Outline

• Movement Basics
• Aiming
• Jumping
• Basic Steering Behaviors
• Steering Behaviors on Vehicles
• Composite Steering Behaviors
• Project 1

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

- Movement Basics
- Aiming
- Jumping
- **Basic Steering Behaviors**
- Steering Behaviors on Vehicles
- Composite Steering Behaviors
- Project 1
Game AI Architecture

AI

- 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
Steering Behaviors: Uses

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.
Why Steering Behaviors?

• The alternative is to use kinematic movement: ignore inertia and acceleration (Pac-Man style)

• Just move the character to the right direction

• Useful for simple or non-realistic games

• E.g.: 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 needs to have during the next execution frame:
  - **Input**: position, orientation, speed, target
  - **Output**: acceleration

- They are executed once per game cycle

- Some return linear acceleration (e.g. accelerate north at $3\text{m/s}^2$), some return angular acceleration (e.g. turn right at $2\text{rad/s}^2$)
Seek

- Move towards a (potentially moving) target
Seek

- Move towards a (potentially moving) target
Seek

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

|D| = sqrt(D.x*D.x + D.y*D.y)
**Seek**

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

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

\[ ND = D / |D| \]

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

\textbf{Return} \ A

\[ |D| = \sqrt{D.x \times D.x + D.y \times D.y} \]
Seek

**Seek** (character, E)

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

ND = D / |D|

A = ND * maxAcceleration

Return A

\[
|D| = \sqrt{D.x \times D.x + D.y \times D.y}
\]
Seek

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

|D| = sqrt(D.x*D.x + D.y*D.y)
**Seek**

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

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

\[ \text{ND} = D \div |D| \]

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

**Return** \( A \)
**Seek**

\[
\text{Seek}(\text{character, E}) \\
D = E - \text{character.position} \\
\text{ND} = D / |D| \\
A = \text{ND} \times \text{maxAcceleration} \\
\text{Return } A
\]

\[|D| = \sqrt{D.x \times D.x + D.y \times D.y}\]
**Seek**

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

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

\[ ND = D / |D| \]

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

**Return** \( A \)

\[ |D| = \sqrt{D.x \times D.x + D.y \times D.y} \]
Seek

\textbf{Seek}(character, E)

D = E - character.position

ND = D / |D|

A = ND \times \text{maxAcceleration}

\textbf{Return} A

|D| = \sqrt{D.x \times D.x + D.y \times D.y}
Physics Code

In the game engine, there would be something like:

```
UpdateCharacter(character, timeDelta)
    A = character.getAI().cycle(timeDelta);
    character.position += character.velocity*timeDelta;
    character.velocity+= A*timeDelta;
```
Physics Code

In the game engine, there would be something like:

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

Inside here you will have your call to “SEEK”
Seek

- The most basic Steering Behavior
- Use to realistically make a character/vehicle move towards a target

- If you were to implement movement by directly moving the character towards the end point (without steering behaviors), you might have characters that make 90 degree turns instantly (physically impossible)
Flee

- Move away from a (potentially moving) target
Flee

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

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

• Would Seek work for trying to move to a specific target and stop there?
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, deltatime)
D = E - character.position
Distance = |D|
If Distance < targetRadius  **Return** (0,0,0)
If Distance > slowRadius  **then**  targetSpeed = maxSpeed
    **else**  targetSpeed = maxSpeed * Distance/slowRadius
targetVelocity = (D/|D|)*targetSpeed
A = (targetVelocity – character.velocity)/deltatime
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 * Length/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 * Length/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 * \( \frac{\text{Length}}{\text{slowRadius}} \)
- targetVelocity = \( \frac{D}{|D|} \)*targetSpeed
- \( A = (\text{targetVelocity} - \text{character.velocity})/\text{time} \)
- If \( |A| > \text{maxAcceleration} \) then \( A = (A/|A|)\ast\text{maxAcceleration} \)
- Return A

![Diagram showing the path from character.position to Target with vectors for character.velocity and targetVelocity, and the calculation of Length and A.](image)
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 * Length/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 * Length/slowRadius
- targetVelocity = (D/|D|)*targetSpeed
- A = (targetVelocity – character.velocity)/time
- If |A|>maxAcceleration **then** A = (A/|A|)*maxAcceleration
- **Return** A

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 * Length/slowRadius
- targetVelocity = (D/|D|)*targetSpeed
- A = (targetVelocity – character.velocity)/time
- If |A|>maxAcceleration then A = (A/|A|)*maxAcceleration
- Return A

```
targetSpeed = maxSpeed
character.position
character.velocity
```
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 \( \text{targetSpeed} = \text{maxSpeed} \times \frac{\text{Length}}{\text{slowRadius}} \)
- \( \text{targetVelocity} = \frac{D}{|D|} \times \text{targetSpeed} \)
- \( A = \frac{(\text{targetVelocity} - \text{character.velocity})}{\text{time}} \)
- If \( |A| > \text{maxAcceleration} \) then \( A = \frac{(A)|A|}{|A|} \times \text{maxAcceleration} \)
- Return \( A \)

