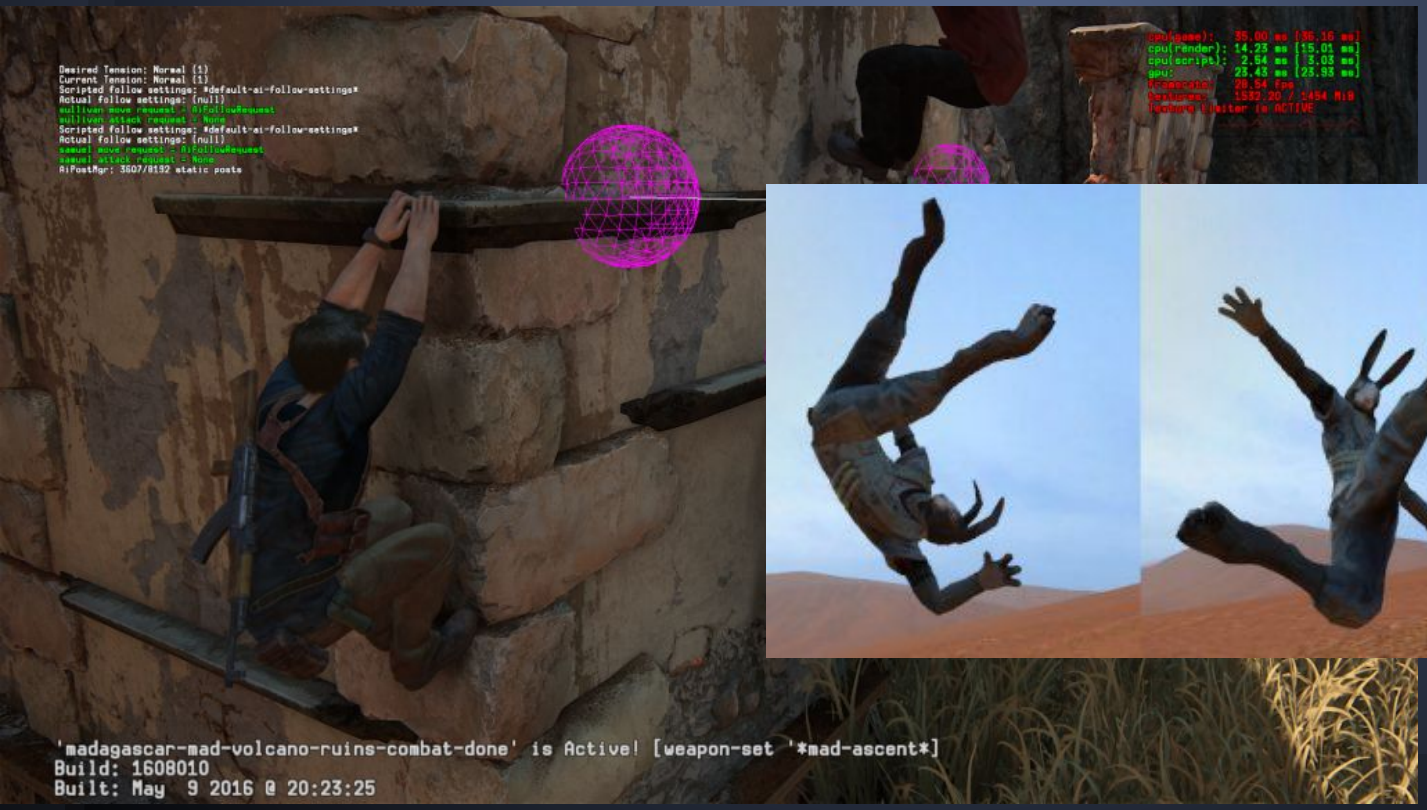
Inteligencia Artificial para Videojuegos

Movimiento

Física y animación

Motivación

- Encontramos solapamiento entre la *IA* y el tema de la Física… y hasta de la Animación



Motivación

- *Mover a los personajes* es un requisito fundamental de la IA para videojuegos
- ¡Pero movimientos hay de muchos tipos!
  - De hecho hay juegos *sin movimiento*, y otros donde la "IA" son únicamente *algoritmos de movimiento*



