Moving Intelligently (1): Steering Behaviours

CE811 Game Artificial Intelligence

Dr Michael Fairbank

1st November 2021

University of Essex

Outline

Introduction

Moving Intelligently

- We need our characters to **move** around our environments
- Ideally, we'd like to move in ways that make sense
- We'd also like them to look natural
- This Lecture is in two parts:
  1. Steering behaviours (This lecture)
  2. Pathfinding (Next lecture)

Outline

3D Vectors

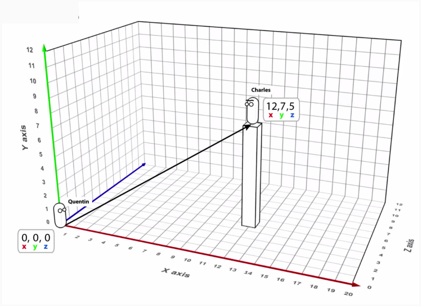
A vector is a geometric object that has **magnitude** (or **length**) and **direction**.

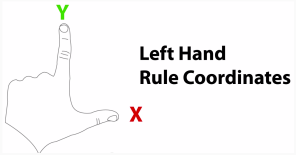
We need vectors in games: to track the position of players and NPCs, and also their velocity and acceleration vectors.

Velocity, accleration, position and displacement are all vectors.

Scalars are quantites that have magnitude only (no direction). Speed and distance are scalars.

3D Vectors





Vectors can be used to represent a **point in space** or a **direction with a magnitude**. 3D vectors are composed of three components, (x, y, z). For vectors starting at the origin, these represent the distance from $(0, 0, 0)$ along the X, Y and Z axis.

2D Vectors

- In 2D, It is convenient to define two fixed vectors: Up $\vec{U} = (0.0, 1.0)$ and Right $\vec{R} = (1.0, 0.0)$.

Length

The length (or magnitude) of a vector is determined by its norm, calculated as $magnitude = \|vec\| = \sqrt{x^2 + y^2 + z^2}$.

Definitions:

1. Speed is the magnitude of velocity.
2. Distance is the magnitude of displacement.

Example: if the velocity vector is $(2, 5, 0)$, the object is moving 2 units per second in the X axis, and 5 units in the Y axis. The magnitude of the vector $\sqrt{2^2 + 5^2 + 0^2} = \sqrt{29} = 5.385$ is the linear speed of the object.

Operations on Vectors

Addition:

$$(1,3,-1)+(2,2,0)=(3,5,-1)$$

$$a+b \equiv b+a$$

Subtraction:

$$(4,3,5)-(3,1,6)=(1,2,-1)$$

$$a-b \not\equiv b-a$$

Scalar multiplication:

$$x(2,3,1)=(2,3,1)x=(2x,3x,1x)$$

$$2(2,3,2)=(4,6,4)$$

Negation:

$$a=(3,0,-1), -a=(-3,0,1)$$

$$b=(2,-2,3), -b=(-2,2,-3)$$

Dot product:

$$a \cdot b = b \cdot a = \sum_{i=1}^{n} a_i b_i$$

$$a \cdot b = |a||b|\cos(\theta)$$

Cross product:

$$a \times b = |a||b|\sin(\theta)n$$

$$a \times b \equiv -(b \times a)$$

n is the unit vector perpendicular to the plane containing *a* and *b*.

Unit Vectors

A **Unit Vector** is any vector with a length of 1

- Normally unit vectors are used simply to indicate direction.

A vector of arbitrary length can be divided by its length to create a unit vector.

- This is known as **normalizing** a vector.
- $\frac{\vec{v}}{||\vec{v}||}$ is a normalised version of vector $\vec{v}$

Dot Product

The **Dot Product** operation between two vectors $\vec{a} = (a_1, a_2, a_3)$ and $\vec{b} = (b_1, b_2, b_3)$ is defined as

$$\begin{aligned} \vec{a} \cdot \vec{b} &= \|\vec{a}\|\|\vec{b}\| cos(\theta) && (1) \\ &= a_1 b_1 + a_2 b_2 + a_3 b_3, && (2) \end{aligned}$$

where $\|\vec{a}\|$ and $\|\vec{b}\|$ are the magnitudes of $\vec{a}$ and $\vec{b}$, and θ is the angle between these two vectors.

The dot product is sometimes called the **inner product**, or, since its result is a scalar, the **scalar product**.

