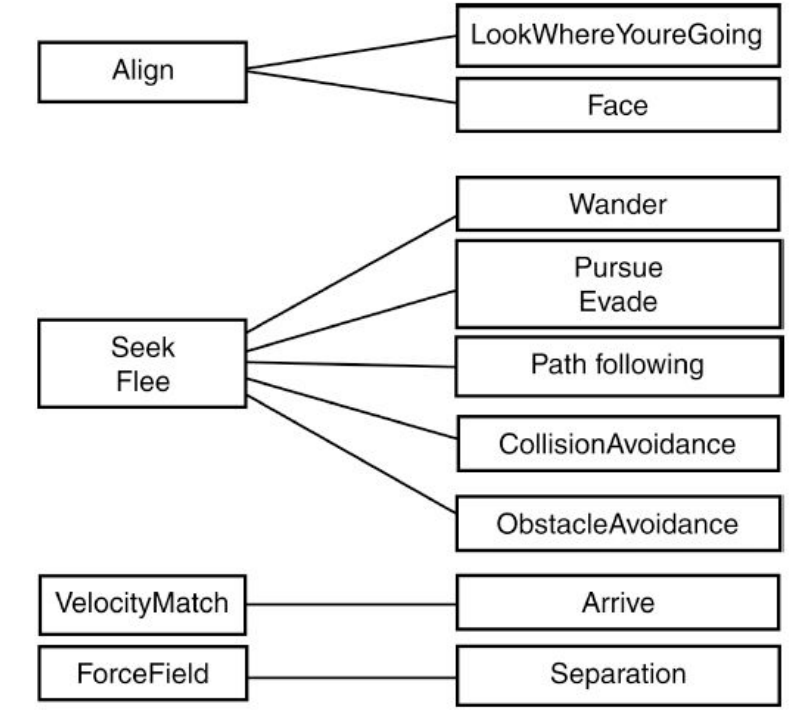
Inteligencia Artificial para Videojuegos

Movimiento

Desplazamiento en grupo

Motivación

- Hay muchos comportamientos de dirección
- Seguiremos hablando de ellos, y diremos cómo *combinarlos de forma avanzada*

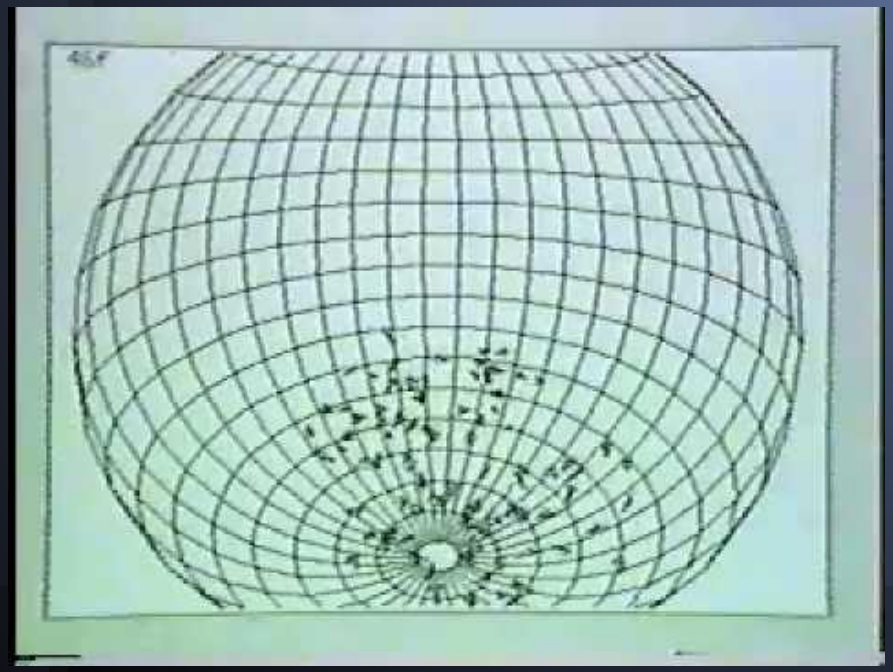


Motivación

- Esto permitirá generar movimientos tan creíbles como el de los pajaroides, de Reynolds (1986)

