Steering I

Movement Algorithms

• kinematic movement
  • computes a **velocity** for the object based on current position
  • no acceleration

• dynamic movement - "steering behaviors"
  • are dynamic - uses current velocity and acceleration
  • computes a **force** on the object
  • multiple simple behaviors can be combined to give more complex ones (e.g., seek enemy+avoid obstacles+align to face enemy)
Movement request: new velocity or forces to apply

Movement Algorithm

Game:
- other characters
- level geometry
- special locations
- paths
- other game state

Character:
- position (velocity)
- other state

Movement request
2D motion

- characters are "points"
- position: \((x, y)\)
- orientation: angle in radians (single float)
  - zero angle == facing along positive x axis
  - convert orientation angle to direction vector:
    - \(\text{dir} = (\cos(\text{angle}), -\sin(\text{angle}))\)
  - convert direction vector to angle:
    - \(\text{angle} = \text{atan2}(-y, x)\)
- velocity: \((x, y)\)
- rotation (angular velocity): radians per time unit (single float)
Kinematic Movement

- using only position and orientation, compute new velocity
- velocity can change suddenly
- can look unrealistic
- ex) Millington: c03_kinematic.exe
Kinematic Seek

velocity = self.target.position - self.character.position
velocity = normalize(velocity)
velocity *= self.maxSpeed
# face in the direction the character is going
self.character.orientationAngle = getOrientationFromVelocity(\  
    self.character.orientationAngle, velocity)
return SteeringOutput(velocity, 0.)
Kinematic Flee

• same as Seek, but in the opposite direction

velocity = self.character.position - self.target.position
Kinematic Arrive

- Seek moves always at full speed, will overshoot and oscillate if it reaches target
- Arrive slows down as it reaches target, stops if within radius of target
- uses timeToTarget: the time spent slowing down. Tries to reach in timeToTarget seconds from current position, but doesn't exceed maximum speed.
Kinematic Arrive

```python
velocity = self.target.position - self.character.position
distsq = velocity[0]**2 + velocity[1]**2
if distsq < self.radius**2:
    return None
velocity /= self.timeToTarget
distsq = velocity[0]**2 + velocity[1]**2
if distsq > self.maxSpeed * self.maxSpeed:
    velocity = normalize(velocity)
    velocity *= self.maxSpeed
# face in the direction the character is going
self.character.orientationAngle = getOrientationFromVelocity(
    self.character.orientationAngle, velocity)
return SteeringOutput ( velocity, 0. )
```
Kinematic Wander

- move in current direction at max speed
- change current direction by random amount

```
velocity = getVelocityFromOrientation(self.character.orientationAngle)
velocity *= self.maxSpeed
# change our orientation randomly
angle = numpy.random.uniform() - numpy.random.uniform()
angle *= self.maxAngularVel
return SteeringOutput ( velocity, angle )
```
Steering Behaviors (dynamic steering)

- try to do one thing, combine with other behaviors
- eg) seek : target position -> force
- eg) collision avoidance: target, geometry -> force
- **variable matching**: try to match one or more attributes of the character to the target's
  - doesn't always make sense, eg match velocity and position (if velocity matches, not moving closer to target)
  - better to use separate matching alg for each attribute, and combine them (eg arbitration)
  - for each behavior, there's an opposite: get as far away from match as possible
def process(self, time_passed):
    if ( self.steeringBehavior != None ):
        steering = self.steeringBehavior.getSteering()
        if steering != None:
            self.velocity += steering.linear * time_passed
            self.angularVelocity += steering.angular * time_passed
        else:
            self.velocity = numpy.array([0.,0.])
            self.angularVelocity = 0.
    self.position += self.velocity * time_passed
    for i in (0,1):
        if self.position[i] > SCREEN_SIZE[i]:
            self.position[i] -= SCREEN_SIZE[i]
        if self.position[i] < 0:
            self.position[i] += SCREEN_SIZE[i]
    self.orientationAngle += self.angularVelocity * time_passed
Seek and Flee