Q: Which definition of the dot product will be easier to program? (1) or (2)?

Dot Product

The **Dot Product** is a useful operation that allows us to determine the angle between two vectors:

$$\begin{aligned} cos(\theta) &= \frac{\vec{a}.\vec{b}}{\|\vec{a}\| \ \|\vec{b}\|} && \text{(by (1))} \\ &= \frac{a_1 b_1 + a_2 b_2 + a_3 b_3}{\|\vec{a}\| \ \|\vec{b}\|} && \text{(by (2))} \qquad (3) \end{aligned}$$

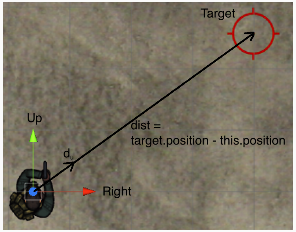
This is very useful to determine the amount of rotation needed to face certain position, angle of views, etc.

For normalized vectors, the dot product is 1 if they point in exactly the same direction; -1 if they point in completely opposite directions; and a number in between for other cases (e.g. Dot returns zero if vectors are perpendicular).

Relative Postion (Displacement Vectors)

Scenario: we want a vector pointing from our player to a target

- `this.position` is the position of the character $\vec{c} = (c_x, c_y)$.
- `target.position` is the position of the target $\vec{t} = (t_x, t_y)$.



- $\vec{d}$ is the displacement vector from the character to the target:

$$\begin{aligned}\vec{d} &= \vec{t} - \vec{c}\\ &= (t_x - c_x, t_y - c_y)\end{aligned}$$

- Tip: In constructing a vector **from** $\vec{c}$ to $\vec{t}$, replace ”from” with ”minus”, to get $-\vec{c} + \vec{t}$.

- $\|\vec{d}\|$ is the distance to the target.

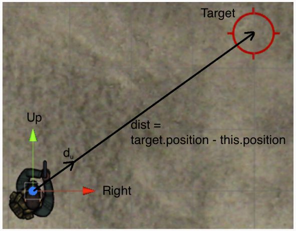
Orientation Vector

In 2D, if we have an angle θ (measured clockwise from North), and we want a unit vector pointing in the direction of θ, then we can use:

$$\vec{v} = \begin{pmatrix} \sin(\theta) \\ \cos(\theta) \end{pmatrix} \tag{4}$$

Orientation to a target

Scenario: the character in the image wants to change its orientation to face the target.



- As before $\vec{d}$ is the displacement vector **from** the player **to** the target
- $\hat{d} = \frac{\vec{d}}{||\vec{d}||}$ is the unit vector towards the target.
- The orientation angle (in radians clockwise from North) from the player to the target is `atan2(d_x, d_y)`.

Movement

Movement can be obtained by adding a velocity $\vec{v} = (v_x, v_y)$ to the current position $\vec{c} = (c_x, c_y)$, applied for a small time step Δt:

$$\vec{c}_{t+1} = \vec{c}_t + (\vec{v}_t \times \Delta t)$$

Here Δt is the time between two consecutive calls to an update function.

Similarly, if the agent has acceleration vector $\vec{a}$, then the velocity will update according to:

$$\vec{v}_{t+1} = \vec{v}_t + (\vec{a} \times \Delta t)$$

In pseudocode,

```
void Update() {
   this.position += this.velocity * deltaTime; // update the position.
   this.velocity += this.acceleration * deltaTime; // update the velocity.
}
```

We assume `Update()` is called at every game cycle.

Steering Behaviours

Steering Behaviours

- **Steering Behaviours** (by Craig W. Reynolds) aim to help our characters to move in a realistic manor
- They're based on applying simple forces to the character to produce life-like, improvisational navigation around an environment
- Unlike some of the other systems we'll be looking at, they're also pretty dumb…
- They're concerned with things that are local to them rather than some complex, path planning system.
- But they're still able to produce some pretty complex (and impressive) movements

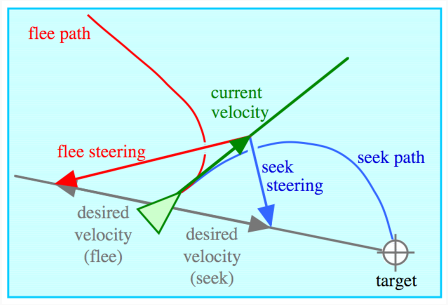
Steering Behaviour References

- These ideas were pioneered by Craig W. Reynolds in “Steering Behaviours for Autonomous Characters” in Game developers conference (1999, pp. 763-782): `https://www.red3d.com/cwr/papers/1999/gdc99steer.pdf`
- See also Millington, Ian, and John Funge. Chapter 3, **Artificial Intelligence for Games.** *CRC Press*, 2012.