BOIDS (BOTS + BIRDS)

  - Parece haber “inteligencia”, y eso es lo importante



Puntos clave

- Separación
- Evitación de colisiones
- Evitación de obstáculos y paredes
- Combinar comportamientos
  - Mezcla por pesos
    - Desplazamiento en bandada
  - Mezcla por prioridad
  - Arbitraje cooperativo

Separación

SEPARATION

- Trata de evitar “pegarse” demasiado a otros que van en su misma dirección
  - Es casi “repulsión”, útil para simular multitudes
  - Según *distancia* al vecino más cercano, *umbral* máximo de vecindad y aceleración *máxima*, vemos separación lineal o ley inversa del cuadrado

LINEAR SEPARATION

```
strength = maxAcceleration * (threshold - distance) / threshold
```

INVERSE SQUARE LAW

```
strength = min(k / (distance * distance), maxAcceleration)
```

*** k** es un coeficiente positivo que indica lo rápido que decae la separación con la distancia

  - También puede calcularse la repulsión total como el sumatorio para los N vecinos más próximos

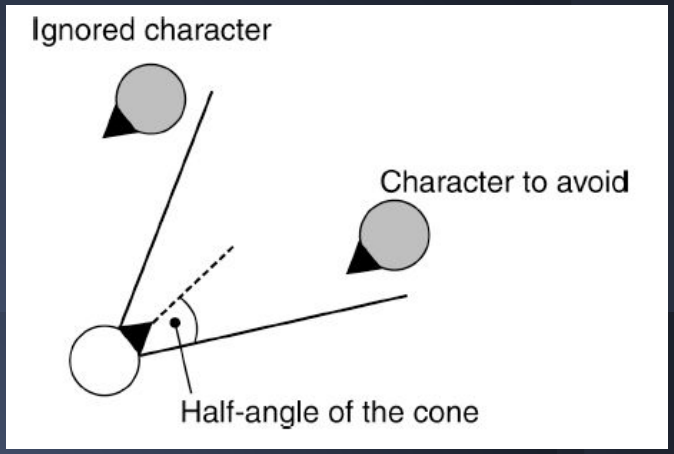
Separación

```
class Separation:
    character: Kinematic
    maxAcceleration: float

    # A list of potential targets.
    targets: Kinematic[]

    # The threshold to take action.
    threshold: float

    # The constant coefficient of decay for the inverse square law.
    decayCoefficient: float

    function getSteering( ) -> SteeringOutput:
        result = new SteeringOutput( )

        # Loop through each target.
        for target in targets:
            # Check if the target is close.
            direction = target.position - character.position
            distance = direction.length( )

            if distance < threshold:
                # Calculate the strength of repulsion
                # (here using the inverse square law).
                strength = min(
                    decayCoefficient / (distance * distance),
                    maxAcceleration)

                # Add the acceleration.
                direction.normalize( )
                result.linear += strength * direction

        return result
```

Tiempo = O(1)
Espacio = O(n)
donde *n* son los agentes vecinos

Evitación de colisiones

COLLISION AVOIDANCE



- Trata de no chocar con otros agentes que van *en dirección distinta*
  - Puede verse como Separación de aquellos vecinos que estén en el *cono de visión* del agente
  - La comprobación debe tener en cuenta *dirección* desde el agente hasta la colisión y *umbral*, el coseno de la mitad del ángulo del cono de visión

```
1  if dotProduct(orientation.asVector(), direction) > coneThreshold:
2      # Do the evasion.
3  else:
4      # Return no steering.
```

  - Existen mejoras a este algoritmo, ya que a veces "evita demasiado" y otras veces se queda corto

Evitación de colisiones

```
class CollisionAvoidance:
    character: Kinematic
    maxAcceleration: float

    # A list of potential targets.
    targets: Kinematic[]

    # The collision radius of a character (assuming all characters
    # have the same radius here).
    radius: float

    function getSteering( ) -> SteeringOutput:
        # 1. Find the target that's closest to collision
        # Store the first collision time.
        shortestTime: float = infinity

        # Store the target that collides then, and other data that we
        # will need and can avoid recalculating.
        firstTarget: Kinematic = null
        firstMinSeparation: float
        firstDistance: float
        firstRelativePos: Vector
        firstRelativeVel: Vector
```

