UNIVERSITÀ DEGLI STUDI
DI MILANO

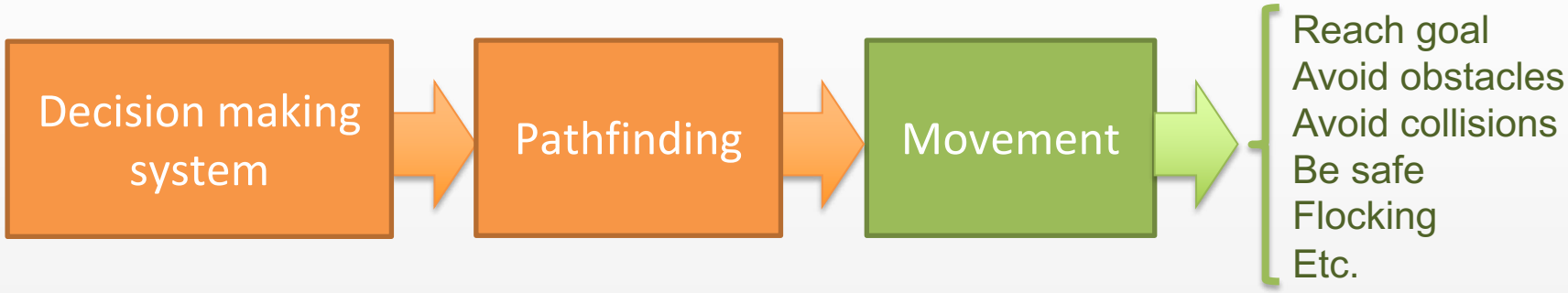
Flocking and Combining Steering Behaviors

A.I. for Video Games

Davie Gadia (gadia@di.unimi.it) and Dario Maggiorini (dario@di.unimi.it)

Combining steering behaviors

- A character generally needs MORE than one steering behaviour (SB)
- Thus, SB must be combined in some way



Blending & arbitration

- Executes all SBs
- Combine results using weights/priorities
- Problems when a lot of constraints are present

- Selects one or more SBs
- Give to the selected SB(s) complete control

Not exclusive approaches:
implemented solution is usually an in-between solution

- Weights/priorities can change in time
- Game state/internal knowledge play a role
- If weight = 0 → SB is switched off

- Arbitration may return a set of blending weights

Weighted blending

- Simplest way to achieve SBs combination
  - each SB has a weight
  - Final accelleration = weighted linear sum of single SB accellerations
  - No normalization → not a weighted mean

- Most known example: **Flocking**

Flocking, Flocking Everywhere

- Nature regroups crowds in flocks
- A good flock simulation will make your game much more lively

A herd of sheep

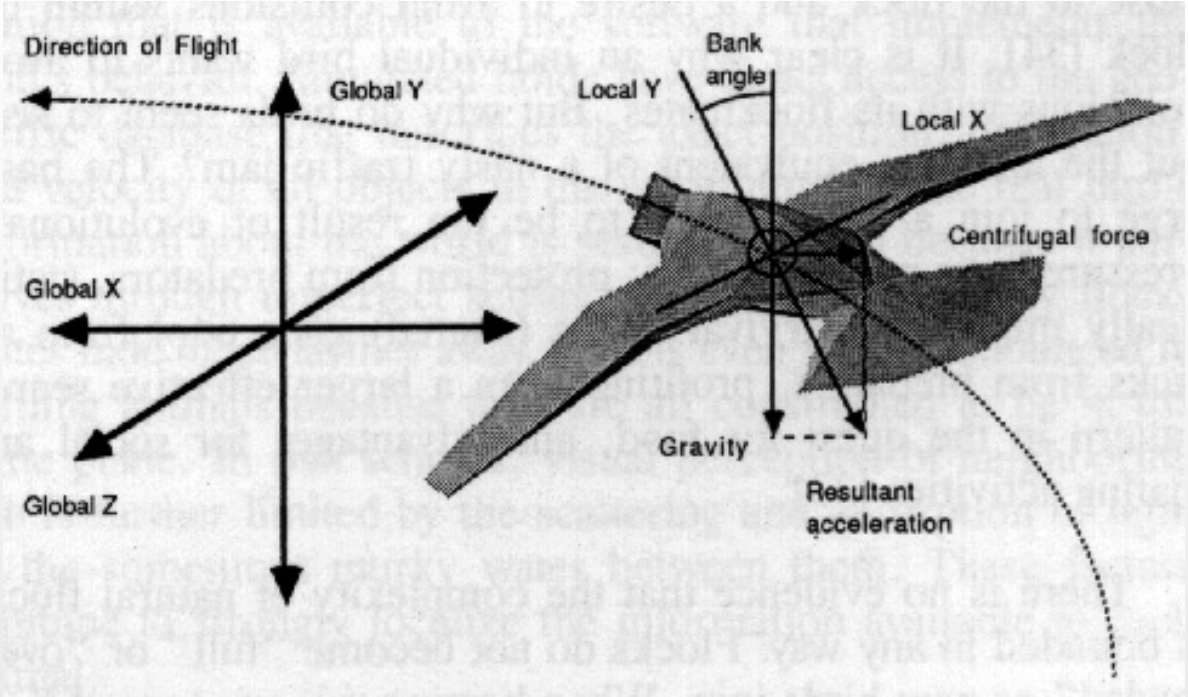
A school of fishes

A flock of birds

… Humans Included (!)