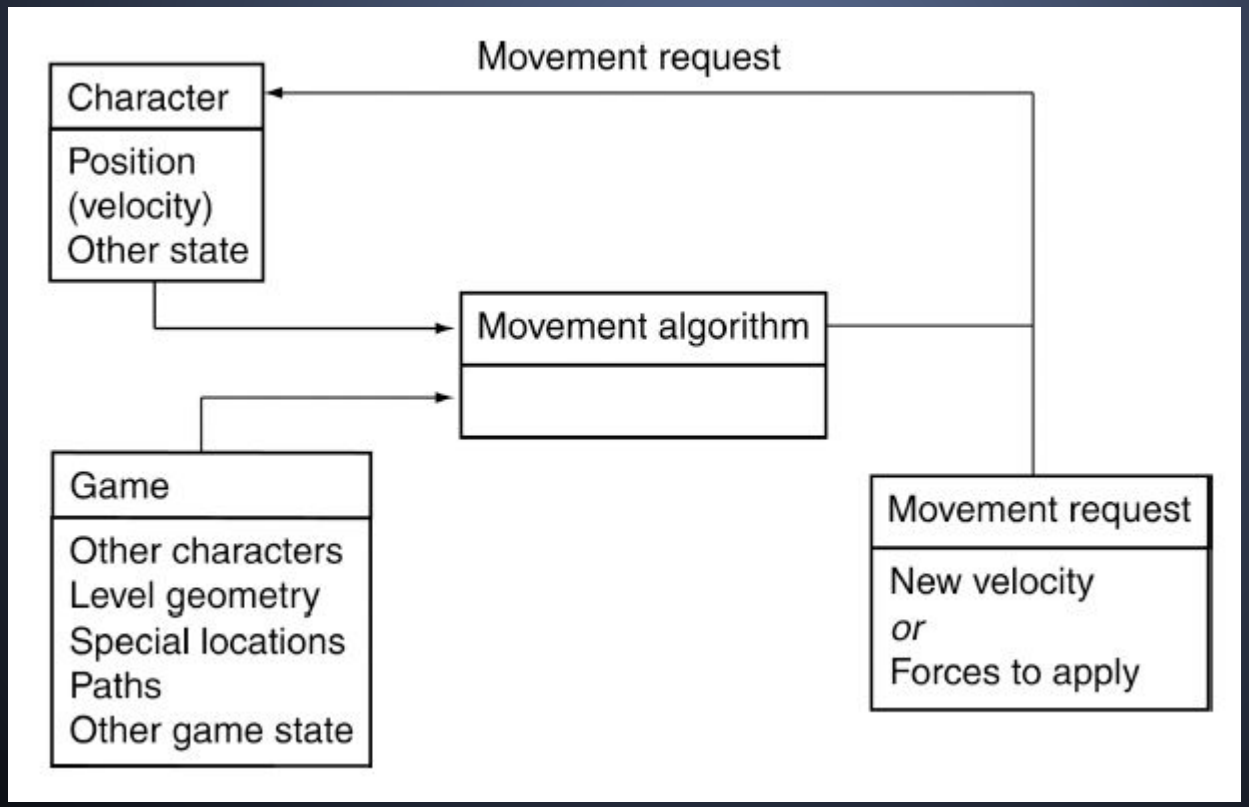



Puntos clave

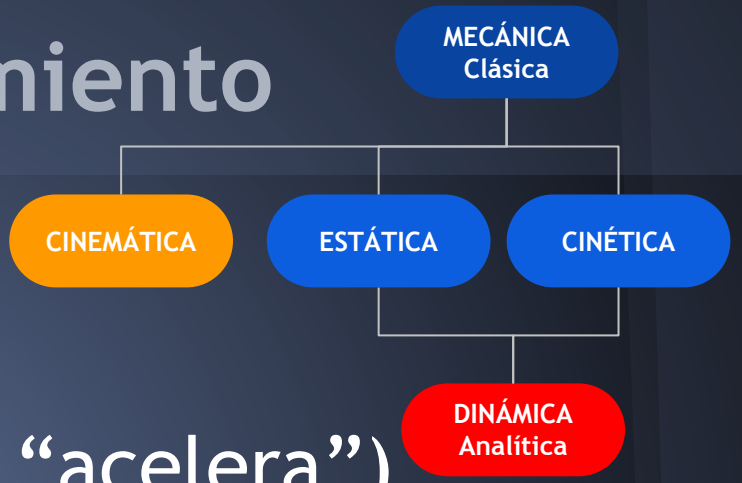
- Fundamentos del movimiento
- Hitos históricos
- Predicción física
- Saltos y movimientos coordinados
- Movimientos 3D
- Movimientos cinemáticos
  - Persecución
  - Huida
  - Llegada
  - Merodeo

