DM842

Computer Game Programming: AI

Lecture 1

AI for Games
Movement Behaviors

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

1. Practicalities
2. Introduction
3. Representations
4. Kinematic Movement
    - Seeking
    - Wandering

Outline

1. Practicalities

2. Introduction

3. Representations

4. Kinematic Movement
    - Seeking
    - Wandering

Introduction

- Web page
- Schedule
- Book
  The textbook is comprehensive, structured, and oriented towards real-life Game AI practice. Author is both well educated in AI, and has 20+ years experience as AI programmer and designer in game industry.

  Book structure (850 pages):
  - Ch. 1-2: Intro to Game AI
  - Ch. 3-8: Game AI techniques
  - Ch. 9-11: Surrounding issues
    (AI execution scheduling, gameworld interfacing, tools and content creation).
  - Ch. 12-13: Game AI technology choices by game genre.

Contents

- Movement algorithms
- 3D Movement
- Pathfinding
- Decision making: Decision trees, State Machine, Behavior Trees
- Behavior Trees and Fuzzy Logic
- Goal-Oriented Behavior
- Tactical and Strategic AI
- Board Games AI

- Bunch of different techniques
- Often a bit limited in depth
- Quite a number of pages to read
- Best if simultaneously trying things out in your projects

Outline

1. Practicalities
2. Introduction
3. Representations
4. Kinematic Movement
    - Seeking
    - Wandering

Classic AI

AI = make machines behave like human beings when solving "fuzzy problems".

Classic AI:

- Philosophical motivation: What is intelligence, what is thinking and decision making?
- Psychological motivation: how does the brain work?
- Engineering motivation: make machine carry out such tasks.

Game AI: Related to third point. Happy to draw on results from AI research, but goal is solely behavior generation in computer games.

Academic AI Eras

- Prehistoric era (-1950): Philosophic questions, mechanical novelty gadgets, what produces thought?
- Symbolic era (1950-1985): symbolic representations of knowledge + reasoning (search) algorithms working on symbols.
  Examples: expert systems, decision trees, state machines, path finding, steering.
- Natural era (1985-): techniques inspired by biological processes.
  Examples: neural networks, genetic algorithms, simulated annealing, ant colony optimization.

Philosophical schools:
weak AI hypothesis: assertion that machines could possibly act intelligently (or, perhaps better, act as if they were intelligent)
strong AI hypothesis: assertion that machines that do so are actually thinking (as opposed to simulating thinking)

AI approach

Bottom up approach:
Agent-based models with emergent behavior.
Movement and individual decision making are the two basic elements.
(eg: computational intelligence, swarm intelligence)

Top down approach:
non agent-based models in which everything is simulated and optimized and then single character's actions are decided.

AI in Games

In games, agents are called game characters.
(In first person shooting games they are called bots)

Author: natural techniques currently more fashionable, but not more successful than symbolic. Game AI techniques often are symbolic.

No AI technique works for everything ("knowledge vs. search trade-off"). In games, simple is often good.

More ad hoc techniques. Both bottom up and top down.

