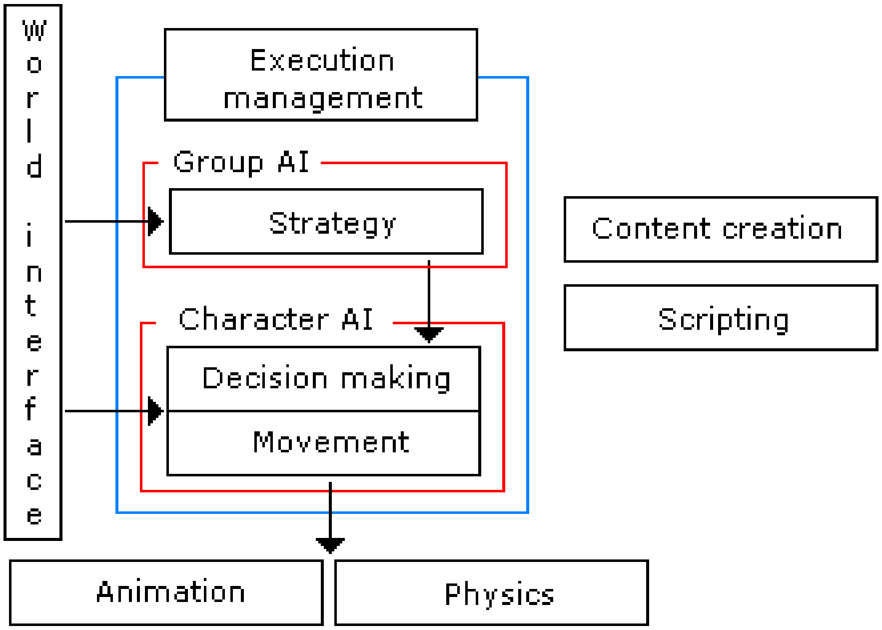
AI: Movement - Introduction

- The AI model for game architecture:

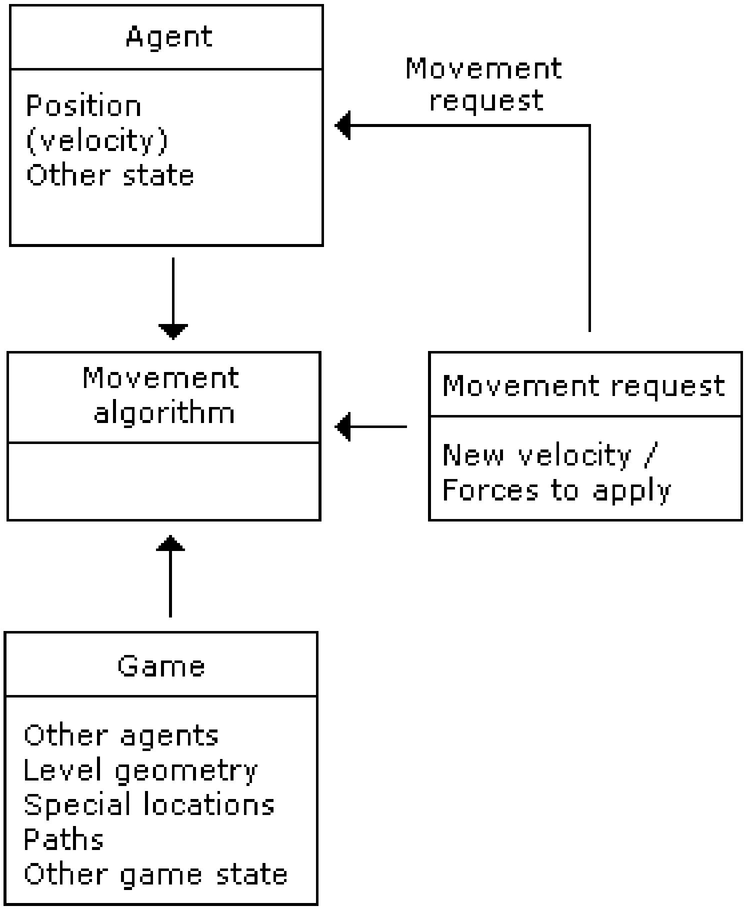


  - Movement is the most basic component of the model

- Movement is distinguished from animation
  - Movement is a large-scale operation
    - It deals with getting agent from one position to another
  - Animation represents a smaller scale
    - Concerned with the details of how agent's components transform
    - This aspect not driven by the AI

AI: Movement - Introduction (2)

- The movement algorithm is represented by the following flow chart



  - Input is geometric data
    - Represents both the agent's and world's states
  - Output is also geometric data representing the agent's and world's states
  - Actual io parameters and outputs depend on context
- Movement can be categorized as *kinematic* or *dynamic* (a *steering behavior*)
  1. Kinematic
     - Does not consider acceleration
     - Is essentially one-speed
     - Agent starts at full speed, moves to desired location, and stops

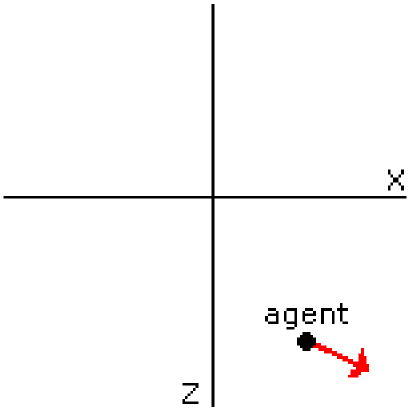
2. Steering behavior
   - Referred to as dynamic since it changes with time
     - Physics-based
   - Called *steering behavior* because it controls direction of agent
     - Term coined by Craig Reynolds, who developed flocking behavior algorithm
   - Algorithm requires agent's current position, velocity, and forces acting on the agent
   - Output is resultant force or acceleration acting on agent
   - The output is used to modify the agent's current velocity
   - the overall behavior for an agent at rest:
     - The agent accelerates to speed
     - Cruises til nears destination
     - Decelerates to a stop

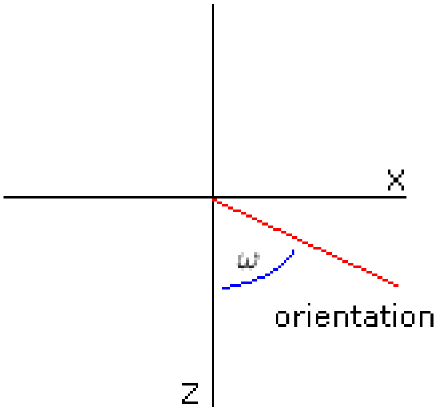
- Vectors
  - Will need a data structure to represent vectors, as they have many uses: direction, velocity, acceleration, etc.
  - A data structure that represents vectors should include
    1. Data members
       (a) x component
       (b) z component
       (c) *magnitude* (*length*), computed as
       $$|v| = \sqrt{x^2 + z^2}$$
    2. Methods
       (a) *normalize()*, which converts a vector into a unit vector
       $$\hat{v} = \left[\ x/magnitude \quad z/magnitude\ \right]^T$$
       (b) Vector addition
       $$v = v_1 + v_2 = \left[\ x_1 + x_2 \quad z_1 + z_2\ \right]^T$$
       where
       $$v_1 = \left[\ x_1 \quad z_1\ \right]^T, v2 = \left[\ x_2 \quad z_2\ \right]^T$$
       (c) Scalar multiplication
       $$v = n * \left[\ x \quad z\ \right]^T = \left[\ n * x \quad n * z\ \right]^T$$

AI: Movement - Introduction (4)

- Representation
  1. Static representation
     - This represents agent position and orientation with no motion data

     (a) 2D representation
       - i. Agent modeled as a 2D point wrt movement
       - ii. Movement is in x-z plane
       - iii. Orientation is represented as an angle between the z-axis and the vector to the agent (rotation about y-axis)
         - ∗ Assumes a right-handed coordinate system



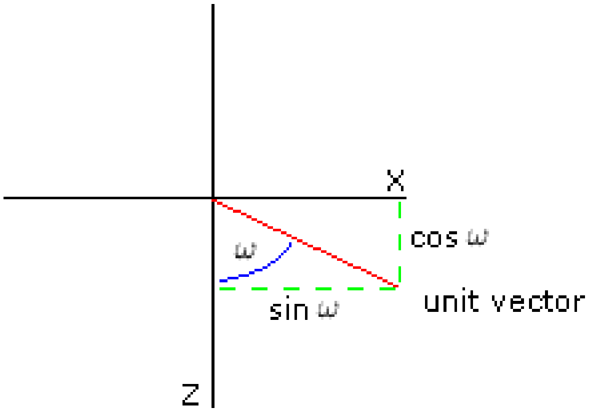


     (b) 2.5D representation
       - ∗ Position represented as a 3D point
       - ∗ Orientation represented as in 2D
       - ∗ Providing agent remains upright, can disregard orientation wrt y- and x-axes
         - · Simplifies math considerably
       - ∗ y coordinate ignored except when agent jumps, falls, or climbs

AI: Movement - Introduction (5)

- An alternative is to represent orientation as a unit vector

  $\hat{\omega} = \left[\, sin\ \omega_s \quad cos\ \omega_s \, \right]^T$



- Implementation
  - ∗ Data members would include
  - (a) A point data type for *position*
  - (b) A float for the orientation
  - ∗ Optionally, you would want to provide a way for generating the vector representation of the orientation
  - (a) It could be another data member, or
  - (b) A method that converts the float representation to a vector (as indicated above)

AI: Movement - Introduction (6)

2. Kinematic representation
   - Velocity/motion data is now incorporated
     - Have both linear (Δ motion) and angular (Δ orientation)
     - Linear represented as a vector: $(\Delta x, \Delta z)$ per second
     - Angular (rotation) represented as radians/degrees per second
   - Implementation
     - Data members would include
     - (a) A point data type for *position*
     - (b) A float for the orientation
     - (c) A vector for the velocity
     - (d) A float for the rotation
     - Do not confuse orientation and rotation
       - Orientation is a direction in which the agent is facing
       - Rotation is the rate of change of the orientation

AI: Movement - Introduction (7)

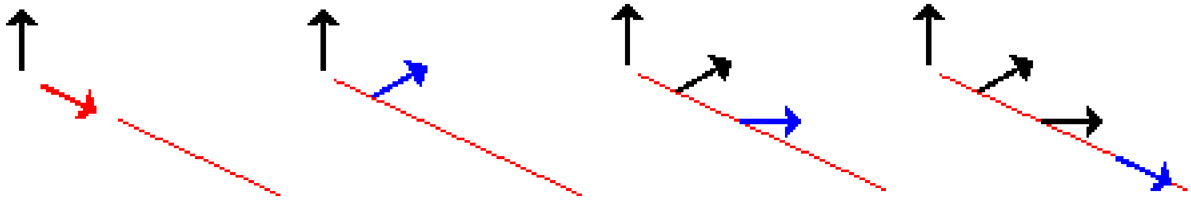
- Steering behaviors return accelerations for linear and angular change
  - Implementation of steering data structure
    - Data members would include, in addition to static members,
      1. A vector for the linear acceleration
      2. A float for the angular acceleration
  - Given a linear acceleration, change in position can be calculated using

    $$\Delta P = vt + \frac{1}{2}at^2$$
  - Since t^2 is generally very small, the Newton-Euler 1 integration update is frequently used instead

    $$\Delta P = vt$$
  - Frame rate can be used in place of time if a steady frame rate is maintained
    - Using time allows for calculations that produce steady Δp based on actual update times
  - The kinematic representation can be modified using the following algorithm

```
update (Steering s, float t)
{
  position += velocity * t;
  orientation += rotation * t;

  velocity += s.linearAcceleration * t;
  rotation += s.angularAcceleration() * t;
}
```

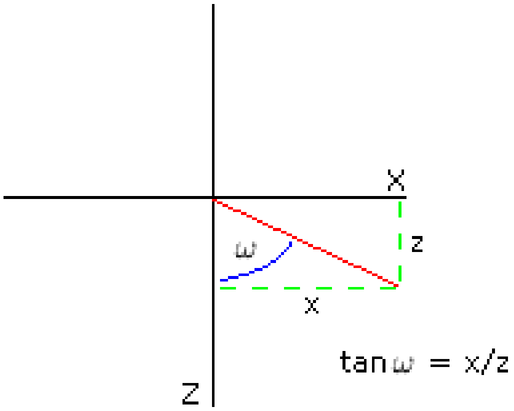
    - The steering data would be returned to the agent by some behavior that the agent is executing
  - Orientation
    - There is nothing that requires an agent to face the direction of motion, but it is usually desired
    - When agent changes direction, could simply change orientation accordingly, but this is unrealistic
    - A simple algorithm is to have the agent rotate by one half the angular difference between the heading and orientation over a sequence of frames



AI: Movement - Kinematic Movement Algorithms - Intro

- These take static agent (and target) data as input and output a velocity
  - They do not use acceleration, but may smooth changes in v
- Simple orientation
  - Generates a new orientation aligned with a new direction of motion
  - Algorithm

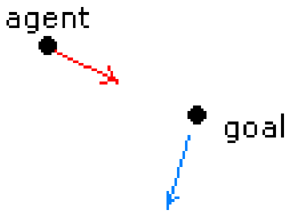
```
getNewOrientation(orientation, velocity)
{
  if (velocity.magnitude != 0.0)
    return arctan(velocity.x/velocity.z);
  else
    return orientation;
}
```

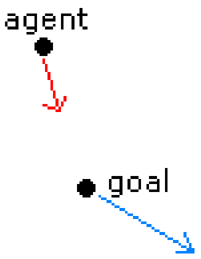


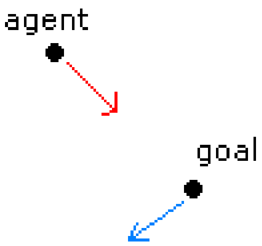
  - If $v = 0$, the current orientation is maintained
- For kinematic movement, must use static model for kinematic *Steering* output, since kinematic motion does not involve accelerations
  - Implementation of kinematic steering data structure
    - ∗ Data members would include
      1. A vector for the velocity
      2. A float for the orientation
- The assumption would be that behaviors are implemented as classes in an object-oriented language