The Boids Algorithm

- Craig Raynolds, in 1986, proposed flocking as the blend of three steering behaviours
- His work was mainly intended for graphics simulation for flocks of birds (hence, the name)
  - Nevertheless, it has been applied to many environments



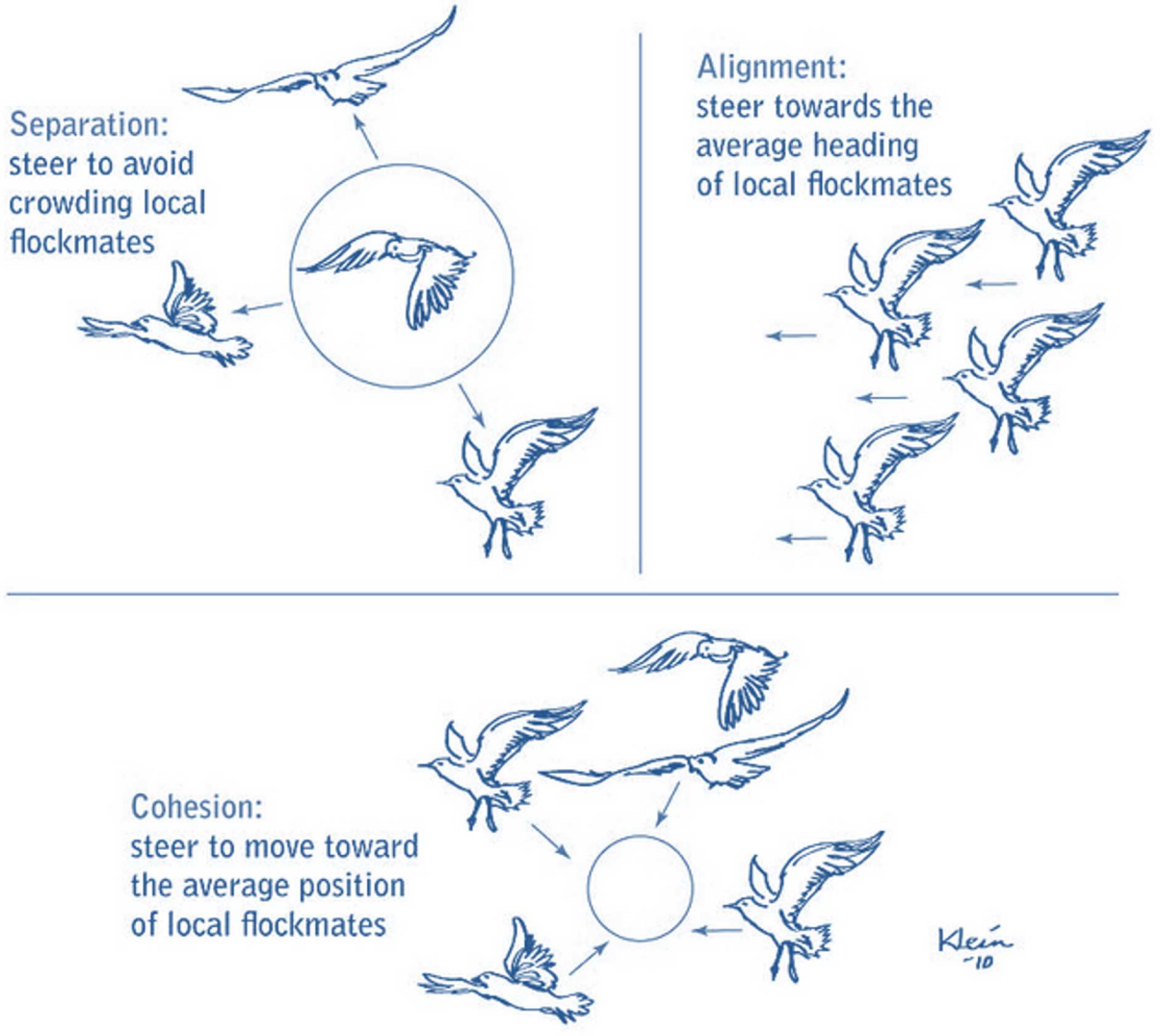
General Logic

- Each component is named a *boid* (from *bird*)
- A behaviour is defined for each boid
  - There is no behaviour defined for the whole group

- Actually, it will be a blend of three behaviours
  - Alignment
  - Cohesion
  - Separation