Outline

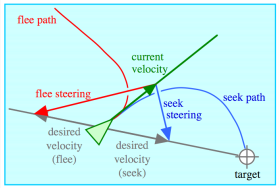
Seek

Seek (or pursuit of a static target) acts to steer the character towards a specified position in global space. This behavior adjusts the character so that its velocity is radially aligned towards the target. Note that this is different from an attractive force (such as gravity) which would produce an orbital path *around* the target point.



Seek Example

Seek (or pursuit of a static target):

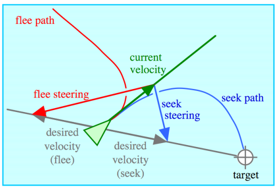


```
Vector2D seek(target) {
   //Desired velocity is the vector to the target.
   desiredVelocity = (target.position - this.position);

   // The desired velocity is along this direction, at full speed.
   desiredVelocity.Normalize ();
   desiredVelocity *= maxSpeed;
   return desiredVelocity;
}

void update_agent(desiredVelocity) {
   //The steering component to steer towards the target:
   desiredDeltaVelocity = desiredVelocity - this.currentVelocity;

   // Gradually change the current velocity towards the desired velocity
   // Here we are interpreting desiredDeltaVelocity as a force
   // and using acceleration=force/mass
   steeringAcceleration=desiredDeltaVelocity / this.mass
   if steeringAcceleration.length()>maxAcc:
      steeringAcceleration=steeringAcceleration.Normalise()*maxAcc

   // Apply the acceleration to the agent...
   this.currentPosition += this.currentVelocity * deltaT
   this.currentVelocity += steeringAcceleration * deltaT
}
```

Flee Example

Flee (or escape from a static target):

- Code for flee is the same as seek, but we the first line (highlighted) has its ± signs reversed.



```
Vector2D flee(target) {
   //Desired velocity is the vector to the target.
   desiredVelocity = (this.position - target.position);

   // The velocity is along this direction, at full speed.
   desiredVelocity.Normalize ();
   desiredVelocity *= maxSpeed;

   return desiredVelocity
}
```

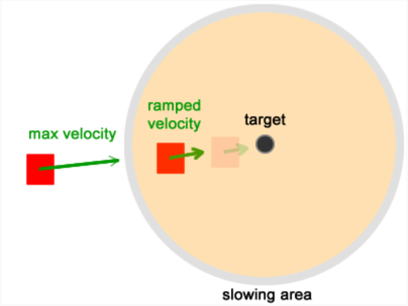
Arrival: Seeking and Stopping

Question
Why might we not want to use seek if we want to arrive at a target location?

Seek doesn’t slow down - it’ll overshoot and go through the target

Arrival

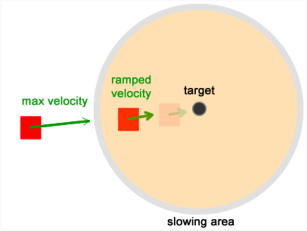
The **Arrival** behaviour is identical to Seek, but instead of moving through the target at full speed, the character will slow down when reaching the target.



- Velocity reduced linearly to 0 when inside the slowing area.
- Adding this effect will (eventually) turn character's velocity to 0.

Arrival

The **Arrival** behaviour is identical to Seek, but instead of moving through the target at full speed, the character will slow down when reaching the target.



```
void adjust_velocity_for_arrival(desired_velocity, target_position) {

   // Calculate distance to the target.
   targetDistance = (target_position - this.position).magnitude();
   targetSpeed=maxSpeed

   //Speed will be unaffected if beyond slowing down distance.
   // ...or less if inside the slowing area's radius.
   if targetDistance<arrivalSlowingDist {
       speedFraction= targetDistance/arrivalSlowingDist
       // rescale velocity vector by fraction:
       targetSpeed *= speedFraction
   }
   if targetDistance<arrivalStoppingDist {
       // if we get beyond a certain narrow distance to the target
       // (inner dark-grey circle in diagram) then stop completely
       targetSpeed = 0
   }
   return desired_velocity.Normalise()*targetSpeed
}
```

See Arrival demo by James Donald

Outline

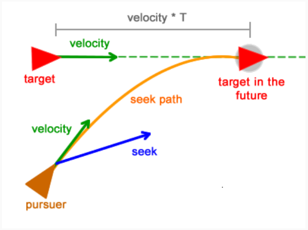
Catching things

Question
What about if our target is moving?

