Lecture 21

Character AI: Thinking and Acting
Take Away for Today

- Review the **sense-think-act** cycle
  - How do we separate actions and thinking?
  - Delay the sensing problem to next time

- What is **rule-based** character AI?
  - How does it relate to sense-think-act?
  - What are its advantages and disadvantages?

- What **alternatives** are there to rule-based AI?
  - What is our motivation for using them?
  - How do they affect the game architecture?
Classical AI vs. Game AI

- **Classical**: Design of *intelligent agents*
  - Perceives environment, maximizes its success
  - Established area of computer science
  - Subtopics: planning, machine learning

- **Game**: Design of *rational behavior*
  - Does not need to optimize (and often will not)
  - Often about “scripting” a personality
  - More akin to cognitive science
Roles of AI in Games

- **Autonomous Characters** (NPCs)
  - Mimics the “personality” of the character
  - May be opponent or support character

- **Strategic Opponents**
  - AI at the “player level”
  - Closest to classical AI

- **Character Dialog**
  - Intelligent commentary
  - Narrative management (e.g. Façade)
Role of AI in Games

- **Autonomous Characters** (NPCs)
  - Mimics the “personality” of the character
  - May be opponent or support character

- **Strategic Opponents**
  - AI at the “player level”
  - Closest to classical AI

- **Character Dialog**
  - Intelligent commentary
  - Narrative management (e.g. Façade)
Review: Sense-Think-Act

- **Sense:**
  - Perceive the world
  - Reading the game state
  - **Example:** enemy near?

- **Think:**
  - Choose an action
  - Often merged with sense
  - **Example:** fight or flee

- **Act:**
  - Update the state
  - Simple and fast
  - **Example:** reduce health

Thinking and Acting
S-T-A: Separation of Logic

- **Loops** = sensing
  - Read other objects
  - *Aggregate* for thinking
  - **Example**: nearest enemy

- **Conditionals** = thinking
  - Use results of sensing
  - Switch between possibilities
  - **Example**: attack or flee

- **Assignments** = actions
  - Rarely need loops
  - Avoid conditionals

```c
move(int direction) {
    switch (direction) {
        case NORTH:
            y -= 1;
            break;
        case EAST:
            x += 1;
            break;
        case SOUTH:
            y += 1;
            break;
        case WEST:
            x -= 1;
            break;
    }
}
```
S-T-A: Separation of Logic

- **Loops** = sensing
  - Read other objects
  - *Aggregate* for thinking
  - **Example**: nearest enemy

- **Conditionals** = thinking
  - Use results of sensing
  - Switch between possibilities
  - **Example**: attack or flee

- **Assignments** = actions
  - Rarely need loops
  - Avoid conditionals

```java
move(int direction) {
    switch (direction) {
        case NORTH:
            y -= 1;
            break;
        case EAST:
            x += 1;
            break;
        case SOUTH:
            y += 1;
            break;
        case WEST:
            x -= 1;
            break;
    }
}
```
S-T-A: Separation of Logic

- **Loops** = sensing
  - Read other objects
  - *Aggregate* for thinking
  - **Example**: nearest enemy

- **Conditionals** = thinking
  - Use results of sensing
  - Switch between possibilities
  - **Example**: attack or flee

- **Assignments** = actions
  - Rarely need loops
  - Avoid conditionals

```c
move(int direction) {
    switch (direction) {
        case NORTH:
            y -= 1;
            break;
        case EAST:
            x += 1;
            break;
        case SOUTH:
            y += 1;
            break;
        case WEST:
            x -= 1;
            break;
    }
}
```

```c
move(int dx, int dy) {
    x += dx;
    y += dy;
}
```
S-T-A: Reducing Dependencies

Actor1 Controller

GameState

Actor2 Controller

Actor1

Actor2

Thinking and Acting
S-T-A: Reducing Dependencies

Actor1 Controller

Actor1

Compute Sensing

GameState

Actor2 Controller

Actor2

Sensing
S-T-A: Reducing Dependencies

Actor1 Controller

Actor2 Controller

GameState

Actor1

Actor2

Compute Thinking

Thinking and Acting
S-T-A: Reducing Dependencies

Actor1 Controller

Compute Actions

Actor2 Controller

GameState

Actor1

Actor2

Thinking and Acting
Review: Sense-Think-Act

- **Sense:**
  - Perceive the world
  - Reading the game state
  - **Example:** enemy near?