Evitación de colisiones

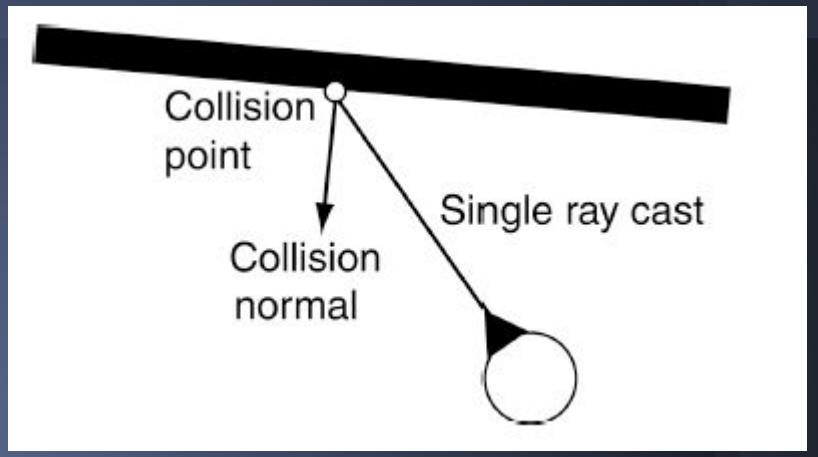
```
25          # Loop through each target.
26          for target in targets:
27              # Calculate the time to collision.
28              relativePos = target.position - character.position
29              relativeVel = target.velocity - character.velocity
30              relativeSpeed = relativeVel.length()
31              timeToCollision = dotProduct(relativePos, relativeVel) /
32                                (relativeSpeed * relativeSpeed)
33
34              # Check if it is going to be a collision at all.
35              distance = relativePos.length()
36              minSeparation = distance - relativeSpeed * timeToCollision
37              if minSeparation > 2 * radius:
38                  continue
39
40              # Check if it is the shortest.
41              if timeToCollision > 0 and timeToCollision < shortestTime:
42                  # Store the time, target and other data.
43                  shortestTime = timeToCollision
44                  firstTarget = target
45                  firstMinSeparation = minSeparation
46                  firstDistance = distance
47                  firstRelativePos = relativePos
48                  firstRelativeVel = relativeVel
```

Evitación de colisiones

```
50          # 2. Calculate the steering
51          # If we have no target, then exit.
52          if not firstTarget:
53              return null
54
55          # If we're going to hit exactly, or if we're already
56          # colliding, then do the steering based on current position.
57          if firstMinSeparation <= 0 or firstDistance < 2 * radius:
58              relativePos = firstTarget.position - character.position
59
60          # Otherwise calculate the future relative position.
61          else:
62              relativePos = firstRelativePos +
63                            firstRelativeVel * shortestTime
64
65          # Avoid the target.
66          relativePos.normalize( )
67
68          result = new SteeringOutput( )
69          result.linear = relativePos * maxAcceleration
70          result.anguar = 0
71          return result
```

Tiempo = O(1)
Espacio = O(n)
donde *n* son
los posibles vecinos

Evitación de obstáculos y paredes



- Trata de no chocar con obstáculos y paredes que *no son circulares*
  - A menudo hay obstáculos muy grandes, como escaleras, o como paredes, cuya forma *no puede aproximarse a un sólo punto*
  - Se usará un detector de colisiones, proyectando uno o varios rayos "unos segundos" por delante y, si alguno colisiona, fija un nuevo objetivo y lo seguirá
    - El objetivo se establecerá cierta distancia sobre la *normal* al objeto desde el punto de la colisión

Evitación de obstáculos y paredes

```
1  class ObstacleAvoidance extends Seek:
2      detector: CollisionDetector
3
4      # The minimum distance to a wall (i.e., how far to avoid
5      # collision) should be greater than the radius of the character.
6      avoidDistance: float
7
8      # The distance to look ahead for a collision
```