1. Kinematic seek behavior
   - Data members
     (a) Static information (position, orientation) for the agent and target
     (b) Max speed of agent
   - Methods
     (a) *getSteering()*
       - Algorithm

```
getSteering (target)
{
  KinematicSteering s;
  s.velocity = target.position - agent.position;
  s.velocity.normalize();
  s.velocity *=  maxSpeed;
  s.orientation = 0.0;
  return s;
}
```







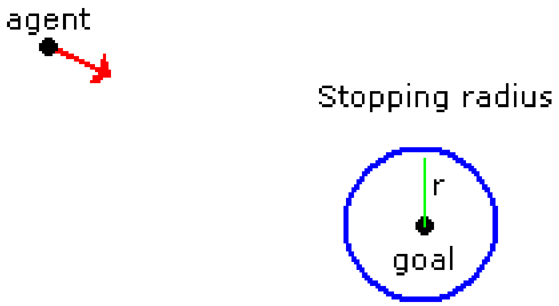
       - Note: The text includes a call to *getNewOrientation()* method
         - ∗ This method should be part of the agent's code to be consistent with steering behavior implementation

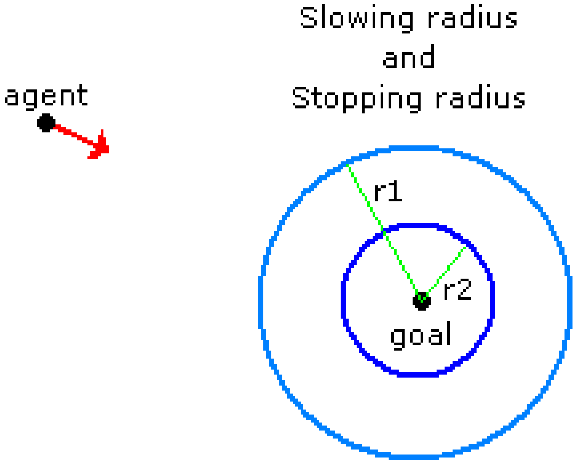
2. Kinematic flee behavior
   - Same as kinematic seek behavior, but with sign of v reversed

3. Kinematic arriving behavior
   - Kinematic seek is intended for pursuing a fleeing target
     - Agent never catches up
   - If the agent can catch or arrive at the target, the behavior will need to be modified
     - Since the agent moves at max speed, it will tend to overshoot the target, which will result in oscillation around the target

AI: Movement - Kinematic Movement Algorithms - Behaviors (2)

- Solutions:
  (a) Agent stops when within some predefined radius of target
  (b) Agent slows down as nears target
    - This can still cause oscillation, but not as severe
  (c) Combine both of the above
    - Have a large radius to signal start of speed reduction
    - Have a smaller radius for stopping





- Text deceleration algorithm based on a fixed time-to-target
- Implementation
  - Data members
    (a) Static information (position, orientation) for the agent and target
    (b) Maximum velocity
    (c) Radius that signals when to start slowing down
    (d) An arbitrary time that indicates how long it should take the agent to reach the target

– Methods

(a) *getSteering()*

- Algorithm

```
getSteering (target)
{
  KinematicSteering s;
  s.velocity = target.position - agent.position;
  if (s.velocity.magnitude < radius)
    return null;
  s.velocity /= timeToTarget;
  if (s.velocity.magnitude > maxSpeed){
    s.velocity.normalize();
    s.velocity *= maxSpeed;
  }
  s.orientation = 0.0;
  return s;
}
```

- The net effect of the *getSteering()* computations for v is demonstrated by the following example

  Consider agent at (0, 0) and target at (64, 0)
  Agent's max speed = 16
  Time to target = 2

time	0	1	2	3	4	5
distance to target	64	48	32	16	8	4
v	$32 \rightarrow 16$	$24 \rightarrow 16$	16	8	4	2

4. *Wandering* behavior
   - The agent simply travels at max speed in the direction of the current orientation
     - As the name implies, the orientation varies randomly
     - This is implemented by the steering method
   - Implementation
     - Data members
       (a) Static information (position, orientation) for the agent
       (b) Maximum velocity
       (c) Maximum rotation
     - Methods
       (a) *getSteering()*
         * Algorithm

```
getSteering (target)
{
  KinematicSteering s;
  s.velocity = agent.orientation(vector rep) * maxSpeed;
  s.orientation = randomBinomial() * maxRotation;
  return s;
}
```

       (b) *randomBinomial()*
         * Method *randomBinomial()* returns a value $-1.0 \leq x \leq 1.0$, generating values near 0.0 with greater probability
         * Algorithm

```
randomBinomial ()
{
  return random() - random();
}
```

AI: Movement - Steering Behaviors, Introduction

- These behaviors incorporate acceleration (both linear and angular) and compute changes in each
- All behaviors are based on the kinematic data of the agent (and target, if warranted)
  - Implementation
    1. Data members would include
       (a) A point data type for *position*
       (b) A float for the orientation
       (c) A vector for the velocity
       (d) A float for the rotation
       (e) A float for the max speed
    2. Methods would include
       (a) *getSteering()*
           - This returns a steering object that contains
             i. Acceleration
             ii. Rotation
  - Agents will include an *update()* method
    - This method use the results of the steering behavior to compute a new velocity and orientation
    - *update()*

```
update (Steering s, float maxSpeed, float time)
  {

    //Update posn and orientation
    position += velocity * time;
    orientation += rotation * time;

    //Update v and rotation
    velocity += s.linearAcceleration * time;
    rotation += s.angularRotation * time;

    //Clamp v
    if (velocity.magnitude > maxSpeed) {
      velocity.normalize();
      velocity *= maxSpeed;
    }
  }
  ...
}
```

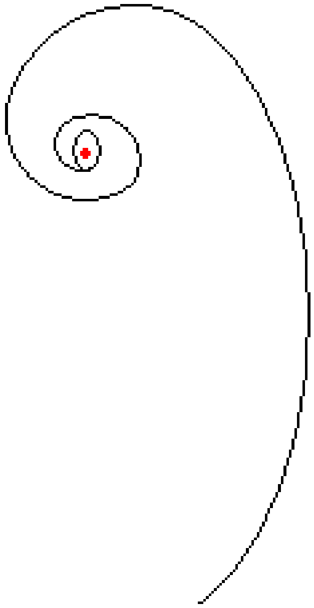
- Some behaviors will require multiple target inputs

AI: Movement - Steering Behaviors, Introduction (2)

- Steering behaviors are hierarchical
  - Start with a set of primitives
  - More complex behaviors arise by combining results of multiple primitive behaviors that an agent is executing
  - Generally do not create specific complex behaviors
- Variable matching
  - Term given to behaviors that try to match one or more kinematic elements of an agent with the corresponding ones of the target
    - E.g., orientation, v, position
  - Generally a behavior will match only one
    - If try to match several, frequently results in conflicts
    - E.g., position and v

AI: Movement - Steering Behavior Primitives, *Seek* and *Flee*

- *Seek* will
    1. Calculate direction to target (like kinematic case)
    2. Accelerate to max speed in that direction
    3. If $v > vMax$, v is clamped to $vMax$
        - Note that this is done *after* the fact, and not by the steering behavior
- Drag can also be taken into account
    - Drag imposes a limit on the max speed an agent can attain
        - ∗ It eliminates the need to check if v has exceeded the max
    - If the target is moving, the agent will spiral in toward the target and orbit around it
        - ∗ With drag, the orbiting is eliminated



AI: Movement - Steering Behavior Primitives, *Seek* and *Flee* (2)

- Implementation
  - Data members
    1. *agent*: Kinematic information (position, orientation, v, rotation) for the agent
    2. *target*: Kinematic information for the target
  - Methods
    1. *getSteering()*
       - Algorithm

         ```
         getSteering (target)
         {
           KinematicSteering s;

           s.linearAcceleration = target.position - agent.position;
           s.linearAcceleration.normalize();
           s.linearAcceleration *=  maxAcceleration;
           s.rotation = 0.0;
           return s;
         }
         ```
       - Orientation is not included since it is its own behavior
- Flee
  - Like its kinematic counterpart, *flee* is simply *seek* with the sign of v reversed

AI: Movement - Steering Behavior Primitives, *Arrive* and *Leave*

- As discussed above, the target of *seek* is assumed to be moving
  - If the agent catches the target, it will spiral in and orbit the target
  - *Arrive* is designed to stop when it reaches the target
- When nearing the target, the agent will decelerate
  - The algorithm will use two radii
    1. One to signal the beginning of deceleration
    2. One to signal a distance acceptably close for stopping
  - The speed up to the outer radius is *maxSpeed*
  - The speed anywhere within the inner radius is 0
  - Speed is interpolated between the two radii
    - Target v is determined based on time, and a is adjusted accordingly
- Implementation
  - Data members
    1. *agent* and *target*: Agent and target kinematic data
    2. *maxAcceleration*, *maxVelocity*: Acceleration and velocity limits of agent
    3. *innerRadius* and *outerRadius*: Arrive and slow radii
    4. *timeToTarget*: Time alloted for reaching target

AI: Movement - Steering Behavior Primitives, *Arrive* and *Leave* (2)

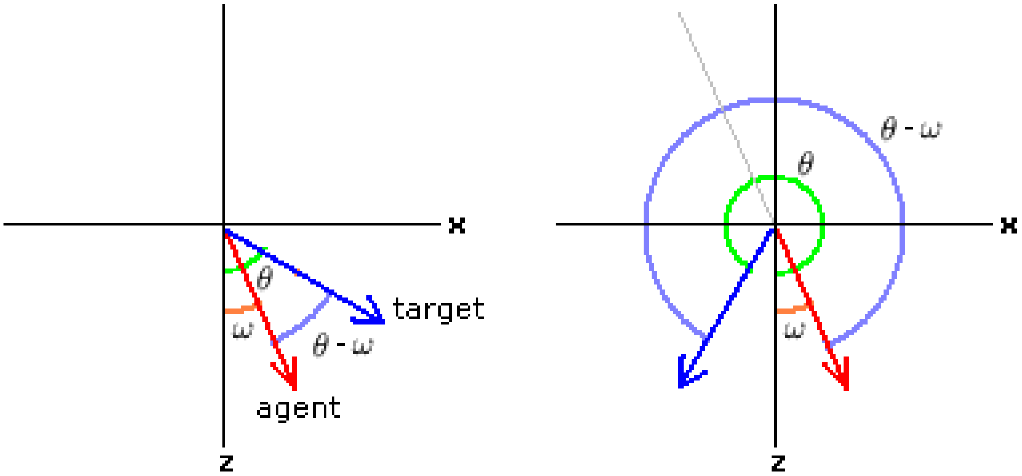
  - Methods
    1. *getSteering()*
       * Algorithm

```
getSteering (target)
{
  kinematicSteering s;

  direction = target.position - agent.position;
  distance = direction.magnitude;
  if (distance < innerRadius)
    return null;
  if (distance > outerRadius)
    targetSpeed = maxSpeed;
  else
    targetSpeed = maxSpeed * distance / outerRadius;
  targetVelocity = direction;
  targetVelocity.normalize();
  targetVelocity *= targetSpeed;
  s.linearAcceleration = targetVelcocity - agent.velocity;
  s.linearAcceleration /= timeToTarget;

  if (s.linearAcceleration.magnitude > maxAcceleration) {
    s.linearAcceleration.normalize();
    s.linearAcceleration *= maxAcceleration;
  }
  s.rotation = 0.0;
  return s;
}
```

- Leave
  - This is not the opposite of *arrive*
  - This behavior most likely wants to reach max accerelation immediately, rather than slowly speeding up
  - A more appropriate opposite is *flee*

AI: Movement - Steering Behavior Primitives, *Align*

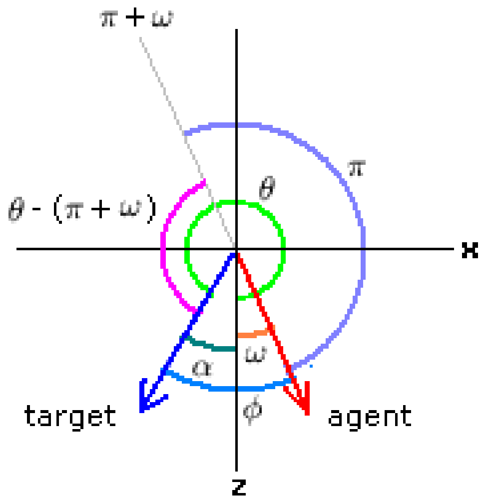
- Tries to match agent's orientation with that of target
- Acts like *arrive*
  - Want $rotation = 0$ when reach target value



- Do not want to simply perform $target.orientation - agent.orientation$
  - Must take into account the fact that the agent could rotate clockwise or counterclockwise to match the target's orientation
  - Unless the rotation is π, one will be smaller than the other
  - Want the agent's rotation to be in range $[-\pi, +\pi]$
  - Can take advantage of the fact that rotation resets every 2π radians

    Let $\phi = \theta - \omega$
    if $-\pi \leq \phi \leq \pi, \phi = \phi$
    if $\phi > \pi, \phi = \phi - 2\pi$
    if $\phi < -\pi, \phi = \phi + 2\pi$

AI: Movement - Steering Behavior Primitives, *Align* (2)



$$\begin{aligned} \phi &= \alpha + \omega \\ &= \pi - (\theta - (\pi + \omega)) \\ &= 2\pi - (\theta - \omega) \\ -\phi &= (\theta - \omega) - 2\pi \end{aligned}$$