- The result will be an emerging (global) behaviour

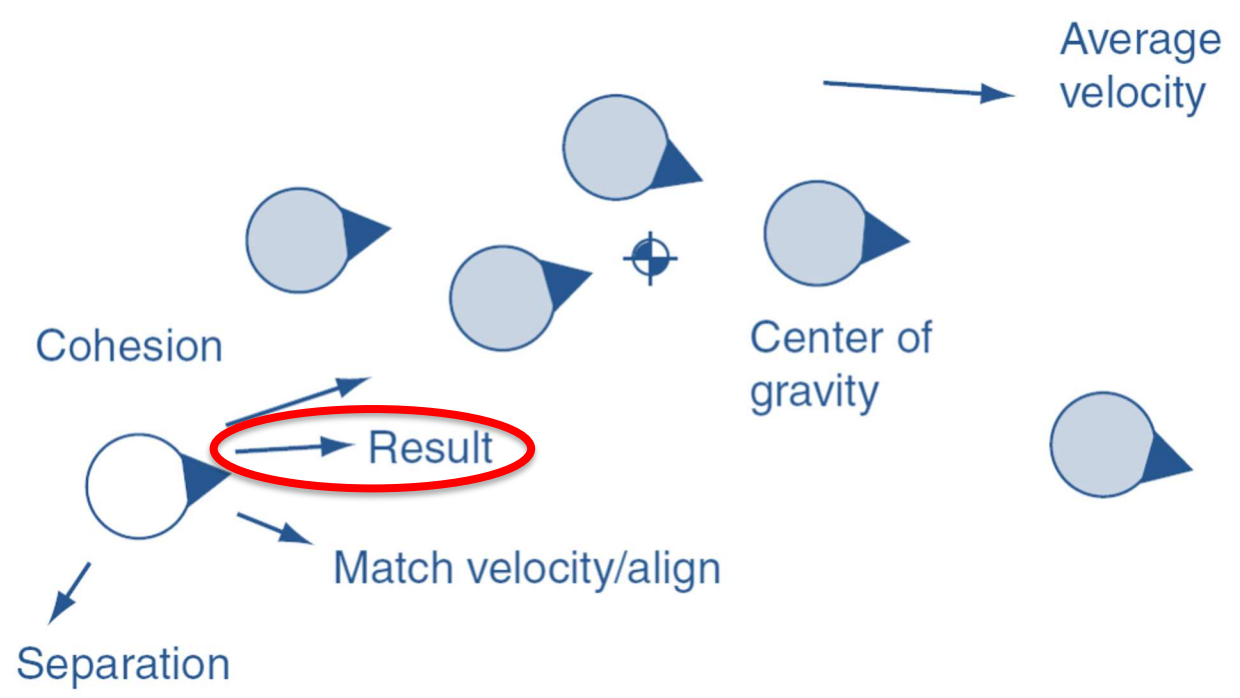
Separation, aligment and cohesion



Separation, aligment and cohesion

Flocking algorithm

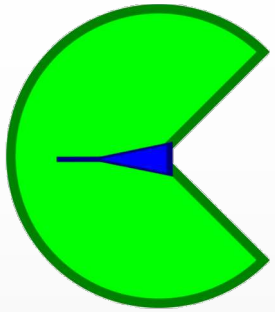
- Move away from too close boids
- Move in the same direction with same velocity
- Move toward centre of mass



- Simpler approach:
  - separation = cohesion = aligment = 1.0
- More realistic simulation:
  - separation > cohesion > aligment

Implementing Flocking

- What we need is a field of view for each boid
  - a sphere or, for a better simulation, it could be something more refined



Distant boids are ignored!

- Then, we must take into consideration all the neighboring boids and calculate components for the steering behaviour
  - Read the manual for *Physics.OverlapSphere* and you may be good

- Last, we blend everything as usual

You will only for small numbers!
Because the number of checks will scale as n^2 with the number of boids

For a more optimized computation, you may use some Spatial Data Structure (see in a following slide)

Before we Start

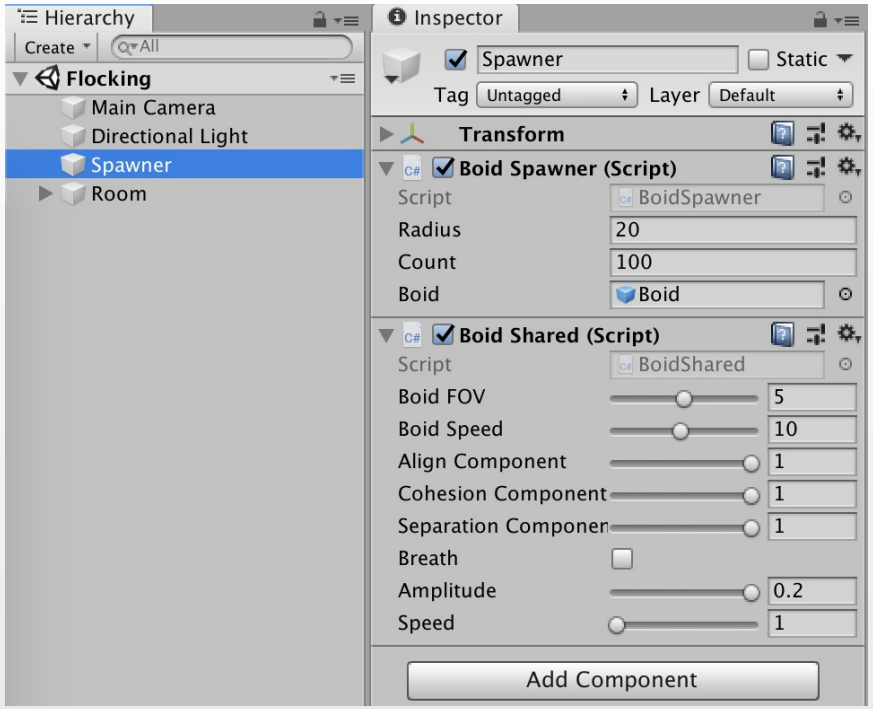
- Main problem in the presented implementation: Scalability
  - We may have hundreds of boids around
  - Every boid will check all its neighbors
    - As a matter of fact, we do not know a priori if two boids are neighbors
      So, every boid is checking all other boids at every update!