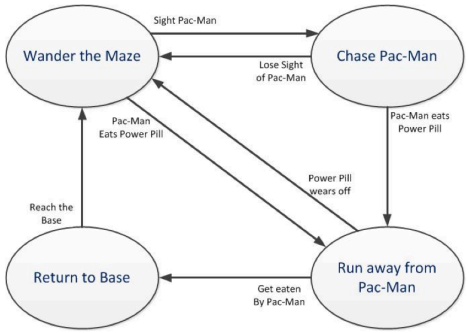
Computer Game Genres

v·t·e	Video game genres	[hide]
Action	Beat 'em up (Hack and slash) · Fighting · Maze (*Pac-Man* clone) · Platform · Shooter (First-person · Light gun · Shoot 'em up · Tactical · Third-person)	
Action-adventure	*Grand Theft Auto* clone · Stealth · Survival horror	
Adventure	Dating sim (Bishōjo · Eroge · Otome) · Graphic adventure (Escape the room) · Interactive fiction · Interactive movie · Visual novel	
Role-playing game	Action role-playing · Roguelike (Dungeon crawl) · MUD (MMORPG) · Tactical role-playing	
Simulation	Construction and management (Business · City · Government) · Life simulation (Dating sim · Digital pet · God game · Social simulation) · Sports game	
Strategy	4X · Real-time strategy (MOBA · Tower defense) · Real-time tactics · Turn-based strategy · Turn-based tactics (Artillery) · Wargame	
Vehicle simulation	Flight simulator (Amateur · Combat · Space) · Racing game (Sim racing) · Submarine simulator · Train simulator · Vehicular combat	
Other genres	*Breakout* clone · Exergame · Music game (Rhythm game) · Non-game · Party · Programming · Puzzle (Sokoban · Tile-matching)	
Related concepts	Arcade game · Art game · Audio game · Casual game · Console game · Christian game · Clone game · Crossover game · Cult game · FMV · Indie game · Minigame · Nonlinear gameplay (Open world) · Online game (Browser game · MMOG · Social network game) · PC game · Serious game (Advergame · Edugame)	

- Video Games and Learning
  https://www.coursera.org/course/videogameslearning
- Documentary: Tetris - From Russia with Love
  http://www.youtube.com/watch?v=NhwNTo_Yr3k

AI in Games, examples

- Pacman (1979) first game using AI
  - Opponent controllers, enemy characters, computer controlled chars.
  - used a finite state machine



- Warcraft 1994
  - path finding, robust formation motion, emotional models, personality
- The Sims 2000, Creatures 1997
  - neural network brain for creatures
- Present:
  - Simple AI
  - Bots in first person games and simulators, advanced AI
  - Real-time strategy (RTS) games, advanced AI
  - Sport and driving games still pose challenges (dynamic path finding)
  - Role-playing games (RPG) conversation still challenging

Tasks of Computer Chars

- Actions:
  attacking, standing still, hiding, exploring, patrolling, ...
- Movement:
  - going to the player before the attack
  - avoiding obstacles on the way
  - a lot done by animation but still need to decide what to do. Eg: eating animation
- Decision making: deciding which action at each moment of the game. (then movement AI + animation technology)
- Strategy:
  coordinate a team while still leaving to each individual its own decision making and movement

Needs for AI in Games

Three main areas:

- Movement (single characters)
- Decision making (where? short term, single character behavior)
- Strategic AI (long term, group behavior)

Not all games have all areas (eg. chess vs. platform game).

Associated issues:

- Gameworld interface (input to AI)
- Execution/scheduling of AI
- Scripting, content creation
- Animation ($\neq$ movement) and Physics (not in book)

The complexity fallacy

Fallacy: More complex AI gives more convincing behavior.

Often, the right simple technique (or combination of simple techniques) looks good.

Complex, intricate techniques often look bad.

"The best AI programmer are those who can use a very simple technique to give the illusion of complexity."

The Perception Window

Most non-player characters (NPC) are met briefly. Adapt AI complexity to players exposure to the character. Advanced AI will look random (so simply use randomization).

Change of behavior is very noticeable (more than behavior itself), and should correspond to relevant events (like being seen).

Algorithms, Hacks and Heuristics

In this part of the course (and in the book) we focuses on general techniques and algorithms for generating behavior and representations for interfacing.

However, real Game AI often employs ad hoc hacks and heuristics.

In games, perceived behavior, not underlying technique, is what matters.

Behaviorist approach: we do not study the principles behind human behavior (as academic AI does), but try to emulate it sufficiently well.
And if an ad hoc method or simple emotional animation will do it, then fine.

Efficiency

AI is done on CPU.
Trend: more tasks taken over by GPU, hence more time left for AI in CPU.
Could be 10-50% of total time cycles.

Heavy AI calculations (eg path finding for 1000 chars) can be distributed (scheduled) over many frames.

Parallelism: easiest when there are several NPCs. SIMD and Multi-core possibilities.

Branch-prediction and, above all, cache-efficiency may impact considerably performance

PC development: should run on varied hardware. Often hard to implement AI that scales with changing hardware (without impacting gameplay). So developers target AI to the minimum hardware requirements. Scaling may be done by reducing number of NPCs.