- Like *arrive*, use two "radii"
  - Are actually intervals, since orientation is a scalar
- Implementation
  - Data members
    1. *agent* and *target*: Agent and target kinematic data
    2. *maxAcceleration*, *maxRotation*: Acceleration and rotation limits of agent
    3. *innerRadius* and *outerRadius*: Arrive and slow radii
    4. *timeToTarget*: Time alloted for reaching target

- Methods
  1. *getSteering()*
     - Algorithm

```
getSteering (target)
{
  kinematicSteering s;

  rotation = target.orientation - agent.orientation;
  rotation = mapToRange(rotation);
  rotationSize = abs(rotation);
  if (rotationSize < innerRadius)
    return null;
  if (rotationSize > outerRadius)
    targetRotation = maxRotation;
  else
    targetRotation = maxRotation * rotationSize / outerRadius;
  targetRotation *= rotation / rotationSize;        //**incorporate direction
  s.angularAcceleration = targetRotation - agent.rotation;
  s.angularAcceleration /= timeToTarget;
  angularAcceleration = abs(s.angularAcceleration);
  if (s.angularAcceleration.magnitude > maxAngularAcceleration) {
    s.angularAcceleration /= angularAcceleration;
    s.angularAcceleration *= maxAngularAcceleration;
  }
  s.linearAcceleration = 0.0;
  return s;
}
```

     - The lines

       angularAcceleration = abs(s.angularAcceleration);
       s.angularAcceleration /= angularAcceleration;

       normalize the angular acceleration
  2. *mapToRange()*
     - Algorithm

```
mapToRange(float rotation)
{
   if (rotation > PI)
     return rotation - 2 * PI;
   if (rotation < -PI)
     return rotation + 2 * PI;
   return rotation;
}
```

- *LookAway*
  - Simply add π to result of *Align*, and use that as the target rotation

AI: Movement - Steering Behavior Primitives, Velocity Matching

- Refers to having agent move with same velocity as target
- Text notes it is most important when dealing with combined behaviors (see following topics)
- *Arrive* behavior can be modified to achieve this
- Implementation
  - Data members
    1. *agent* and *target*: Agent and target kinematic data
    2. *maxAcceleration*, *maxRotation*: Acceleration and rotation limits of agent
    3. *timeToTarget*: Time alloted for reaching target
  - Methods
    1. *getSteering()*
       * Algorithm

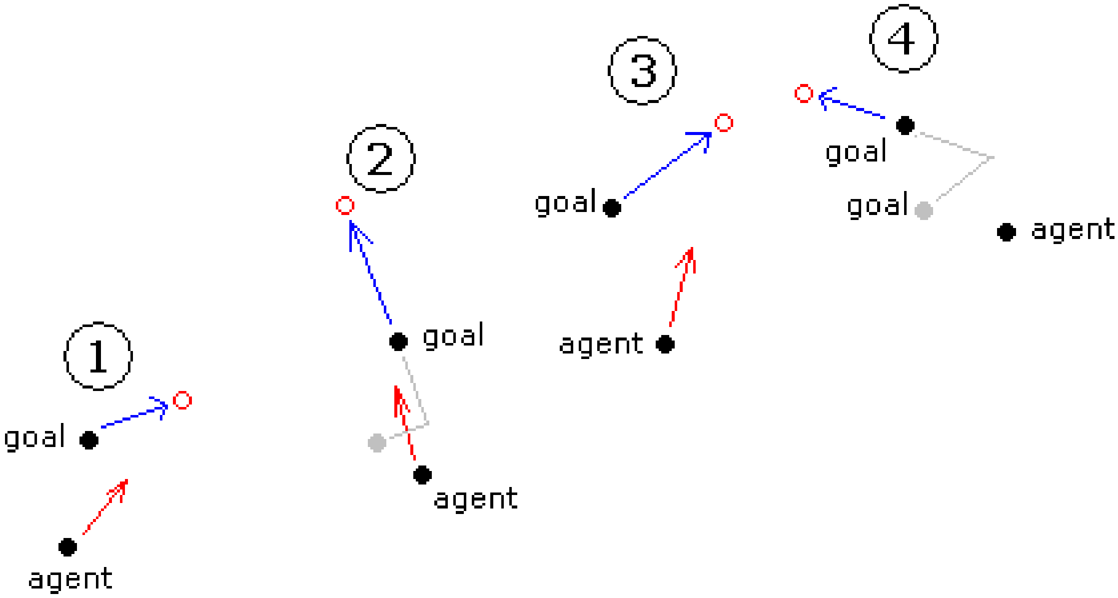
```
getSteering (target)
{
  kinematicSteering s;

  s.linearAcceleration = target.velocity - agent.velocity;
  s.linearAcceleration /= timeToTarget;
  if (s.linearAcceleration.magnitude > maxAcceleration) {
    s.linearAcceleration.normalize();
    s.linearAcceleration *= maxAcceleration;
  }
  s.rotation = (0.0);
  return s;
}
```

AI: Movement - Delegated Steering Behaviors, Introduction

- This type of behavior delegates actions to other behaviors
- It calculates a target value (e.g., position, orientation, velocity), and calls on another behavior to generate the steering output
- This effectively represents a hierarchy of behaviors
- The text implements these in terms of inheritance

AI: Movement - Delegated Steering Behaviors, *Pursue* and *Evade*

- In *Seek*, an agent moves toward a target based on the target's position
  - When the target is moving - assumed by *Seek* - the agent is alway moving toward a location that the target will no longer be occupying by the time the agent gets there
- *Pursue* attempts to predict where the target is heading, and determines an intercepting direction based on the target's current position and velocity



- The algorithm presented here is based on the one developed by Craig Reynolds
  - It assumes the target will maintain its current velocity in the short term
  - Steps:
    1. Find distance to target
    2. Find time to target if travel at max speed
       - This is used as the look ahead time
    3. Determine the target's position at the end of this time (assuming current velocity)
       - This is the target for the agent
  - The result is that the agent will make a series of adjustments as it closes in on target
  - Since the calculated time could be large, and the target may try evasive maneuvers, want a limit on the time
- *Pursue* delegates to *Seek*

AI: Movement - Delegated Steering Behaviors, *Pursue* and *Evade* (2)

- Implementation
  - Data members
    1. *agent* and *target*: Agent and target kinematic data
    2. *maxPredictionTime*: Look ahead time limit
    3. *newTarget*: a simulated target that represents the goal the agent is aiming for
  - Methods
    1. *getSteering()*
       - Algorithm

```
getSteering (target)
{
  direction = target.position - agent.position;
  distance = direction.magnitude;
  speed = agent.velocity.magnitude;
  if (speed <= distance / maxPredictionTime)
    predictionTime = maxPredictionTime;
  else
    predictionTime = distance / speed;
  newTarget = copy(target);
  newtarget.position = target.position + target.velocity * prediction;
  agent.Seek.getSteering(newTarget);
}
```

       - Note: The text has *Pursue* extend the *Seek* class
       - *Seek* has a target data member, which the text directly manipulates in the above algorithm - not good
         - I instead chose to pass the goal target as a parameter to *Seek*'s *getSteering()* method (the original takes no arguments)
- *Evade*
  - Delegates to *Flee*
  - If the target is not moving, will result in oscillation of agent around target
  - To remedy, delegate to *Arrive* instead

AI: Movement - Delegated Steering Behaviors, *Face*

- The agent faces the target
- The algorithm determines the orientation of the target wrt the agent
  - It then rotates to face in that direction
- *Face* delegates to *Align* for the orientation
- Implementation
  - Data members
    1. *agent* and *target*: Agent and target kinematic data
    2. *newTarget*: a simulated target that represents the goal the agent is aiming for
  - Methods
    1. *getSteering()*
       * Algorithm

```
getSteering (target)
{
  direction = target.position - agent.position;
  if (direction.magnitude == 0.0)
    return target;
  newTarget = copy(target);
  newtarget.orientation = atan(direction.X/direction.Z);
  agent.align.getSteering(newTarget);
}
}
```

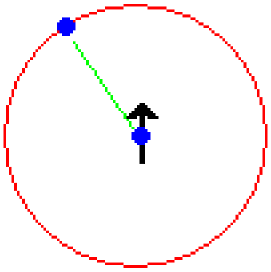
AI: Movement - Delegated Steering Behaviors, *FaceHeading*

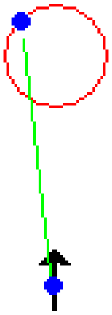
- Note: Text refers to this as ”Looking where you’re going”
- In the kinematic behaviors, the agent’s orientation is simply set - a discontinuous change
  - *FaceHeading* will delegate to *Align* after computing *rotation*
- This is coded similarly to *Face*
- Implementation
  - Data members
    1. *agent* and *target*: Agent and target kinematic data
    2. *newTarget*: Simulated target that represents the goal the agent is aiming for
  - Methods
    1. *getSteering()*
       - Algorithm

```
getSteering ()
{
  if (agent.velocity.magnitude == 0.0)
    return;
  newTarget = copy(agent);
  newtarget.orientation = atan(-agent.velocity.X/agent.velocity.Z);
  agent.align.getSteering(newTarget);
}
```

AI: Movement - Delegated Steering Behaviors, *Wander*

- *Wander* can be implemented similarly to *Seek*
    - Consider a circle centered on the agent to which a target is constrained
    - The target moves around the circle randomly
    - To preclude sharp reversals of direction, shrink the radius of the circle and have it centered in front of the agent





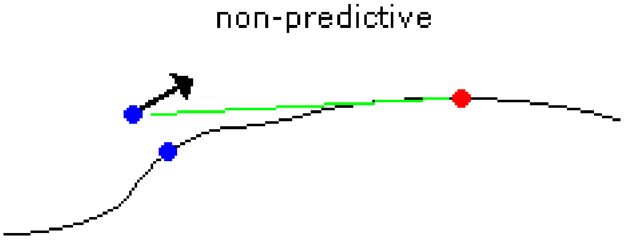
- Will implement by delegating to *Face* to have agent facing along heading
    - Could achieve this by incorporating *FaceHeading* with *Seek*
    - The angle subtended by the direction between the agent and target and the agent's orientation determines rate of rotation
    - Will maintain full acceleration toward target
- Implementation
    - Data members
        1. *wanderOffset*: Distance to circle
        2. *wanderRadius*: Radius of circle
        3. *maxRotation*: Max rate of rotation (*wanderRate* in text)
        4. *wanderOrientation*: Current orientation of target
        5. *maxAcceleration*: Max acceleration of character

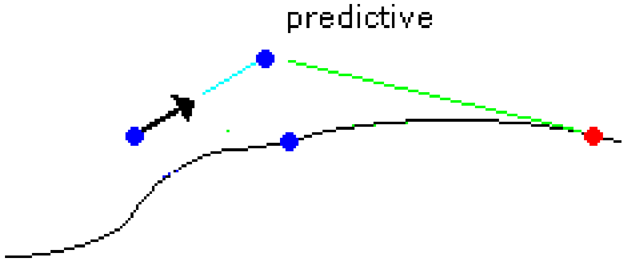
- Methods
  1. *getSteering()*
     - Algorithm

```
getSteering ()
{
  kinematicSteering s;

  //Note: Using vector version of orientation
  wanderOrentation += randomBinomial() * wanderRate;
  newTarget = copy(agent);
  newTarget.orientation = wanderOrientation + agent.orientation;
  newTarget.position = agent.position + wanderOffset * agent.orientation;
  newTarget.position += wanderRadius * targetOrientation;
  s = Face.getSteering(newTarget);
  s.linearAcceleration = maxAcceleration * agent.orientation;
  return s;
}
```

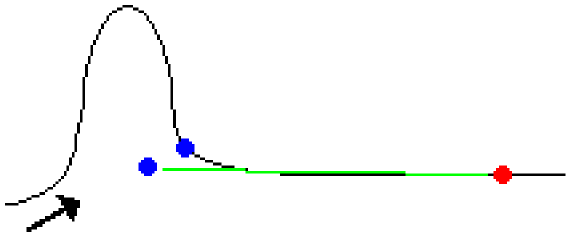
AI: Movement - Delegated Steering Behaviors, *FollowPath*

- In path following, the goal of the agent is to follow a path as closely as possible
- The strategy is to identify a target on the path, and then delegate to *Seek*
  1. Find the point on the path closest to the agent
  2. Find the point on the path that is some fixed distance away (along the general heading of the agent)
- Predictive path following alters this strategy
  1. Determine where agent will be in a short time
  2. Find the point on the path closest to this position
  3. Proceed as above





- Predictive path following produces smoother motion, but may result in taking shortcuts



- Locating the target position on the path is not simple, especially for complex curves

- Implementation (non-predictive)
  - Data members
    1. *path*: The path being followed
    2. *pathOffset*: Parametric distance to advance the target
    3. *currentParam*: Parameter corresponding to the agent's current location on the path
    4. *nearestPosition*: Nearest position on path to agent's current position
    5. *newTarget*: Simulated target that represents the goal the agent is aiming for
  - Methods
    1. *getSteering()*
       - Algorithm

```
getSteering (target)
{
  kinematicSteering s;

  newTarget = copy(agent);
  currentParam = path.getParam(agent.position, currentPos);
  targetParam = currentParam + pathOffset;
  newTarget.position = path.getPosition(targetParam);
  return Seek.getSteering(newTarget);
}
```

- The *Path* data type
  - The implementation assumes that the path is represented by a parameterized equation
    - In these representations, distance along the path is represented by a parameter in the range $[0.0, 1.0]$
    - Given points $p_0(x_0, z_0)$ and $p_1(x_1, z_1)$, the parametric equation for the line segment from p_0 to p_1 is

      $p_x(t) = x_0 + t(x_1 - x_0)$

      $p_z(t) = z_0 + t(z_1 - z_0)$
    - For example, the standard parametric representation of a line segment between points $(2, 8)$ and $(18, 0)$ is