[Diagram showing the movement of character to target with vectors and calculations.]
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 * Length/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 * Length/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 * Length/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 * Length/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*)
- \( D = E - \text{character.position} \)
- Length = \(|D|\)
- If Length<targetRadius **Return** (0,0,0)
- If Length>slowRadius then targetSpeed = maxSpeed
  - else targetSpeed = maxSpeed * Length/slowRadius
- targetVelocity = \((D/|D|)*\text{targetSpeed}\)
- \( A = (\text{targetVelocity} - \text{character.velocity})/\text{time} \)
- If |\(A|>\text{maxAcceleration} then \( A = (A/|A|)*\text{maxAcceleration} \)
- **Return** \( A \)
Arrive

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

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

- But it can prevent error in cases of high speeds, low frame rates or low maximum acceleration
Align

• Try 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**

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

![Diagram showing the alignment of an avatar (A) with a target (0.7 rad).]
Align

**Align**\((\text{character}, E, \text{targetInterval}, \text{slowInterval}, \text{time})\)

rotation = \(E - \text{character.angle}\)

rotationAmount = \(|D|\)

**If** rotationAmount < targetInterval **Return** 0

**If** rotationAmount > slowInterval **then** targetRotation = maxRotation

**else**

targetRotation = maxRotation \* rotationAmount/slowInterval

targetRotation = targetRotation \* (rotation / rotationAmount)

AA = (targetRotation – \text{character.angularvelocity})/time

**If** \(|AA|>\text{maxAngulatAcc}\) **then** AA = (AA/\(|AA|\)) \* \text{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

\[
\text{MatchingVelocity}(\text{character}, \text{targetVelocity}, \text{time})
\]
\[
A = (\text{targetVelocity} - \text{character.velocity})/\text{time}
\]

**If** \(|A|>\text{maxAcceleration}** then** \(A = (A/|A|) \times \text{maxAcceleration}**

Return A
Basic Steering Behaviors

• Seek
• Flee
• Arrive
• Align
• Velocity Matching
Outline

• Movement Basics
• Aiming
• Jumping
• Basic Steering Behaviors
• **Steering Behaviors on Vehicles**
• Composite Steering Behaviors
• Project 1
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:

What would happen here?
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)

Example results:
https://www.youtube.com/watch?v=Mh1E4Bdwxkk
Exercise:

- Consider the following situation:
  - Vehicle is a car that can:
    - Accelerate forward at $2 \text{m/s}^2$
    - Accelerate backwards at $5 \text{m/s}^2$
    - Accelerate turning right/left at $1 \text{rad/s}^2$
  - 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 $0.1 \text{s}$

- 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: -5, 0, or 2
  - The closest to -6.36 is -5: brake!
Exercise:

- Angular acceleration (easy method):
  - determine whether A is “to the left” or “to the right” of V:
    - Get vector “right” (just rotate V 90 degrees clockwise): (10,0,-10)
    - Project A on “right”: positive means turn right, negative means turn left. In this case: turn right.
  - You could then have a “threshold” (if it’s positive, but it’s a very small value, e.g. 0.01, you might not want to turn).
Exercise:

- Angular acceleration (better, more elaborate method):
  - Angle between $V$ and $A$
  - 2 steps (first is same as before):
    - 1) determine whether it’s left or right
      - Get vector “right” (just rotate $V$ 90 degrees clockwise): $(10,0,-10)$
      - Project $A$ on “right”: positive means turn right, negative means turn left. In this case: turn right.
    - 1) raw angle: $V \cdot A = |V||A|\cos(\alpha)$
      - $\alpha = \arccos\left(\frac{V \cdot A}{|V||A|}\right) = \arccos(-0.045) = 1.6158$
      - 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 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

- Movement Basics
- Aiming
- Jumping
- Basic Steering Behaviors
- Steering Behaviors on Vehicles
- Composite Steering Behaviors
- Project 1
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
Seek vs Pursue

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
        ...
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
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

• 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
    • Might involve finding the closest point in a segment, or in a split to the player character
  • 2) Determine a point that is further along the path
Collision Avoidance

- Avoid collision with other vehicles/characters in the game

- Idea:
  - 1) detect if a possible collision might happen
  - 2) Delegate to “flee” if collision possible
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
Wall Avoidance

• This is needed for project 1

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

• So, I recommend:
  1. Start by implementing a “rayCast” method: use the RotatedRectangle class to move a rectangle along the “ray” until it collides with something (this is very slow, and in a real game you would never do it this way, but for this project, it suffices)
  2. Once you have that, cast 3 rays: in front, 45 degrees to the left, and 45 degrees to the right.
  3. Based on those, if any ray hits a wall, set a safe target and “seek” there.
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.

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

• 3D:
  • Most behaviors we saw today work in 3D equally well
  • Some 3D-specific steering behaviors

• Flocking:
  • You can combine steering behaviors and apply them to large collections of characters to achieve flocking, crowd behaviors, etc.
  • There are many cool effects you can achieve with this in games: e.g. NPCs queuing up to enter buildings, NPCs realistically navigating around a city, mops reacting to disasters, animal behavior (groups of birds, fish, etc.).
Outline

• Movement Basics
• Aiming
• Jumping
• Basic Steering Behaviors
• Steering Behaviors on Vehicles
• Composite Steering Behaviors
• Project 1
Project 1: Steering Behaviors

- Implement steering behaviors (explained week 2)
- Game Engine:
  - Simple car driving (Java)

- Goals:
  - Seek
  - Arrive
  - Wall avoidance

- But please feel free to go beyond these!