- AVOID (at all costs)
  - Allocating and de-allocating memory
    - Garbage collector will kill your framerate at random points
  - Computationally complex operations
    - Such as strings comparison

Global Parameters

Source: BoidSpawner & BoidShared
Folder: Flocking

- To control the system, we can put two components with static fields in an empty object
  - The value of these fields will be available to all boids at once



How many boids to spawn and where to spawn them (distance from the spawner)

Field of view

Weight for each behaviour

Give a sine shrink/enlarge effect to the flock for a more natural result (enable with the flag)

Blending Revisited

Source: BoidBlending
Folder: Flocking

```
public abstract class BoidComponent : MonoBehaviour {
    public abstract Vector3 GetDirection (Collider[] neighbors, int size);
}

public class BoidBlending : MonoBehaviour {

    private Collider [] neighbors = new Collider [200];
    // this must be large enough

    void FixedUpdate () {

        Vector3 globalDirection = Vector3.zero;

        int count = Physics.OverlapSphereNonAlloc (transform.position, BoidShared.BoidFOW, neighbors);

        foreach (BoidComponent bc in GetComponents<BoidComponent> ()) {
            globalDirection += bc.GetDirection (neighbors, count);
        }

        if (globalDirection != Vector3.zero) {
            transform.rotation = Quaternion.LookRotation ((globalDirection.normalized + transform.forward) / 2f);
        }

        transform.position += transform.forward * BoidShared.BoidSpeed * Time.deltaTime;
    }
}
```

This abstract class let us find all boid components in a gameobject

We assume the vision is spherical, and we select neighboring boids using the physics subsystem

Recycle your memory!

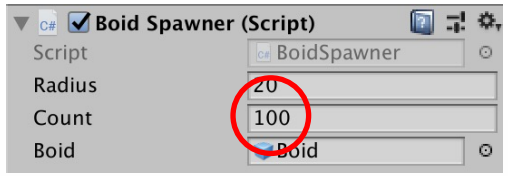
Collect all directions suggested by the satellite components

Avoid snapping.
At every update, a boid will turn only halfway toward the new direction calculated by the satellite components

Blending Revisited

Source: BoidBlending
Folder: Flocking

```
public abstract class BoidComponent : MonoBehaviour {
    public abstract Vector3 GetDirection (Collider[] neighbors, int size);
}

public class BoidBlending : MonoBehaviour {

    private Collider [] neighbors = new Collider [200];
    // this must be large enough

    void FixedUpdate () {

        Vector3 globalDirection = Vector3.zero;

        int count = Physics.OverlapSphereNonAlloc (transform.position, BoidShared.BoidFOW, neighbors);

        foreach (BoidComponent bc in GetComponents<BoidComponent> ()) {
            globalDirection += bc.GetDirection (neighbors, count);
        }

        if (globalDirection != Vector3.zero) {
            transform.rotation = Quaternion.LookRotation ((globalDirection.normalized + transform.forward) / 2f);
        }

        transform.position += transform.forward * BoidShared.BoidSpeed * Time.deltaTime;
    }
}
```



The size of this array must be larger than the number of generated boids

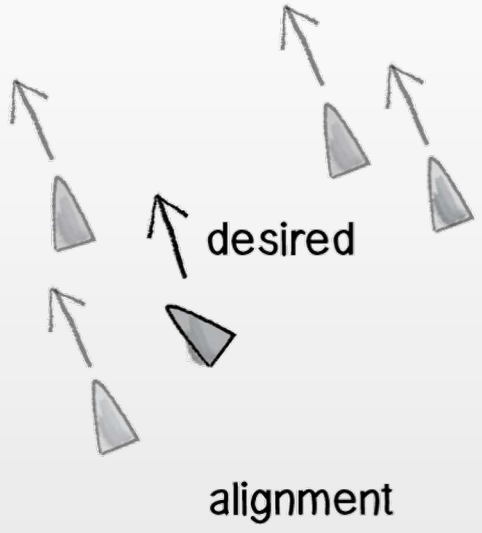
Align

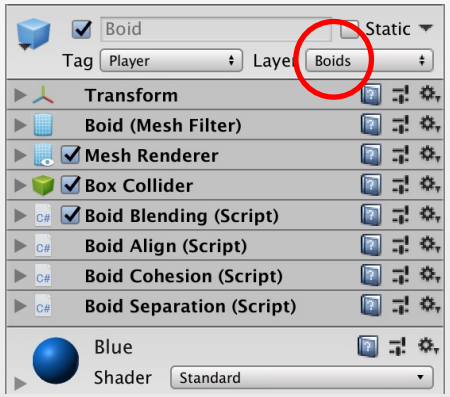
Source: BoidAlign
Folder: Flocking/Delegates

```
public class BoidAlign : BoidComponent {

    override public Vector3 GetDirection (Collider [] neighbors, int size) {

        Vector3 alignment = Vector3.zero ;
        for (int i = 0; i < size; i +=1) {
            if (neighbors[i].gameObject.layer == gameObject.layer) {
                alignment += neighbors [i].gameObject.transform.forward;
            }
        }
        return alignment.normalized * BoidShared.AlignComponent;
    }

}
```

Only align to object on the same layer (other boids).
Comparing tags is NOT a good idea, because they are strings

The requested direction is the “average forward direction” of all neighboring boids weighted by the global parameter





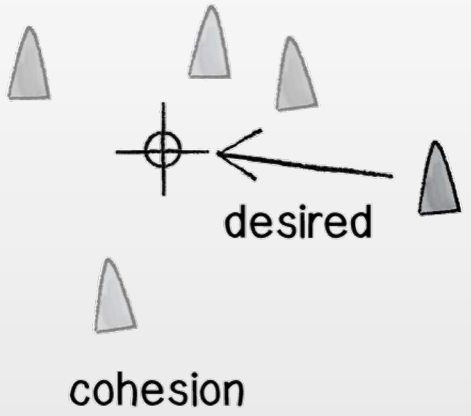
Cohesion

Source: BoidCohesion
Folder: Flocking/Delegates

```
public class BoidCohesion : BoidComponent {

    override public Vector3 GetDirection (Collider [] neighbors, int size) {
        Vector3 cohesion = Vector3.zero;
        float counter = 0f;
        for (int i = 0; i < size; i += 1) {
            if (neighbors [i].gameObject.layer == gameObject.layer) {
                cohesion += neighbors [i].transform.position;
                counter += 1f;
            }
        }
        cohesion /= (float) counter;
        cohesion -= transform.position;
        return cohesion.normalized * BoidShared.CohesionComponent;
    }
}
```

The requested direction is toward the geometric center described by all positions of the neighboring boids.

