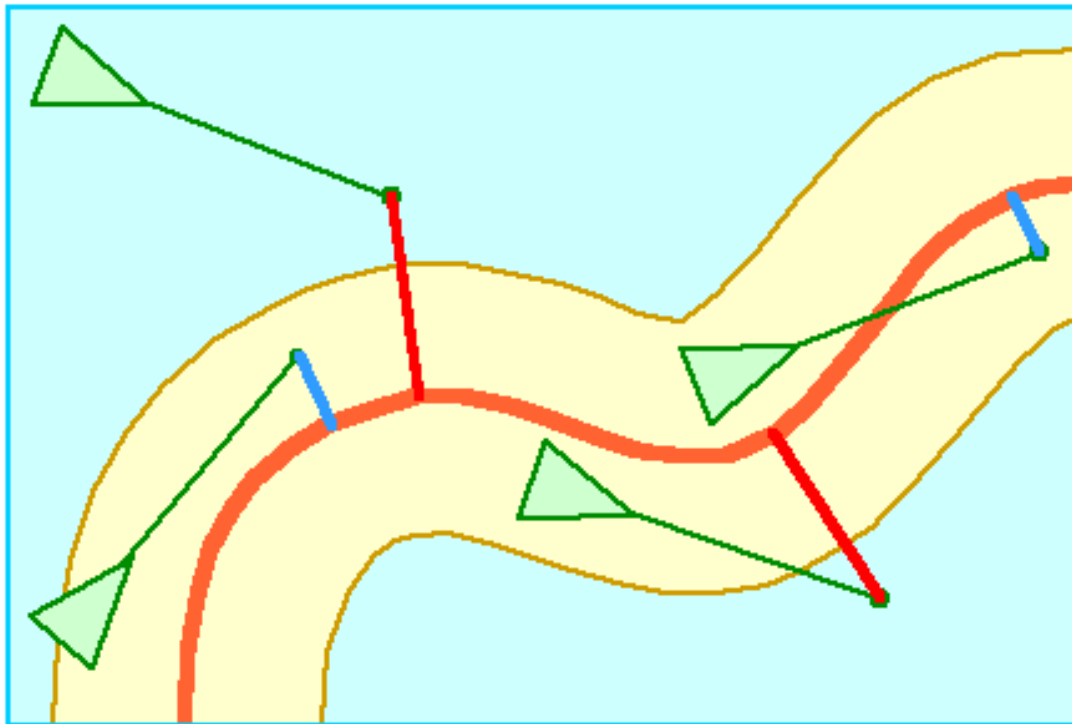
# Steering Behaviors – Overview

- Wander:



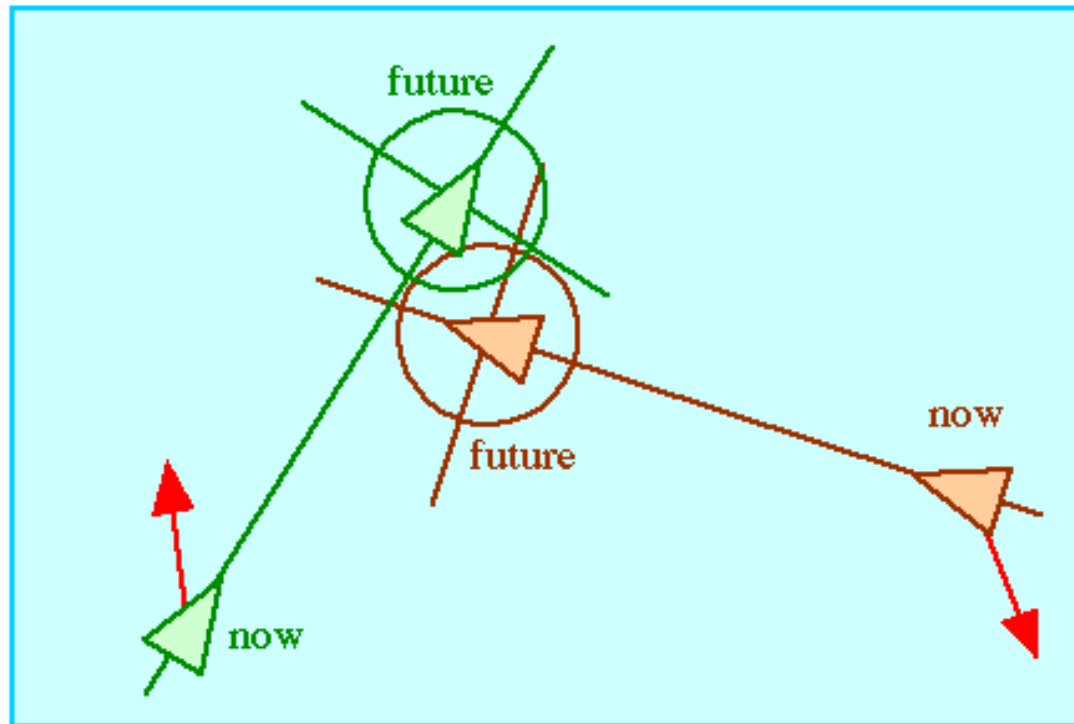
# Steering Behaviors – Overview

- Path Following:



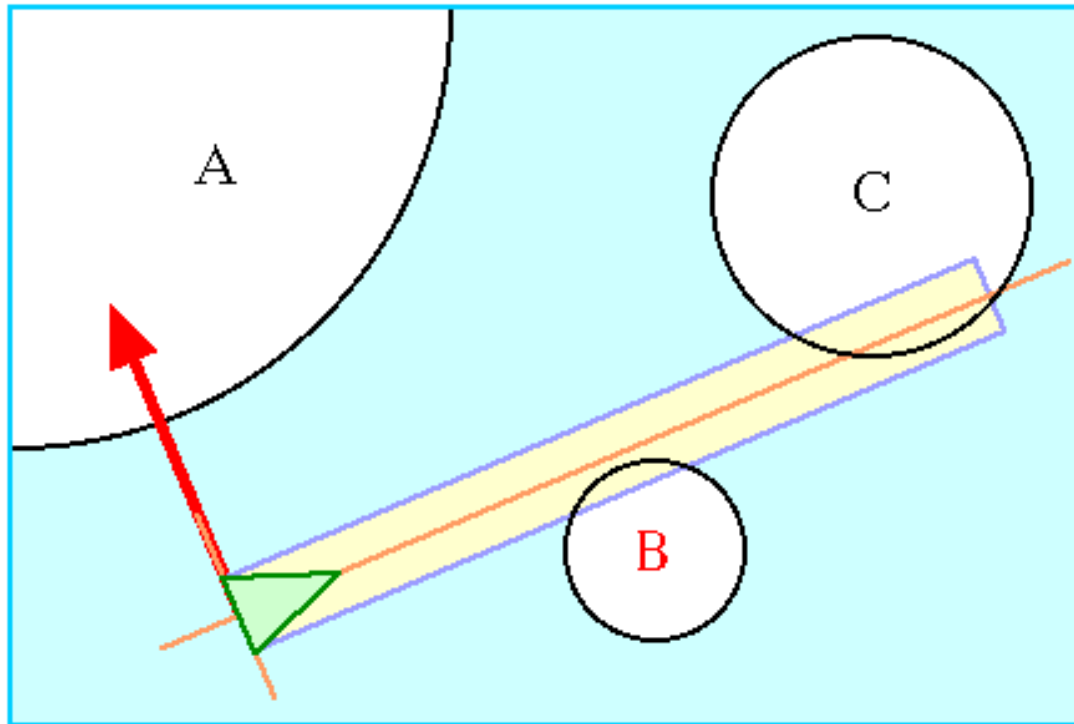
# Steering Behaviors – Overview

- Collision Avoidance:



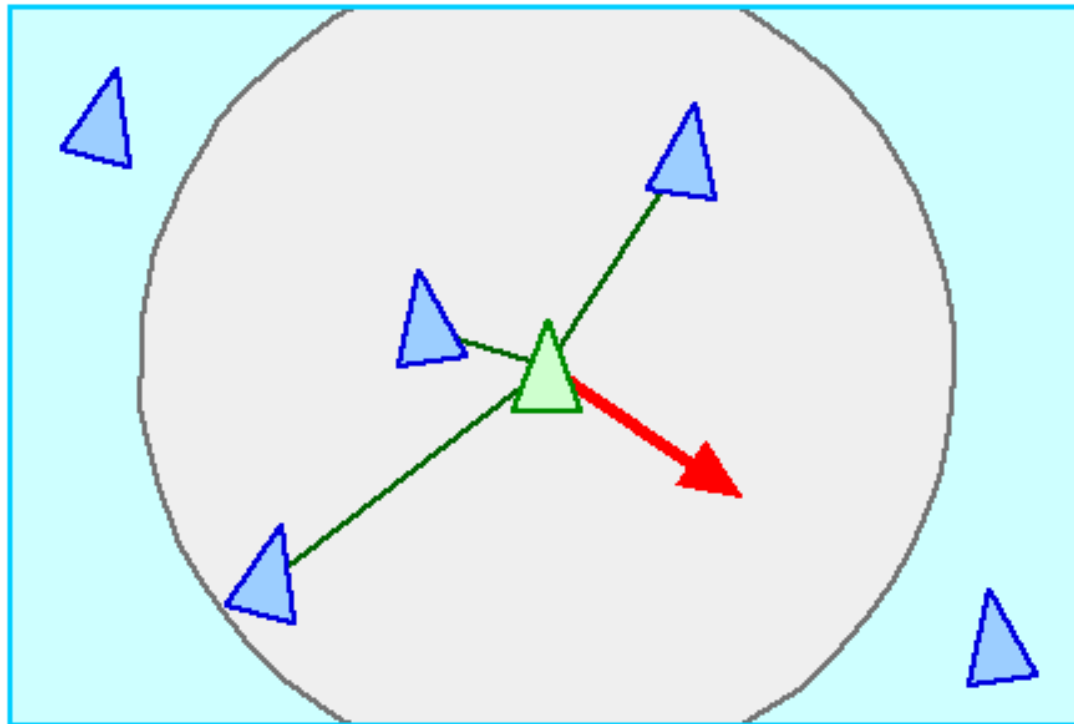
# Steering Behaviors – Overview

- Obstacle Avoidance:



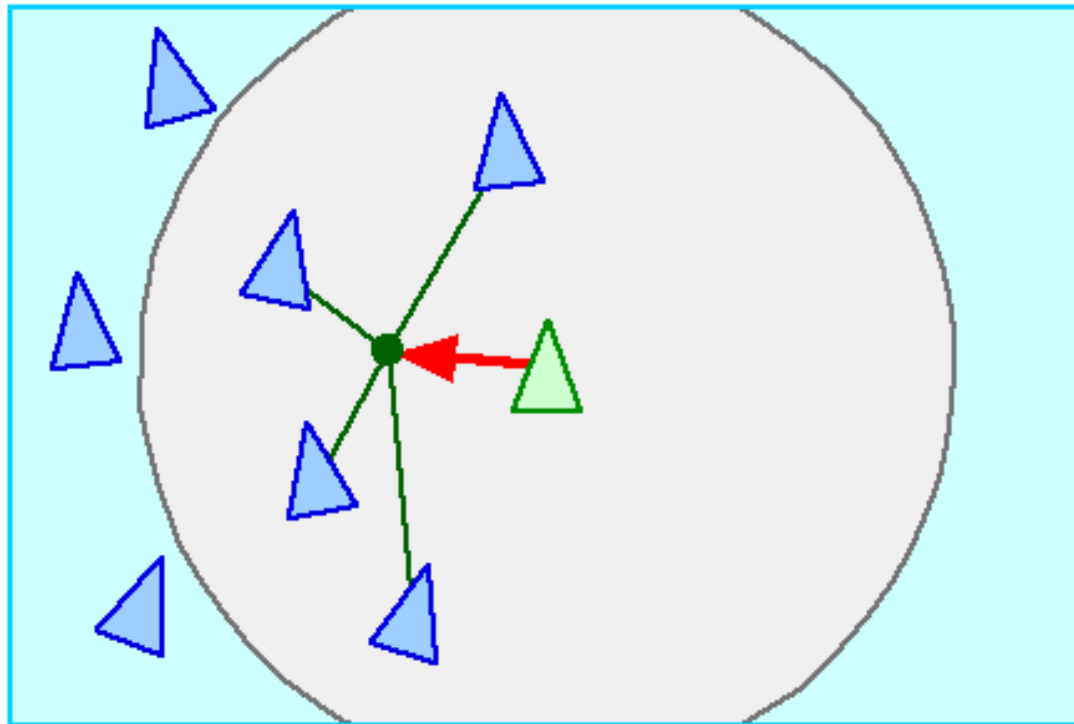
# Steering Behaviors – Overview

- Separation:



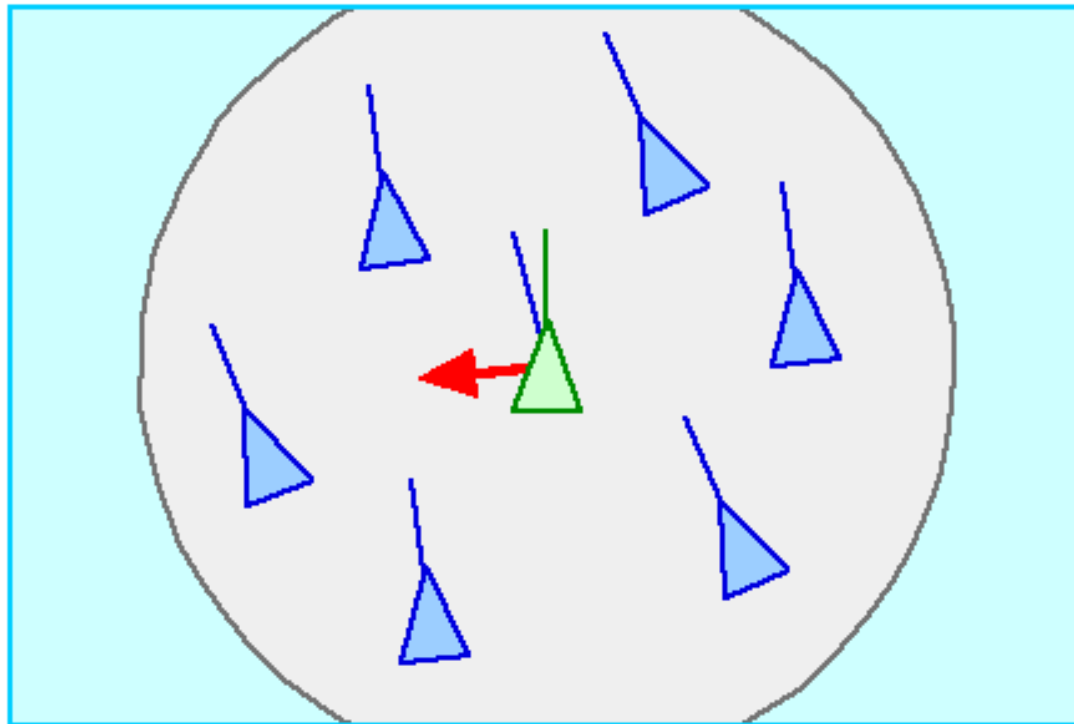
# Steering Behaviors – Overview

- Cohesion:



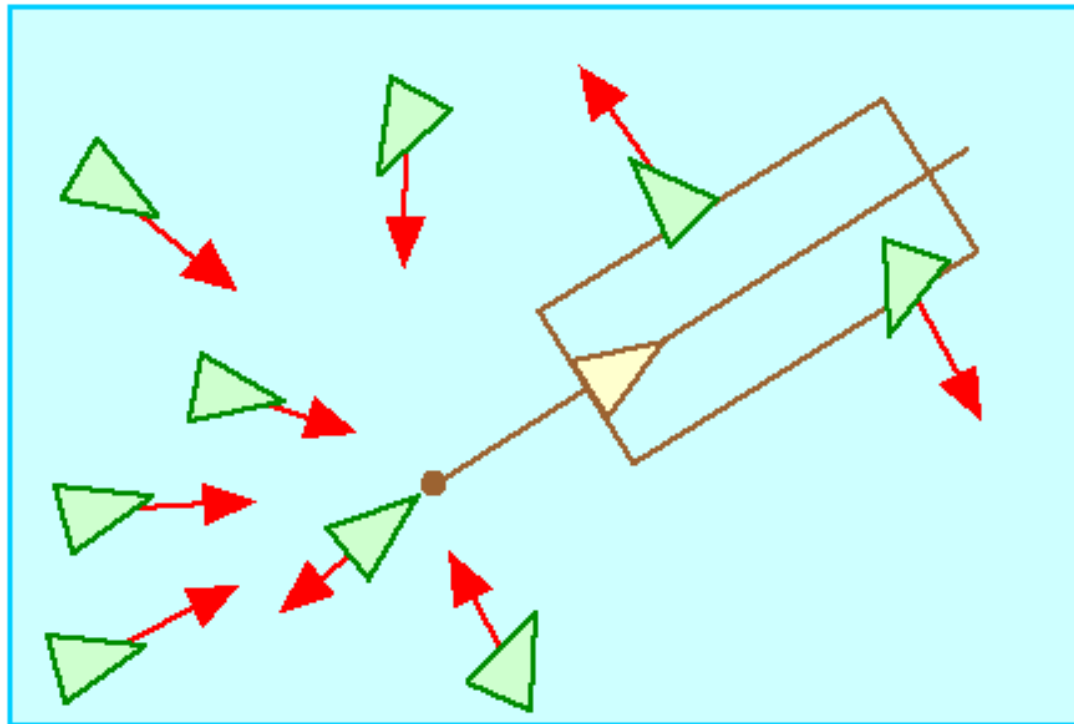
# Steering Behaviors – Overview

- Alignment:



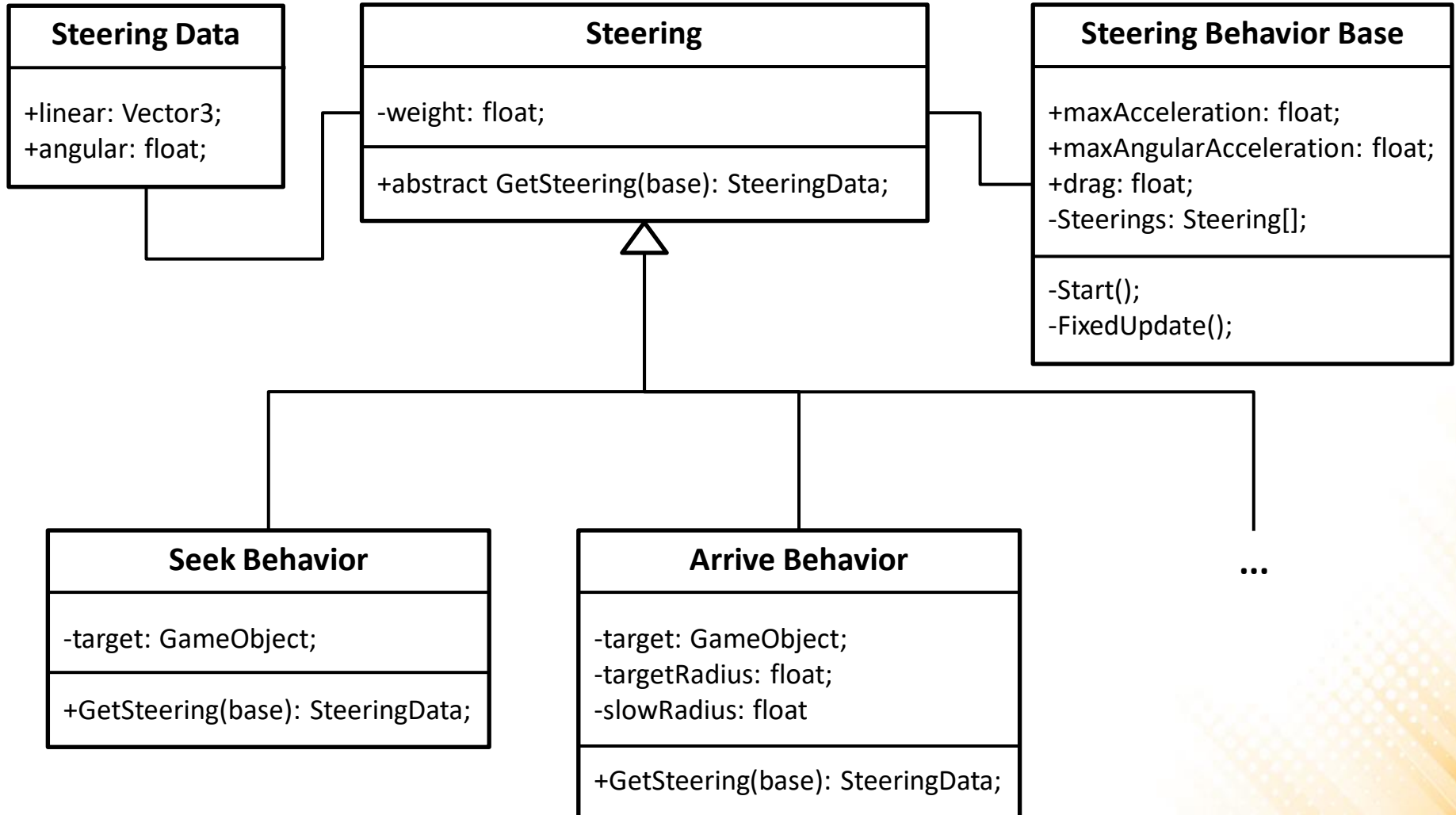
# Steering Behaviors – Overview

- Flocking (Leader Following):





# Unity Implementation – Class Diagram



# Base Classes

- **Steering Class:**

```
public abstract class Steering : MonoBehaviour
{
    public abstract SteeringData GetSteering(SteeringBehaviorBase
                                             steeringbase);
}
```

- **SteeringData Class:**

```
public class SteeringData{
    public Vector3 linear;
    public float angular;

    public SteeringData(){
        linear = Vector3.zero;
        angular = 0f;
    }
}
```

# Base Classes

- **SteeringBehaviorBase Class:**

```
public class SteeringBehaviorBase : MonoBehaviour
{
    private Rigidbody rb;
    private Steering[] steerings;
    public float maxAcceleration = 10f;
    public float maxAngularAcceleration = 3f;
    public float drag = 1f;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
        steerings = GetComponents<Steering>();
        rb.drag = drag;
    }

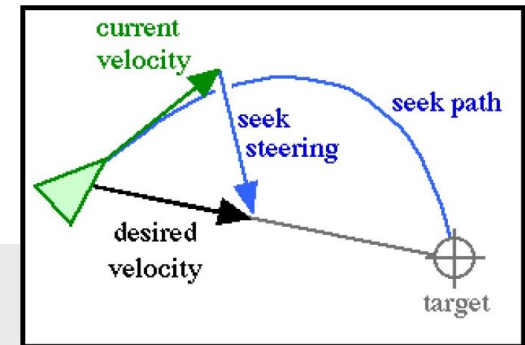
    ...
}
```

# Base Classes

```
void FixedUpdate ()
{
    Vector3 acceleration = Vector3.zero;
    float rotation = 0f;
    foreach (Steering behavior in steerings)
    {
        SteeringData steering = behavior.GetSteering(this);
        acceleration += steering.linear;
        rotation += steering.angular;
    }
    if (acceleration.magnitude > maxAcceleration){
        acceleration.Normalize();
        acceleration *= maxAcceleration;
    }
    rb.AddForce(acceleration);
    if (rotation != 0)
    {
        rb.rotation = Quaternion.Euler(0, rotation, 0);
    }
}
}
```

# Seek Behavior

- Seek behavior tries to match the position of the character with the position of the target.

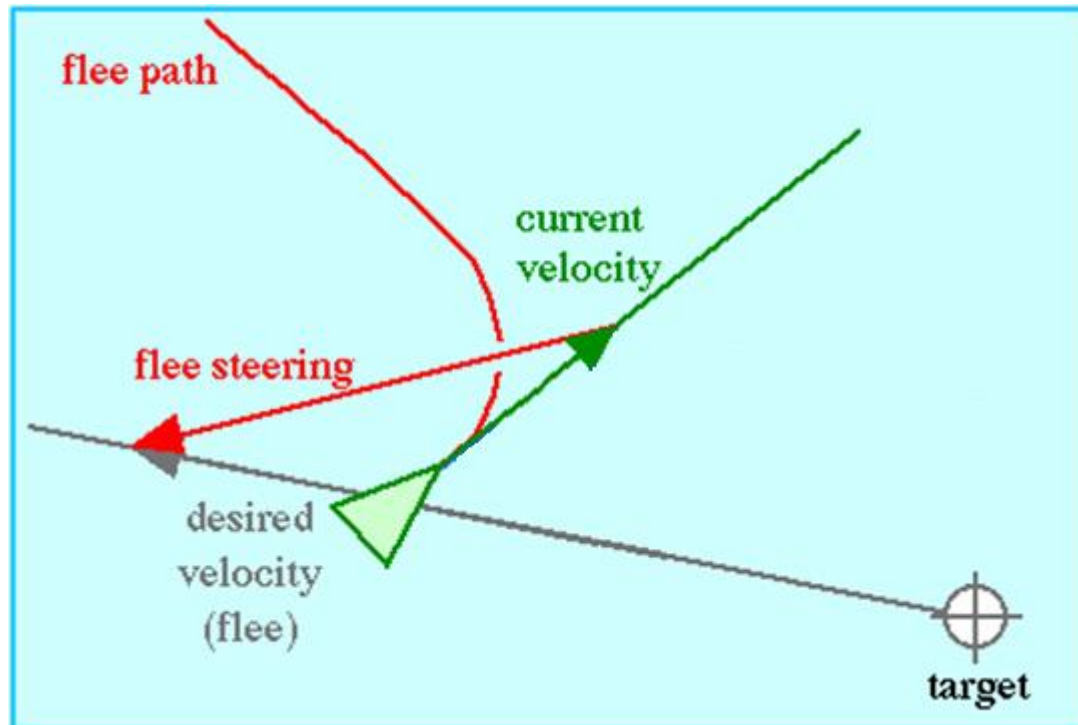


```
public class SeekBehavior : Steering
{
    [SerializeField] private GameObject target;

    public override SteeringData GetSteering(SteeringBehaviorBase
                                             steeringbase) {
        SteeringData steering = new SteeringData();
        steering.linear = target.transform.position - transform.position;
        steering.linear.Normalize();
        steering.linear *= steeringbase.maxAcceleration;
        steering.angular = 0;
        return steering;
    }
}
```

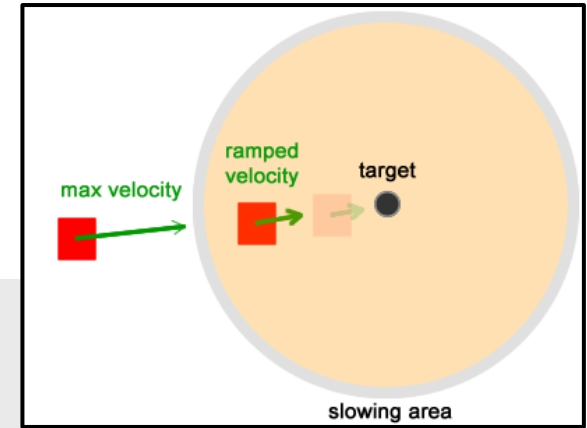
# Exercise 1

- 1) Flee is the opposite of seek. It tries to get as far from the target as possible. Implement and test the flee behavior class.



