DM810

Computer Game Programming II: AI

Lecture 2
Movement

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

1. Representations

2. Kinematic Movement
   - Seeking
   - Wandering

3. Steering Behaviors

Outline

1. Representations

2. Kinematic Movement
   - Seeking
   - Wandering

3. Steering Behaviors

Movement

Movement of characters around the level (not about movement of faces)

Input: geometric data about the state of the world + current position of character + other physical properties

Output: geometric data representing movement (velocity, accelerations)

For most games, characters have only two states: stationary + running

Running:

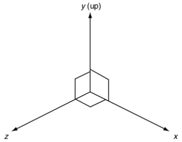
- Kinematic movement: constant velocity, no acceleration nor slow down.
- Steering behavior: dynamic movement with accelerations. Takes into account current velocity of the character and outputs acceleration (eg, Craig Reynolds, flocking)

Examples: Kinematic algorithm from A to B returns direction.
Dynamic/steering algorithm from A to B returns acceleration and deceleration

Static Representations

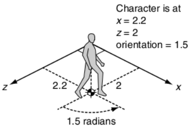
Characters represented as points, center of mass (collision detection, obstacle avoidance need also size but mostly handled outside of movement algorithms).

In 2D:



x, z orthonormal basis of 2D space
2D movement takes place in $x, z \rightsquigarrow$ (x, z) coordinates

Orientation value θ:
counterclockwise angle, in radiants from positive z-axis




```
struct Static:
  position # a 2D vector
  orientation # single floating point value
```

then rendered in 3D (θ determines the rotation matrix)

Static Representations

In 3D movement is more complicated: orientation implies 3 parameters

May be needed in flight simulators

But often one dim is gravity and rotation about the upright direction is enough, the rest can be handled by animations)

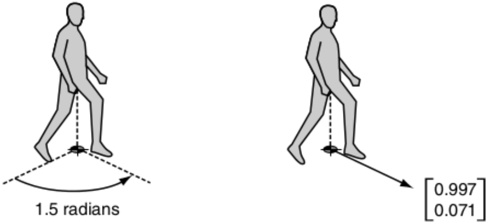
Hybrid model:

In $2\frac{1}{2}$D

- full 3D position (includes possibility for jumps)
- orientation as a single value

huge simplification in math in change of a small loss in flexibility

Orientation in Vector Form



from angle θ to unit length vector in the direction that the character is facing

$$\boldsymbol{\theta} = \begin{bmatrix} sin\theta \\ cos\theta \end{bmatrix}$$

Kinematic Representations

- Kinematic algorithms:
  position + orientation + velocity

	2D	$2\frac{1}{2}$D
linear velocity $\mathbf{v}$	v_x, v_z components	v_x, v_y, v_z components
angular velocity θ'	π/s	π/s

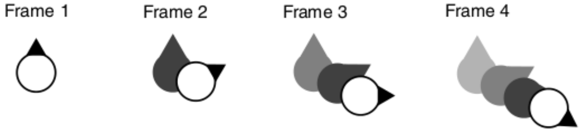
```
struct Kinematic:
  position # 2 or 3D vector
  orientation # single floating point value
  velocity # 2 or 3D vector
  rotation # single floating point value
```

- Steering algorithms:
  return linear acceleration $\mathbf{a}$ and angular acceleration θ''

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

Independent facing

Characters mostly face the direction of movement. Hence steering algs often ignore rotation. To avoid abrupt changes orientation is moved proportionally towards moving direction:



Kinematic Representations

- Updates (classical mechanics)

$$\mathbf{v}(t) = \mathbf{r}'(t) \qquad \mathbf{a}(t) = \mathbf{r}''(t)$$

$$\begin{aligned} \mathbf{r} &= \mathbf{v}t + \tfrac{1}{2}\mathbf{a}t^2 & \qquad \mathbf{v} &= \mathbf{a}t \\ \theta &= \theta' t + \tfrac{1}{2}\theta'' t^2 & \qquad \theta' &= \theta'' t \end{aligned}$$

```
struct Kinematic:
  position
  orientation
  velocity
  rotation
  def update(steering, time):
     position += velocity * time + 0.5 *
           steering.linear * time * time
     orientation += rotation * time + 0.5 *
            steering.angular * time * time
     velocity += steering.linear * time
     orientation += steering.angular * time
```

```
struct Kinematic:
  position
  orientation
  velocity
  rotation
  def update(steering, time):
     position += velocity * time
     orientation += rotation * time
     velocity += steering.linear * time
     orientation += steering.angular * time
```

Velocities expressed as $\mathrm{m/s}$ thus support for variable frame rate.
Eg.: If $v = 1\mathrm{m/s}$ and the frame duration is $20\mathrm{ms}$ ➡ $x = 20\mathrm{mm}$

Netwon's Physics

Accelerations are determined by forces and inertia ($F = ma$)

To model object inertia:

- object's mass for the linear inertia
- moment of inertia (or inertia tensor in 3D) for angular acceleration.