Easy	Leave seek as it is - inaccurate
Better	Predict where the target is going to be

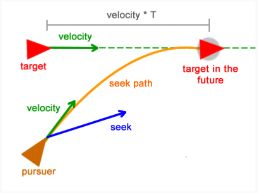
Pursuit

The **Pursuit** behaviour is equivalent to the *Seek* behaviour, but the target is moving.



- Linear velocity-based predictor: assumes target will not turn.
- This is fine: evaluated at every frame (error is small).
- *Seek* to a predicted position *T* frames ahead.
- *T* should decrease as target is closer.

Pursuit Example



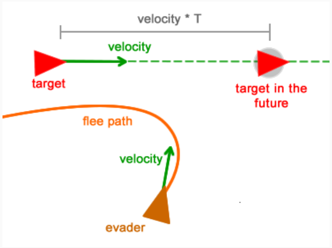
```
distance=(target.position-this.position).magnitude()
T=min(distance/maxSpeed,maxT)
advanced_target =
    target.position + target.velocity*T
desiredVelocity=seek(advanced_target)
```

- In this implementation, *T* is calculated dynamically based on the intercept distance.
- Hence *T* will reduce automatically as the target gap closes.
- Don’t want to anticipate too far ahead though, because the target might change direction (hence line 2 of the code above).

See Pursuit demo by James Donald

Evade

The **Evade** behaviour is equivalent to the *Flee* behaviour, but the target to flee from is moving (or to *Pursuit* but fleeing instead of seeking). No need to decrease *T* with distance!



Code for **Evade**:

```
advanced_target = target.position + target.velocity*T
flee(advanced_target)
```

See Evade demo by James Donald

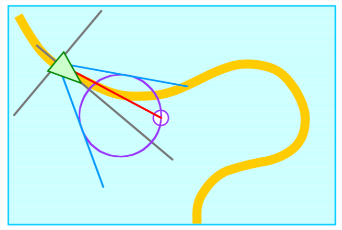
Wandering about

Question
What about just wandering about?

Easy	a different random steering component each frame.
Better	keep steering component, changing it randomly each frame.

Wander

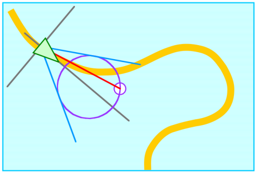
The **Wander** behaviour is a type of random steering.



- We want the wandering to be smooth (not jittery), so we let a small imaginary target circle drift randomly around the larger imaginary circle, and the agent always ”seeks” the small target circle.
- The large circle is a fixed radius and distance, always anchored directly in front of the agent - when the agent turns the large circle moves with it.
- Even if the small target circle jitters around the wider circle, the agent which is permanently in seek mode, shows smooth wandering motion.
- The larger the radius of the larger circle, the more extreme the steering of the agent, and the more pronounced the wandering.

Wander

The **Wander** behaviour is a type of random steering.



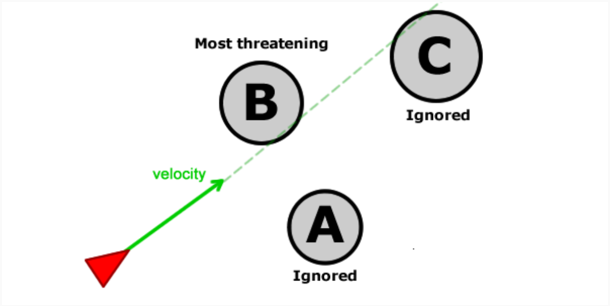
```
void wander() {
   // randomly move the target circle around the larger circle's perimeter
   // wanderOrientation is the angle of the small circle within the larger circle
   this.wanderOrientation += Random.Range (-1f, 1f)*wanderRate;

   // Calculate the combined target orientation.
   // The following 3 orientation variables are all angles (floats)
   float targetOrientation = this.wanderOrientation + character.orientation

   // calculate the centre of the large circle:
   // wanderOffset is the distance the large circle is in front of the agent.
   Vector largeCircleCentre = character.position
        + wanderOffset * character.orientation.asVector()

   // Calculate the target location.
   // wanderRadius is radius of larger circle
   Vector target = largeCircleCentre + wanderRadius * targetOrientation.asVector()

   // Seek towards the target small circle:
   seek(target)
}
```