To find this center, we just average all the positions



Separation

Source: BoidSeparation
Folder: Flocking/Delegates

```
public class BoidSeparation : BoidComponent {

    override public Vector3 GetDirection (Collider [] neighbors, int size) {

        Vector3 separation = Vector3.zero;
        Vector3 tmp;
        for (int i = 0; i < size; i += 1) {
            tmp = (transform.position - neighbors [i].ClosestPointOnBounds (transform.position));
            separation += tmp.normalized / (tmp.magnitude + 0.0001f);
        }
        return separation.normalized * BoidShared.SeparationComponent;
    }
}
```

In this case we do not check the collider to be in the same layer, because we must avoid also obstacles (such as walls)

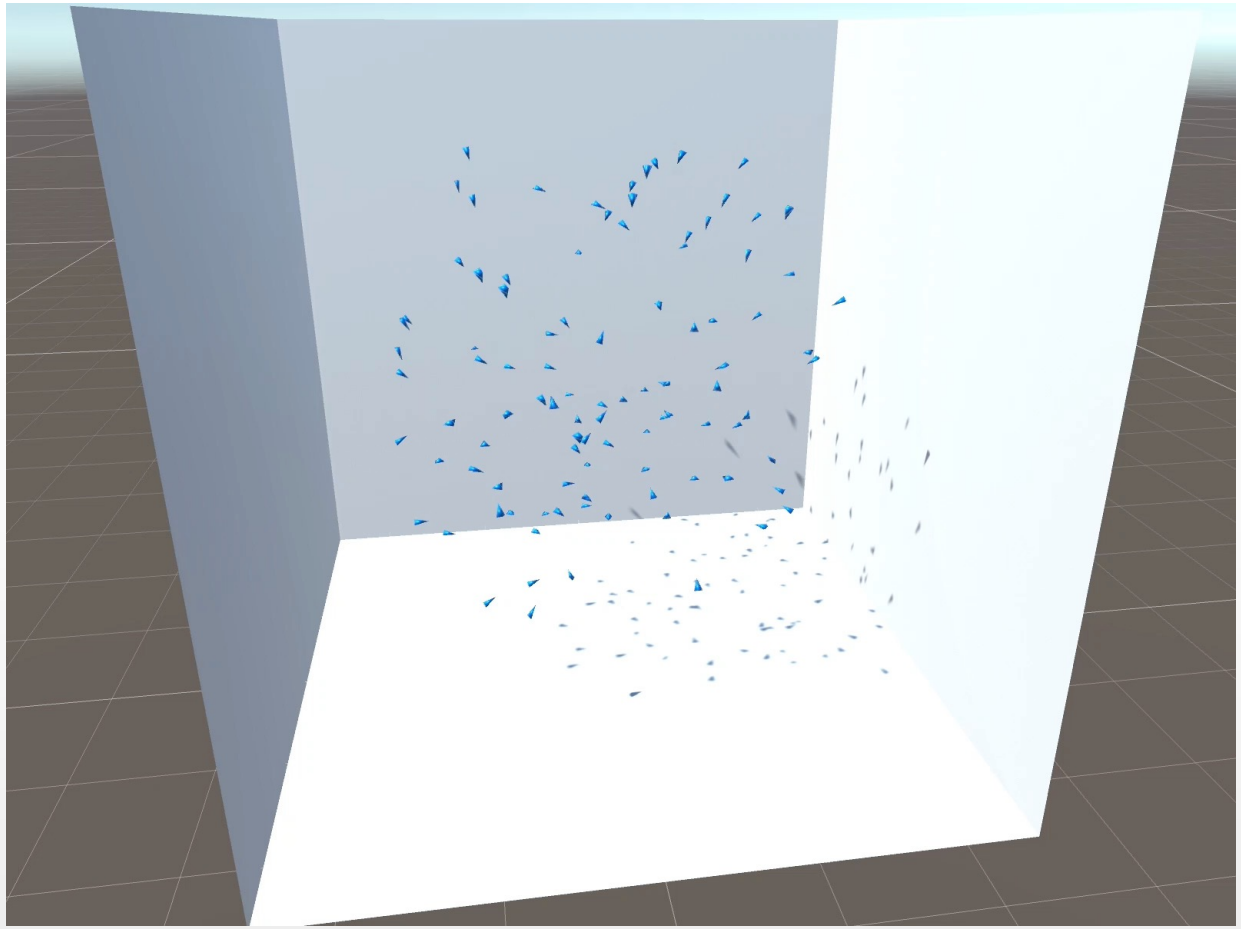
The closest we are, the more we get repulsed

We get repulsed by all neighboring colliders based on the closest mesh point. This will avoid weird behaviors if we have a very large object nearby, like a large wall repulsing us from its geometric center.

Please, mind the sign in the vectors difference

separation

Final Result



Beware of the Parameters!