Console development: development platform is often PCs, so many small tweaks during development is harder.

The AI Engine

- Reuse of code saves programming time.
- Content creation takes up the bulk of a game development time. Tools for this are necessary.

⇝ For AI (as for graphics), generic (in-house) engines are now common.

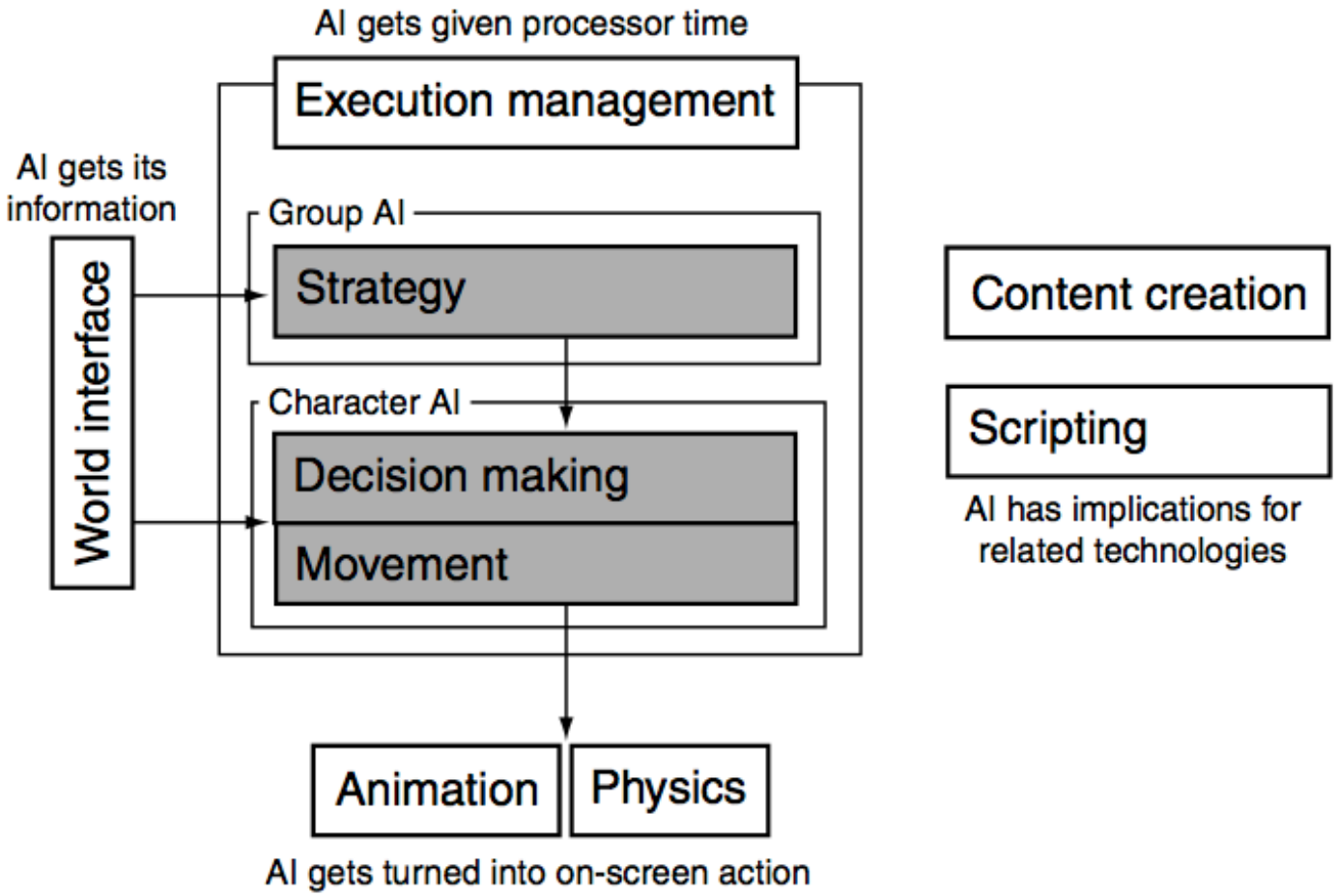
⇝ interfacing

⇝ infrastructure, action, support technology

(Example, CD code, python game)

For this, AI knowledge representation forms needs to be decided upon and put into level editing tools. (See Figure 2.2. More details in Ch. 11.)