For the Orientation.as_vector(), use equation (4)

Outline

Obstacle Avoidance

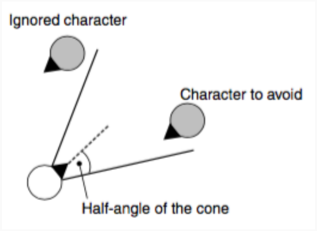
The basic idea behind **Obstacle** (or **Collision**) **Avoidance** is to generate a steering force to avoid colliding with other agents in the environment.



The Cone Approach

The idea:

- Simple: detect obstacles within a cone around the velocity of the agenct.
- Use the *Evade* behaviour.
- If there are several obstacles, evade the average of all predicted targets.



- We can compute the angle between the vector to a potential obstacle ($\vec{e}$) and the orientation vector of the agent $\vec{o}$.

$$cos(\theta) = \frac{\vec{e} \cdot \vec{o}}{||\vec{e}|| \ ||\vec{o}||} \tag{5}$$

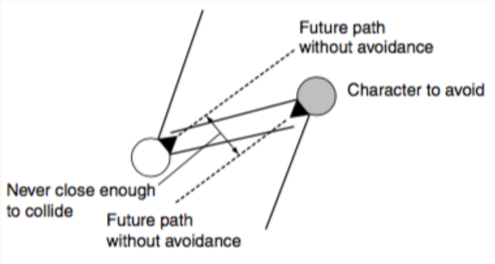
Cone Problems

Question
What might some limitations with this approach be?

The Cone Approach: Problems

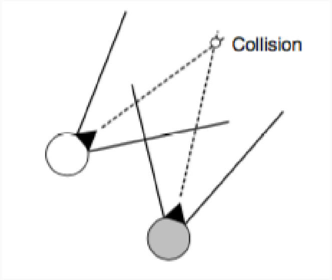
Two *minor* problems:

Problem 1:


Future path without avoidance
Character to avoid
Never close enough to collide
Future path without avoidance

We avoid things we won’t hit

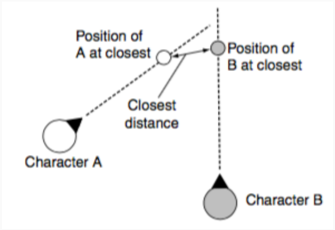
Problem 2:



We don’t avoid things we will hit

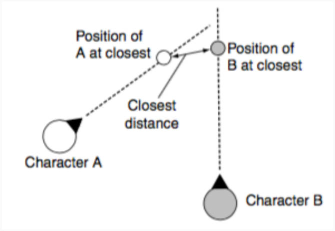
Improvement: Collision Prediction

To avoid these problems, we can do a look-ahead to check if agents really are going to collide: Assume that all characters keep their current velocity. We need then to find the point at which the characters will be the closest:



- If the closest distance is less than a threshold, a collision is predicted.
- Note that this point is different to the point where the trajectories cross: need to take velocity into account.

Improvement: Collision Prediction



- Mathematically, the time when two agents are at their closest is given by

$$t = -\frac{\vec{x}_{REL} . \vec{v}_{REL}}{\vec{v}_{REL} . \vec{v}_{REL}} \tag{6}$$

where $\vec{x}_{REL}$ is the objects' current relative positions, and $\vec{v}_{REL}$ is the objects' relative velocity.