# Arrive Behavior

- Arrive is similar to seek, but the character slows down as he gets close to the target.
  - It arrives exactly at the right location.

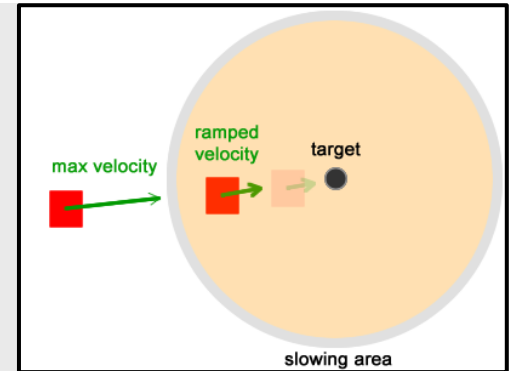


```
public class ArriveBehavior : Steering
{
    [SerializeField] private Transform target;
    [SerializeField] private float targetRadius = 1.5f;
    [SerializeField] private float slowRadius = 5f;

    public override SteeringData GetSteering(SteeringBehaviorBase
                                             steeringbase) {
        SteeringData steering = new SteeringData();
        Vector3 direction = target.position - transform.position;
        float distance = direction.magnitude;
        if (distance < targetRadius) {
            steeringbase.GetComponent<Rigidbody>().velocity = Vector3.zero;
            return steering;
        }
    }
}
```

# Arrive Behavior

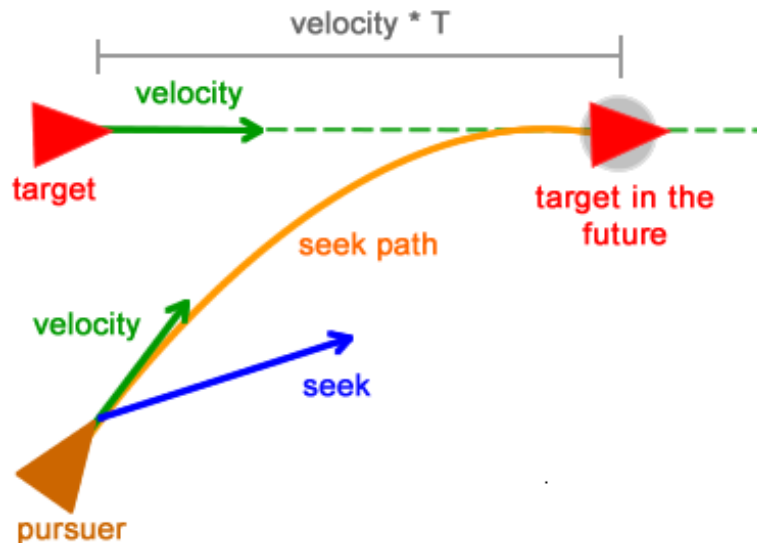
```
float targetSpeed;  
if (distance > slowRadius)  
    targetSpeed = steeringbase.maxAcceleration;  
else  
    targetSpeed = steeringbase.maxAcceleration *  
                (distance / slowRadius);  
Vector3 targetVelocity = direction;  
targetVelocity.Normalize();  
targetVelocity *= targetSpeed;  
steering.linear = targetVelocity -  
                 steeringbase.GetComponent<Rigidbody>().velocity;  
if (steering.linear.magnitude > steeringbase.maxAcceleration){  
    steering.linear.Normalize();  
    steering.linear *= steeringbase.maxAcceleration;  
}  
steering.angular = 0;  
return steering;  
}  
}
```





# Pursue Behavior

- Pursue behavior tries to predict where the target will be at some time in the future and aim toward that point.



- How to predict the future position?
  - We assume the target will continue moving with the same velocity it currently has.
  - This is a reasonable assumption over short distances.

# Pursue Behavior

- Pursue behavior tries to predict where the target will be at some time in the future and aim toward that point.

```
public class PursueBehavior : Steering{
```

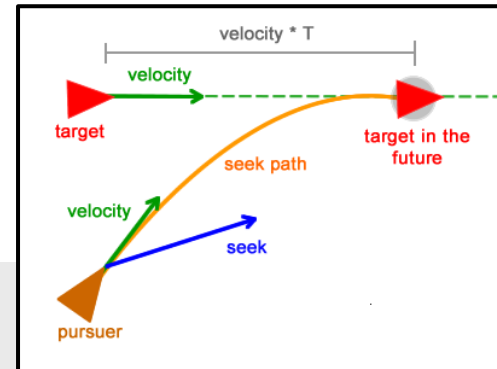
```
    [SerializeField] private float maxPrediction = 2f;  
    [SerializeField] private GameObject target;
```

```
    public override SteeringData GetSteering(SteeringBehaviorBase  
                                             steeringbase) {
```

```
        SteeringData steering = new SteeringData();
```

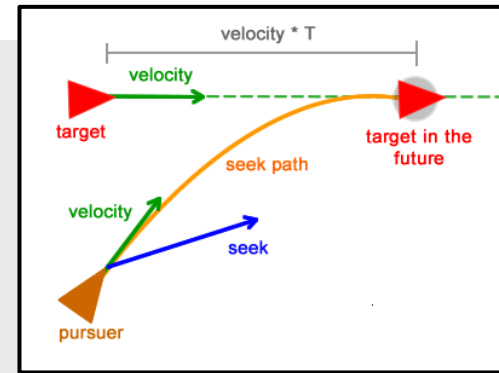
```
        Vector3 direction = target.transform.position - transform.position;  
        float distance = direction.magnitude;  
        float speed = GetComponent<Rigidbody>().velocity.magnitude;
```

```
        ...
```



# Pursue Behavior

```
float prediction;  
if (speed <= (distance / maxPrediction))  
    prediction = maxPrediction;  
else  
    prediction = distance / speed;
```

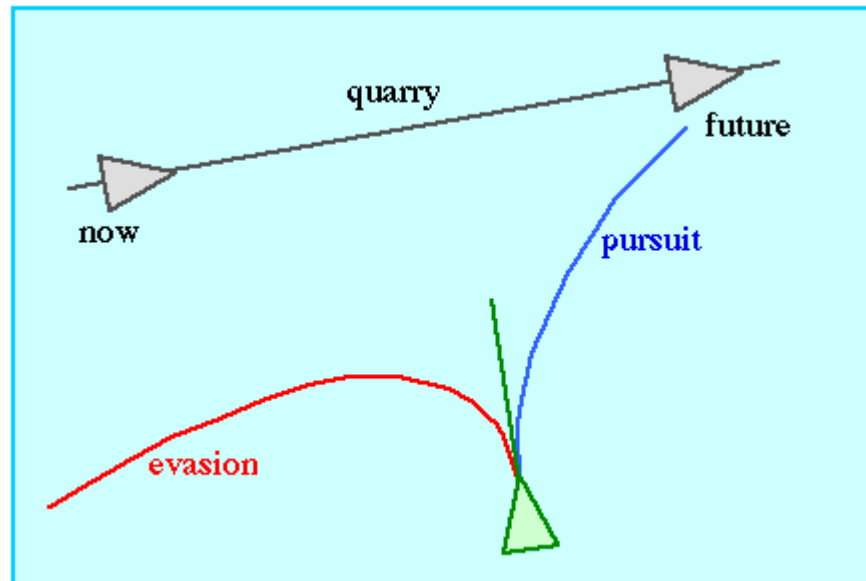


```
Vector3 predictedTarget = target.transform.position +  
    (target.GetComponent<Rigidbody>().velocity * prediction);  
  
steering.linear = predictedTarget - transform.position;  
steering.linear.Normalize();  
steering.linear *= steeringbase.maxAcceleration;  
steering.angular = 0;  
  
return steering;  
}  
}
```

Seek behavior code.

# Exercise 2

- 2) The opposite behavior of pursuit is evade. Once again, we calculate the predicted position of the target, but rather than delegating to seek, we delegate to flee. Implement and test the evade behavior class.



# Face Behavior

- The face behavior makes a character look at its target.

```
public class FaceBehavior : Steering{

    [SerializeField] private Transform target;

    public override SteeringData GetSteering(SteeringBehaviorBase
                                             steeringbase){
        SteeringData steering = new SteeringData();
        Vector3 direction = target.position - transform.position;
        float angle = Mathf.Atan2(direction.x, direction.z) *
            Mathf.Rad2Deg;
        steering.angular = Mathf.LerpAngle(transform.rotation.eulerAngles.y,
                                           angle, steeringbase.maxAngularAcceleration *
                                           Time.deltaTime);
        steering.linear = Vector3.zero;
        return steering;
    }
}
```

# Look Where You're Going Behavior

- The “look where you're going” behavior calculates the orientation using the current velocity of the character.