Evitación de obstáculos y paredes

```
9       # (i.e., the length of the collision ray).
10      lookahead: float
11
12      # ... Other data is derived from the superclass ...
13
14      function getSteering( ):
15          # 1. Calculate the target to delegate to seek
16          # Calculate the collision ray vector.
17          ray = character.velocity
18          ray.normalize( )
19          ray *= lookahead
20
21          # Find the collision.
22          collision = detector.getCollision(character.position, ray)
23
24          # If have no collision, do nothing.
25          if not collision:
26              return null
27
28          # 2. Otherwise create a target and delegate to seek.
29          target = collision.position + collision.normal * avoidDistance
30          return Seek.getSteering( )
```

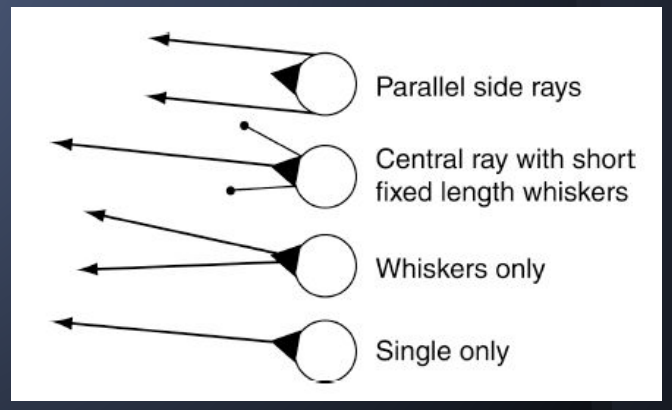
Evitación de obstáculos y paredes

- El detector de colisiones tiene esta interfaz

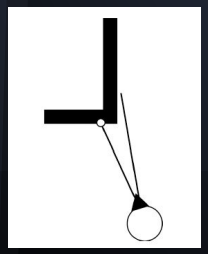
```
1  class CollisionDetector:
2      function getCollision(position: Vector,
3                            moveAmount: Vector) -> Collision
```

  - Usando una determinada configuración de rayos, devuelve la primera colisión encontrada

```
1  class Collision:
2      position: Vector
3      normal: Vector
```



  - ¡Las esquinas suponen una verdadera *trampa* para la IA!, existiendo soluciones avanzadas



Participación

tiny.cc/IAV

- El cono de visión del agente
  A. Sirve para “repeler” a otros agentes
  B. Sirve para percibir pero también como umbral
  C. Ayuda a filtrar obstáculos y paredes colisionables
  D. Ayuda a filtrar los agentes que podrían colisionar
- Desarrolla tu respuesta (en texto libre)

Combinar comportamientos

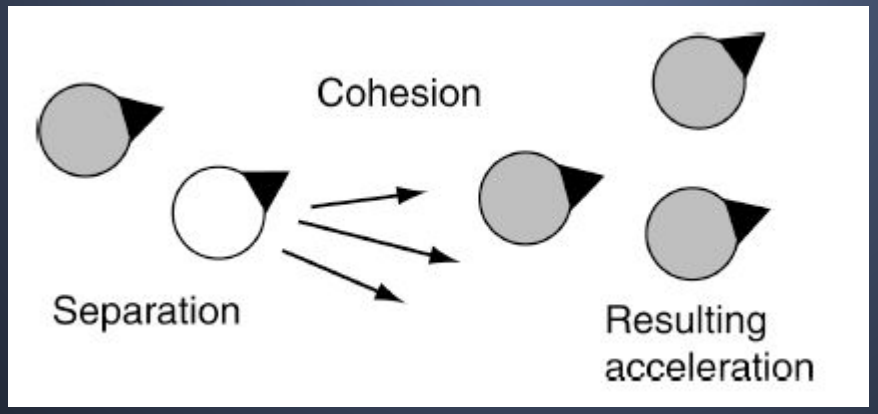
- Es habitual necesitar varios a la vez
  - Ej. Seguir un objetivo, evitar colisiones con agentes, obstáculos y paredes, llegar al destino…
  - O si queremos desplazarnos en grupo/formación
- Combinar estos comportamientos requiere usar mezcla (por pesos o por prioridades), arbitraje (cediéndose el control) o ambos (arquitecturas híbrida hay muchas)