      $p_x(t) = 2 + 16t$, $p_y(t) = 8 - 8t$
      - At $t = 0.0$, $p(t) = (2, 8)$
      - At $t = 0.5$, $p(t) = (10, 4)$
      - At $t = 1.0$, $p(t) = (18, 0)$

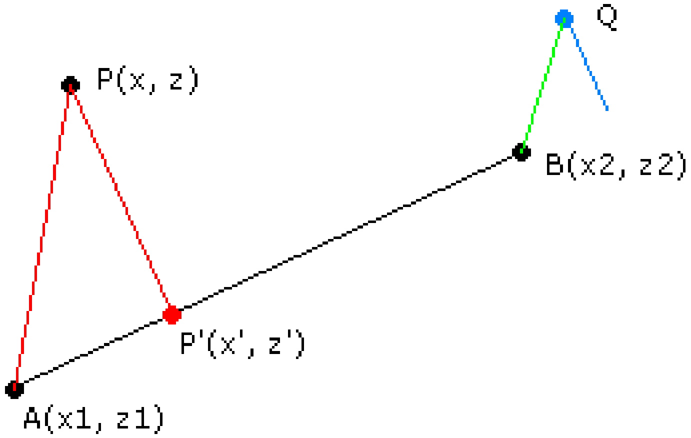
AI: Movement - Delegated Steering Behaviors, *FollowPath* (3)

    - To define the line through p_0 and p_1, $-\infty \le t \le +\infty$
    - Many curves (e.g., splines) have parametric representations
  - The method *getParam(position, nearestPosition)* returns the parameter for agent's nearest point on the curve
  - *getPosition(param)* returns the point on the path that corresponds to the *param* argument

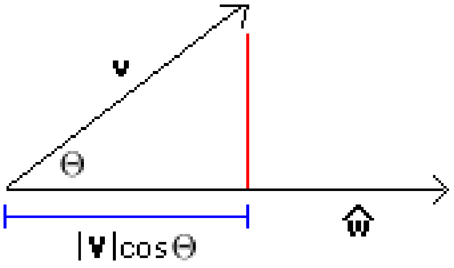
- Finding closest point to a curve

  1. Lines

     - Consider the following



     - Given line segment $\overline{AB}$ and point $P(x, z)$, find point $P'(x', z')$ on $\overline{AB}$ closest to P
       - Assume a parameterized representation for $\overline{AB}$
       - We know that the dot product of a vector $\mathbf{v}$ with a unit vector $\widehat{\mathbf{w}}$ is the magnitude of the projection of $\mathbf{v}$ onto $\widehat{\mathbf{w}}$



$$\begin{aligned} \mathbf{v} \cdot \widehat{\mathbf{w}} &= |\mathbf{v}||\widehat{\mathbf{w}}|\cos\Theta \\ &= |\mathbf{v}|\cos\Theta \end{aligned}$$

- ∗ This is the basis for the algorithm:
  - (a) Normalize $\overline{AB}$ to give $\widehat{\mathbf{AB}}$
  - (b) Take the dot product of $\mathbf{AP}$ and $\widehat{\mathbf{AB}}$ to give $|\mathbf{AP}'|$
  - (c) Calculate $t = \frac{|\mathbf{AP}'|}{|\mathbf{AB}|}$
    - · t is the value in the parametric equation for $\overline{AB}$ that corresponds to P

– If P lies beyond the line segment (as does Q in the above figure), it must be handled as a special case

- ∗ The above algorithm would find the point that represents the closest distance to the line that extends beyond $\overline{AB}$, which is not correct
- ∗ For Q, the closest point would be B
- ∗ To check for this special case, examine the value of t: if $t \notin [0.0, 1.0]$, then the closest point is A or B

2. Curves

– Consider point $P(a, b)$ and the curve defined by $y = f(x)$

- ∗ The distance between P and point $Q(x, y)$ on the curve is given by

$$d = \sqrt{(x-a)^2 + (f(x) - b)^2}$$

- ∗ The minimal distance d_m can be found by finding d', and solving for x when $d' = 0$
  - · This is generally simple to solve for quadratic curves, but becomes difficult for curves of higher order
- ∗ The *Newton-Raphson* approximation can be used to solve for x in all cases
  - · The formula used is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

AI: Movement - Delegated Steering Behaviors, *FollowPath* (5)

- Algorithm:

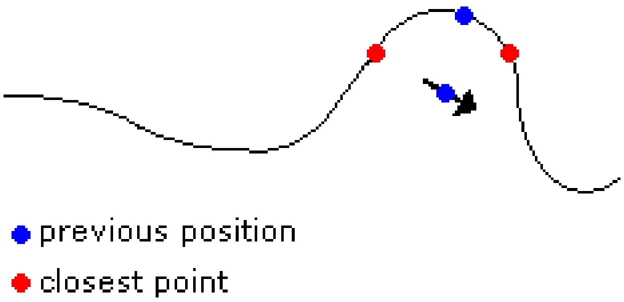
```
x1 = "close" guess for x
epsilon = some arbitrarily small convergence value
repeat {
  x0 = x1
  x1 = x0 - f(x0)/f'(x0)
} until x1 - x0 <= epsilon
return x1
```

AI: Movement - Delegated Steering Behaviors, *FollowPath* (6)

- Implementation (predictive)
  - Data members
    1. Same as in non-predictive version
    2. *predictTime*: Used to determine agent's future position
  - Methods
    1. *getSteering()*
       - ∗ Algorithm

```
getSteering (target)
{
  kinematicSteering s;

  newTarget = copy(agent);
  futurePosition = agent.position + agent.velocity * predictTime;
  currentParam = path.getParam(futurePos, currentPos);
  targetParam = currentParam + pathOffset;
  newTarget.position = path.getPosition(targetParam);
  return Seek.getSteering(newTarget);
}
```

  - The method *getParam(position, lastParam)* returns the parameter for agent's nearest point on the curve
  - *getPosition(param)* returns the point on the path that corresponds to the *param* argument
  - *getParam()* will generate the new parameter not too far from the previous one
    - ∗ This is called *coherence*
    - ∗ If the algorithm allows large distances between consecutive parameters, it may result in shortcuts as above



AI: Movement - Steering Behaviors, *Separation*

- *Separation* keeps agents in a crowd from getting too close
  - Sometimes referred to as *repulsive steering*
- If agents' paths cross, obstacle avoidance (collision detection) is a better choice
- If there are no agents close, the ouput is 0
- This behavior is similar to *Avoid*, but repulsive strength varies with distance
  - Could use linear dropoff

    $strength = maxAcceleration * (threshold - distance)/threshold$
    where *threshold* is the maximum distance at which the behavior has effect
  - or inverse-square

    $strength = min(k/distance^2, maxAcceleration)$
    where k is an arbitrary positive value that determines how fast the decay is
- If multiple agents are within the effective radius, the calculation is performed for each and the results summed
- Implementation
  - Data members
    1. *agent*: Holds kinematic data of agent
    2. *targets*: A list of target agents
    3. *threshold*: As described above
    4. *k*: As described above
    5. *maxAcceleration*

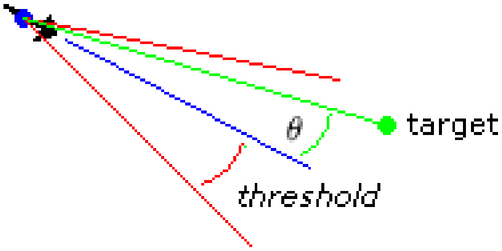
- Methods
    1. *getSteering()*
        * Algorithm

```
getSteering (target)
{
  kinematicSteering s;

  s.linear = 0;
  for (target in targets) {
    direction = target.position - agent.position;
    distance = direction.magnitude;
    if (distance < threshold) {
      strength = min(k / (distance * distance), maxAcceleration);
      direction.normalize();
      s.linear += strength * direction;
    }
  }
  return s;
}
```

- For large numbers of agents, there are data structures that can be used to represent proximity among agents
    - These preclude the brute force computation of distance between an agent and every other one (an n^2 algorithm) at the cost of greater complexity in representation
- By reversing the sign, we have an attractive behavior
    - When there are attractive and repulsive forces, a number of issues must be dealt with
    - These will be discussed in later behaviors

- This behavior can be implemented as a variation of *Evade* or *Separation*
  - It is triggered when a target is within an arbitrary cone projected in front of the agent
  - We can use the dot product to determine this:

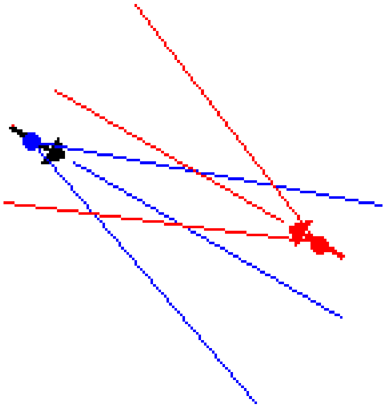
    Let *orientation* be the *normalized* vector representation of the agent's orientation
    Let *direction* be the *normalized* direction from the agent to the target
    Since $\mathbf{v} \cdot \mathbf{w} = |\mathbf{v}||\mathbf{w}| cos\ \theta$,
      where θ is the angle between $\mathbf{v}$ and $\mathbf{w}$, and we're using normalized vectors
    We simply need to check whether
      $arccos(orientation \cdot direction) \in [-threshold, threshold]$,
      where *thresold* is the cutoff angle from
    the axis of orientation

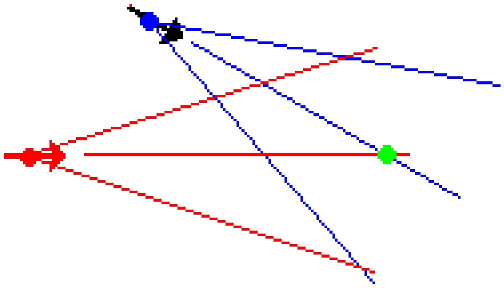


- If multiple targets lie within cone, could
  1. Use average position of group
  2. Use target closest to agent

AI: Movement - Steering Behaviors, *Collision Avoidance* (2)

- The above strategy is too simplistic
  - Consider the following cases





    - In the left figure, each agent will detect a collision even though none will occur given the current trajectories
    - In the right figure, neither agent senses a collision even though one is imminent
  - The problems arise because velocity is not taken into account
- Amended strategy:
  - Using the current velocities, find the point of closest approach
  - If the distance between the agents at this point is less than a threshold value, activate collision avoidance
  - Note that the point of closest approach is not necessarily the point where the trajectories intersect
  - The following calculations can be used

    Let $\mathbf{delta_p} = p_t - p_a$ be the direction from the agent to the target
    Let $\mathbf{delta_v} = \mathbf{v_t} - \mathbf{v_a}$ be the *relative* velocity of the agent toward the target
    Then, $t_{closest} = -\frac{\mathbf{delta_p} \cdot \mathbf{delta_v}}{|\mathbf{delta_v}|^2}$, where t represents time
    Finally, $p'_a = p_a + \mathbf{v_a} t_{closest}, p'_t = p_t + \mathbf{v_t} t_{closest}$,
    where p'_a and p'_t are the points of closest approach for the agent and target
  - If $t < 0$, it means that the agents ahve already passed each other
  - If a collision is detected, use the projected point as the position to avoid

  - May want to perform a panic check to see whether a collision is about to occur now
    - Could do this first, to avoid the above computations if collision were imminent
    - If indicated, take immediate avoidance measures
- For groups of targets, using a center of mass will not work
  - Rather, base avoidance on closest target, then update wrt next closest, etc.
- Implementation
  - Data members
    1. *agent*: Holds kinematic data of agent
    2. *targets*: A list of target agents
    3. *radius*: Collision radius of agent
    4. *maxAcceleration*

AI: Movement - Steering Behaviors, *Collision Avoidance* (4)

– Methods

1. *getSteering()*
    - ∗ Algorithm

```
getSteering (target)
{
  kinmaticSteering s;

  s.linear = 0;
  shortestTime = $\infty$;
  firstTarget = null;
  for (target in targets) {
    relativePos = target.position - agent.position;
    reletiveVelocity = target.velocity - agent.velocity;
    relativeSpeed = relativeVelocity.magnitude;
    timeToCollision = dotProduct(relativePos, relativeVelocity) /
                        (relativeSpeed * relativeSpeed);
    distance = relativePos.magnitude;
    minSeparation = distance - relativeSpeed * timeToCollsion;
    if (minSeparation <= 2 * radius) {
      if ((timeToCollision > 0.0) && (timeToCollision < shortestTime)) {
        shortestTime = timeToCollision;
        firstTarget = target;
        firstMinSeparation = target;
        firstDistance = distance;
        firstRelativePos = relativePos;
        firstRelativeVelocity = relativeVelocity;
      }
    }
  }
  if (firstTarget == null)
    return null;
  if ((firstMinSeparation <= 0.0) || (firstDistance <= 2 * radius))
    relativePos = firstTarget.position - agent.position;
  else
    relativePos = firstRelativePos + firstRelativeVelocity * shortestTime;
  relativePos.normalize();
  s.linear = relativePos * maxAcceleration;
  return s;
}
```

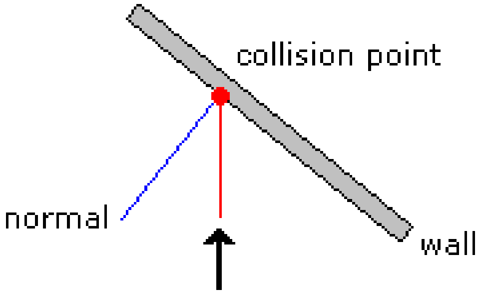
AI: Movement - Delegated Steering Behaviors, *Obstacle and Wall Avoidance*

