CS 480: GAME AI

STEERING BEHAVIORS

4/12/2012
Santiago Ontañón
santi@cs.drexel.edu
https://www.cs.drexel.edu/~santi/teaching/2012/CS480/intro.html
Reminders

• Check BBVista site for the course regularly
• Also: https://www.cs.drexel.edu/~santi/teaching/2012/CS480/intro.html

• Paper presentation dates are available on the course website

• Project 1 is already available on the course website
Outline

• Basic Steering Behaviors
• Steering Behaviors on Vehicles
• Composite Steering Behaviors

• (Chapter 3 of Millington’s and Funge book)
Outline

- Basic Steering Behaviors
- Steering Behaviors on Vehicles
- Composite Steering Behaviors
Game AI Architecture

- Strategy
- Decision Making
- Movement
- World Interface (perception)
Game AI Architecture

- Strategy
- Decision Making
- Movement

World Interface (perception)

Steering behaviors executed in this module
Steering Behaviors

- Basic building blocks for continuous movement
- Whole family of steering behaviors (we will cover only the most important ones)
- Widespread use among commercial computer games
Steering Behaviors: Uses
Steering Behaviors: Uses

Decision Making

Movement
In car racing games, Decision Making is typically hard coded. The game designers create a set of waypoints in the track (or in the track pieces), and cars go to them in order.
Steering Behaviors: Uses

Movement is in charge of driving the car to each of the waypoints, avoiding opponents, braking, accelerating, turning, etc.
Steering Behaviors: Uses

• Not just racing games

• Any games with vehicles (helicopters, tanks, planes, boats)

• Or even characters moving in a 3D environment (continuous movement) with inertia (e.g. sports games)
  • Most FPS games just assume there is no inertia and characters can move in any direction at any time (bad physics 😞)
Alternative to Steering Behaviors

• Kinematic movement: ignores inertia and acceleration

• Just move the character to the right direction

• Useful for simple or non-realistic games

• E.g.: Pac-Man (you can move Pac-Man in any direction without inertia)
Basic Steering Behaviors

- Seek
- Flee
- Arrive
- Align
- Velocity Matching
Steering Behaviors

• Defined as methods that return the acceleration that the body/vehicle controller by the AI needs to have during the next execution frame:
  • **Input**: position, orientation, speed of AI, target
  • **Output**: acceleration

• They are executed once per game cycle

• Some return linear acceleration (accelerate north at 3m/s²), some return angular acceleration (turn right at 2rad/s²)
Seek

- Move towards a (potentially moving) target
Seek

- Move towards a (potentially moving) target

\[ D = E - S \]

- S: Start coordinates
- E: End coordinates
- V: Current Speed

Difference

Target

E: End coordinates
Seek

Seek(character, E)
D = E - character.position
ND = D / |D|
A = ND * maxAcceleration
Return A
Seek

\textbf{Seek}(\text{character, E})

D = E - \text{character.position}

ND = D / |D|

A = ND \times \text{maxAcceleration}

\textbf{Return} A
Seek

Seek(character, E)

D = E - character.position

ND = D / |D|

A = ND * maxAcceleration

Return A
Seek

\( \text{Seek}(\text{character}, E) \)

\[ D = E - \text{character.position} \]

\[ ND = \frac{D}{|D|} \]

\[ A = ND \times \text{maxAcceleration} \]

Return \( A \)
Seek

Seek(character, E)
D = E - character.position
ND = D / |D|
A = ND * maxAcceleration
Return A
Seek

\textbf{Seek}(\text{character, E})

D = E - \text{character.position}

ND = D / |D|

A = ND \times \text{maxAcceleration}

\textbf{Return} \ A
Seek

Seek(character, E)
D = E - character.position
ND = D / |D|
A = ND * maxAcceleration
Return A
**Seek**

Seek(character, E)
D = E - character.position
ND = D / |D|
A = ND * maxAcceleration
Return A
Physics Code

In the game engine, there would be something like:

```java
UpdateCharacter(character, timeDelta)
    A = character.getAI().cycle(timeDelta);
    character.position += character.position*timeDelta;
    character.speed += A*timeDelta;
```
Seek

- The most basic Steering Behavior
- Use to realistically make a character/vehicle move towards a target
- If you were to implement it by directly moving the character towards the end point, you might have characters that make 90 degree turns instantly (physically impossible)
Flee