- Cohesion and separation cannot diverge too much
  - Otherwise, you will just get chaos or everything collapsing
- Separation must not be too low
  - Otherwise, a boid getting closer to a wall will get a larger contribution from align and cohesion (from the other boids) than separation (from the wall). The result will be the boid *punching* through the wall and escaping the containment room
    - This can be fixed giving a different weight to separation from the wall and separation from the other boids
- View distance for each boid should (must?) be smaller than the average swarm radius
  - Otherwise, everything will just move in sync, like a platoon

How to Make a Swarm More Realistic

1. Change cohesion and separation amplitudes over time

```
private void Update () {
    if (breath) {
        float c = 1f - ((Mathf.Cos (Time.realtimeSinceStartup * speed) + 1) * amplitude / 2f);
        float s = 1f - ((Mathf.Sin (Time.realtimeSinceStartup * speed) + 1) * amplitude / 2f);
        CohesionComponent = _CohesionComponent = c;
        SeparationComponent = _SeparationComponent = s;
    }
}
```

2. Set different parameters for each boid (in a range)

3. Add some noise

Possible Improvements - 1

- Flocking/swarming techniques are computationally expensive
  - high number of characters to manage
  - performances depends on number of characters, and computational capabilities of the gaming device

- To optimize the search of neighbors, Spatial Data Structures can be adopted
  - E.g., grids, octree,….
  - in advanced collision management systems, this is called *broadphase stage*

Possible Improvements - 2

- Move from Object-Oriented to **Data-Oriented** approach
  - all the characters share several characteristics and data
  - separation and organization of the data from the code
  - exploit of multi-thread jobs to transform the data

- Unity has introduced the Entity Component System (ECS)
  - https://bit.ly/3nIZ8Sj
  - part of a rebuild of the Unity core following the Data-oriented approach

Unity ECS

- Beware: it is still a WIP (most of the packages are in Preview)
  - **(very) advanced topic**: it requires a complete refactoring of your code
    - *pure* ECS do not use GameObjects, Prefabs, ''classic'' Physics
    - *pure* ECS requires a different Physics library, and a heavy use of Jobs and Burst compiler (to create optimized multi-threading native machine code on the target platform)
    - However, there are some *hybrid* ECS solutions to keep prefabs etc

- You can find an advanced Unity flocking example using ECS here:
  - https://bit.ly/3Fs2ChM
  - and here a talk explaining the approach: https://bit.ly/33x33Kl

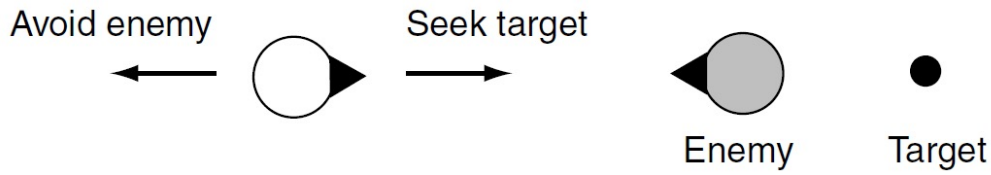
Weighted blending: problems …

- WB works well in open environments, but in urban areas/closed environments characters often get stuck

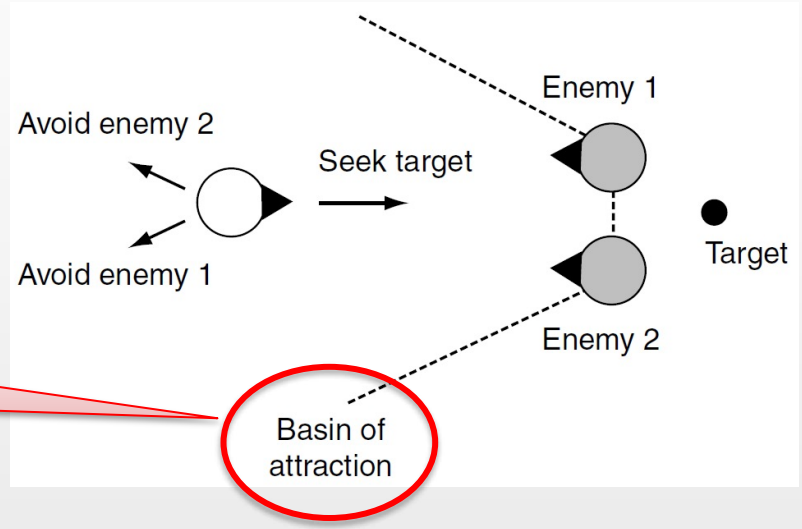
- Problems may arise from:
  - **Stable equilibria**
  - **Constrained environments**
  - **Nearsightedness**

Weighted blending issues: (un)stable equilibria

- **Unstable equilibrium**
  - Numerical instability will sort out the equilibrium



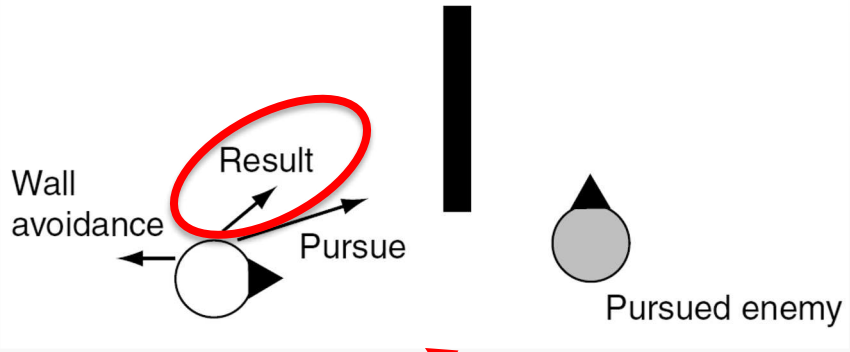
- **Stable equilibrium**
  - No escape, character trapped into a **basin of attraction**