- Algorithms to this point based on bounding spheres
  - Won't work for obstacles like walls
- Instead, use the following strategy
  - Have agents cast rays along direction of travel
  - Check if they intersect anything
  - If detect an obstacle, generate an avoidance target and delegate to *Seek*
  - Rays are not infinite
    - Are long enough to account for a few seconds of agent movement
- Implementation
  - Data members
    1. *agent*: Kinematic data for agent
    2. *target*: Kinematic data for target
    3. *collisionDetector*
    4. *avoidDistance*: Min acceptable distance to a wall
    5. *lookahead*: Lookahead distance
  - Methods
    1. *getSteering()*
       - Algorithm

```
name getSteering (target)
{
  rayVector = agent.velocity;
  rayvector.normalize();
  rayvector += lookahead;
  collision = collisionDetector.getCollision(agent.position, rayVector);
  if (collision == null)
    return null;
  target.position = collision.position + collision.normal * avoidDistance;
  return Seek.getSteering(target);
}
```

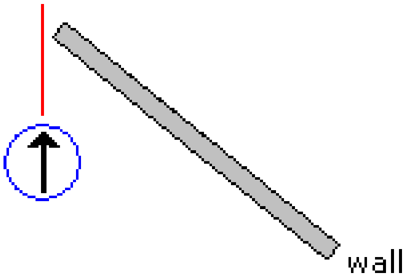
AI: Movement - Delegated Steering Behaviors, *Obstacle and Wall Avoidance* (2)

- Class *collisionDetector*:
  - Implementation
    - Data members
      1. *collision*: Data structure that holds
         (a) Point of collision
         (b) Normal to the point



    - Methods
      1. *getCollision(Point2D position, Vector2D v)*
         - Returns a *Collision* object if a collision is detected; *null* otherwise

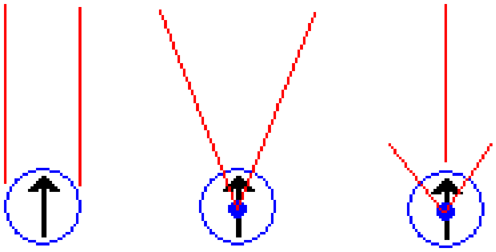
- Issues
  1. One ray is not sufficient



  - A single ray can indicate that no obstacles lie ahead, and yet the side of the agent will collide

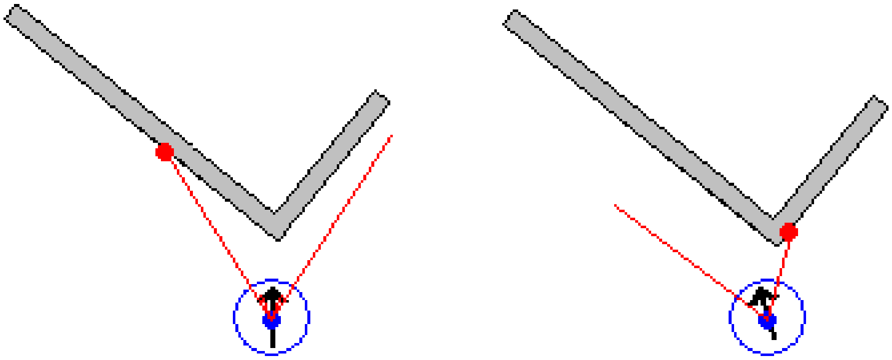
AI: Movement - Delegated Steering Behaviors, *Obstacle and Wall Avoidance* (3)

- The solution is to use multiple rays



- Each of the configurations has its strengths and weaknesses

2. Corner trap
   - This problem is associated with acute corners



   - In the figure, the left ray intersects the wall
     - ∗ This results in a target being generated along the normal to the wall at the point of intersection
     - ∗ The agent rotates counter-clockwise
     - ∗ Now, the right ray intersects, causing the agent to rotate clockwise
   - The result is back-and-forth oscillation as the agent nears the corner
   - Solutions
     (a) Use a wide fan angle
       - ∗ Prohibits agent from traversing narrow hallways
       - ∗ May result in collisions with agent's sides because will not detect obstacles directly ahead along sides

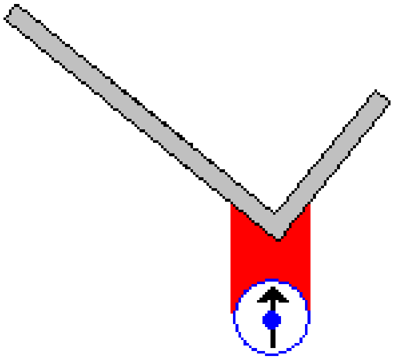
     (b) Use adaptive fan width
       - ∗ When no collisions detected, use narrow width
       - ∗ On collisions, widen the angle
       - ∗ The greater the successive hits, the greater the angle

AI: Movement - Delegated Steering Behaviors, *Obstacle and Wall Avoidance* (4)

(c) Check corner trap as special case

  - ∗ When detected, arbitrarily turn toward one side, ignoring the rays on the other

(d) Project an area (volume)



AI: Movement - Combining Steering Behaviors, Introduction

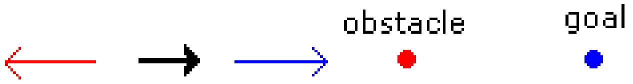
- Complex behavior arises from the interaction of many simpler ones
- Steering behaviors can be combined in two ways:
  - Blending
  - Arbitration
- Each takes in one or more steering behaviors and generates a single output based on the outputs of the input behaviors
  - Blending executes all of its inputs and combines the results based on weighted sums (priorities)
  - Arbitration selects one or more from the input set and uses their outputs to control the agent
  - These represent the opposite ends of a continuum
- The sets of behaviors are referred to as a *portfolio*

AI: Movement - Combining Steering Behaviors, Weighted Blending

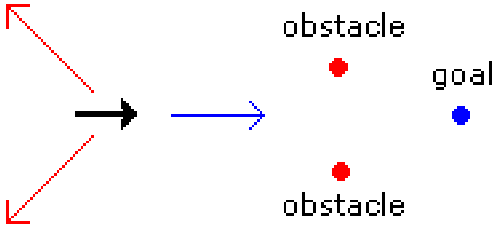
- Strategy:
  1. Each behavior is polled for its acceleration requirement
  2. Generate a weighted linear sum
  3. If $A > max$, clamp to max
- The key is in the weighting
  - There are no definitive algorithms for doing this
  - Manual assignment is as good as anything
- Implementation
  - Data members
    1. *behaviorAndWeight*: Data structure that holds these
    2. *behaviors*: A list of *behaviorAndWeight*s
    3. *maxAcceleration*
    4. *maxRotation*
  - Methods
    1. *getSteering()*
       * Algorithm

```
name getSteering (target)
{
  kinematicSteering s;

  s.linear = 0.0;
  s.angular = 0.0;
  for (item in behaviors) {
    s.linear += item.weight * item.behavior.getSteering().linear;
    s.angular += item.weight * item.behavior.getSteering().angular;
  }
  s.linear = min(s.linear, maxAcceleration);
  s.angular = min(s.angular, maxRotation);
  return s;
}
```

- Issues
  1. Agents may freeze
     - Result of opposing attractive and repulsive forces



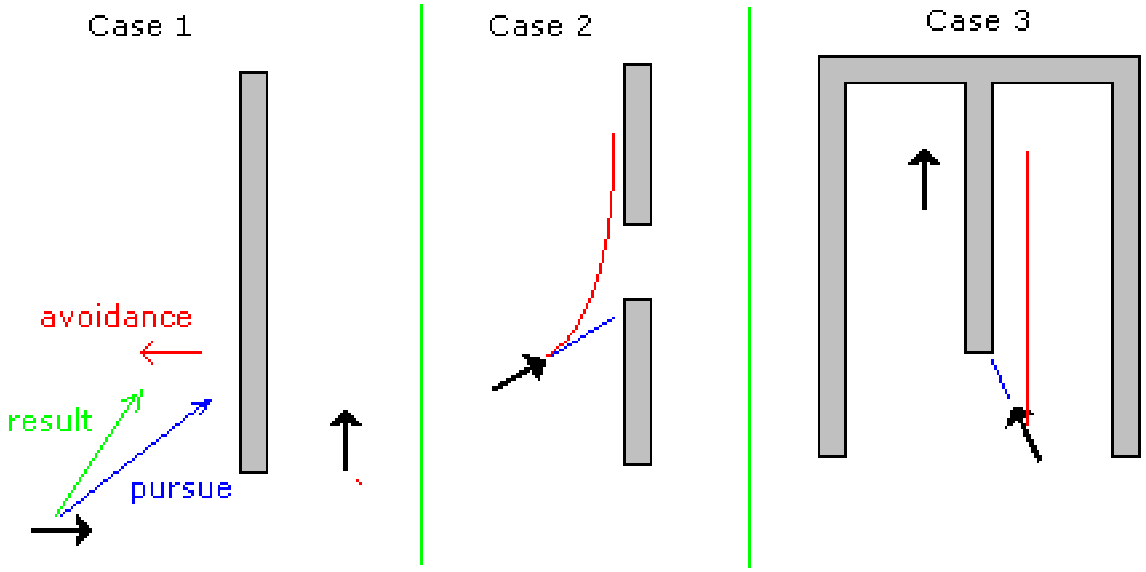
     - In the case of a single obstacle, *numerical instability* will usually result in small lateral velocities that will eventually move the agent around the obstacle
       - This is referred to as *unstable equilibrium*
     - *stable equilibrium* is situation in which accelerations sum to 0.0



       - If the agent moves upwards or downwards, the repulsion from the closer obstacle will increase, sending the agent back to the equilibrium point
       - The *basin of attraction* is the area in which an agent will be forced to the equilibrium point
       - The larger the basin, the greater the probability that agents will get trapped

AI: Movement - Combining Steering Behaviors, Weighted Blending (3)

2. Constrained environments
    - Blended steering behaviors work best in environments with few constraints
    - The greater the constraints, the greater the chance of poor behavior



    - In case 1 in the above figure, the combined attractive and repulsive vectors force the agent on the wrong side of the partition
    - In case 2, the agent again stays on the wrong side of the partition instead of going through the doorway
    - Case 3 reflects case 1
  - This has caused some to design around these issues, e.g. make doorways over-wide
  - The problem is that the blended behaviors are nearsighted
    - They only take into account local aspects
    - They do not use lookahead
    - To remedy the situation, path finding (planning) can be incorporated (see later)

3. Accelerations are weakened
    - When blending several accelerations, individual accelerations tend to be weakened in that they are averaged with the others in the mix
    - This can be critical when a collision is imminent
      - If the avoidance vector is weakened (whose value is determined in isolation from any other behaviors) , the collision may occur

AI: Movement - Blending Steering Behaviors, *Flocking*

- *Flocking* combines the following behaviors:
  1. *Separation* to prevent agents from getting too close
  2. *Cohesion* to have agents move toward the center of gravity of the group
  3. *Alignment* to have agents move in the direction of the group
  4. *Velocity matching* to have agents move at the same speed as the group
- They are usually prioritized in the above order
- An agent usually only considers it nearest neighbors - not all agents in the flock
  - Usually consider those within an arbitrary radius
  - Could use an angular cutoff instead
    - Would represent the agents that a given agent could see

AI: Movement - Combining Steering Behaviors, Priority-based

- Priority-based behaviors eliminate the problems discussed above
- Behaviors are grouped together based on weights
  - The groups are then prioritized
  - Groups are considered in order of priority
  - Within a group, behaviors are considered as in the standard weighted blending approach
  - The first group whose acceleration exceeds the minimal value controls the agent
- Implementation
  - Data members
    1. *groups*: A set of *BlendingSteering* instances
    2. *epsilon*: Minimal value in order for group to be considered
  - Methods
    1. *getSteering()*
       - Algorithm

```
name getSteering (target)
{
  KimematicSteering s;

  for (group in groups) {
    s = group.();
    if ((s.linear.magnitude > epsilon) || (s.linear.angular > epsilon))
      return s;
  }
  return s;
}
```

  - Note that assumes the groups are ordered in *groups* by priority
  - If none is large enough, the lowest priority value is returned
    - This is default behavior (often *Wander*)
    - Avoids equilibrium situation in many cases
      - If basin is large enough, will still be caught
      - As move away from basin center, other behaviors will be triggered, sending it back

AI: Movement - Combining Steering Behaviors, Priority-based (2)

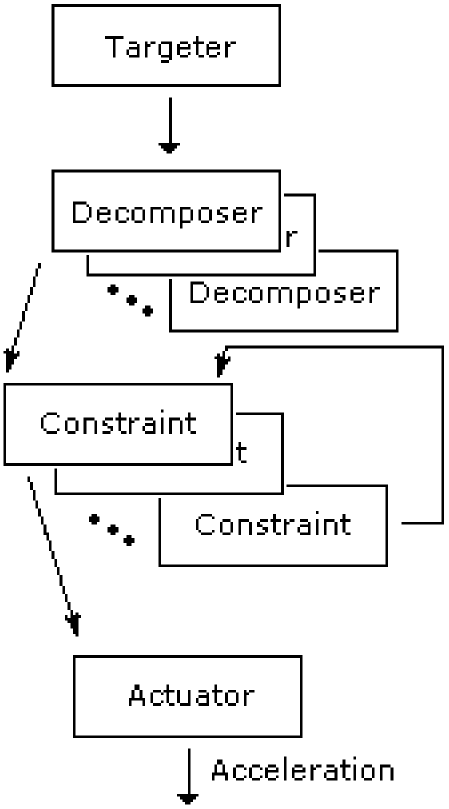
- An alternative is to use variable priorities
  - For example, priority of collision avoidance group increases as collision becomes more imminent
  - Need to include a sort in the above algorithm
  - Additional overhead not considered worth the effort

AI: Movement - Combining Steering Behaviors, *Cooperative Arbitration*
Introduction