Summary


AI gets given processor time
Execution management
AI gets its information
World interface
Group AI
Strategy
Character AI
Decision making
Movement
Content creation
Scripting
AI has implications for related technologies
Animation
Physics
AI gets turned into on-screen action

Outline

1. Practicalities

2. Introduction

3. Representations

4. Kinematic Movement
    - Seeking
    - Wandering

Movement

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

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

Output: geometric data representing movement (velocity, accelerations)

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

Running:

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

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

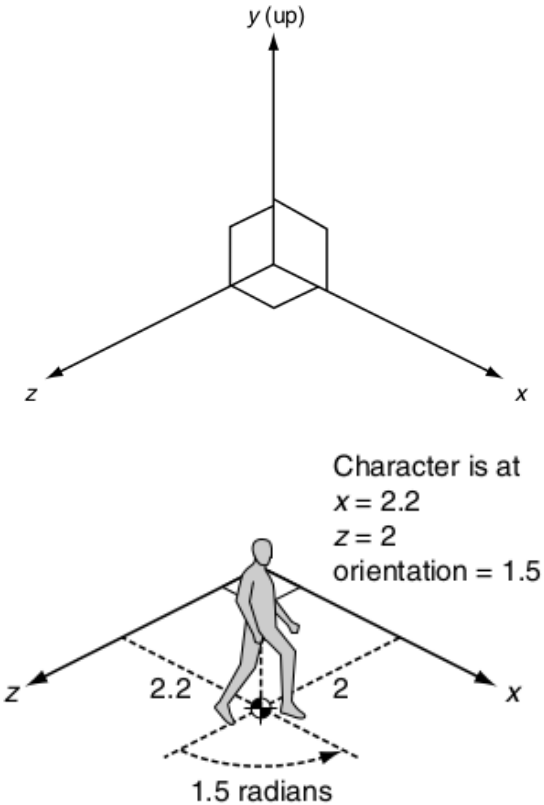
Static Representations

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

In 2D:

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

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




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

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

Static Representations

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

Full 3D may be needed in flight simulators

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

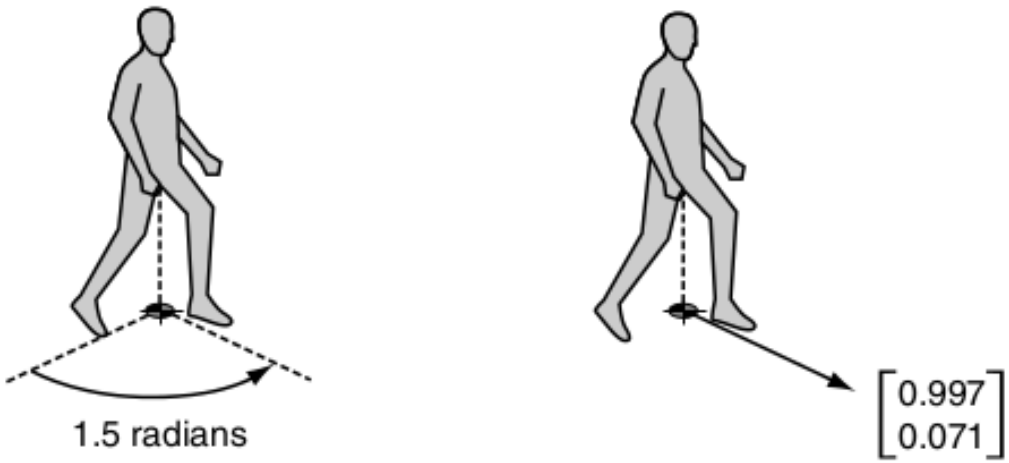
Hybrid model:

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

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

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

Orientation in Vector Form



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

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

Kinematic Representations

- Kinematic algorithms:
  position + orientation + velocity

	2D	$2\frac{1}{2}$D
linear velocity **v**	v_x, v_z components	v_x, v_y, v_z components
angular velocity θ'	π/s	π/s
(scalar, tangential to radius)		