• Move away from a (potentially moving) target
Flee

\textbf{Seek}(\text{character, E})
\begin{align*}
D &= E - \text{character.position} \\
ND &= D / |D| \\
A &= ND \times \text{maxAcceleration} \\
\text{Return A}
\end{align*}

\textbf{Flee}(\text{character, E})
\begin{align*}
D &= \text{character.position} - E \\
ND &= D / |D| \\
A &= ND \times \text{maxAcceleration} \\
\text{Return A}
\end{align*}
Arrive

- Seek is good for pursuing a moving object, but not for arriving at spots. This is what Seek would do:
  - Orbit around the target (since it reaches it at full speed)
Arrive

• Arrive would do this:

• Decelerate when arriving at the target, to stop right on the spot
Arrive

• Idea:
  • Define two radii around the target point
  • Small radius: define the target (to make the job easier)
  • Big radius: define the area of deceleration
Arrive

Arrive(character, E, targetRadius, slowRadius, time)
D = E - character.position
Length = |D|
If Length<targetRadius Return (0,0,0)
If Length>slowRadius then targetSpeed = maxSpeed
    else targetSpeed = maxSpeed * distance/slowRadius
targetVelocity = (D/|D|)*targetSpeed
A = (targetVelocity - character.velocity)/time
If |A|>maxAcceleration then A = (A/|A|)*maxAcceleration
Return A
Arrive

- **Arrive** (character, E, targetRadius, slowRadius, time)
- D = E - character.position
- Length = |D|
- If Length < targetRadius **Return** (0, 0, 0)
- If Length > slowRadius **then** targetSpeed = maxSpeed
- **else** targetSpeed = maxSpeed * distance/slowRadius
- targetVelocity = (D/|D|)*targetSpeed
- A = (targetVelocity – character.velocity)/time
- If |A| > maxAcceleration **then** A = (A/|A|)*maxAcceleration
- **Return** A
Arrive

- **Arrive**\( (\text{character}, \text{E}, \text{targetRadius}, \text{slowRadius}, \text{time}) \)
- \( D = \text{E} - \text{character.position} \)
- Length = \(|D|\)
- If Length<targetRadius Return (0,0,0)
- If Length>slowRadius then targetSpeed = maxSpeed
  - else targetSpeed = maxSpeed \* \text{distance/slowRadius}
- targetVelocity = \((D/|D|)\)*targetSpeed
- \( A = (\text{targetVelocity} - \text{character.velocity})/\text{time} \)
- If \(|A|>\text{maxAcceleration}\) then \( A = (A/|A|)\)*maxAcceleration
- Return \( A \)
Arrive

- **Arrive** (character, E, targetRadius, slowRadius, time)
- \( D = E - \text{character.position} \)
- Length = |D|
- If Length<targetRadius **Return** (0,0,0)
- If Length>slowRadius then targetSpeed = maxSpeed
- else targetSpeed = maxSpeed * distance/slowRadius
- targetVelocity = (D/|D|)*targetSpeed
- A = (targetVelocity – character.velocity)/time
- If |A|>maxAcceleration then A = (A/|A|)*maxAcceleration
- **Return** A
Arrive

• **Arrive**(character, E, targetRadius, slowRadius, time)
• D = E - character.position
• Length = |D|
• **If** Length<targetRadius **Return** (0,0,0)
• **If** Length>slowRadius **then** targetSpeed = maxSpeed
• **else** targetSpeed = maxSpeed * distance/slowRadius
• targetVelocity = (D/|D|)*targetSpeed
• A = (targetVelocity – character.velocity)/time
• **If** |A|>maxAcceleration **then** A = (A/|A|)*maxAcceleration
• **Return** A
**Arrive**

- **Arrive** (character, E, targetRadius, slowRadius, time)
- \( D = E - \text{character.position} \)
- \( \text{Length} = |D| \)
- If \( \text{Length} < \text{targetRadius} \) Return \((0,0,0)\)
- If \( \text{Length} > \text{slowRadius} \) then \( \text{targetSpeed} = \text{maxSpeed} \)
- Else \( \text{targetSpeed} = \text{maxSpeed} \times \frac{\text{distance}}{\text{slowRadius}} \)
- \( \text{targetVelocity} = \frac{D}{|D|} \times \text{targetSpeed} \)
- \( A = \left( \text{targetVelocity} - \text{character.velocity} \right) / \text{time} \)
- If \( |A| > \text{maxAcceleration} \) then \( A = \left( A / |A| \right) \times \text{maxAcceleration} \)
- Return \( A \)