Collision Prediction Example i

Compute the average point of those positions at the closest distance, and *Evade* it.

```
void CollisionPrediction(){
   //Relative positions and velocities.
   Vector3 relPosition = target.position - this.position;
   Vector3 relVelocity = target.velocity - this.velocity;

   //calculate time to the closest approach.
   float time = -DotProduct(relPosition, relVelocity) / DotProduct(relVelocity, relVelocity);
   if(time < 0.0) return; //There will be no collision.

   //This and target's predicted positions at the closest point.
   Vector3 posFuture = this.position + this.velocity * time;
   Vector3 targetPosFuture = target.position + target.velocity * time;

   //Distance at the closest point (this is, min distance ever).
   float minDistance = (posFuture - targetPosFuture).Magnitude();

   if(minDistance > THRESHOLD) //possible threshold: radius * 2
      return; //There will be no collision.

   //Evade the point in between
   Vector3 pointToEvade = MidPoint(posFuture, targetPosFuture);
   Evade(pointToEvade);
}
```

Outline

Spherical Cows

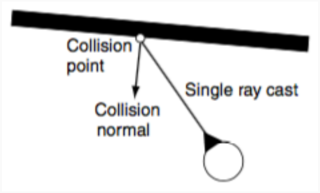
- So far we've assumed the things we're going to hit have been spherical
- This is a good approximation but it doesn't scale well to other shapes
- Specificity, it doesn't work well with walls.
- We'll need to use something a bit more complex...

Obstacle and Wall Avoidance

For this, we need to use **rays**:

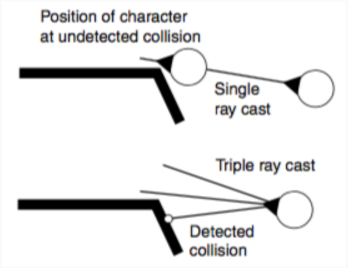
- A **ray** is an infinite line starting at origin and going in some direction. Typically a Ray has two attributes: `Vector3 origin`, the origin of the ray; and `Vector3 direction`, the direction of the ray.
- A **raycast** is a procedure that consists of casting a ray against all or certain colliders in the scene.

Apply *Seek* to a `target = cPoint + cNormal * avoidDist`:



Single Ray Casting Problems

Single raycasting can be suboptimal in many situations:

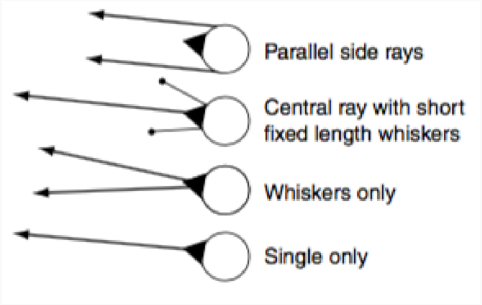


The top diagram shows the single ray does not account for the width of the player.

The lower diagram shows how multiple raycasting can be more effective.

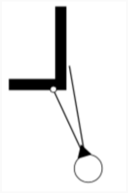
Multiple Rays

A potential solution is to use multiple rays. Here are some possible configurations:



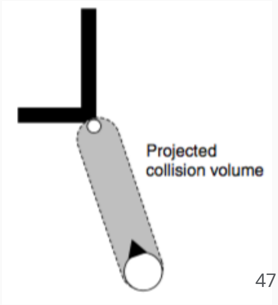
There is no clear best solution, as they might work better or worse depending on the situation (i.e., multiple whiskers avoid corners, but brings problems when navigating through narrow corridors).

Multiple Rays: Problems



The *corner trap*: multiple rays in a Whiskers configuration lead to the agent getting stuck in corners. The agent will turn alternatively right and left as one or the other ray hits the wall.

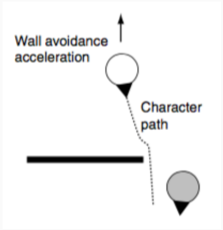
The best approach to collision avoidance is acutally to project the collision volume of the agent along the velocity. However, this is typically quite expensive (and some game engines don’t provide this functionality).

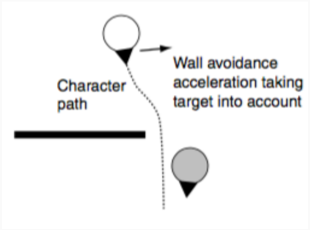


Combining Multi-Objective Behaviours

Multi-Objective Behaviours: Example

In reality, these behaviours are not used in isolation. We might be, for instance, pursuing a target **and** avoiding collisions with walls at the same time.[1]



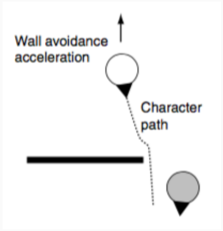


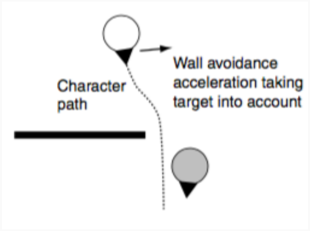
- There are 3 main ways to tackle this multi-objective behaviour:
    1. Weighted Blending
    2. Priorities
    3. Steering Pipeline

[1]Source: Millington, Ian, and John Funge. **Artificial Intelligence for Games.** *CRC Press*, 2012.