- **Think:**
  - Choose an action
  - Often merged with sense
  - **Example:** fight or flee

- **Act:**
  - Update the state
  - Simple and fast
  - **Example:** reduce health
Mainly use assignments
- Avoid loops, conditionals
- Similar to getters/setters
- Complex code in thinking

Helps with serializability
- Record and undo actions

Helps with networking
- Keep doing last action
- Recall: dead reckoning

move(int direction) {
    switch (direction) {
        case NORTH:
            y -= 1;
            break;
        case EAST:
            x += 1;
            break;
        case SOUTH:
            y += 1;
            break;
        case WEST:
            x -= 1;
            break;
    }
}

move(int dx, int dy) {
    x += dx;
    y += dy;
}
Delivering Actions

Sequential Actions are Bad

Think (Choose) & Act (Apply)

Choose Action; Apply Later

Think (Choose)

Act (Apply)
Thinking: Primary Challenge

- A mess of conditionals
  - “Spaghetti” code
  - Difficult to modify
- Abstraction requirements:
  - Easy to visualize models
  - Mirror “cognitive thought”
- Want to separate talent
  - **Sensing:** Programmers
  - **Thinking:** *Designers*
  - **Actions:** Programmers
Rule-Based AI

If $X$ is true, Then do $Y$

Three-Step Process

- **Match**
  - For each rule, check if
  - Return *all* matches

- **Resolve**
  - Can only use one rule
  - Use metarule to pick one

- **Act**
  - Do *then*-part
If $X$ is true, Then do $Y$

- **Thinking**: Providing a list of several rules
- But what happens if there is more than one rule?
- Which rule do we choose?
Rule-Based AI

If $X$ is true, Then do $Y$

- **Thinking**: Providing a list of several rules
  - But what happens if there is more than one rule?
  - Which rule do we choose?
Conflict Resolution

- Often **resolve by order**
  - Each rule has a priority
  - Higher priorities go first
  - “Flattening” conditionals

- **Problems:**
  - Predictable
    - Same events = same rules
  - Total order
    - Sometimes no preference
  - Performance
    - On average, go far down list

\[
\begin{align*}
R_1 &: \text{if } \text{event}_1 \text{ then } \text{act}_1 \\
R_2 &: \text{if } \text{event}_2 \text{ then } \text{act}_2 \\
R_3 &: \text{if } \text{event}_3 \text{ then } \text{act}_3 \\
R_4 &: \text{if } \text{event}_4 \text{ then } \text{act}_4 \\
R_5 &: \text{if } \text{event}_5 \text{ then } \text{act}_5 \\
R_6 &: \text{if } \text{event}_6 \text{ then } \text{act}_6 \\
R_7 &: \text{if } \text{event}_7 \text{ then } \text{act}_7
\end{align*}
\]
Conflic Resolution

- **Specificity:**
  - Rule with most "components"

- **Random:**
  - Select randomly from list
  - May "weight" probabilities

- **Refractory Inhibition:**
  - Do not repeat recent rule
  - Can combine with ordering

- **Data Recency:**
  - Select most recent update

\[ R_1: \text{if } A, B, C, \text{ then} \]
\[ R_2: \text{if } A, B, D, \text{ then} \]
Impulses

- Correspond to certain events
  - **Global**: not tied to NPC
  - Must also have duration
- Used to **reorder** rules
  - Event makes rule important
  - Temporarily up the priority
  - Restore when event is over
- Preferred conflict resolution
  - Simple but flexible
  - Used in *Halo 3* AI.

\[
\begin{align*}
R_1 &: \text{if event}_1 \text{ then act}_1 \\
R_2 &: \text{if event}_2 \text{ then act}_2 \\
R_5 &: \text{if event}_5 \text{ then act}_5 \\
R_3 &: \text{if event}_3 \text{ then act}_3 \\
R_4 &: \text{if event}_4 \text{ then act}_4 \\
R_6 &: \text{if event}_6 \text{ then act}_6 \\
R_7 &: \text{if event}_7 \text{ then act}_7
\end{align*}
\]
Rule-Based AI: Performance

- Matching = **sensing**
  - If-part is expensive
  - Test *every* condition
  - Many unmatched rules

- Improving performance
  - Optimize sensing (make if-part cheap)
  - Limit number of rules
  - Other solutions?

- Most games limit rules
  - Reason for *state machines*
Rule-Based AI: Performance

- Matching = **sensing**
  - If-part is expensive
  - Test *every* condition
  - Many unmatched rules

- Improving performance
  - Optimize sensing (make if-part cheap)
  - Limit number of rules
  - Other solutions?