\( \text{targetSpeed} = \text{maxSpeed} \)
**Arrive**

- **Arrive** (character, E, targetRadius, slowRadius, time)
- D = E - character.position
- Length = |D|
- If Length < targetRadius Return (0,0,0)
- If Length > slowRadius then targetSpeed = maxSpeed
- else targetSpeed = maxSpeed * distance/slowRadius
- targetVelocity = (D/|D|)*targetSpeed
- A = (targetVelocity - character.velocity)/time
- If |A| > maxAcceleration then A = (A/|A|)*maxAcceleration
- Return A
**Arrive**

- **Arrive**(character, E, targetRadius, slowRadius, time)
- \( D = E - \text{character.position} \)
- \( \text{Length} = |D| \)
- If \( \text{Length} < \text{targetRadius} \) **Return** (0,0,0)
- If \( \text{Length} > \text{slowRadius} \) then \( \text{targetSpeed} = \text{maxSpeed} \)
  - else \( \text{targetSpeed} = \text{maxSpeed} \times \text{distance/slowRadius} \)
- \( \text{targetVelocity} = (D/|D|) \times \text{targetSpeed} \)
- \( A = (\text{targetVelocity} - \text{character.velocity})/\text{time} \)
- If \( |A| > \text{maxAcceleration} \) then \( A = (A/|A|) \times \text{maxAcceleration} \)
- **Return** A

\( \text{targetSpeed} = \text{maxSpeed} \)
**Arrive**

- **Arrive**(character, E, targetRadius, slowRadius, time)
- D = E - character.position
- Length = |D|
- If Length<targetRadius Return (0,0,0)
- If Length>slowRadius then targetSpeed = maxSpeed
  - else targetSpeed = maxSpeed * distance/slowRadius
- targetVelocity = (D/|D|)*targetSpeed
- A = (targetVelocity – character.velocity)/time
- If |A|>maxAcceleration then A = (A/|A|)*maxAcceleration
- Return A

![Diagram showing the calculation process]

targetSpeed = maxSpeed
Arrive

- **Arrive**(character, E, targetRadius, slowRadius, time)
- D = E - character.position
- Length = |D|
- If Length<targetRadius **Return** (0,0,0)
- If Length>slowRadius then targetSpeed = maxSpeed
  - else targetSpeed = maxSpeed * distance/slowRadius
- targetVelocity = (D/|D|)*targetSpeed
- A = (targetVelocity – character.velocity)/time
- If |A|>maxAcceleration then A = (A/|A|)*maxAcceleration
- **Return** A
Arrive

- **Arrive** (character, E, targetRadius, slowRadius, time)
- D = E - character.position
- Length = |D|
- If Length<targetRadius Return (0,0,0)
- If Length>slowRadius then targetSpeed = maxSpeed
- else targetSpeed = maxSpeed * distance/slowRadius
- targetVelocity = (D/|D|)*targetSpeed
- A = (targetVelocity – character.velocity)/time
- If |A|>maxAcceleration then A = (A/|A|)*maxAcceleration
- Return A
**Arrive**

- **Arrive**(character, E, targetRadius, slowRadius, time)
- D = E - character.position
- Length = |D|
- If Length<targetRadius **Return** (0,0,0)
- If Length>slowRadius then targetSpeed = maxSpeed
  - else targetSpeed = maxSpeed * distance/slowRadius
- targetVelocity = (D/|D|)*targetSpeed
- A = (targetVelocity – character.velocity)/time
- If |A|>maxAcceleration then A = (A/|A|)*maxAcceleration
- **Return** A
Arrive

- **Arrive** (character, E, targetRadius, slowRadius, time)
- D = E - character.position
- Length = |D|
- If Length<targetRadius Return (0,0,0)
- If Length>slowRadius then targetSpeed = maxSpeed
  - else targetSpeed = maxSpeed * distance/slowRadius
- targetVelocity = (D/|D|)*targetSpeed
- A = (targetVelocity – character.velocity)/time
- If |A|>maxAcceleration then A = (A/|A|)*maxAcceleration
- Return A
Arrive