• similar to kinematic version, but doesn't control orientation (will use a separate Align behavior for that)

```python
accel = self.target.position - self.character.position
accel = normalize(accel)
accel *= self.maxAcceleration
# use a separate facing behavior for the orientation
return SteeringOutput ( accel, 0. )
```

• For Flee, flip direction
Arrive

- Move at max acceleration until close
- when close, use timeToTarget to slow down
- stop when within radius
- use Flee for opposite behavior
def getSteering(self):
    direction = self.target.position - self.character.position
    distance = length(direction)
    if distance < self.targetRadius:
        return None
    if distance > self.slowRadius:
        targetSpeed = self.maxSpeed
    else:
        targetSpeed = self.maxSpeed * distance / self.slowRadius

    targetVelocity = direction
    targetVelocity = normalize(targetVelocity)
    targetVelocity *= targetSpeed

    linear = targetVelocity - self.character.velocity
    linear /= self.timeToTarget

    if length(linear) > self.maxAcceleration:
        linear = normalize(linear)
        linear *= self.maxAcceleration
    angular = 0.

    return SteeringOutput (linear, angular)
Align

- tries to match target's orientation
- similar to Arrive, using 1-D angle
  - "slow radius" : angle difference at which we start to slow rotation speed
- convert difference in orientations to the range \([\text{-pi, pi}]\) by adding/subtracting 2\(\text{pi}\).
  - ensures we take the "shortest turn"
- no opposite: angles wrap around
  - to face in opposite direction as target: align to target angle + pi
Velocity Matching

• Also like Arrive, but tries to match target velocity instead of position
Delegated Behaviors

- **calculate** a (dummy) target position, and delegate to another behavior to compute the steering

- Seek, Align, and VelocityMatching are the **only** fundamental behaviors
Pursue and Evade

- use Seek to move to target's **predicted** position (instead of current position)
- predict based on target's position and current velocity
- assume current velocity will not change during prediction interval
# calculate a target to delegate to Seek
direction = self.mytarget.position - self.character.position
distance = length(direction)
speed = length(self.character.velocity)
if speed <= distance / self.maxPrediction:
    prediction = self.maxPrediction
else:
    prediction = distance / speed
Seek.target.position = self.mytarget.position
Seek.target.position += self.mytarget.velocity * prediction
# delegate to Seek
return Seek.getSteering()
Evade

- predict target position as in Pursue
- delegate to Flee
Face

- makes character look at target
- computes desired orientation, delegates to Align

```python
direction = self.mytarget.position - self.character.position
if length(direction) == 0:
    return None
Align.target.orientation = atan2(-direction.y, direction.x)
return Align.getSteering()
```
Look Where You're Going

- face in direction of current motion
- create a target for Align with an orientation equal to our velocity

```python
Align.target.orientation = 
    atan2(-self.character.velocity[1],self.character.velocity[0])
return Align.getSteering()
```
Wander

- can think of KinematicWander as a delegated Seek
  - pick a random spot on a circle surrounding character, move towards it
  - direction changes "twitchy"
Wander

• can reduce twitching by moving circle in front of character, and shrinking it down

• use Face or LookWhereYoureGoing to align to the target
Path Following

- A delegated behavior:
  - choose a point on the path and use Seek to follow it. (Arrive only if the point stops moving.)
- Chase the rabbit:
  - Find closest point on path to current position
    - can be hard. esp in complex regions of path with many segments
  - Select as target a point on the path at a fixed distance ahead of closest point
Predictive Path Following

- Can smooth out motion in complex curves
- First predict the future position of the character, use it to find closest point and target
- Can "cut corners" in some cases
  - prevent this with **coherence** (assuming that next value is close to previous one)
  - limit the closest point to have a parameter value close to (but greater than) that of the previous closest point
Paths

- parameterized: \( (x,y) = P(u) \)
- spline curve
- sequence of line segments

- closest point on path:
  - find closest line segment, then closest point on that segment
FollowPath

def getSteering(self):
    currentParam = self.path.getParam (
        self.character.position, currentPos)
    targetParam = currentParam + self.pathOffset
    target.position = path.getPosition(targetParam)
    return Seek.getSteering()