- Basic problem with above approaches is that blending does not take into account goals of other behaviors
  - It is localized (based on a single behavior or group)
  - This frequently does not produce the best result
  - Better approach is to take context (other goals) into account
  - Such approaches called *cooperative*
- Types of cooperative architectures:
  1. Decision trees
  2. Finite state machines
  3. Blackboards
  4. Etc.
- There is no definitive cooperative approach

AI: Movement - Combining Steering Behaviors, *Cooperative Arbitration Steering Pipeline*

- The steering pipeline architecture



- Overview
  1. *Targeter*
     - Identifies a movement goal
  2. *Decomposer*
     - Generates subgoals for achieving the top-level goal
     - Each is accessed in sequence
  3. *Constraint*
     - Imposes limits on how a goal can be achieved
     - These accessed in sequence, and may need to loop through them to get a consistent result
  4. *Actuator*
     - Generates the accelerations needed to acieve the goal given the constraints

AI: Movement - Combining Steering Behaviors, *Steering Pipeline Targeter*

- There can be multiple goals
- Goals specify *target*s to be achieved (e.g., orientation, v, position, etc.)
- A target is called a *channel*
- Any goal may specify any of the channels
    - However, a given channel may only be specified by a single goal (i.e., can't have multiple targeters generate goals that have the same target)
    - Algorithm assumes targeters cooperate in this manner
- May not be able to satisfy all channels as their achievement may conflict

AI: Movement - Combining Steering Behaviors, *Steering Pipeline Decomposer*

- Given a goal, a decomposer determines whether the goal can be directly achieved
  - If it can, it simply passes it on
  - If not, it generates a path (plan for achieving the goal) and passes the first step in the plan to the next decomposer
- The decomposers form a pipeline
  - Each passes its output the the next in the chain
  - They represent a hierarchical problem solver

AI: Movement - Combining Steering Behaviors, *Steering Pipeline Constraints*

- These consist of
  1. The constraint itself - a condition that must be met
  2. A method to determine if achieving a goal will violate the constraint
  3. A method for suggesting an alternative that will not violate the constraint
- They work in conjunction with the *Actuator*
  - The actuator develops the sequence of actions needed to achieve a goal
  - Each constraint reviews the current sequence of actions and returns new subgoals if a problem is identified
  - The process is iterative
- Different constraints are concerned with specific channels
- As noted above, some constraints may be mutually exclusive
  - This can result in infinitely looping
  - To preclude this, a special steering behavior named *Deadlock* is given absolute control when this condition is recognized
- This approach to plan refinement is considered ”lightweight”
  - Complete planning system could be used, but adds complexity
  - Want to keep things simple

AI: Movement - Combining Steering Behaviors, *Steering Pipeline Actuator*

- Only one actuator per agent
- Determines *how* to achieve the goal
  - Generates the path (plan)
  - Sets priorities
  - Etc.
- Can be very simple, or complex
- Implementation
  - Data members
    1. *targeters*
    2. *decomposers*
    3. *constraints*
    4. *actuator*
    5. *constraintSteps*: Number of iterations allowed for constraint satisfaction
    6. *deadlock*: The deadlock steering behavior
    7. *s*: kinematic steering behavior

AI: Movement - Combining Steering Behaviors, *Steering Pipeline Actuator* (2)

- Methods
  1. *getSteering()*
     - Algorithm

```
name getSteering (target)
{
  Goal g;
  KinematicSteering s;

  for (targeter in targeters)
    g.updateChannels(targeter.getGoal(s);
  for (decomposer in decomposers)
    g = decomposer.decompose(s, g);
  validPath = false;
  i = 0;
  while ((!validPath) && (i < constraintSteps)) {
    validPath = true;
    path = actuator.getPath(s, g);
    constraint = constraints.getNext();
    while (validPath && constraint) {
      if (constraint.isViolated(path)) {
        g = constraint.suggest(path, s, g);
        validPath = false;
      }
      constraint = constraints.getNext();
    }
    i++;
  }
  if (validPath)
    return actuator.output(path, s, g);
  else
    return deadlock.getSteering();
}
```

AI: Movement - Combining Steering Behaviors, *Steering Pipeline* Data Structures

1. *Targeter*
   - Methods
     (a) *Goal getGoal (KinematicSteering)*: Returns the goal for this targeter
2. *Decomposer*
   - Methods
     (a) *Goal decompose (KinematicSteering, Goal)*: Returns the input goal if no refinemment needed; first subgoal of a refined plan otherwise
3. *Constraint*
   - Methods
     (a) *Boolean willViolate (Path)*: Indicates if the path violates the constraint
     (b) *Goal suggest (Path, KinematicSteering, Goal)*: Returns a new goal that will not result in the constraint violation
4. *Actuator*
   - Methods
     (a) *Path getPath (Path)*: Returns a path to the goal
     (b) *KinematicSteering output (Path)*: Returns the steering behavior for achieving the goal
5. *Deadlock*
   - Methods
     (a) *KinematicSteering getSteering ()*
   - This can be any general behavior
6. *Goal*
   - Data members
     (a) *hasPosition*, *hasOrientation*, *hasVelocity*, *hasRotation*: Flags to indicate if the goal includes these channels
     (b) *position*, *orientation*, *velocity*, *rotation*: The corresponding channel data
   - Methods
     (a) *updateChannel (Goal)*: Sets the above data members based on whether the input goal has a given channel
7. *Path*
   - This depends on your implementation

AI: Movement - Predicting Physics, Intro

- This topics concerns the physics of objects interacting with other objects
    - In particular, wrt trajectories
- This material deals with aiming and shooting
- Characters should be able to
    1. Shoot accurately
    2. Respond to enemy fire
        - This is relevant for conventional weapons (mortars, rifles)

AI: Movement - Predicting Physics, Trajectories

- A projectile under gravity follows a parabola described by

$$\mathbf{p_t} = \mathbf{p}_0 + \mathbf{u}s_m t + \frac{1}{2}\mathbf{g}t^2$$

  where $\mathbf{p_t}$ is position at time t

  $\mathbf{p_0}$ is initial position

  $\mathbf{u}$ is direction of aiming

  s_m is speed

  $\mathbf{g}$ is the gravitational constant ($\mathbf{g} = \begin{bmatrix} 0 & -9.81 & 0 \end{bmatrix} ms^{-2}$)

  - Many developers recommend using a larger value for $\mathbf{g}$

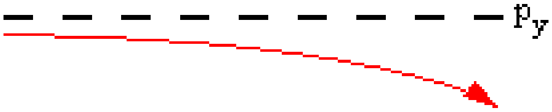
- To determine where a projectile lands (simple case: flat terrain):
  - Solve above equation for a fixed height (e.g., height from which projectile is fired)

$$t_i = \frac{-u_y s_m \pm \sqrt{u_y^2 s_m^2 - 2g_y(p_{0y} - p_{iy})}}{g_y}$$

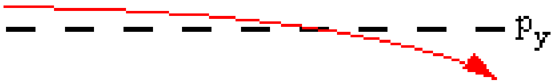
    where

    p_{iy} is vertical location of projectile at time t_i

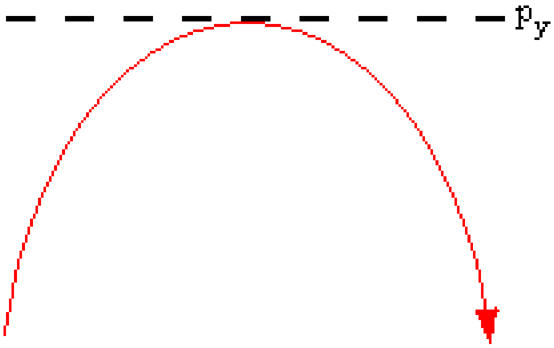
  - Solution set:
    - ∗ If no solution, projectile never reaches p_{iy}



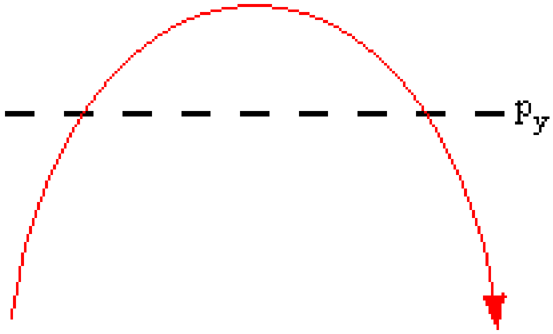
    - ∗ If one solution, two cases



OR



  - ∗ If two solutions, projectile reaches p_{iy} once on the way up and once on the way down



  - ∗ Text only considers case where projectile is descending, so want larger value of t_i

- – The point of impact is then computed using

$$\mathbf{p_i} = \left[\ p_{0x} + u_x s_m t_i + \tfrac{1}{2} g_x t_i^2 \quad p_{iy} \quad p_{0z} + u_z s_m t_i + \tfrac{1}{2} g_z t_i^2\ \right]$$

- – Since $g_x = g_z = 0$, this reduces to

$$\mathbf{p_i} = \left[\ p_{0x} + u_x s_m t_i \quad p_{iy} \quad p_{0z} + u_z s_m t_i\ \right]$$

- For situations where the terrain is uneven, use an iterative approach
  1. Calculate the (x, y) coordinate for the computed y value
  2. Use the actual value of y wrt (x, y)
  3. Recompute t_i using this new y value
  4. Continue refining the y value as above until $|y_{i-1} - y_i|$ is arbitrarily small
  5. In practice, theh y values usually converge

AI: Movement - Predicting Physics, The Firing Solution

- Consider a shooter at point $\mathbf{s}$ who wants to hit a target at point $\mathbf{e}$
  - Muzzle velocity s_m is known
    - For some weapons, this value is variable
    - Assume the largest possible value in the derivation below
  - Want to find the direction to aim: $\mathbf{u}$
- To solve for $\mathbf{u}$, first solve for t_i in

$$\mathbf{p_t} = \mathbf{p_0} + \hat{\mathbf{u}} s_m t + \frac{1}{2}\mathbf{g}t^2$$

where $\mathbf{s} \equiv \mathbf{p_0}$
$\mathbf{e} \equiv \mathbf{p_t}$
$\hat{\mathbf{u}}$ is *normalized*: $\hat{\mathbf{u}} = \left[\ u_x/|\mathbf{u}| \quad u_y/|\mathbf{u}| \quad u_z/|\mathbf{u}|\ \right]$, $u_x^2 + u_y^2 + u_z^2 = 1$

  - This gives four equations in 4 unknowns:

$$\begin{aligned} E_x =&\ S_x + u_x s_m t_i + \tfrac{1}{2} g_x t_i^2 \\ E_y =&\ S_y + u_y s_m t_i + \tfrac{1}{2} g_y t_i^2 \\ E_x =&\ S_z + u_z s_m t_i + \tfrac{1}{2} g_z t_i^2 \\ 1 =&\ u_x^2 + u_y^2 + u_z^2 \end{aligned}$$

  - Solving these we get

$$|\mathbf{g}^2| t_i^2 - 4(\mathbf{g} \cdot \mathbf{\Delta} + s_m^2) t_i^2 + 4|\mathbf{\Delta}|^2 = 0$$

where $\mathbf{\Delta} = \mathbf{E} - \mathbf{S}$

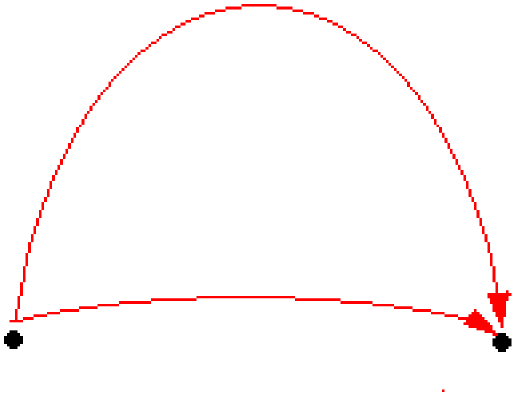
  - Using the quadaritic equation:

$$t_i = +2\sqrt{\frac{\mathbf{g} \cdot \mathbf{\Delta} + s_m^2 \pm \sqrt{(\mathbf{g} \cdot \mathbf{\Delta} + s_m^2)^2 - |\mathbf{g}|^2 |\mathbf{\Delta}|^2}}{2|\mathbf{g}|^2}}$$

Only positive solutions are considered

AI: Movement - Predicting Physics, The Firing Solution (2)

- No solution exists when $\mathbf{g} \cdot \mathbf{\Delta} + s_m^2 < |\mathbf{g}|^2 |\mathbf{\Delta}|^2$
- If only 1 solution exists, the target is at the limit of the weapons range (s_m)
- If 2 solutions exists, they correspond to 2 arcs:



  - Will assume the shorter is more desireable as it gives the opponent less time to react
- Once t_i is determined, solve for $\mathbf{u}$ using

$$\mathbf{u} = \frac{2\mathbf{\Delta} - \mathbf{g}t_i}{2s_m t_i}$$

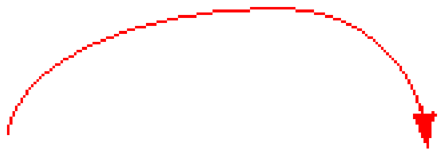
AI: Movement - Predicting Physics, The Firing Solution (3)

- Pseudocode

```
Vector calculateFiringSolution (source, target, muzzle_v, g)
{
  delta = target - source;

  //quadratic equation factors
  a = g * g;
  b = -4 * (g * delta + muzzle_v * muzzle_v);
  c = 4 * delta * delta;

  if (4 * a * c > b * b)
    return null;
  time0 = sqrt((-b + sqrt(B * b - 4 * a * c)) / (2 * a));
  time1 = sqrt((-b - sqrt(B * b - 4 * a * c)) / (2 * a));
  if (time0 < 0)
    if (time1 < 0)
      return null;
    else
      tt = time1;
  else
    if (time1 < 0)
      tt = time0;
    else
      tt = min(time0, time1);
  return (2 * delta - g * tt * tt) / ( 2 * muzzle_v * tt);
}
```