We could extend char data and movement algorithms with these, but mostly needed for physics games, eg, driving game.

Actuation is a post-processing step that takes care of computing forces after steering has been decided to produce the desired change in velocity (poses feasibility problems)

Outline

1. Representations

2. Kinematic Movement
   - Seeking
   - Wandering

3. Steering Behaviors

Kinematic Movement Algorithms

Input: static data

Output: velocity (often: on/off full speed or being stationary + target direction)

From $\mathbf{v}$ we calculate orientation using trigonometry:

$$\tan\theta = \frac{\sin\theta}{\cos\theta} \qquad \theta = \arctan(-v_x/v_z)$$

(sign because counterclockwise from z-axis)

```
def getNewOrientation(currentOrientation, velocity):
    if velocity.length() > 0:
        return atan2(-static.x, static.z)
    else: return currentOrientation
```

Seeking

Input: character's and target's static data

Output: velocity along direction to target

```
struct Static:
    position
    orientation
```

```
struct KinematicSteeringOutput:
    velocity
    rotation
```

```
class KinematicSeek:
    character # static data char.
    target # static data target
    maxSpeed

    def getSteering():
        steering = new KinematicSteeringOutput()
        steering.velocity = target.position - character.position # direction
        steering.velocity.normalize()
        steering.velocity *= maxSpeed
        character.orientation = getNewOrientation(character.orientation, steering.
              velocity)
        steering.rotation = 0
        return steering
```

Performance in time and memory? $O(1)$

- `getNewOrientation` can be taken out
- flee mode:

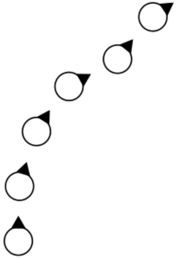
```
steering.velocity = character.position - target.position
```

- problem: arrival must be stationary not wiggling back and forth
  - use large radius of satisfaction to target
  - use a range of movement speeds, and slow the character down as it reaches its target

```
class KinematicArrive:
    character
    target
    maxSpeed
    radius # satisfaction radius
    timeToTarget = 0.25 # time to target constant
    def getSteering():
        steering = new KinematicSteeringOutput()
        steering.velocity = target.position - character.position # direction
        if steering.velocity.length() < radius:
        return None
        steering.velocity /= timeToTarget # set vel. wrt time to target
        if steering.velocity.length() > maxSpeed:
        steering.velocity.normalize()
        steering.velocity *= maxSpeed
        character.orientation = getNewOrientation(character.orientation, steering.
             velocity)
        steering.rotation = 0
        return steering
```

Wander

A kinematic wander behavior moves in the direction of the character's current orientation with maximum speed.
Orientation is changed by steering.



```
class KinematicWander:
    character
    maxSpeed
    maxRotation # speed
    def getSteering():
        steering = new KinematicSteeringOutput()
        steering.velocity = maxSpeed * character.orientation.
              asVector()
        steering.rotation = random(-1,1) * maxRotation
        return steering
```

Demo

Outline

1. Representations

2. Kinematic Movement
   - Seeking
   - Wandering

3. Steering Behaviors

Steering, Intro

- movement algorithms that include accelerations
- present in driving games but always more in all games.
- range of different behaviors obtained by combination of fundamental behaviors: eg. seek and flee, arrive, and align.
- each behavior does a single thing, more complex behaviors obtained by higher level code
- often organized in pairs, behavior and its opposite (eg, seek and flee)

Input: kinematic of the moving character + target information
(moving char in chasing, representation of the geometry of the world in obstacle avoidance, path in path following behavior; group of targets in flocking – move toward the average position of the flock.)

Output: steering, ie, accelerations

Variable Matching

- Match one or more of the elements of the character's kinematic to a single target kinematic (additional properties that control how the matching is performed)
- To avoid incongruencies: individual matching algorithms for each element and then right combination later. (algorithms for combinations resolve conflicts)

Seek and Flee

Seek tries to match the position of the character with the position of the target. Accelerate as much as possible in the direction of the target.

```
struct Kinematic:
  position
  orientation
  velocity
  rotation
  def update(steering, maxSpeed, time):
    position += velocity * time
    orientation += rotation * time
    velocity += steering.linear * time
    orientation += steering.angular * time
    if velocity.length() > maxSpeed:
      velocity.normalize()
      velocity *= maxSpeed


struct SteeringOutput
  linear # accleration
  angular # acceleration
```