Fundamentos del movimiento

- Todos los algoritmos de movimiento comparten el mismo esquema



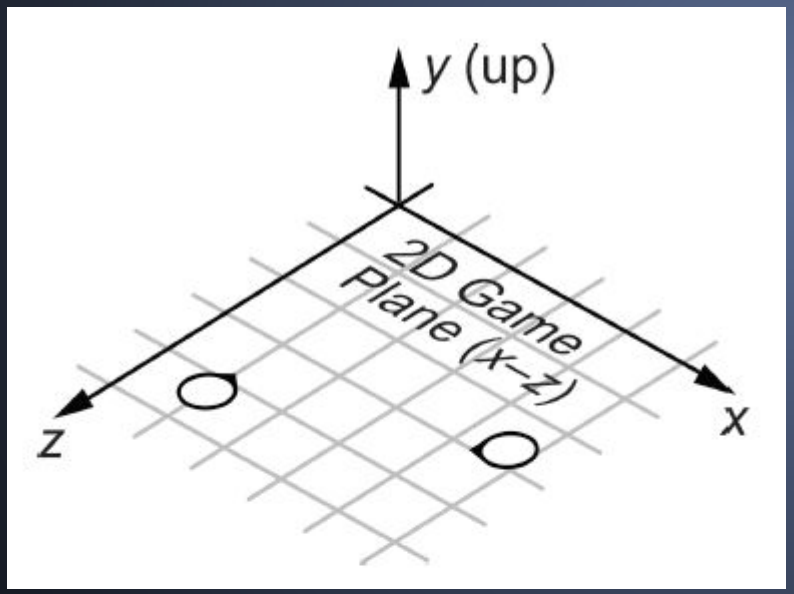
Fundamentos del movimiento

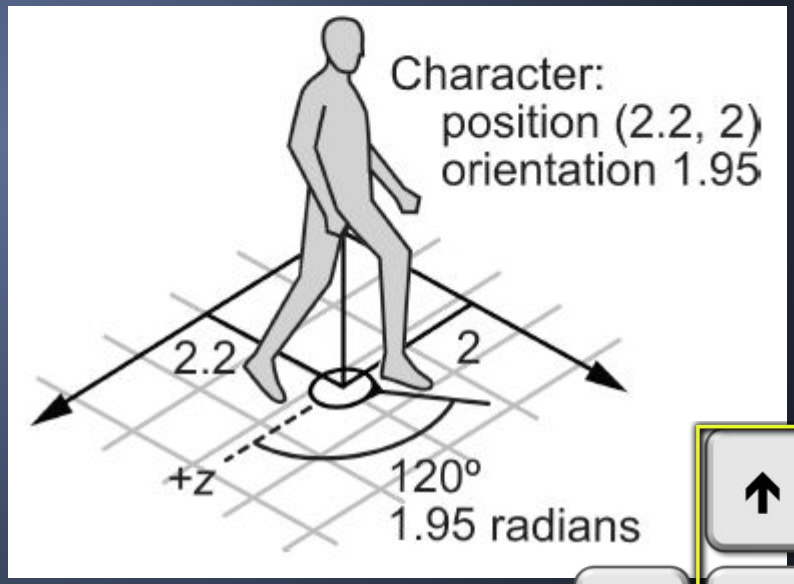


- La cinemática estudia el movimiento (posición, incluso velocidad… y hasta "acelera") pero sin considerar *causas* que lo originan
  - Ej. Algoritmos que reciben sólo posición y orientación, y devuelven velocidad deseada en *t*
  - Pueden ser aceleración sencilla que refina caminos
- La dinámica considera también esas causas (energía, masa, aplicación de fuerzas…)
  - Ej. Algoritmos que además reciben velocidad, y según aceleración deseada en *t*, devuelven fuerzas a aplicar (tipo motor de física)

Fundamentos del movimiento

- Movimiento en 2D (todo suelo lo es)





Z positivo es hacia abajo
Z negativo es hacia arriba

Hitos históricos

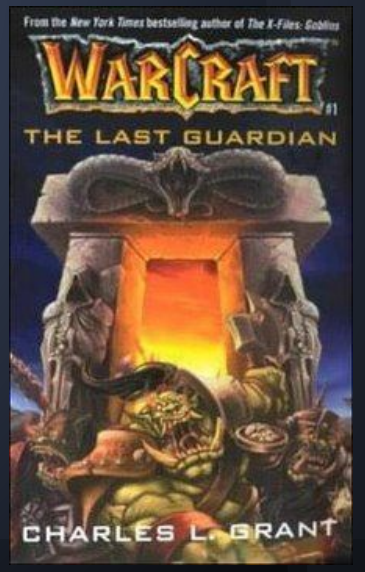
- Arranca la industria del videojuego en 1972
  - Los primeros videojuegos no tienen IA
- Las primeras IAs aparecieron en clones de Pong con oponentes controlados por ordenador, o clones de Space Invaders
  - Básicamente lo único que hacían era moverse… moverse hacia el lado que debían :-)
- Se popularizaron mucho las recreativas como Speed Race (1974)
  - Tenía una "IA primitiva" (coches que se acercan) y dos niveles de dificultad

Hitos históricos

- Space Invaders, el *boom* de 1978