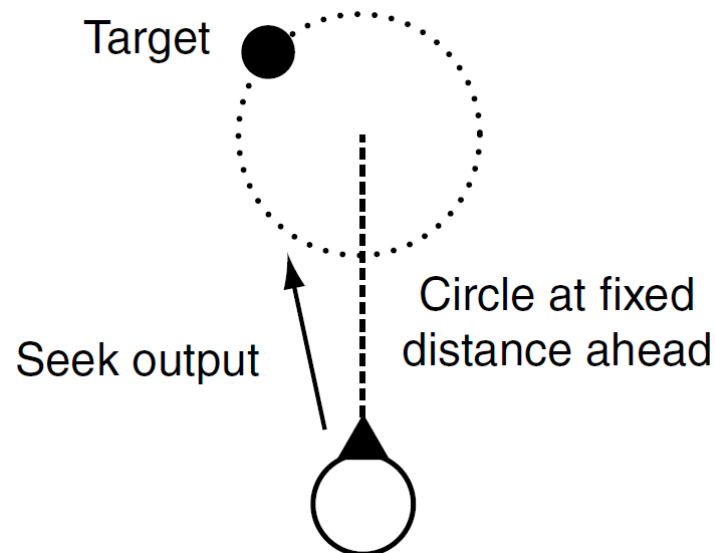
```
public class LookWhereYoureGoingBehavior : Steering{

    public override SteeringData GetSteering(SteeringBehaviorBase
                                             steeringbase){
        SteeringData steering = new SteeringData();
        if (GetComponent<Rigidbody>().velocity.magnitude == 0)
            return steering;

        float angle = Mathf.Atan2(GetComponent<Rigidbody>().velocity.x,
                                   GetComponent<Rigidbody>().velocity.z) * Mathf.Rad2Deg;
        steering.angular = Mathf.LerpAngle(transform.rotation.eulerAngles.y,
                                           angle, steeringbase.maxAngularAcceleration *
                                           Time.deltaTime);
        steering.linear = Vector3.zero;
        return steering;
    }
}
```

# Wander Behavior

- The wander behavior moves a character aimlessly. The character must retain steering direction and make small random displacements to it each frame.
  - This behavior can be implemented in several ways, but one that produces good results is to constrain the steering force to the surface of a circle located slightly ahead of the character.



# Wander Behavior

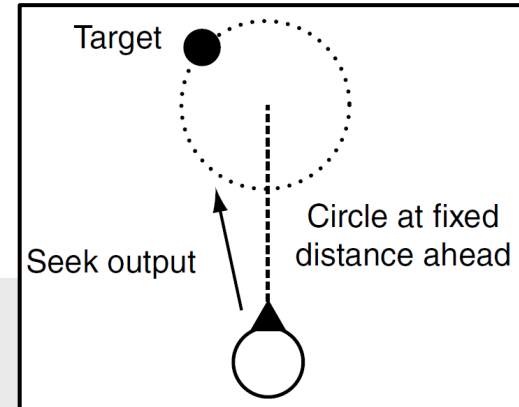
- The wander behavior moves a character aimlessly.

```
public class WanderBehavior : Steering
{
    [SerializeField] private float wanderRate = 0.4f;
    [SerializeField] private float wanderOffset = 1.5f;
    [SerializeField] private float wanderRadius = 4f;

    private float wanderOrientation = 0f;

    private float RandomBinomial() {
        return Random.value - Random.value;
    }

    private Vector3 OrientationToVector(float orientation) {
        return new Vector3(Mathf.Cos(orientation), 0,
            Mathf.Sin(orientation));
    }
}
```





# Wander Behavior

```
public override SteeringData GetSteering(SteeringBehaviorBase
                                         steeringbase) {
    SteeringData steering = new SteeringData();
    wanderOrientation += RandomBinomial() * wanderRate;

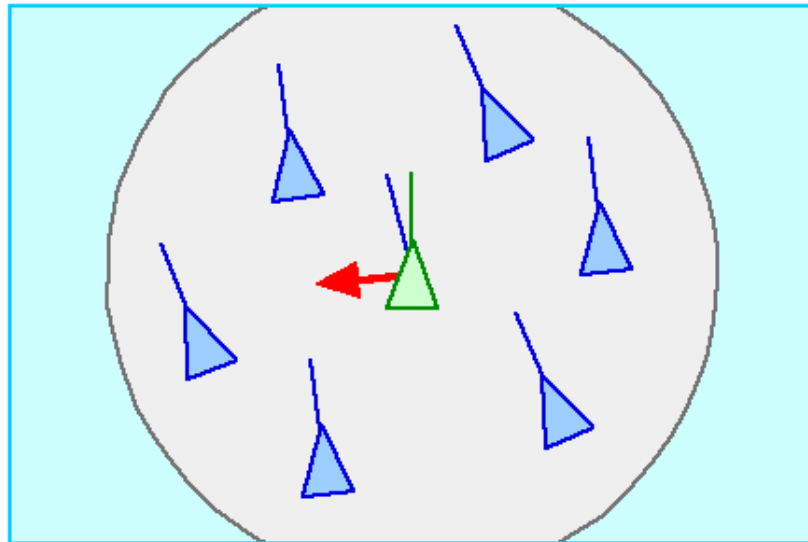
    float characterOrientation = transform.rotation.eulerAngles.y *
                                Mathf.Deg2Rad;
    float targetOrientation = wanderOrientation +
                                characterOrientation;

    Vector3 targetPosition = transform.position + (wanderOffset *
                                                  OrientationToVector(characterOrientation));
    targetPosition += wanderRadius *
                    OrientationToVector(targetOrientation);
    steering.linear = targetPosition - transform.position;
    steering.linear.Normalize();
    steering.linear *= steeringbase.maxAcceleration;

    return steering;
}
}
```

# Alignment Behavior

- Alignment behavior gives an character the ability to align itself with (that is, head in the same direction and/or speed as) other nearby characters.
  - Can be implemented by finding all characters in the local neighborhood and averaging together the velocity of the nearby characters.



# Alignment Behavior

```
public class AlignmentBehavior : Steering
{
    private Transform[] targets;
    [SerializeField] private float alignDistance = 8f;

    void Start() {
        SteeringBehaviorBase[] agents =
            FindObjectsOfType<SteeringBehaviorBase>();
        targets = new Transform[agents.Length - 1];
        int count = 0;
        foreach (SteeringBehaviorBase agent in agents) {
            if (agent.gameObject != gameObject) {
                targets[count] = agent.transform;
                count++;
            }
        }
    }

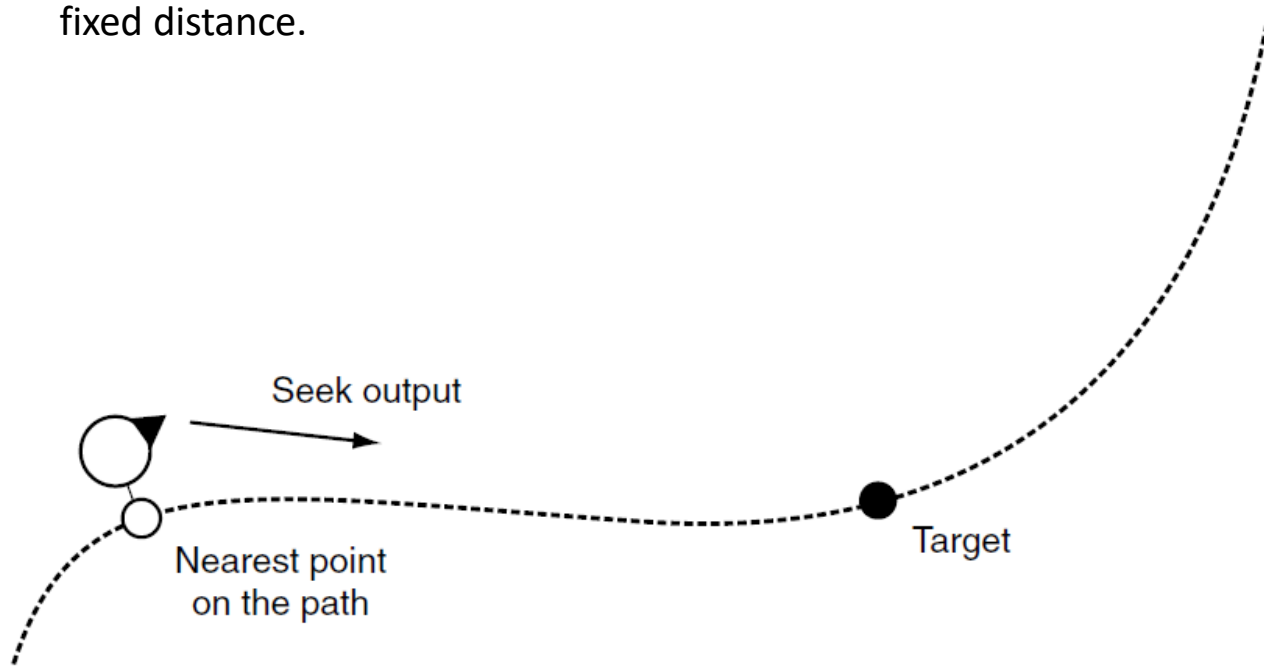
    ...
}
```

# Alignment Behavior

```
public override SteeringData GetSteering(SteeringBehaviorBase
                                         steeringbase) {
    SteeringData steering = new SteeringData();
    steering.linear = Vector3.zero;
    int count = 0;
    foreach (Transform target in targets) {
        Vector3 targetDir = target.position - transform.position;
        if (targetDir.magnitude < alignDistance) {
            steering.linear += target.GetComponent<Rigidbody>().velocity;
            count++;
        }
    }
    if (count > 0) {
        steering.linear = steering.linear / count;
        if (steering.linear.magnitude > steeringbase.maxAcceleration) {
            steering.linear = steering.linear.normalized *
                steeringbase.maxAcceleration;
        }
    }
    return steering;
}
```

# Path Following Behavior

- Path following behavior takes a whole path as a target. The character moves along the path in one direction.
  - The target position is calculated in two stages:
    1. The current character position is mapped to the nearest point along the path;
    2. A target is selected which is further along the path than the mapped point by a fixed distance.