BLENDING

ARBITRATION

Mezcla por pesos

- La forma más simple de combinar comportamientos de dirección
  - Muy típico para mantener agrupaciones
    - Cohesión (Llegada al centro de masas del grupo)
    - Separación con respecto a los vecinos



* Los pesos no suelen sumar 1 (no es *media ponderada*) pero da igual porque hay *máximos* de seguridad

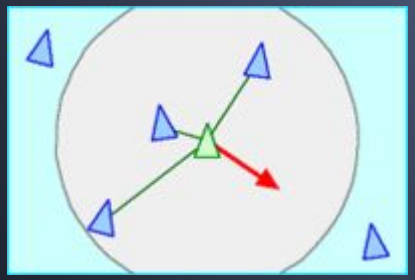
Mezcla por pesos

```
class BlendedSteering:
    class BehaviorAndWeight:
        behavior: SteeringBehavior
        weight: float

    behaviors: BehaviorAndWeight[]

    # The overall maximum acceleration and rotation.
    maxAcceleration: float
    maxRotation: float

    function getSteering( ) -> SteeringOutput:
        result = new SteeringOutput( )

        # Accumulate all accelerations.
        for b in behaviors:
            result += b.weight * b.behavior.getSteering( )

        # Crop the result and return.
        result.linear = max(result.linear, maxAcceleration)
        result.angular = max(result.angular, maxRotation)
        return result
```

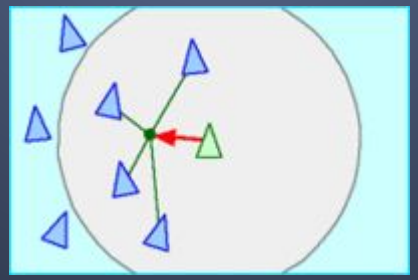
Tiempo = O(1)
Espacio = O(n)
donde *n* son los comportamientos de dirección

Desplazamiento en bandada

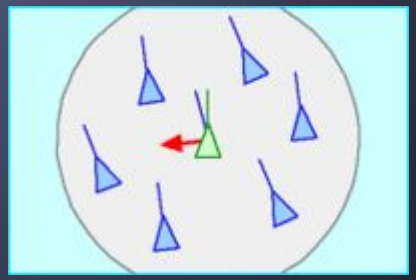
- Los *pajaroides* posiblemente tienen el primer comportamiento de dirección combinado, y el más famoso, el desplazamiento en bandada FLOCKING
  - Mezclaba por pesos, tres comportamientos de dirección de los básicos