  - Tenía una dificultad *incremental*
  - Usaba distintos patrones de movimiento dependientes de la entrada del jugador
- Numerosas secuelas y títulos similares fueron añadiendo patrones de movimiento más complejos

Hitos históricos

- Karate Champ (1984)
  - Fue el primer juego de lucha en usar notablemente la IA
  - Diferentes patrones de lucha (otra forma de movimiento) para representar personalidades distintas en sus luchadores
- Juegos como Warcraft (1994) incluyen sutilezas como movimientos sutiles de las tropas para romper filas y aproximarse al enemigo



Movimientos cinemáticos

- Los más simples, aunque muy utilizados
  - Son 2D y se usan para no cambiar la velocidad bruscamente, sino suavizándola en el tiempo, SMOOTHING refinando así el camino que recorre el agente

```
1  class Static:
2      position: Vector
3      orientation: float
```

Estado **ESTÁTICO** del agente, con posición (2D) y orientación (0..2π radianes)

```
1  class Kinematic:
2      position: Vector
3      orientation: float
4      velocity: Vector
5      rotation: float
```

Estado **CINEMÁTICO**, añadiendo velocidad lineal y angular (rotación en radianes/segundo)

```
1  class SteeringOutput:
2      linear: Vector
3      angular: float
```

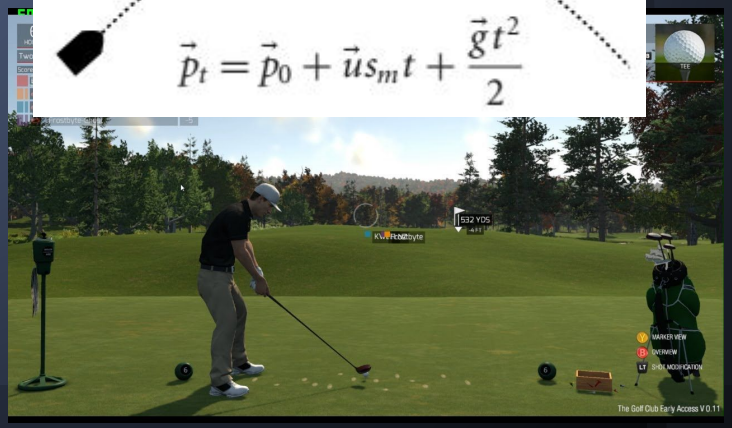
Aceleración lineal y angular que devuelven estos algoritmos como **DIRECCIÓN** de salida