# Path Following Behavior

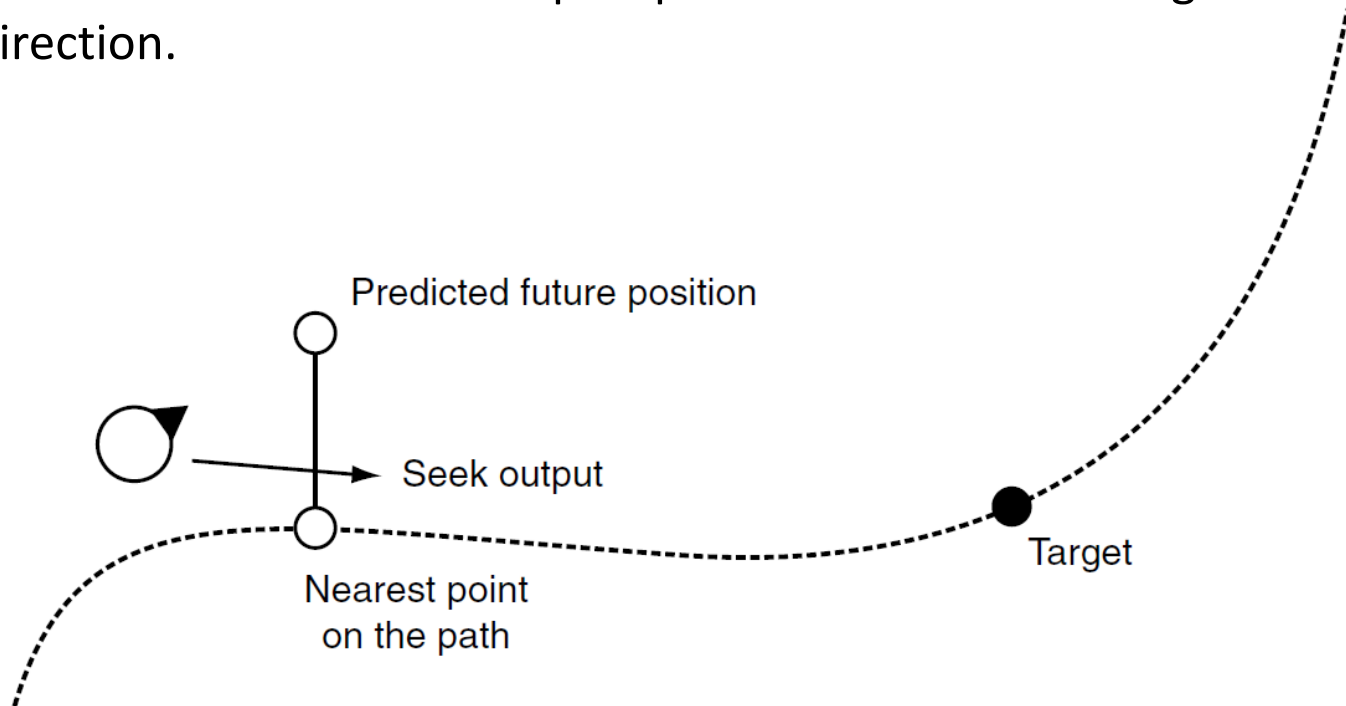
```
public class FollowPathBehavior : Steering{
    [SerializeField] private PathLine path;
    [SerializeField] private float pathOffset = 0.71f;
    private float currentParam = 0;

    public override SteeringData GetSteering(SteeringBehaviorBase
                                             steeringbase){
        SteeringData steering = new SteeringData();
        Vector3 targetPosition;
        if (path.nodes.Length == 1)
            targetPosition = path.nodes[0];
        else{
            currentParam = path.GetParam(transform.position);
            float targetParam = currentParam + pathOffset;
            targetPosition = path.GetPosition(targetParam);
            steering.linear = targetPosition - transform.position;
            steering.linear.Normalize();
            steering.linear *= steeringbase.maxAcceleration;
        }
        return steering;
    }
}
```

PathLine class: <http://www.inf.puc-rio.br/~elima/game-ai/PathLine.cs>

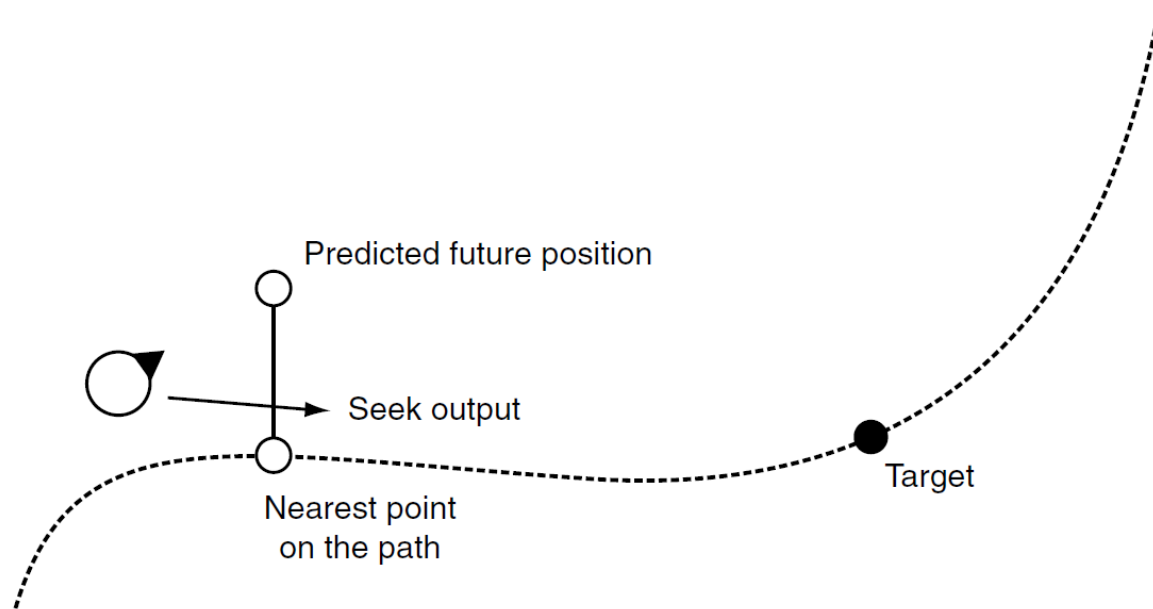
# Path Following Behavior

- In an alternative implementation, we first predict where the character will be in a short time and then map this to the nearest point on the path.
  - Smoother solution for complex paths with sudden changes of direction.



# Exercise 3

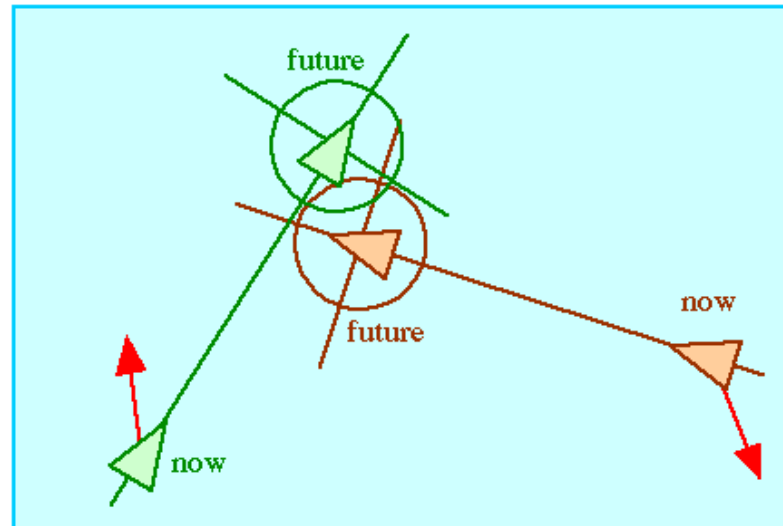
- 3) Based on the first implementation of the Path Following behavior, implement the alternative version of the behavior.
- First predict where the character will be in a short time and then map this to the nearest point on the path.