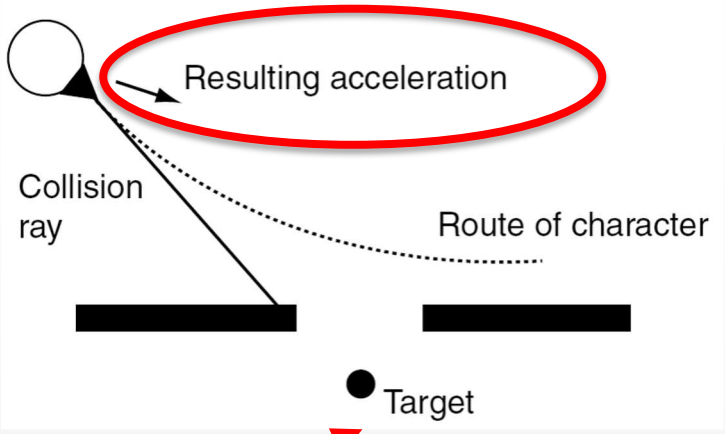


Difficult to visualize and debug!

Weighted blending issues: constrained environments

- SBs (with/out WB) work well with no/few constraints

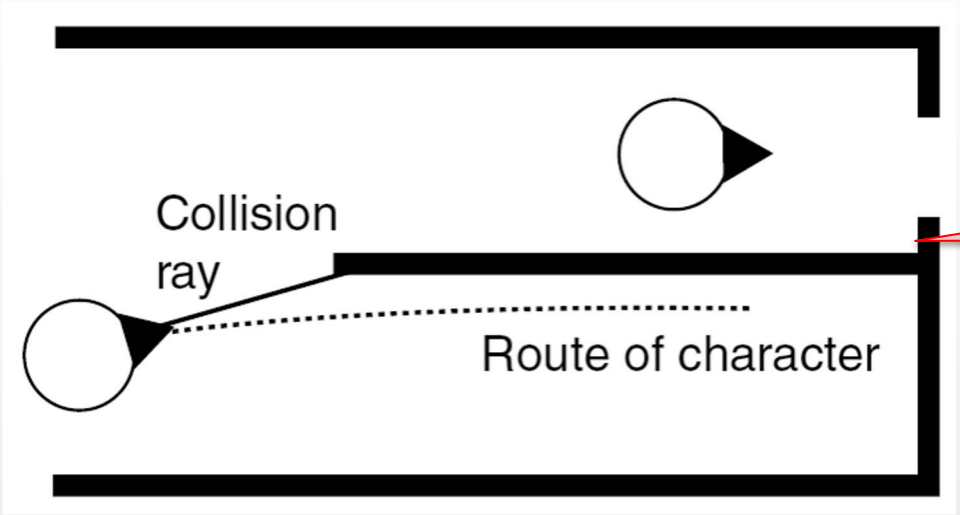




The problem is so complex that the «generally adopted» solution is:
design levels with wide passages … !

Weighted blending issues: nearsightedness

- SBs act locally: decisions are made on the immediate surroundigs
  - unable of tactic/strategic decisions → may take wrong course of action to reach their goal



No possible solution

Workaround: incorporate **pathfinding** into SBc

Behaviour blending: priorities

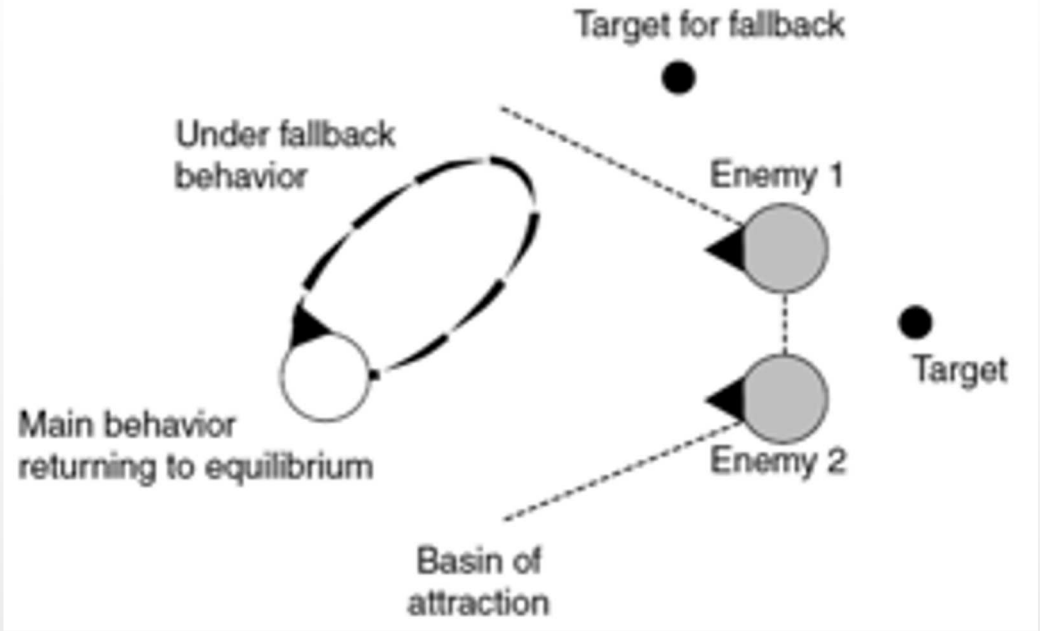
- Several SBs produce an accelleration only in few cases, but with high priority
  - e.g., collision avoidance
  - → they should be handled immediately