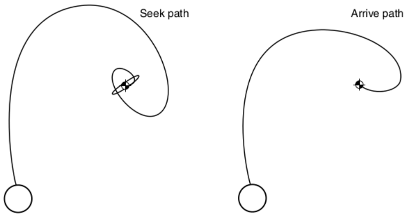
```
class Seek:
  character # kinematic data
  target # kinematic data
  maxAcceleration

  def getSteering():
    steering = new SteeringOutput()
    steering.linear = target.position -
                  character.position #
                        change here for
                        flee
    steering.linear.normalize()
    steering.linear *= maxAcceleration
    steering.angular = 0
    return steering
```

Demo

if velocity exceeds the maximum speed it is trimmed back in a post-processing step of the update function.
Note, orientation removed: like before or by matching or proportional

Arrive

Seek always moves to target with max acceleration. If target is standing it will orbit around it. Hence we need to slow down and arrive with zero speed.



Two radii:

- arrival radius, as before, lets the character get near enough to the target without letting small errors keep it in motion.
- slowing-down radius, much larger. max speed at radius and then interpolated by distance to target

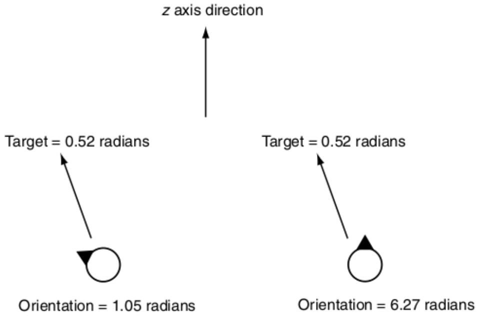
Direction as before
Acceleration dependent on the desired velocity to reach in a fixed time (0.1 s)

Arrive

```
class Arrive:
    character # kinematic data
    target
    maxAcceleration
    maxSpeed
    targetRadius
    slowRadius
    timeToTarget = 0.1 # time to arrive at target
    def getSteering(target):
        steering = new SteeringOutput()
        direction = target.position - character.position
        distance = direction.length()
        if distance < targetRadius
            return None
        if distance > slowRadius:
            targetSpeed = maxSpeed
        else:
            targetSpeed = maxSpeed * distance / slowRadius
        targetVelocity = direction
        targetVelocity.normalize()
        targetVelocity *= targetSpeed
        steering.linear = targetVelocity - character.velocity
        steering.linear /= timeToTarget
        if steering.linear.length() > maxAcceleration:
            steering.linear.normalize()
            steering.linear *= maxAcceleration
        steering.angular = 0
        return steering
```

Align

match the orientation of the character with that of the target (just turn, no linear acceleration). Angular version of Arrive.
Issue:
avoid rotating in the wrong direction because of the angular wrap



convert the result into the range $(-\pi, \pi)$ radians by adding or subtracting $m \cdot 2\pi$

Align


```
class Align:
    character
    target
    maxAngularAcceleration
    maxRotation
    targetRadius
    slowRadius
    timeToTarget = 0.1
    def getSteering(target):
        steering = new SteeringOutput()
        rotation = target.orientation - character.orientation
        rotation = mapToRange(rotation)
        rotationSize = abs(rotationDirection)
        if rotationSize < targetRadius
            return None
        if rotationSize > slowRadius:
            targetRotation = maxRotation
        else:
            targetRotation = maxRotation * rotationSize / slowRadius
        targetRotation *= rotation / rotationSize
        steering.angular = targetRotation - character.rotation
        steering.angular /= timeToTarget
        angularAcceleration = abs(steering.angular)
        if angularAcceleration > maxAngularAcceleration:
            steering.angular /= angularAcceleration
            steering.angular *= maxAngularAcceleration
        steering.linear = 0
        return steering
```

Velocity Matching

- So far we matched positions
- Matching velocity becomes relevant when combined with other behaviors, eg. flocking steering behavior
- Simplified version of arrive

```
class VelocityMatch:
   character
   target
   maxAcceleration
   timeToTarget = 0.1
   def getSteering(target):
      steering = new SteeringOutput()
      steering.linear = target.velocity - character.velocity
      steering.linear /= timeToTarget
      if steering.linear.length() > maxAcceleration:
         steering.linear.normalize()
         steering.linear *= maxAcceleration
      steering.angular = 0
      return steering
```

Delegated Behaviors

- we saw the building blocks: seek and flee, arrive, align, and velocity matching
- calculate a target, either position or orientation, and delegate the steering
- author uses polymorphic style of programming (inheritance, subclasses) to avoid duplicating code