# Collision Avoidance Behavior

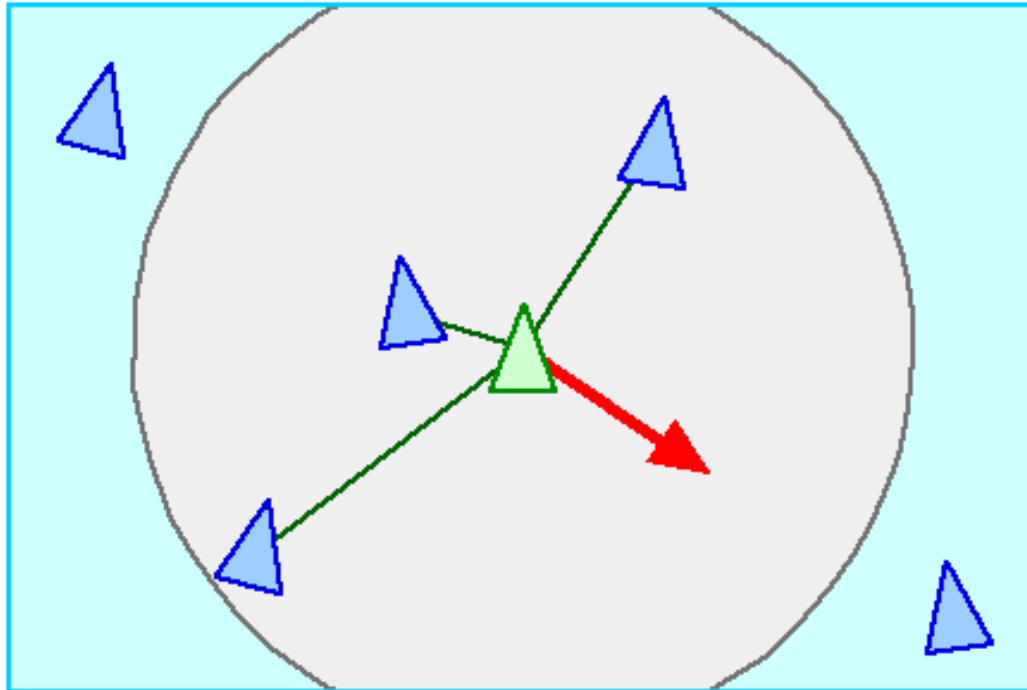
- Collision avoidance behavior tries to keep characters (which are moving in arbitrary directions) from running into each other.
  - The character must consider all other characters to determine (based on current velocities) when and where the two will make their nearest approach.



Code: <http://www.inf.puc-rio.br/~elima/game-ai/CollisionAvoidanceBehavior.cs>

# Separation Behavior

- Separation behavior 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 Behavior

```
public class SeparationBehavior : Steering{

    private Transform[] targets;
    [SerializeField] private float threshold = 2f;
    [SerializeField] private float decayCoefficient = -25f;

    void Start(){
        SteeringBehaviorBase[] agents =
            FindObjectsOfType<SteeringBehaviorBase>();
        targets = new Transform[agents.Length - 1];
        int count = 0;
        foreach (SteeringBehaviorBase agent in agents){
            if (agent.gameObject != gameObject){
                targets[count] = agent.transform;
                count++;
            }
        }
    }
    ...
}
```

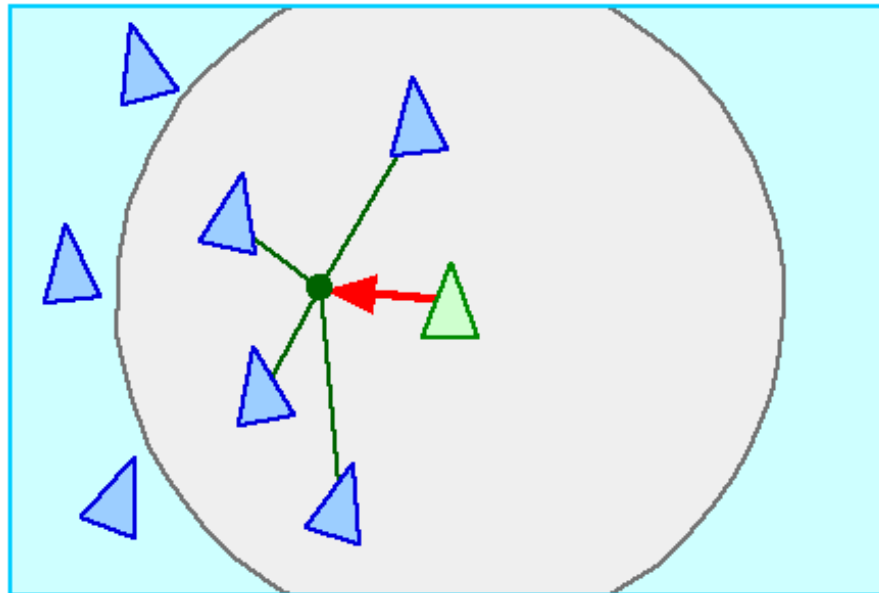
# Separation Behavior

```
public override SteeringData GetSteering(SteeringBehaviorBase
                                         steeringbase) {
    SteeringData steering = new SteeringData();
    foreach (Transform target in targets) {
        Vector3 direction = target.transform.position -
                               transform.position;

        float distance = direction.magnitude;
        if (distance < threshold) {
            float strength = Mathf.Min(decayCoefficient / (distance *
                                                           distance), steeringbase.maxAcceleration);
            direction.Normalize();
            steering.linear += strength * direction;
        }
    }
    return steering;
}
```

# Cohesion Behavior

- Cohesion behavior gives an character the ability to form a group with other nearby characters.
  - Its implemented by computing the “average position” (or “center of gravity”) of the nearby characters and then setting it as the target for seek behavior.



# Cohesion Behavior

```
public class CohesionBehavior : Steering{

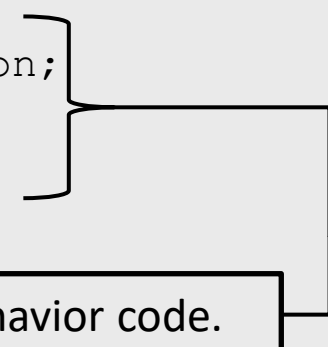
    private Transform[] targets;
    [SerializeField] private float viewAngle = 60f;

    void Start(){
        SteeringBehaviorBase[] agents =
            FindObjectsOfType<SteeringBehaviorBase>();
        targets = new Transform[agents.Length - 1];
        int count = 0;
        foreach (SteeringBehaviorBase agent in agents){
            if (agent.gameObject != gameObject){
                targets[count] = agent.transform;
                count++;
            }
        }
    }

    ...
}
```

# Cohesion Behavior

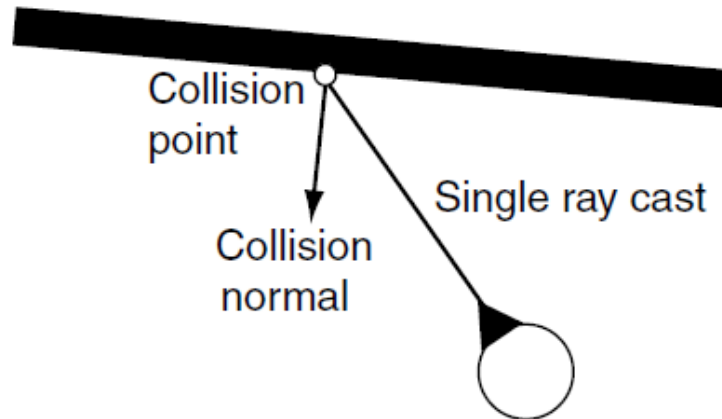
```
public override SteeringData GetSteering(SteeringBehaviorBase
                                         steeringbase) {
    SteeringData steering = new SteeringData();
    Vector3 centerOfMass = Vector3.zero;
    int count = 0;
    foreach (Transform target in targets) {
        Vector3 targetDir = target.position - transform.position;
        if (Vector3.Angle(targetDir, transform.forward) < viewAngle) {
            centerOfMass += target.position;
            count++;
        }
    }
    if (count > 0) {
        centerOfMass = centerOfMass / count;
        steering.linear = centerOfMass - transform.position;
        steering.linear.Normalize();
        steering.linear *= steeringbase.maxAcceleration;
    }
    return steering;
}
}
```



Seek behavior code.

# Obstacle Avoidance Behavior

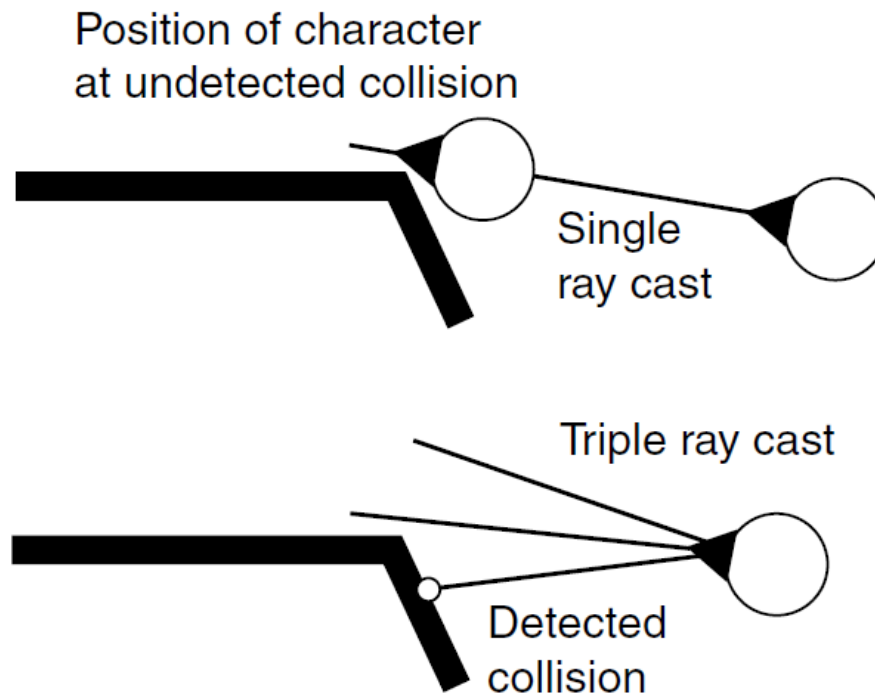
- Obstacle avoidance behavior gives a character the ability to maneuver in a cluttered environment by dodging around obstacles.
  - Implementation:
    - The moving character casts one or more rays out in the direction of its motion.
    - If these rays collide with an obstacle, then a target is created that will avoid the collision, and the character does a basic seek on this target.