Para movimientos dinámicos

Predicción física

- Algún *comportamiento de dirección* la tiene, y se usa en *juegos online*, pero la típica es la del lanzamiento de proyectiles
  - Requiere conocer el punto de impacto y si la trayectoria atraviesa o no a un personaje
  - A veces existe rozamiento y si *la complejidad es grande* se hace un apuntado iterativo (simulando N lanzamientos)
  - Ej. Incluso las balas trazan un *arco parabólico* y hay videojuegos que lo simulan

$$\vec{p}_t = \vec{p}_0 + \vec{u}s_m t + \frac{\vec{g}t^2}{2}$$



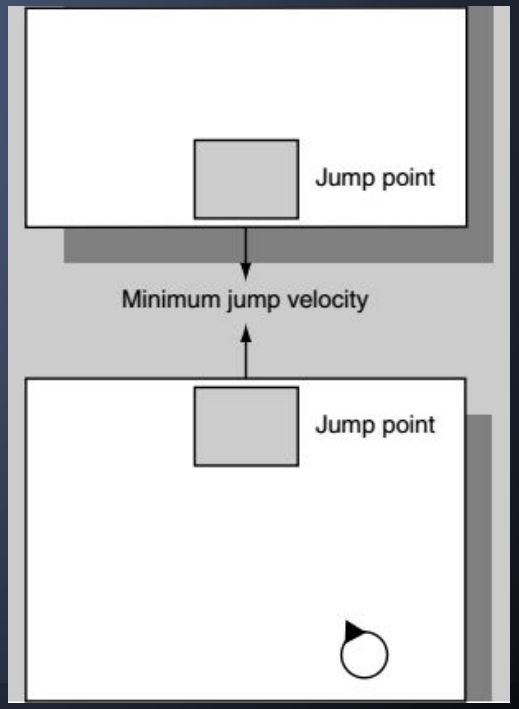
* La gravedad acelera a 9,81 m/s^2 aunque en videojuegos a veces se usa el doble

Saltos y movimientos coordinados

- Hay movimientos como el salto en los que el agente se lo juega *todo* en un instante
  - Truco: El diseñador puede marcar puntos de salto (¡incluso evitar saltos *demasiado difíciles* en sus niveles!)
  - En vez de marcar la velocidad mínima de salto, se puede marcar la zona de aterrizaje y que el agente calcule su velocidad
  - Y mejor aún: se pueden marcar los huecos saltables y que luego el agente los salte como quiera