AI: Movement - Predicting Physics, Adding Drag

- Adding drag complicates trajectory calculations
- Path will no longer be a parabola
  - The second half is 'flattened' and not symmetric with the first half:



- Drag is represented by $D = -kv - cv^2$

  where k is the viscous drag coefficient
  c is the aerodynamic (ballistic) coefficient

- Incorporating these into the motion equation results in

$$\mathbf{p}''_t = \mathbf{g} - k\mathbf{p}'_t - c\mathbf{p}'_t|\mathbf{p}'_t|$$

  - The third term relates drag in one direction to that in another
  - Motion in one direction is no longer independent of that in another
    - This is what complicates the situation

- There are two approaches to handling this

  1. One solution is to ignore the third term:

$$\mathbf{p}''_t = \mathbf{g} - k\mathbf{p}'_t$$

     - This can be solved for $\mathbf{p}_t$:

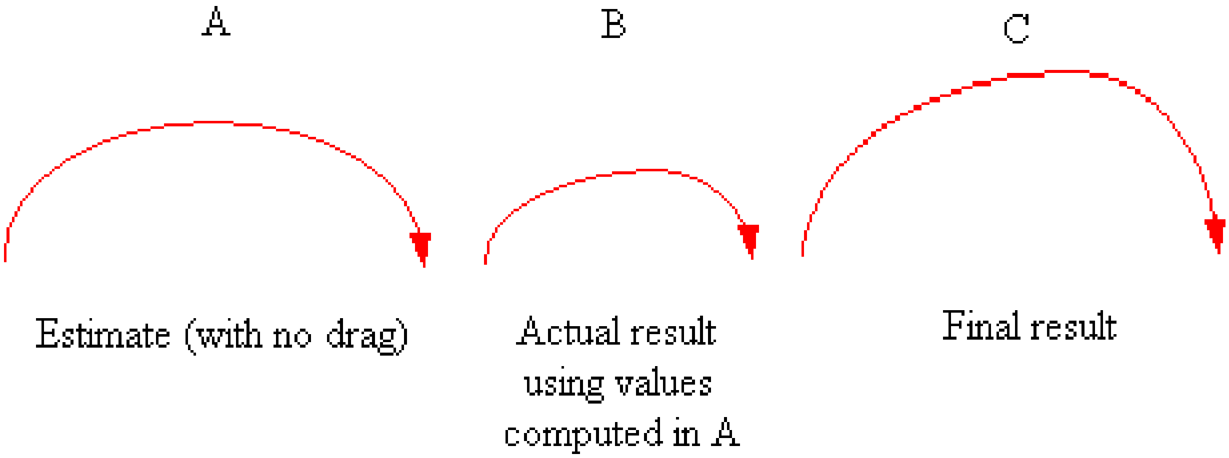
$$\mathbf{p}_t = \frac{\mathbf{g}t - \mathbf{A}e^{-kt}}{k} + \mathbf{B}$$

     where $\mathbf{A} = s_m\mathbf{u} - \mathbf{g}/k$
     $\mathbf{B} = \mathbf{p}_0 - \mathbf{A}/k$

AI: Movement - Predicting Physics, Adding Drag (2)

2. The alternative is to use an iterative approach
   - The strategy is to
     - Generate an initial solution using the basic firing equation, ignoring drag
     - Determine how close it is to the actual target by computing points on the path at fixed time intervals
     - Refine the initial conditions
     - Repeat until the result is sufficiently close
   - There are two aspects to refinement:
     (a) Bearing
       - This is of concern if wind is a factor

     (b) Elevation



   - Best approach is to alternate between the two

AI: Movement - Predicting Physics, Adding Drag (3)

– Pseudocode (based on elevation refinement)

```
Vector refineTargeting (source, target, muzzle_v, g, epsilon)
{
  delta = target - source;

  //initial guess
  direction = calculateFiringSolution(source, target, muzzle_v, g);
                  //corresponding firing angle
  minBound = atan(direction.y / direction.length());     //*** text uses asin
  distance = distanceToTarget(direction, source, target, muzzle_v);
                  //does it work?
  if (distance * distance < epsilon * epsilon)
    return direction;
                  //overshot?
  else (if distance > 0) {                               //*** text uses minBoundDistance
                  //set bounds
    maxBound = minBound;
    minBound = 90;                                       //*** text uses -90
  }
  else {          //adjust for undershot
    maxBound = 45;
                  //calc distance of bound
    direction = convertToDirection(deltaPosition, angle);
    distance = distanceToTarget(direction, source, target, muzzle_v);
    if (distance * distance < epsilon * epsilon)
      return direction;
    else if (distance < 0)
      return null;
  }
  //Use binary search to find correct angle
                                                        //*** text has distance = epsilon;
  while (distance * distance > epsilon * epsilon) {     //*** text has <
    angle = (maxBound - minBound ) * 0.5;
    direction = convertToDirection(deltaPosition, angle);
    distance = distanceToTarget(direction, source, target, muzzle_v);
    if (distance < 0)
      minBound = angle;
    else
      maxBound = angle;
  }
  return direction;
}
```

– Other factors to be considered:

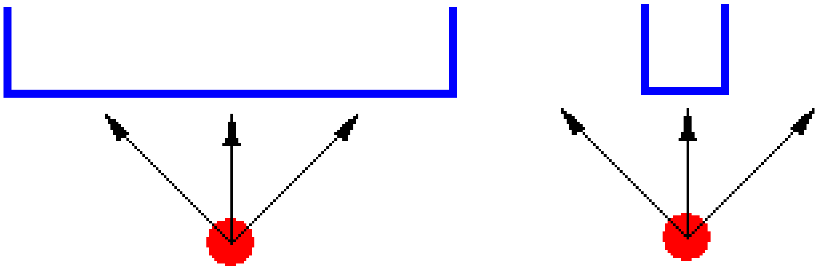
(a) Spin: This creates additional lift

(b) Gravity wells: These are local attractors

(c) Etc.

AI: Movement - Jumping, Jump Points

- Jumping is more exacting than steering
  - Consequences of a poorly executed jump can be serious
- A *jump point* is a predefined area from which a character can launch
  - Specified by game designer
  - Usually have a minimum required velocity associated
- Direction of jump may or may not be important
  - Not critical if jumping onto a broad area
  - Not so for narrow landing areas



- General steps associated with jumping
  1. Character decides to jump
     - May be the result of pathfinding, steering behavior, ...
  2. Character recognizes which jump to make
     - Requires look-ahead so have enough time to accelerae
  3. Steering behavior toward jump point takes over
     - Character speed must match jump point minimum speed
  4. Jump executed when jump point reached
- Jump point may have additional info associated with it:
  1. Direction
  2. Danger
  3. ...

  - These would be hard-coded by designer
- NPCs are often restricted wrt jumps to preclude stupid behaviors

AI: Movement - Jumping, Landing Pads

- A *landing pad* is a target location for a corresponding jump point
  - This allows computation of jumping info instead of storing it with the jump point
  - It provides greater flexibility
    - Other factors (e.g., load) can now be taken into account
- To compute a jumping trajectory for a given v_y (ignoring drag), need to compute
  1. x and z components of $\mathbf{v}$
  2. t

$$\begin{aligned} E_x &= S_x + v_x t \\ E_y &= S_y + v_y t + \frac{1}{2} g t^2 \\ E_z &= S_z + v_z t \end{aligned}$$

  - Solving for t:

$$t = \frac{-v_y \pm \sqrt{2g(E_y - S_y) + v_y^2}}{g}$$

$$v_x = \frac{E_x - S_x}{t}$$

$$v_z = \frac{E_z - S_z}{t}$$

  - The equation for t has 2 solutions - usually want the smaller one
    - But the smaller result may require a velocity that cannot be attained
- To achieve the jump, implement a steering behavior from the jump point to the landing pad
- *Jump* must be independent of other steering behaviors
  - If combined with others, usually results in a failed jump

AI: Movement - Jumping, Class *Jump*

- Data members
  1. jumpPoint
     - An instance of class *JumpPoint*
     - Contains members
       (a) jumpLocation - *Point* representing jump point's location
       (b) landingLocation - *Point* representing landing pad's location
       (c) delta - float representing scalar distance between the above
  2. agent - dynamic character
  3. velocityMatch - instance of velocity matching class
- Methods
  1. *getSteering()*

```
Steering getSteering ()
{
  canAchieve = FALSE;
  if (null(target))
    target = calculateTarget();
  if (!canAchieve)               //set by calculateTarget()
    return new(Steering);
  if (agent.position.near(target.position) && agent.velocity.near(target.velocity) {
    scheduleJumpAction();
    return new(Steering);
  }
  return velocityMatch.getSteering();
}
```

  2. *calculateTarget()*

```
 calculateTarget ()
{
  target = new(Kinematic);
  target.position = jumpPoint.jumpLocation;
  sqrtTerm = sqrt(2 * g.y * jumpPoint.delta.y + agent.maxVelocity.y * agent.maxVelocity.y);
  time = (agent.maxVelocity.y - sqrtTerm) / g.y;
  checkJumpTime(time);
}
```

  3. *checkJumpTime()*

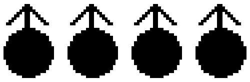
```
 checkJumpTime ()
{
  vx = jumpPoint.delta.x / time;
  vz = jumpPoint.delta.z / time;
  speedSq = vx * vx + vz * vz;
  if (speedSq < agent.maxSpeed * agent.maxSpeed) {
    target.velocity.x = vx;
    target.velocity.y = vz;
    canAchieve = true;
  }
}
```

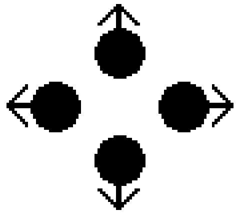
AI: Movement - Jumping, Hole Fillers

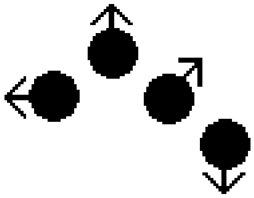
- The most general solution is to have the character choose its own jumping point and landing pad
- This is accomplished by placing a *jumpable gap* in the hole
  - This is an invisible object
- Character behavior is implemented as an inverse collision steering behavior
  - The character accelerates toward the jumpable gap
  - At point of collision, jump is executed
- This eliminates need for jump points and landing pads
- This introduces a new problem:
  - Since there is no prescribed landing spot, there is no specific target velocity
  - This approach tends to result in more failed jumps

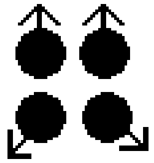
AI: Movement - Coordinated Movement, Single Level

- This topic deals with movement of formations
  - Having characters move as a group
  - Basic formations


Line


Defensive Circle


V/Finger Four


Two Abreast in Cover

- Basic approaches:
  1. Fixed formations
     - Simplest approach
     - Have a predefined shape
     - Formation defined as a set numer of *slots*
       - One slot (slot 0) is reserved for the leader
       - Remaining slots defined relative to the leader slot
       - The leader position defines the location and orientation of the formation
     - The leader character moves as an independent character
       - The remaining characters' motions are based on the leader's

- Given slot s located r_s relative to slot 0, the position of the character associated with slot s is

  $$p_s = p_l + \Omega_l r_s$$

  where p_s is the position associated with slot s
  p_l is the position associated with slot 0
  Ω_l is the orientation of the leader

- The orientation of the character in slot s is given by

  $$\omega_s = \omega_l + \omega_s$$

- While the leader moves as an independent character, must also consider that it is part of a formation
  - For example, on collisions, must take overall size of the formation into account
  - When turning, must do so slowly enough so that all members of the formation can keep up

2. Scalable formations
   - The number of slots is not fixed but based on the number of characters
   - Orientation of the slots based on the number of slots
3. Emergent formations
   - Each character now has its own steering system using the arrive behavior
     - Target is pseudo-target bassed on another character in the formation
   - The steering behavior is based on the shape of the formation
   - When selecting a target, the character will not select one already chosen, or one in the process of being chosen
     - This requires cooperation among the characters
     - Once chosen, the character keeps that target until the time at which is is lost
     - If a target is lost, another is chosen
   - There is no explicit leader (although it is recommended that one slot has no target, and so is the *de facto* leader)
   - This approach allows individuals to deal with obstacles
   - Since the formation shape is emergent, may result in degenerate cases
   - Trying to create steering behaviors to produce a given shape is difficult

AI: Movement - Coordinated Movement, Two Level

- This approach is a synthesis of the slot and emergent formation approaches
- Variations:
  1. Basic approach
     - Uses a geometric formation with slots
     - Slots are used as targets by characters each of which has its own steering behavior
     - Called a two-level approach because
       - The leader steers the formation
       - The characters steer to maintain their positions
  2. Anchor point approach
     - Problem with leader is that
       - If a leader executes an avoidance behavior, the whole formation follows the leader's motions
       - In reality, it may not be necessary for all characters in the formation to avoid the obstacle
     - An *anchor point* takes the place of a leader
       - It acts like an invisible leader
       - It has a fixed location in the formation like a leader
       - It does not have a character associated with it
     - The anchor only has to worry about large obstacles
       - Small obstacles are handled by individual characters' steering behaviors
       - This can pose a problem:
         - Since the anchor doesn't worry about obstacles but characters must, they may have a hard time keeping up
       - Solutions:
       (a) Set speed of anchor $\sim 1/2$ that of characters
         - But this makes normal formation movement too slow

AI: Movement - Coordinated Movement, Two Level (2)

(b) Make the kinematics of the anchor the average of the position, orientation, and rotation of the characters

- · Move the anchor first, then the characters
- · Reset the anchor's properties after each set of moves
- · If characters cannot keep up, the whole formation will eventually slow to a stop
- · To remedy this, add an offset to the center of mass of the formation:

$$\mathbf{p}_a = \mathbf{p}_c + k\mathbf{v}c$$

where $\mathbf{p}_c$ and + $\mathbf{v}c$ are the position and velocity of the center of mass