- **Arrive** (character, E, targetRadius, slowRadius, time)
- $D = E - \text{character.position}$
- $\text{Length} = |D|$
- If $\text{Length} < \text{targetRadius}$ Return $(0,0,0)$
- If $\text{Length} > \text{slowRadius}$ then targetSpeed = maxSpeed
  - else targetSpeed = maxSpeed * distance/slowRadius
- targetVelocity = $(D/|D|) \times \text{targetSpeed}$
- $A = (\text{targetVelocity} - \text{character.velocity})/\text{time}$
- If $|A| > \text{maxAcceleration}$ then $A = (A/|A|) \times \text{maxAcceleration}$
- Return $A$
Arrive

- **Arrive**(character, E, targetRadius, slowRadius, time)
- D = E - character.position
- Length = |D|
- If Length<targetRadius **Return** (0,0,0)
- If Length>slowRadius **then** targetSpeed = maxSpeed
  - **else** targetSpeed = maxSpeed * distance/slowRadius
- targetVelocity = (D/|D|)*targetSpeed
- A = (targetVelocity - character.velocity)/time
- If |A|>maxAcceleration **then** A = (A/|A|)*maxAcceleration
- **Return** A
Arrive

• Most of the times targetRadius (the small one) is not needed

• It can prevent errors in cases of high speeds, low frame rates or low maximum acceleration
Align

• Tried to match the orientation of the character with that of the target
• Identical to “arrive”, but dealing with angles instead of with positions:
  • Instead of two radii, we have two intervals (targetInterval, slowInterval)
  • Rather than linear acceleration, we have angular acceleration and speed
Align

- Tried to match the orientation of the character with that of the target
- Identical to “arrive”, but dealing with angles instead of with positions:
  - Instead of two radii, we have two intervals (targetInterval, slowInterval)
  - Rather than linear acceleration, we have angular acceleration and speed

Target (0.7 rad)
Align

Align(character, E, targetInterval, slowInterval, time)
rotation = E – character.angle
rotationAmount= |D|
If rotationAmount<targetInterval Return 0
If rotationAmount>slowInterval then targetRotation = maxRotation
    else targetRotation = maxRotation* rotationAmount/slowInterval
targetRotation = targetRotation * (rotation / rotationAmount)
AA = (targetRotation – character.angularvelocity)/time
If |AA|>maxAngulatAcc then AA = (AA/|AA|)*maxAngulatAcc
Return AA
Velocity Matching

• Not very useful by itself, but can be combined with others to defined useful steering behaviors

• Can be defined by simplifying “Arrive”
Matching Velocity

MatchingVelocity(character, targetVelocity, time)
A = (targetVelocity – character.velocity)/time
If |A|>maxAcceleration  then  A = (A/|A|)*maxAcceleration
Return A
Outline

• Basic Steering Behaviors
• Steering Behaviors on Vehicles
• Composite Steering Behaviors
Steering Behaviors in Vehicles

- As defined so far, steering behaviors assume:
  - Character/vehicle under control can exert a force at an arbitrary angle
  - The direction of movement is independent of the direction being faced

- None of those assumptions are satisfied by vehicles

- Are Steering behaviors still useful then?
Motor Control Layer

- Steering Behaviors generate the “desired accelerations”. An underlying “motor layer” translates that into commands like “accelerate, brake, turn right, turn left”:
Motor Control Layer

- Two approaches:
  - Output Filtering (simple)
  - Capability-sensitive steering (complex)
Output Filtering

• Idea:
  • Use the steering behavior to produce an acceleration request
  • Project the request onto the accelerations that the vehicle at hand can perform, and ignore the rest

Steering Request:  Vehicle capabilities:  Projection:
Output Filtering

- Very simple idea
- Sometimes produces suboptimal results
- It works most of the times! Some times it does not:

If a car is still and the steering behavior wants to move perpendicularly, this will translate to only rotation and no acceleration (and the car will stand still forever)
Output Filtering

• Very simple idea
• Sometimes produces suboptimal results
• It works most of the times! Some times it does not:

This could be treated as a special case (since it’s a very unlikely situation anyway)
Exercise:

• Consider the following situation:
  • Vehicle is a car that can:
    • Accelerate forward at 5m/s$^2$
    • Accelerate backwards at 2m/s$^2$
    • Accelerate turning right/left at 1rad/s$^2$
  
  • Position of the car: (10,0,20)
  • Car is currently moving at velocity: (10,0,10), angular velocity: 0
  • Steering behavior returns A = (1,0,-10)
  • Time interval 0.1s

• Apply **output filtering** to determine what the car should do:
  • Decisions: accelerate/nothing/brake, turn left/nothing/right
Exercise:

- Linear acceleration:
  - Projection of $A$ onto the capabilities:
  - Direction vector: $(\sqrt{2}/2, 0, \sqrt{2}/2)$
  - Projection of $A$: $\sqrt{2}/2 + 0 - 5\sqrt{2} = -6.363961$
  - The car can do: -2, 0, or 5
  - The closest to -6.36 is -2: brake!
Exercise:

- Angular acceleration:
  - Angle between V and A
  - 2 steps:
    1) raw angle: \( V \cdot A = |V||A|\cos(\alpha) \)
    - \( \alpha = \arccos(V.A/|V||A|) = \arccos(-0.045) = 1.6158 \)
    2) determine whether it’s left or right
    - Find line equation passing by car, in V direction:
      - \( X - Z - 10 = 0 \)
    - Substitute (car.position + A):
      - \((10+1) - (20-10) - 10\)
      - \(-9 -> right hand side (positive means left hand side)\)
    - We need to turn “right” 1.6158 radians
    - The car can turn \(-1\text{rad/s}^2\), 0 or \(1\text{rad/s}^2\) during 0.1 seconds
      - Turn \textbf{right} is the one that will get us closest!
Capability-Sensing Steering

- Define specific steering behaviors for each vehicle
- No standard algorithms
- One possible approach:
  - If there is a limited set of commands: try all of them and select the one that gets us closer to the target
Outline

- Basic Steering Behaviors
- Steering Behaviors on Vehicles
- Composite Steering Behaviors
Composite Steering Behaviors

- Pursue and Evade
- Face
- Looking where you are going
- Wander
- Path Following
- Separation
- Collision Avoidance
- Obstacle/Wall Avoidance

- General Combination
Composite Steering Behaviors

- Pursue and Evade
- Face
- Looking where you are going
- Wander
- Path Following
- Separation
- Collision Avoidance
- Obstacle/Wall Avoidance

- General Combination
Pursue

• Pursue a moving target

Idea:
• Rather than move towards where the target is (like in “seek”):
  • Estimate where the target will be in the future and then move there

• Pursue uses “seek” as a subroutine
Pursue

**Pursue** (character, target, maxtime)

time = (target.position – character.position)/maxspeed

**If** time > maxtime **then** time = maxtime

prediction = target.position + target.velocity*time

A = seek(character,prediction)

**Return** A
Figure 3.12 Seek and pursue

Pseudo-Code

The pursue behavior derives from seek, calculates a surrogate target, and then delegates to seek to perform the steering calculation:

```python
class Pursue(Seek):
    # Holds the maximum prediction time
    maxPrediction
    # OVERRIDES the target data in seek (in other words
    # this class has two bits of data called target:
    # Seek.target is the superclass target which
    # will be automatically calculated and shouldn't
    # be set, and Pursue.target is the target we're
    # pursuing).
    target
    # ... Other data is derived from the superclass ...

def getSteering():
    # 1. Calculate the target to delegate to seek
```
Evade

- Evade moving target

- Same as pursue, but this time we delegate to “flee” rather than to “seek” (useful in some domains, but not for your project 1)
Path Following

• Follow a continuous path (this one is useful for project 1!)

• Idea:
  • 1) Estimate the closest point in the path to the character
  • 2) Compute a point that is slightly further ahead than the character in the path
  • 3) Delegate to “seek”
Path Following

So far we've seen behaviors that take a single target or no target at all. Path following is a steering behavior that takes a whole path as a target. A character with path following behavior should move along the path in one direction.

Path following, as it is usually implemented, is a delegated behavior. It calculates the position of a target based on the current character location and the shape of the path. It then hands its target off to seek. There is no need to use arrive, because the target should always be moving along the path. We shouldn’t need to worry about the character catching up with it.