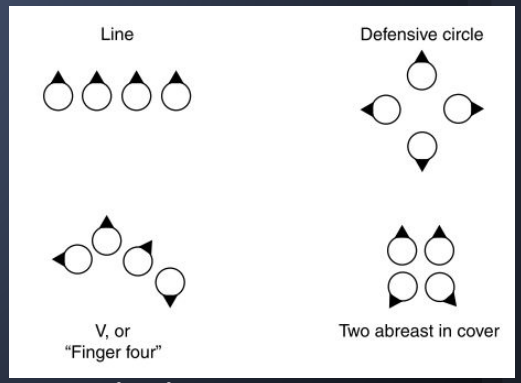
LANDING PAD

HOLE FILLER



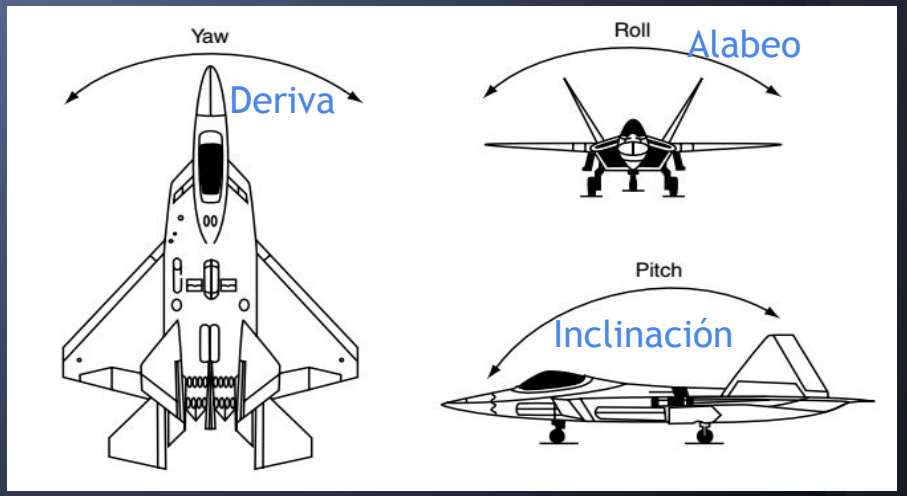
Saltos y movimientos coordinados

- Los agentes puede coordinarse para moverse de forma centralizada o distribuida
  - Sin llegar a *tomar decisiones tácticas*, una primera opción es hacer que se muevan en formación
  - Las formaciones pueden tener un líder (o *punto de referencia invisible* como el centro de masas) o no
  - Pueden ser escalables y adaptarse a un número variable de agentes (incluso de manera emergente)
  - A veces hay un gestor de formación a dos niveles, para evitar que un agente se descuelgue o para asignar posiciones en la formación a cada tipo de agente



Movimientos 3D

- La orientación tridimensional requiere cálculos matemáticos en *cuaterniones*
  - Los *comportamientos de dirección* son los mismos que en 2D, salvo los de componente *angular* que requieren de estos nuevos cálculos
    - Alinear, encarar, orientar al movimiento del objetivo y merodear
  - Orientarse en 3D es difícil, por lo que a veces se ofrece un eje Z o se simula la deriva con *alabeo + inclinación*, como en un avión real



Participación

tiny.cc/IAV

- ¿Qué maneja el movimiento cinemático?
  A. La aceleración lineal y angular
  B. La posición y la orientación
  C. Lo del estático, más velocidad lineal y rotación
  D. Lo del estático, más lo del dinámico
- Desarrolla tu respuesta (en texto libre)

Movimientos cinemáticos

- Algoritmo *sencillo* para actualizar la posición y orientación según el movimiento

```
class Kinematic:
    # ... Member data as before ...

    function update(steering: SteeringOutput, time: float):
        # Update the position and orientation.
        half_t_sq: float = 0.5 * time * time
        position += velocity * time + steering.linear * half_t_sq
        orientation += rotation * time + steering.angular * half_t_sq

        # and the velocity and rotation.
        velocity += steering.linear * time
        rotation += steering.angular * time
```

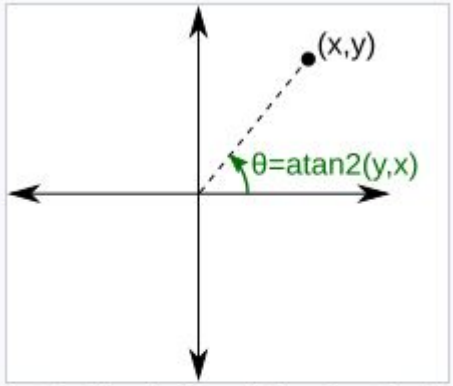
Movimientos cinemáticos

- Con función para modificar la *orientación según la velocidad*

```
function newOrientation(current: float, velocity: Vector) -> float:
    # Make sure we have a velocity.
    if velocity.length( ) > 0:
        # Calculate orientation from the velocity.
        return atan2(-static.x, static.z)

    # Otherwise use the current orientation.
    else:
        return current
```