# Obstacle Avoidance Behavior

- Number of rays:





# Obstacle Avoidance Behavior

```
rayVector[1] = new Vector3(Mathf.Cos(rightRayOrientation), 0,
                           Mathf.Sin(rightRayOrientation));
rayVector[1].Normalize();
rayVector[1] *= lookahead;
rayVector[2] = new Vector3(Mathf.Cos(leftRayOrientation), 0,
                           Mathf.Sin(leftRayOrientation));
rayVector[2].Normalize();
rayVector[2] *= lookahead;
for (int i = 0; i < rayVector.Length; i++){
    RaycastHit hit;
    if (Physics.Raycast(transform.position, rayVector[i], out hit,
                        lookahead)) {
        Vector3 target = hit.point + (hit.normal * avoidDistance);
        steering.linear = target - transform.position;
        steering.linear.Normalize();
        steering.linear *= steeringbase.maxAcceleration;
        break;
    }
}
return steering;
}
```

# Combining Steering Behaviors

- Individually, steering behaviors can achieve a good degree of movement sophistication. However, by combining steering behaviors together, even more complex movement can be achieved.
- There are two methods of combining steering behaviors: blending and arbitration.
  - Blending combines behaviors by executing all the steering behaviors and combining their results using a set of weights;
  - Arbitration selects one or more steering behaviors to have complete control over the character.

# Blending Behaviors

- We can modify the base Steering class in order to add a weight to each behavior:

```
public abstract class Steering : MonoBehaviour
{
    [SerializeField] private float weight = 1f;

    public abstract SteeringData GetSteering(SteeringBehaviorBase
                                             steeringbase);

    public float GetWeight(){
        return weight;
    }
}
```

# Blending Behaviors

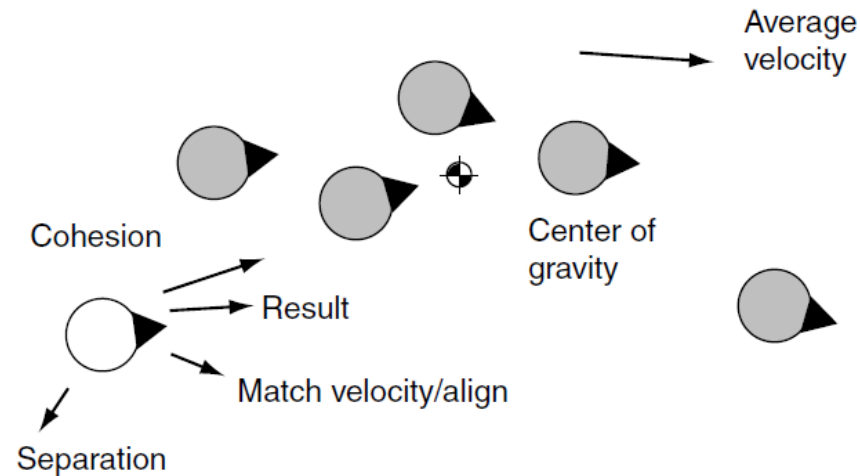
- The weights are used in the SteeringBehaviorBase class to compute how much each behavior affects the movement:

```
void FixedUpdate() {
    Vector3 acceleration = Vector3.zero;
    float rotation = 0f;
    foreach (Steering behavior in steerings) {
        SteeringData steering = behavior.GetSteering(this);
        acceleration += steering.linear * behavior.GetWeight();
        rotation += steering.angular * behavior.GetWeight();
    }
    if (acceleration.magnitude > maxAcceleration) {
        acceleration.Normalize();
        acceleration *= maxAcceleration;
    }
    rb.AddForce(acceleration);

    ...
}
```

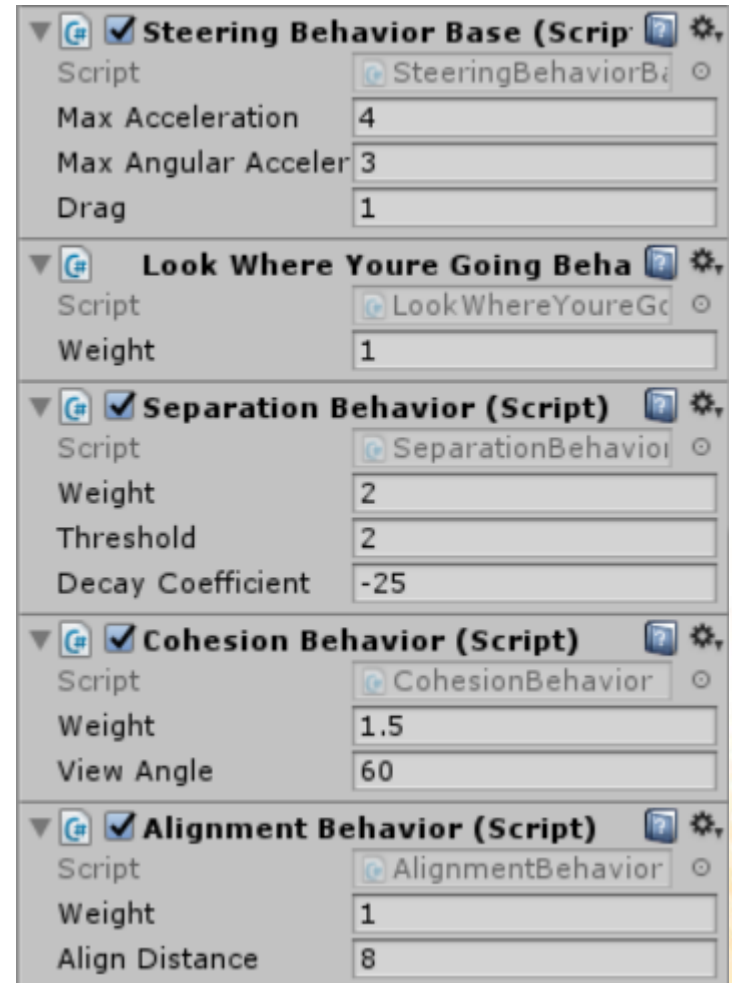
# Group Behavior: Flocking

- The flocking behavior relies on blending three simple steering behaviors:
  - Separation: move away from characters that are too close;
  - Alignment: move in the same direction and at the same velocity as the group;
  - Cohesion: move toward the center of mass of the flock.



# Group Behavior: Flocking

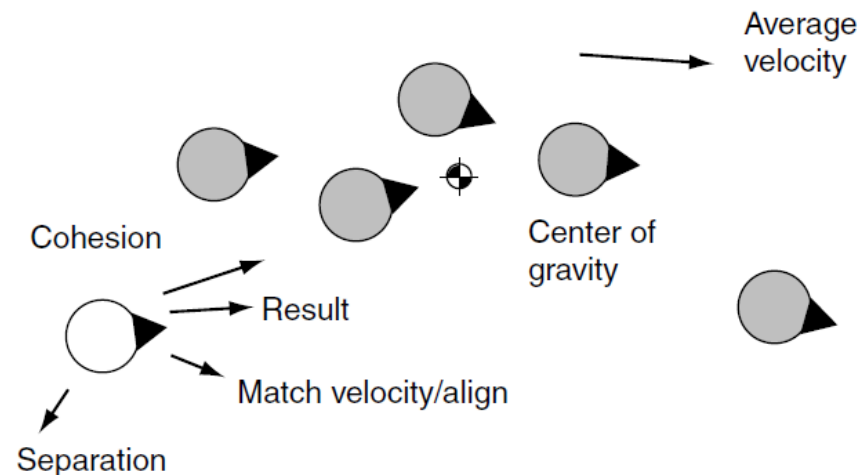
- In flocking behavior, separation is more important than cohesion, which is more important than alignment.
- The importance of each behavior can be established by setting the behaviors' weights.





# Exercise 4

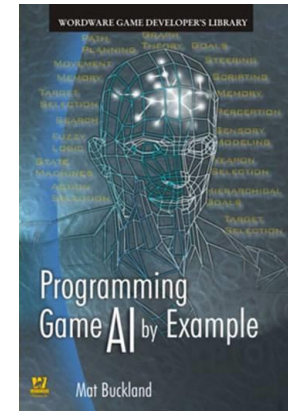
- 4) Create a scene with several characters and test the flocking behavior.
- Test different parameters to create a realistic flocking simulation.
  - Evaluate what happens when other steering behaviors are combined with the flocking behavior, such as Collision Avoidance and Wander.



# Further Reading

- Buckland, M. (2004). **Programming Game AI by Example**. Jones & Bartlett Learning. ISBN: 978-1-55622-078-4.

- **Chapter 3: How to Create Autonomously Moving Game Agents**



- Millington, I., Funge, J. (2009). **Artificial Intelligence for Games (2nd ed.)**. CRC Press. ISBN: 978-0123747310.

- **Chapter 3.3: Steering Behaviors**
- **Chapter 3.4: Combining Steering Behaviors**