- Most games limit rules
  - Reason for *state machines*

- 90-95% of time
  - Updated State
  - Matching Rules
  - Resolve Conflicts
  - Selected Rule
  - Act
Finite State Machines

Slide courtesy of John Laird

Thinking and Acting
Finite State Machines

Events
- E=Enemy Seen
- S=Sound Heard
- D=Die

Only check rules for outgoing edges

Slide courtesy of John Laird
Implementation: Model-View-Controller

- Games have **thin** models
  - Methods = get/set/update
  - Controllers are heavyweight

- AI is a **controller**
  - Uniform process over NPCs

- But behavior is **personal**
  - Diff. NPCs = diff. behavior
  - Do not want unique code

- What can we do?
  - Data-Driven Design

---

**Controller**
- Updates model
- Updates view

**Model**
- Manages the data
- Reacts to requests

**View**
- Displays model
- Provides interface
Implementation: Model-View-Controller

- **Actions** go in the model
  - Lightweight updates
  - Specific to model or role

- **Controller** is framework for general **sensing, thinking**
  - Standard FSM engine
  - Or FSM alternatives (later)

- **Process** stored in a model
  - Represent thinking as **graph**
  - Controller processes graph
An Aside: Animations

- AI may need many actions
  - Run, jump, duck, slide
  - Fire weapons, cast spells
  - Fidget while idling

- Want animations for all
  - Is loop appropriate for each?
  - How do we transition?

- Idea: shared boundaries
  - End of loop = start of another
  - Treat like advancing a frame
An Aside: Animations

- AI may need many actions
  - Run, jump, duck, slide
  - Fire weapons, cast spells
  - Fidget while idling

- Want animations for all
  - Is loop appropriate for each?
  - How do we transition?

- Idea: shared boundaries
  - End of loop = start of another
  - Treat like advancing a frame
An Aside: Animations

- AI may need many actions
  - Run, jump, duck, slide
  - Fire weapons, cast spells
  - Fidget while idling

- Want animations for all
  - Is loop appropriate for each?
  - How do we transition?

- Idea: shared boundaries
  - End of loop = start of another
  - Treat like advancing a frame
Animation and State Machines

- **Idea**: Each sequence a state
  - Do sequence while in state
  - Transition when at end
  - Only loop if loop in graph

- A graph edge means...
  - Boundaries match up
  - Transition is allowable

- Similar to data driven AI
  - Created by the designer
  - Implemented by programmer
  - Modern engines have tools

Scene Graphs
Animation and State Machines

- **Idea**: Each sequence a state
  - Do sequence while in state
  - Transition when at end
  - Only loop if loop in graph

- A graph edge means…
  - Boundaries match up
  - Transition is allowable

- Similar to data driven AI
  - Created by the designer
  - Implemented by programmer
  - Modern engines have tools
Complex Example: Jumping

- **Stand**
  - Stand2crouch
    - Crouch
      - Takeoff
    - Hop
      - Float
    - Land
Complex Example: Jumping

- Stand
- Stand2Crouch
- Crouch
- Hop
- Takeoff
- Float
- Near Ground
- Jump Press
- Jump Release
- Land

Scene Graphs
Complex Example: Jumping

Transition state needed to align the sequences
LibGDX Interfaces

StateMachine\(<E>\>

- Attached to an entity
- Constructor
- Updates current state.
- Must implement methods
  - `update()`
  - `changeState(State<A> state)`
  - `revertToPreviousState()`
  - `getCurrentState()`
  - `isInState(State<A> state)`
- DefaultStateMachine provided

State\(<E>\>

- Not attached to an entity
- `StateMachine` sets state
- `StateMachine` passes entity
- Must implement methods
  - `enter(E entity)`
  - `exit(E entity)`
  - `update(E entity)`

Transition
logic external
to the state machine.

Thinking and Acting
# LibGDX Interfaces

## StateMachine\(<E>\>

- **Attached to an entity**
  - Set the entity in constructor
  - New entity, new state machine
- **Must implement methods**
  - `update()`
  - `changeState(State<A> state)`
  - `revertToPreviousState()`
  - `getCurrentState()`
  - `isInState(State<A> state)`
- **DefaultStateMachine provided**

## State\(<E>\>

- **Not attached to an entity**
  - StateMachine sets state
  - StateMachine passes entity
- **Must implement methods**
  - `enter(E entity)`
    - When machine enters state
  - `exit(E entity)`
    - When machine enters state
  - `update(E entity)`
    - When machine stays in state
Problems with FSMs

Events
- E = Enemy Seen
- S = Sound Heard
- D = Die

No edge from Attack to Chase

Slide courtesy of John Laird

Thinking and Acting
Problems with FSMs

Events

- E=Enemy Seen
- S=Sound Heard
- D=Die

Requires a redundant state

Slide courtesy of John Laird
Problems with FSMs

Events
- E=Enemy Seen
- S=Sound Heard
- D=Die
- L=Low Health

Adding a new feature can double states

Slide courtesy of John Laird
Problems with FSMs

Adding a new feature can double states

Might as Well Go Back to Rule Based AI

Events
- E=Enemy Seen
- S=Sound Heard
- L=Low Health

Slide courtesy of John Laird
An Observation

- Each state has a set of **global attributes**
  - Different attributes may have same actions
  - Reason for redundant behavior

- Currently just cared about attributes
  - Not really using the full power of a FSM
  - Why don’t we just check attributes directly?