Separación



Cohesión

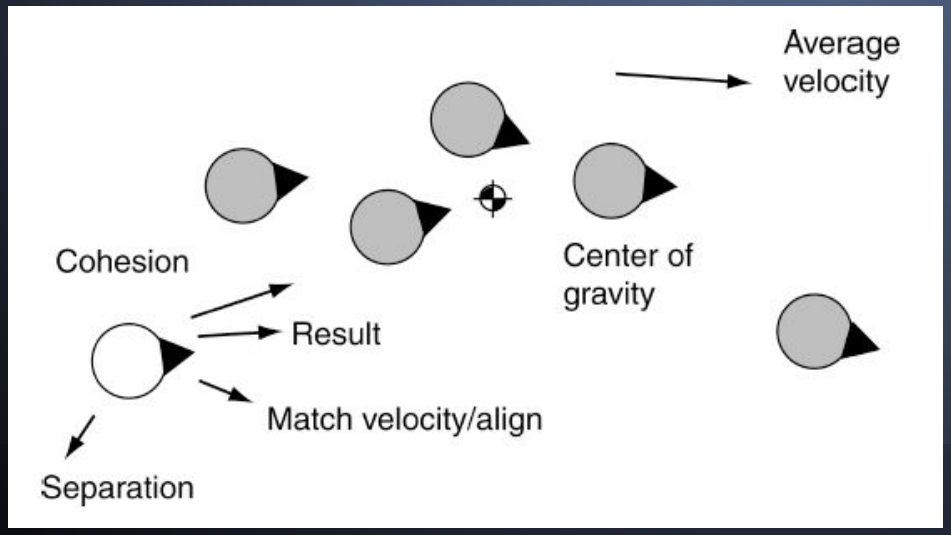


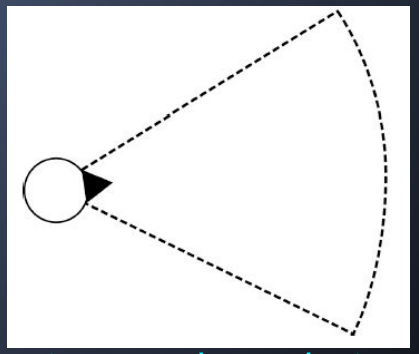
Alineamiento

(y equiparamiento de velocidad)

Desplazamiento en bandada

- Simula una bandada o enjambre de seres
  - Mezcla Separación, Cohesión, Alineamiento y Equiparación de velocidad, usando pesos
    - Generalmente los pesos influyen así: Separación > Cohesión > Alineamiento y equiparación de velocidad

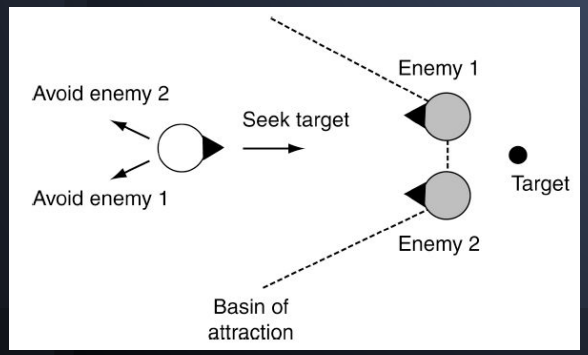


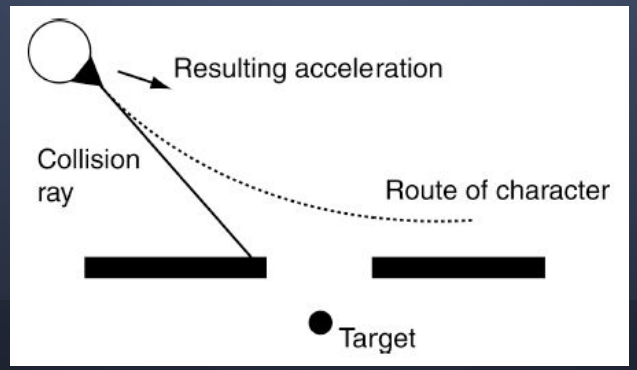


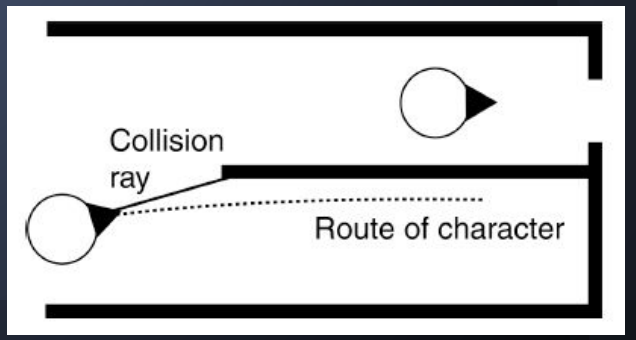
* Los vecinos pueden reducirse usando el típico *cono de visión*

Desplazamiento en bandada

- En la práctica, surgen problemas si la bandada se mueve en entornos cerrados
  - Equilibrio estable, los comportamientos de dirección se contraponen y *bloquean* al agente
  - Entornos restringidos, con puertas o pasillos por donde el agente inteligente *no es capaz de pasar*
  - Miopía, los comportamientos de dirección no ven más allá de lo local... pueden fallar en lo global







Mezcla por prioridad

- Algunos comportamientos de direccionamiento *no suelen devolver aceleración* salvo en casos muy concretos
  - ¡Conviene hacerles caso al 100% entonces!
- Un posible algoritmo de mezcla *divide estos comportamientos en grupos del mismo peso* y luego ordena estos grupos por prioridad
  - Se revisan las salidas y la primera que esté por encima de un umbral ε (cercano a 0), se usará
  - Ej. Primero *Evitaciones diversas*, después *Separación* y finalmente el grupo de la *Persecución*