The target position is calculated in two stages. First, the current character position is mapped to the nearest point along the path. This may be a complex process, especially if the path is curved or made up of many line segments. Second, a target is selected which is further along the path than the mapped point by a fixed distance. To change the direction of motion along the path, we can change the sign of this distance. Figure 3.15 shows this in action. The current path location is shown, along with the target point a little way farther along. This approach...
Path Following

- The 2 complex steps are:
  - 1) Determine the closest point in the path to the character
    - In PTSP this is not needed, since the path calculated to reach the closest point always starts in the character
    - In F1 Spirit the example code I provide already does this, so, you don’t have to worry. But if you are interested, it’s basically some basic geometry (like what we saw in the exercise before)
  - 2) Determine a point that is further along the path
    - In PTSP you can use the next point in the path returned by A* (of a point that is n steps ahead)
    - In F1 Spirit, the example code I provide also already does this. But you might need to fine-tune it to provide a good target point
Collision Avoidance

• Avoid collision with other vehicles/characters in the game (useful for the F1 Spirit version of the project)

• Idea:
  • 1) detect if a possible collision might happen
  • 2) Delegate to “evade” if collision possible

The cone check can be carried out using a dot product:

\[
\text{if } \text{orientation.asVector().dot(direction)} > \text{coneThreshold}:
\]

# do the evasion

Else:

# return no steering

where \( \text{direction} \) is the direction between the behavior’s character and the potential collision. The \( \text{coneThreshold} \) value is the cosine of the cone half-angle, as shown in Figure 3.20.

If there are several characters in the cone, then the behavior needs to avoid them all. It is often sufficient to find the average position and speed of all characters in the cone and evade that target. Alternatively, the closest character in the cone can be found and the rest ignored.

Unfortunately, this approach, while simple to implement, doesn’t work well with more than a few characters. The character does not take into account when it will actually collide but instead has a “panic” reaction to even coming close. Figure 3.21 shows a simple situation where the character will never collide, but our naive collision avoidance approach will still take action.

Figure 3.22 shows another problem situation. Here the characters will collide, but neither will take evasive action because they will not have the other in their cone until the moment of collision.

A better solution works if both the characters keep their current velocity. This involves working out the closest approach of the two characters and determining if the distance at this point is less than some threshold radius. This is illustrated in Figure 3.23.

Note that the closest approach will not normally be the same as the point where the future trajectories cross. The characters may be moving at very different velocities, and so are likely to reach the same point at different times. We simply can’t see if their paths will cross to check if the characters will collide. Instead, we have to find the moment that they are at their closest, use this to derive their separation, and check if they collide.
Collision Avoidance

- Previous idea is easy to implement, but fails in many occasions

- Solution, use the same technique as for “pursue”, use the current speed of the characters to predict where will they be
Wall Avoidance

- Idea:
  - Detect if at the current speed and direction there will be a collision to a wall
  - If yes, then find a safe target and use “seek” to go there

![Diagram of Wall Avoidance](image)

Pseudo-Code

```python
class ObstacleAvoidance (Seek):
    # Holds a collision detector
    collisionDetector

    # Holds the minimum distance to a wall (i.e., how far to avoid collision) should be greater than the radius of the character.
```
Wall Avoidance

- This can be useful for project 1 (PTSP version)

- Typically collision detection is done using a “ray” and testing if it collides with any wall

- If this is complex, an easy way is:
  - Test if there is a wall in the 10 – 30 pixels ahead (just be checking if there are wall pixels or not in those points, in the case of the PTSP)
  - Try to find a point that is 5 – 10 pixels away from the collision point and close to the path: use “seek” to go there. To find this pixel you can just search in 4 or 8 possible directions
Combining Steering Behaviors

• What if you want to combine path-following + evade? (for example)
  • Blending:
    • Run each steering behavior separately
    • Average their outputs
    • You can use priorities(weights): each steering behavior outputs a priority in addition to the acceleration, which is used to blend:
      • No point on averaging with 0 each time you have a collision avoidance that returns nothing

• Arbitration
  • Define the conditions under which each steering behavior takes control, and select only one or a subset at a time
Outline

- Basic Steering Behaviors
- Steering Behaviors on Vehicles
- Composite Steering Behaviors
Project 1: Steering Behaviors

• Physical Travel Salesman Problem
• F-1 Spirit
Next Week

• Pathfinding