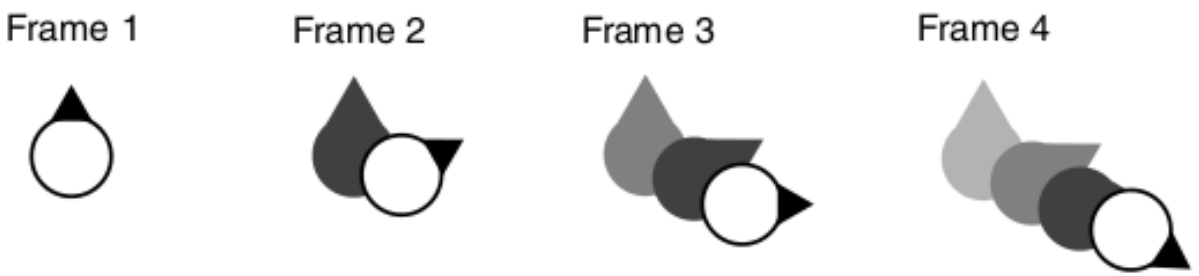
struct Kinematic:
  position *# 2 or 3D vector*
  orientation *# single floating point value*
  velocity *# 2 or 3D vector*
  rotation *# single floating point value*

- Steering algorithms:
  return linear acceleration **a** and angular acceleration θ'' that modify Kinematic:

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

Independent facing

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



Kinematic Representations

- Updates (classical mechanics)

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

$$\begin{aligned} \mathbf{r} &= \mathbf{v}t + \tfrac{1}{2}\,\mathbf{a}t^2 + \mathbf{r}_0 \\ \theta &= \theta' t + \tfrac{1}{2}\theta'' t^2 + \theta_0 \end{aligned} \qquad \begin{aligned} \mathbf{v} &= \mathbf{a}t + \mathbf{v}_0 \\ \theta' &= \theta'' t + \theta'_0 \end{aligned}$$

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

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

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

Netwon's Physics

Accelerations are determined by forces and inertia ($\mathbf{F} = m\mathbf{a}$)

To model object inertia:

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

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

In most of the cases steering algorithms are defined with accelerations.

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

Outline

1. Practicalities

2. Introduction

3. Representations

4. Kinematic Movement
    - Seeking
    - Wandering

Kinematic Movement Algorithms

Input: static data

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

From **v** we calculate orientation using trigonometry:

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

(sign because counterclockwise from z-axis)

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

Seeking

Input: character's and target's static data

Output: velocity along direction to target

```
struct Static:
    position
    orientation
```

```
struct KinematicSteeringOutput:
    velocity
    rotation
```

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

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

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

- getNewOrientation can be taken out
- flee mode:

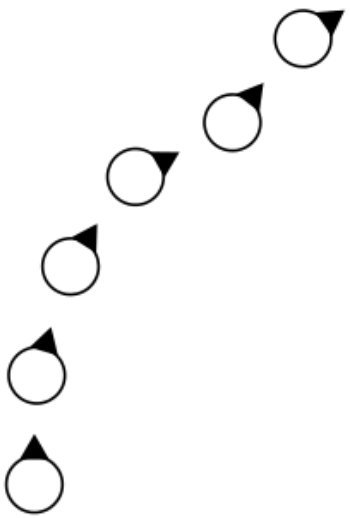
```
steering.velocity = character.position − target.position
```

- what happens at arrival? wiggling back and forth while it should be stationary
  - use large radius of satisfaction to target
  - use a range of movement speeds, and slow the character down as it reaches its target

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

Wandering

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



```
class KinematicWander:
    character
    maxSpeed
    maxRotation # speed
    def getSteering():
        steering = new KinematicSteeringOutput()
        steering.velocity = maxSpeed * character.orientation.asVector()
        steering.rotation = (random(0,1)-random(0,1)) * maxRotation
        return steering
```

Demo

Outline

1. Practicalities
2. Introduction
3. Representations
4. Kinematic Movement
    - Seeking
    - Wandering