*__atan2__ es la función arcotangente de dos parámetros



La función atan2 (*y*, *x*) devuelve el ángulo θ entre el rayo que une el origen de coordenadas con un punto (*x*, *y*) y el eje positivo *x*, limitado a -π, π

Movimientos cinemáticos

SEEK

* Algoritmo de Seguimiento

(no confundir con Persecución, que lleva *predicción*) PURSUE

```
class KinematicSteeringOutput:
    velocity: Vector
    rotation: float
```

Devolvemos una **dirección cinemática** de salida (sólo velocidad lineal y rotación)

FLEE

* Modificación para Huida

(no confundir con Evasión, que lleva *predicción*) EVADE

```
class KinematicSeek:
    character: Static
    target: Static

    maxSpeed: float

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

        # Get the direction to the target.
        result.velocity = target.position - character.position

        # The velocity is along this direction, at full speed.
        result.velocity.normalize( )
        result.velocity *= maxSpeed

        # Face in the direction we want to move.
        character.orientation = newOrientation(
            character.orientation,
            result.velocity)

        result.rotation = 0
        return result
```

```
# Get the direction away from the target.
steering.velocity = character.position - target.position
```

Movimientos cinemáticos

```
class KinematicArrive:
    character: Static
    target: Static
    maxSpeed: float

    # The satisfaction radius.
    radius: float

    # The time to target constant.
    timeToTarget: float = 0.25

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

        # Get the direction to the target.
        result.velocity = target.position - character.position

        # Check if we're within radius.
        if result.velocity.length( ) < radius:
            # Request no steering.
            return null

        # We need to move to our target, we'd like to
        # get there in timeToTarget seconds.
        result.velocity /= timeToTarget
```

ARRIVE

* Algoritmo de Llegada (para que los agentes no se queden "orbitando" alrededor del destino)

Movimientos cinemáticos

```
28          # If this is too fast, clip it to the max speed.
29          if result.velocity.length( ) > maxSpeed:
30              result.velocity.normalize( )
31              result.velocity *= maxSpeed
32
33          # Face in the direction we want to move.
34          character.orientation = newOrientation(
35              character.orientation,
36              result.velocity)
37
38          result.rotation = 0
39          return result
```

Movimientos cinemáticos

WANDER

```
class KinematicWander:
    character: Static
    maxSpeed: float

    # The maximum rotation speed we'd like, probably should be smaller
    # than the maximum possible, for a leisurely change in direction.
    maxRotation: float

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

        # Get velocity from the vector form of the orientation.
        result.velocity = maxSpeed * character.orientation.asVector()

        # Change our orientation randomly.
        result.rotation = randomBinomial() * maxRotation

        return result
```

* Algoritmo para Merodeo

***randomBinomial** equivale a pedir dos números aleatorios entre 0 y 1, y restarlos

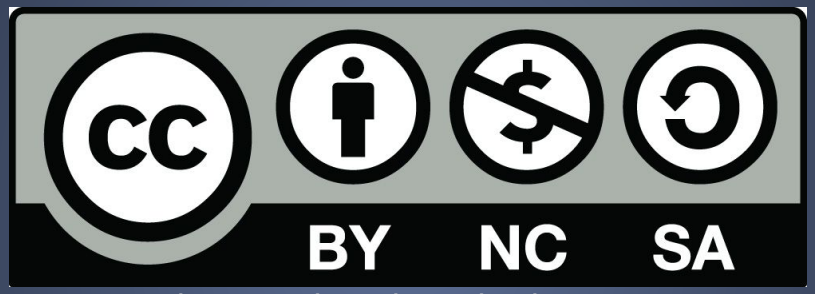
Resumen

- El movimiento dinámico añade fuerzas al cinemático, que es más sencillo de usar
- Hay movimientos cinemáticos básicos como la persecución, la huida, la llegada y el merodeo
- Es típico predecir la física de proyectiles, para ver donde llegan nuestros disparos
- Saltar o controlar un coche requiere trucos o heurísticas, y el 3D, de mayor cálculo

Más información

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

Críticas, dudas, sugerencias...



Federico Peinado (2019-2020)

www.federicopeinado.es