- Drift
  - The center of mass is computed by

$$p_c = \frac{1}{n}\sum_{i=1}^{n} \begin{cases} p_{s_i} & ifoccupied \\ 0 & ifnot \end{cases}$$

  - Slot movement is then updated as

$$p'_{s_i} = p_{s_i} - p_c$$

  - Since movement is performed in 2 stages (anchor, then characters), and anchor's properties represent an average of the characters', the formation may drift when the anchor is 'stationary'
  - Rotational drift around the anchor's location can occur if the anchor does not use the formation's average orientation:

$$\omega_c = \frac{\mathbf{v}_c}{|\mathbf{v}_c|}$$

where

$$\mathbf{v}_c = \frac{1}{n}\sum_{i=1}^{n} \begin{cases} p_{s_i} & \text{if occupied} \\ 0 & \text{if not} \end{cases}$$

$$\omega_c = \omega_{s_i} - \omega_c$$

AI: Movement - Coordinated Movement, Class *FormationManager*

- Data members
  1. slotAssignment
     - Instance of class that represents assignment of one character to a slot
     - Has members
       (a) character
       (b) slotNumber
  2. slotAssignments - list of assignments
  3. driftOffset - Static behavior structure
  4. pattern - the formation
- Methods
  1. *updateSlotAssignments()*

```
updateSlotAssignments ()
{
  for (i = 0 to slotAssignments.length)
    slotAssignments[i].slotNumber = i;
}
```

  2. *addCharacter()*

```
Boolean addCharacter (character)
{
  occupiedSlots = slotAssignments.length;
  if (pattern.supportsSlots(occupiedSlots + 1)) {
    slotAssignment = new SlotAssignment();
    slotAssignment.character = character;
    slotAssignments.append(slotAssignment);
    updateSlotAssignments();
    return true;
  }
  return false;
}
```

  3. *removeCharacter()*

```
removeCharacter (character)
{
  slot = characterInSlots.find(character);
  if (slot > 0) {
    slotAssignments.removeElementAt(slot);
    updateSlotAssignments();
  }
}
```

4. *updateSlots()*

```
updateSlots ()
{
  anchor = getAnchorPoint();
  orientationMatrix = anchor.orientation.asMatrix();
  for (i = 0 to slotAssignments.length) {
    relativeLoc = pattern.getSlotLocation(slotAssigments[i].slotNumber);
    location = new Static();
    location.position = relativeLoc.position * orientationMatrix + anchor.position;
    location.orientation = relativeLoc.orientation  + relativeLoc.orientation;
    location.position = -= driftOffset.position;
    location.orientation = -= driftOffset.orientation;
    slotAssignments[i].character.setTarget(location);
  }
}
```

AI: Movement - Coordinated Movement, Add'l Classes

1. class *FormationPattern*
    - Used to represent specific patterns

      ```
      interface FormationPattern
      {
        numberOfSlots;

        int getDriftOffset(slotAssignments);
        int getSlotLocation(slotNumber);
        Boolean supportsSlots(slotCount);
      }
      ```

2. class *Character*

   ```
   class Character
   {
     setTaregt(static);
   }
   ```

AI: Movement - Coordinated Movement, Example Class *DefensiveCirclePattern*

- Data members
  1. characterRadius
- Methods
  1. *calculateNumberOfSlots()*

```
int calculateNumberOfSlots(assignments)
{
  filledSlots = 0;
  for (assignment in assignments)
    if (assignment.slotNumber >= maxSlotNumber)
      filledSlots = assignment.slotNumber;
  numberOfSlots = filledSlots + 1;
  return numberOfSlots;
}
```

  2. *getDriftOffset()*

```
Static getDriftOffset (assignments)
{
  center = new Static();
  for (assignment in assignments) {
    location = getSlotLocation(assignment.slotNumber);
    center.position += location.position;
    center.orientation += location.orientation;
  }
  numberOfAssignments = assignments.length;;
  center.position /= numberOfAssignments;
  center.orientation /= numberOfAssignments;
  return center;
}
```

  3. *getSlotLocation()*

```
Static getSlotLocation (slotNumber)
{
  angleAroundCircle = slotNumber / numberOfSlots * PI * 2;
  radius = characterRadius / sin(PI / numberOfSlots);
  location = new Static();
  location.position.x = radius * cos(angleAroundCircle);
  location.position.y = radius * sin(angleAroundCircle);
  return location;
}
```

AI: Movement - Coordinated Movement, Hierarchical Formations

- Note: Text refers to this as formations having more than two levels
- In this version, slots can be filled with *formations*, in addition to characters
  - This should be no problem implementationally if both characters and formations use the same interface
- Any of the previously discussed steering approaches can be used
- Example (pp 138-139 text):

AI: Movement - Coordinated Movement, Slot Roles

- An issue that needs to be addressed:
  - In some situations, certain slots can only be filled by specific roles
  - The basic formation approach must be refined to incorporate this
- A *slot role* limits the type of filler that can used with a formation slot
- There are two types
  1. Hard roles
     - Hard roles can only be filled by candidates that fulfill that role
     - The difficulty with this arises when the candidates do not match up with the slots
       - ∗ Some slots will remain unfilled
       - ∗ Some candidates will remain unassigned
     - To address this issue, could
       (a) Provde a set of alternate formations for the set of candidates and roles
         - ∗ This puts additional work on the designer
         - ∗ Theoretically, would require a very large number of possibilities

       (b) Implement a function that generates an optimal formation for the given set of candidates and roles
         - ∗ This problemmatic due to the increased load on the programmer, and the additional computation costs
  2. Soft roles
     - A *soft role* associates a cost with a candidate (essentially a priority)
       - ∗ Note: A candidate will have a cost associated with each role in the formation
     - A cost indicates how appropriately a candidate type fulfills the role
     - Low values imply good fit
       - ∗ To preclude a candidate from occupying a slot, assign a value of ∞
     - In this approach, a slot can be filled by any candidate type (except those with infinite values for that slot)
       - ∗ The goal is to fill the formation with assignments that result in the least overall cost
       - ∗ An assignment usually occurs only once
         - · Once assigned, candidates usually remain in their slots

AI: Movement - Coordinated Movement, Slot Roles (2)

- The assignment problem is *NP-complete*:
  - Try all possible combinations and choose the one with minimal cost
  - Given k candidates and n slots:

$$\text{number of combinations} = \frac{n!}{(n-k)!}$$

- A more practical approach assigns each candidate in turn to an unassigned slot with the lowest cost
  - To make this more efficient, associate an ’*ease of assignment*’ value to candidates
    - This represents how easy it is to find a slot for the candidate

$$ease = \sum_{i=1}^{n} \begin{cases} \frac{1}{1+c_i} \text{ if } c_i < k \\ 0 \text{ otherwise} \end{cases}$$

where c_i is the cost to occupy slot i

n is the number of slots

k is the slot cost limit (cost beyond which the slot is considered too expensive to fill)

- The fewer the number of slots a candidate can fill, the lower its *ease* score
- Candidates with low *ease* scores will be assigned first

- Additional factors can be incorporated into slot costs:
  1. Distance between candidate and slot location
  2. Protection slot location provides
  3. Etc.

AI: Movement - Coordinated Movement, Class *FormationManager*: Amended for Slot Roles

- Data members - As before
- Methods
  1. *updateSlotAssignments()*

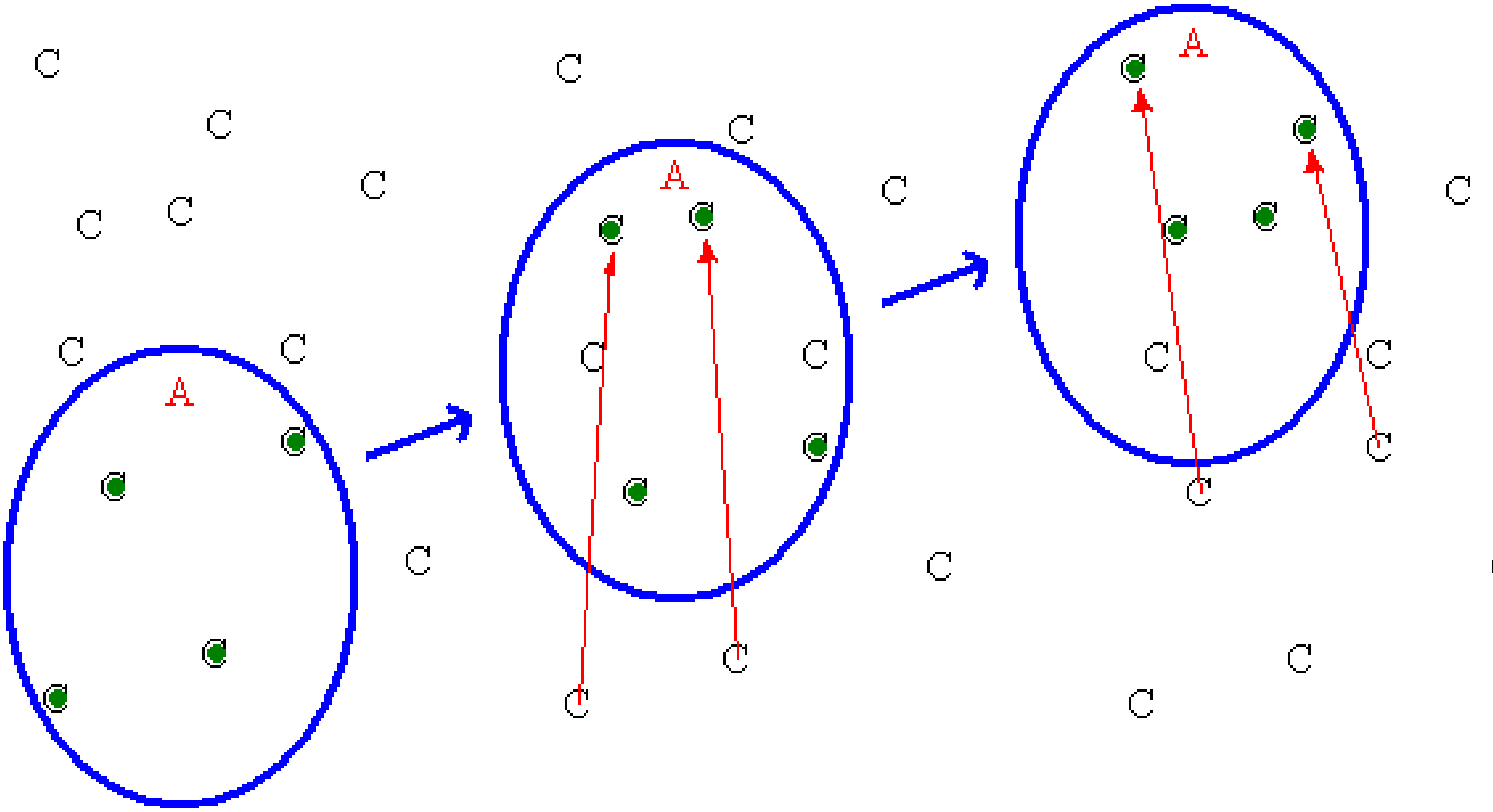
```
updateSlotAssignments ()
{
  struct CostAndSlot {
    cost
    slot
  };

  struct CharacterAndSlots {
    character
    assignmentEase
    costAndSlots
  };

  characterData

  for (assigment in slotAssignments){
    datum = new CharacterAndSlots();
    datum.character = assigment.character;
    for (slot = 0; slot < patter.numberOfSlots; slot++) {
      cost = pattern.getSlotCost(assignment.character);
      if (cost >= LIMIT)
        continue;
      slotDatum = new CostAndSlot();
      slotDatum.slot = slot;
      slotDatum.cost = cost;
      datum.costAndSlots.append(slotDatum);
      datum.assignmentEase += 1/(1 + cost);
    }
  }
  filledSlots = new Boolean[pattern.numberOfSlots];
  assignments = [];
  characterData.sortByAssignmentEase();
  for (characterDatum in characterData) {
    characterDatum.costAndSlots.sortByCost()
    for (slot in characterDatum.costAndSlots) {
      if (!filledSlots[slot]) {
        assignment = new SlotAssignment();
        assignment.character = characterDatum.character;
        assignment.slot = slot;
        assignments.append(assignment);
        filledSlots[slot] = true;
        break continue;  //***
      }
    }
    error;
  }
  slotAssignments = assignment;
}
```

AI: Movement - Coordinated Movement, Dynamic Slots

- Slots can be modeled so that they are dynamically positioned relative to the anchor instead of being fixed
  - This is useful for sports games (e.g., play strategies)
- In this approach, the formation can change over time
  1. Characters can follow predefined paths, or
  2. Slots can move to new positions and use *arrive* behaviors to move the characters
- Time must be added to code
  - Since slots change position, drift may occur
  - Since two-level steering is frequently not used in play situations, this isn't usually an issue
- Slot position can be
  1. Predetermined, or
  2. Set by AI based on current environment (e.g., in response to opponent's behaviors)

AI: Movement - Coordinated Movement, Tactical Movement

- *Tactical movement* is an extension/application of dynamic slots
- It relates to the tactical movement of squads
- *Bounding overwatch* is an example of miltary squad movement
  - Squad members move from cover to cover while team mates provide protection from covered positions
  - Movement determined by cover locations
    - Set of cover points closest to anchor are id'd (one per slot of formation)
    - Formation shape will be dynamic, based on environment
    - As anchor moves, some cover points will fall out of range (furthest behind) while new ones will come into range (at the forefront)
    - Characters associated with the lost cover points will be reassigned to the new ones and move to them



AI: Movement - Coordinated Movement, Tactical Movement (2)

- Moderation of the anchor point must be turned off
  - Since characters at fixed locations as the anchor moves, center of mass may be static for a time span
  - Consequently, must have anchor move slowly enough so characters can keep up