- Use **priorities** together with weights:
  1. Behaviours are arranged in **groups** with regular blending weights, then the **groups are ordered by priority**
  2. Steering systems considers each group in turn and blends behaviours in the same group, if the result is $\cong 0$ is ignored and next group considered
  3. First group with a result > 0 is used to steer the character

Behaviour blending: priorities

- Priority-based approach copes well with stable equilibria:
  - add a low priority fallback behaviour (e.g., wandering) to break equilibria

Still issues with large stable equilibria:
if the fallback behaviour has not moved the character out of the basin → the following higher level behaviour will drag it into equilibrium again



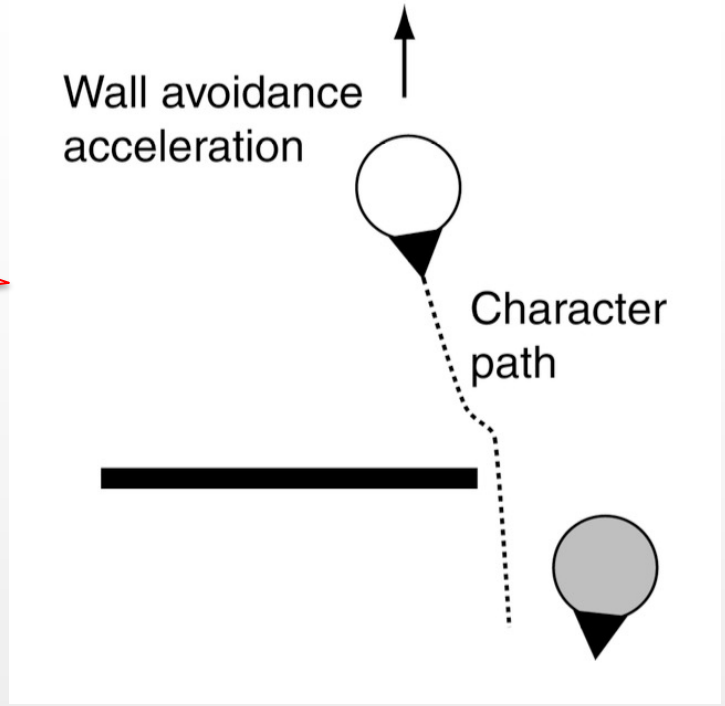
Cooperative arbitration

- Approaches based into **cooperation among different behaviours**

Example: Chasing (Pursue) + Avoid Collision

- when imminent → avoidance takes precedence, and character accelerates away from the wall
- Character slows dramatically near the wall, because avoidance behavior provides only a tangential accelleration

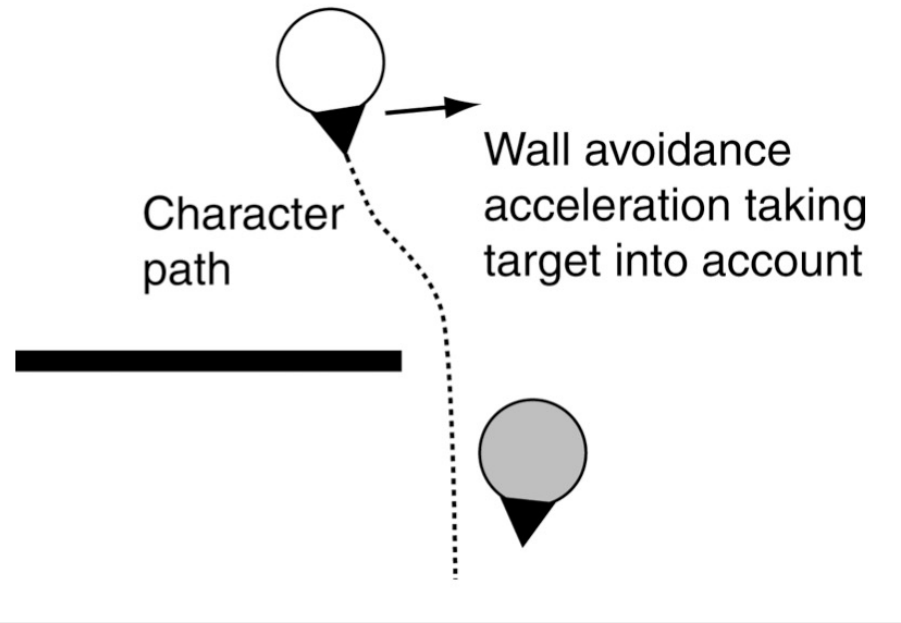
→ Behavior is not ‘’natural’’



Cooperative arbitration

- Approaches based into **cooperation among different behaviours**

- Avoidance should take into account what **Pursue** is trying to achieve
- **Context sensitive behaviours:**
  - returns an acceleration which takes both concerns into account
  - Implementation based on **decision making** (e.g., DT, FSM)



References

- On the book
  - § 3.4 (excluded 3.4.5)
- Original paper from Craig Reynolds
  - C. W. Reynolds, “Flocks, Herds, and Schools: A Distributed Behavioral Model”, in Computer Graphics, 21(4), SIGGRAPH '87 Conference Proceedings, pages 25-34
  - You can find a copy here: https://stanford.io/3tkOLaJ