- Attribute-based selection: **decision trees**
Decision Trees

- Thinking **encoded as a tree**
  - Attributes = tree nodes
  - Left = true, right = false
  - Actions = leaves (reach from the root)

- Classify by **descending** from root to a leaf
  - Start with the test at the root
  - Descend the branch according to the test
  - Repeat until a leaf is reached
Decision Tree Example

Start Here

D?

<table>
<thead>
<tr>
<th>t</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spawn</td>
<td>E?</td>
</tr>
</tbody>
</table>

E?

<table>
<thead>
<tr>
<th>t</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>L?</td>
<td>S?</td>
</tr>
</tbody>
</table>

L?

<table>
<thead>
<tr>
<th>t</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retreat</td>
<td>Attack</td>
</tr>
</tbody>
</table>

S?

<table>
<thead>
<tr>
<th>t</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retreat</td>
<td>Chase</td>
</tr>
</tbody>
</table>

Action

Slide courtesy of John Laird
Decision Tree Example

Slide courtesy of John Laird
**FSMs vs. Decision Trees**

**Finite State Machines**
- Not limited to attributes
- Allow “arbitrary” behavior
- Explode in size very fast

**Decision Trees**
- Only attribute selection
- Much more manageable
- Mixes w/ machine learning

![Finite State Machines Diagram](image)

![Decision Trees Diagram](image)
Behavior Trees

- Part rule-based
- Part decision tree
- Freedom of FSM (almost)

- Node is a list of actions
- Select action using rules
- Action leads to subactions
Behavior Trees

Ordered Rules

Act  Root

Ordered Rules with Actions

Flee  Hide

Shoot  Charge  Grenade

Wander  Guard

Rule Outcome

Thinking and Acting
Behavior Trees

Ordered Rules

Rule Outcome

Act

Root

Ordered Rules with Actions

Flee Hide Shoot Charge Grenade Wander Guard

Thinking and Acting
LibGDX Behavior Trees

- Base actions are defined at the leaves
- Internal nodes to **select** or even **combine** tasks

![Diagram of LibGDX Behavior Trees]

- Task
- Task
- Task
- Task
- Task
- Basic Task
- Composite Task

Thinking and Acting
LibGDX Behavior Trees

- Base actions are defined at the leaves
- Internal nodes to select or even combine tasks

Use classes in LibGDX
(sub)Classes you create

Can be either condition (if) or an action (then)
LibGDX Rules

• **Selector** rules
  - Tests each subtask for success
  - Tasks are tried independently
  - Chooses first one to succeed

• **Sequence** rules
  - Tests each subtask for success
  - Tasks are tried in order
  - Does all if succeeds; else none

• **Parallel** rules
  - Tests each subtask for success
  - Tasks are tried simultaneously
  - Does all if succeeds; else none
Decorator Rules

- Rules with a single child
  - Wrap subtree as single task
  - Modify the meaning of task

- Example decorators
  - AlwaysFail
  - AlwaysSucceed
  - Invert (do the opposite)
  - Limit (# of times to do)

- Supports dynamic sequences
  - UntilFail (repeat until fail)
  - UntilSuccess
Tactical Managers

- “Invisible NPC”
  - Assigned to NPC Group
  - Performs all thinking
  - NPCs just follow orders

- Applications
  - Protecting special units
  - Flanking
  - Covering fire
  - Leapfrogging advance
Protecting Special Units

Slide courtesy of Dave Mark

Thinking and Acting
Protecting Special Units
Protecting Special Units

Slide courtesy of Dave Mark

Thinking and Acting
Protecting Special Units

Slide courtesy of Dave Mark
Protecting Special Units

Slide courtesy of Dave Mark

Thinking and Acting
Protecting Special Units

Flanking!!!

Slide courtesy of Dave Mark
Protecting Special Units

Flanking!!!
Protecting Special Units

Flanking!!!

Slide courtesy of Dave Mark
Protecting Special Units

Flanking!!!

Slide courtesy of Dave Mark
Character AI is a software engineering problem
- Sense-think-act aids code reuse and ease of design
- Least standardized aspect of game architecture

Rule-based AI is the foundation for all character AI
- Simplified variation of sense-think-act
- Alternative systems made to limit number of rules

Games use graphical models for data-driven AI
- Controller outside of NPC model processes AI
- Graph stored in NPC model tailors AI to individuals