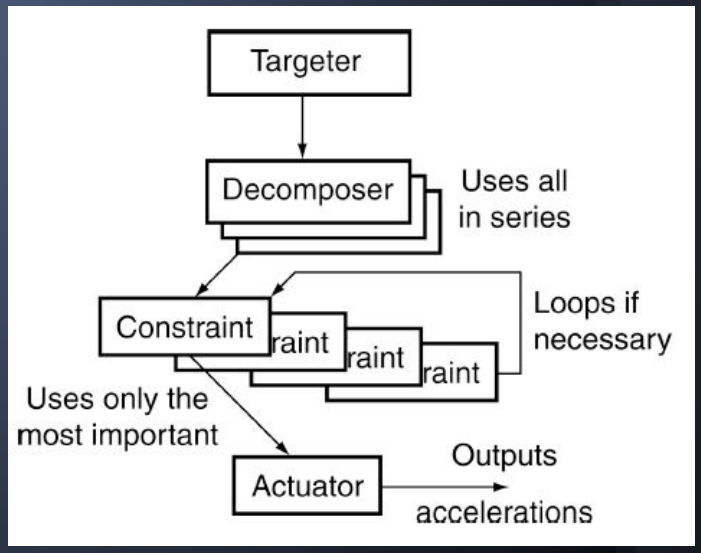
Mezcla por prioridad

```
# Should be a small value, effectively zero.
epsilon: float

class PrioritySteering:
    # Holds a list of BlendedSteering instances, which in turn
    # contain sets of behaviors with their blending weights.
    groups: BlendedSteering[]

    function getSteering( ) -> SteeringOutput:
        for group in groups:
            # Create the steering structure for accumulation.
            steering = group.getSteering( )

            # Check if we're above the threshold, if so return.
            if steering.linear.length( ) > epsilon or
               abs(steering.angular) > epsilon:
                return steering

        # If we get here, it means that no group had a large enough
        # acceleration, so return the small acceleration from the
        # final group.
        return steering
```

Tiempo = O(1)
Espacio = O(n)
donde *n* son los comportamientos de dirección

Arbitraje cooperativo

- Cuando necesitamos un control más fino, el siguiente paso es que los comportamientos de dirección cooperen entre sí
  - Implica *reprogramarse los movimientos*, sin tener algoritmos independientes
  - Tiene mucho que ver con la Toma de Decisiones, aunque Millington propone aquí un sistema compuesto de *un flujo que va desglosando cada componente* del comportamiento de dirección



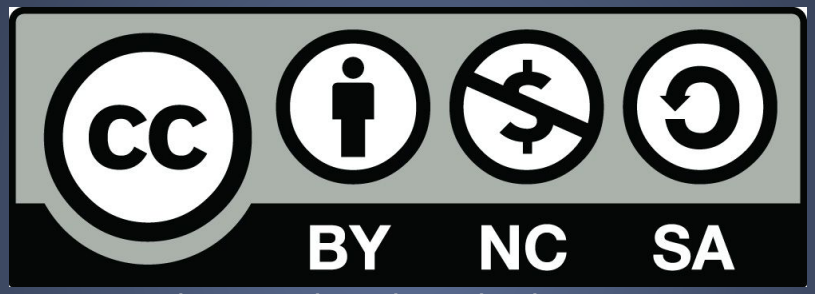
Resumen

- Continuamos viendo comportamientos de dirección como Separación, Evitación de colisiones, y de obstáculos y paredes
- Es importante aprender a combinar estos comportamientos de manera organizada
- Una de las formas es mezclar por pesos, como en el desplazamiento en bandada
- Otra es la mezcla por prioridad
- Finalmente, existen arquitecturas más complejas, como el arbitraje cooperativo

Más información

- Millington, I.: Artificial Intelligence for Games. CRC Press, 3rd Edition (2019)

Críticas, dudas, sugerencias...



Federico Peinado (2019-2020)

www.federicopeinado.es