Outline

Multi-Objective Behaviours: Weighted Blending

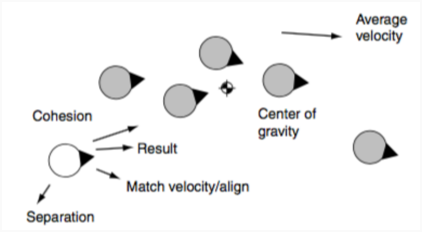




- One way to tackle this problem is to consider all objectives, and add them up using **weighted blending**:
- E.g. apply 50% of the motion decided by wall avoidance plus 50% of the motion decided by the pursuit of opponent.

Weighted Blending

The steering of all behaviours are weighted to produce a single output. Used for **flocking** behaviours:



It combines three elemental behaviours:

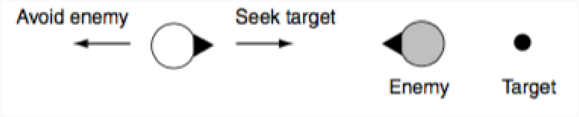
- **Separation:** Move away from agents (boids) that are too close.
- **Alignment:** Move in the same direction and velocity as the flock.
- **Cohesion:** Move towards the center of mass of the flock.

Weighted Blending: Flocking

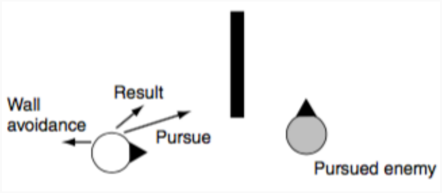
See boids demo of flocking.

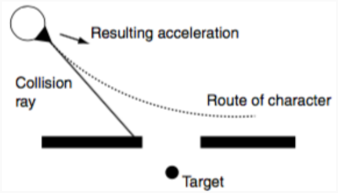
Weighted Blending

Weighted Blending can, however, lead to an unstable equilibrium…



… or simply to errors:





Outline

Prioritizing behaviours

This is a variation of Weighted Blending, but using weights and **priorities**.

This idea works as follows:

- Put similar (semantically speaking) behaviours in groups: avoidance, approaching targets, fleeing from targets, etc.
- Each group provides an output, result of their *blended* behaviours.
- Sort these groups by priority.
- If the steering output by a group is small (less than a defined threshold), it’s ignored.

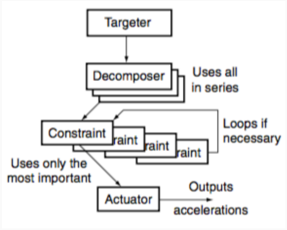
Typically, you would give higher priority to collision avoidance behaviours. But this might depend on the effect that you want to create! You could even change these priorities dynamically according to the state of the game.

Outline

Steering Pipeline

It's a more involved approach that includes constraints and subgoals.[2]

Targeters	What's the goal? (i.e. position, orientation, velocity, etc.)
Decomposers	Find sub-goals.
Constraints	Limit how goals can be achieved (i.e. obstacles).
Actuator	Returns the steering for the next sub-goal.



[2]See Millington+Funge, section 3.4.5 for more details

Further Reading

- Millington, Ian, and John Funge. Chapter 3, **Artificial Intelligence for Games.** *CRC Press*, 2012.
  See interesting sections there, not covered in this lecture, on:
    - Path Following,
    - Motor Control (Steering vehicles),
    - Jumping,
    - Predictive Physics,
    - Coordinating movement (e.g. teams of attackers, sports teams).

Further Reading

- Youtube series (by The Coding Train) on Craig W. Reynolds "Steering Behaviours for Autonomous Characters". Individual videos covering Steering Agents, Seek, Flee/Pursue/Evade, Arrive, Wander

Quizzes

- One quiz on Steering Behaviours
- Two quizzes on vectors (including one to program a Vector2D class in Python)

Questions?