Lecture 3 – Finite State Machines

CMPS 146, Fall 2013

Josh McCoy
Readings

First three will be posted right after class.
- First lecture: 19-35
- Decision Tree lecture: 293-309
- FSM/today's lecture: 309-333

All three lectures on the website.

Is anyone having issues accessing the readings or course material?

Readings will only be on soe.ucsc.edu for copyright reasons.
Pac-Man AI Competition Information

http://www.pacman-vs-ghosts.net/

Project information and official assignment Monday!

Team Size: 2

Requests for 3:
• under consideration
• 50% more work: HFSM or DT/FSM hybrid
DRC Accommodations

If you qualify for classroom accommodations because of a disability, please get an Accommodation Authorization from the Disability Resource Center (DRC) and submit it to the instructor in person outside of class (e.g., office hours) within the first two weeks of the quarter. Contact DRC at 459-2089 (voice), 459-4806 (TTY), or http://drc.ucsc.edu for more information on the requirements and/or process.
The AI Model

- **Execution Management**
- **Group AI**: Strategy
- **Character AI**: Decision Making, Movement
- **World Interface**
- **Content Creation**
- **Scripting**
- **Animation**
- **Physics**

CMPS 146, Fall 2013
Decision Making AI

- Internal Knowledge
- External Knowledge
- Decision Maker
- Internal changes
- External changes
- Action Request

expressiveintelligencestudio
If (visible) {
  if (distance < 10) {
    attack();
  } else {
    if (flank){
      move();
    } else  {
      Attack();
    }
  }
} else if (audible){
  creep();
}
Lots of branching code:
Difficult to author
Difficult to maintain
Difficult to debug
Hard to insert or modify behaviors
Hard to reason temporally
Benefits of Decision Tree Representation

- Provides a simple abstraction above raw code
- Easier to understand and modify
- Supports team development
- Fast
Drawbacks of Decision Trees
Drawbacks of Decision Trees

- Have to explicitly decide what to do from ground truth
  - No previous context or history used.

- Modifications in the middle of tree can be problematic.

- Big trees can make thinking about current context difficult.
State-based Can Be Cleaner

Diagram:
- Chasing
- Scattering
- In Ghost House
- Dead
- Frightened

Connections:
- Chasing to Scattering
- Chasing to In Ghost House
- Chasing to Dead
- Scattering to Frightened
- In Ghost House to Dead
- In Ghost House to Frightened
- Dead to Frightened
Finite State Machines

- States have no internal structure
- Transition tests are simple
- Often introduced in the context of accepting languages
  - Start state and accept state – here S0 is both
  - This one accepts binary strings that are multiples of 3 (but not all multiples)
Game FSM Example

- States represent potentially complex behavior
- Transitions make non-trivial tests of environment
- No accepting or terminating state
class NPC:
    enum State
        GUARD
        FIGHT
        RUN_AWAY
        INIT
    State myState

    def update():
        if myState == INIT:
            myState = GUARD
        elif myState == GUARD:
            if canSeeEnemy(SMALL): myState = FIGHT
            elif canSeeEnemy(BIG): myState = RUN_AWAY
            else: guard()
        elif myState == FIGHT:
            if health < HEALTH_THRESHHOLD:
                myState = RUN_AWAY
            else: fight()
        elif myState == RUN_AWAY:
            if !canSeeEnemy(): myState = GUARD
            else: run()
Hardcoded FSMs

- Easy to write for small FSMs
- Very efficient
- Notoriously hard to debug and maintain
- No abstraction of FSM architecture from specific FSM
Different Approaches to Abstracting FSMs

Hardcoding via macros to make code more readable

- E.g. a macro language with `BeginStateMachine`, `EndStateMachine`, `State(a)`, `OnEnter()`, ...

State machine compiler

- Takes a clean state machine language and turns it into messy code

OO implementation of abstract state machine logic
Abstracting the Decision Logic

class State:
    def getAction()
    def getEntryAction()
    def getExitActions()
    def getTransitions()

Class Transition:
    def isTriggered()
    def getTargetState()
    def getAction()

class StateMachine:
    states
    initialState
    currentState = initialState

    def update():
        triggeredTransition = None
        for transition in currentState.getTransitions():
            if transition.isTriggered():
                triggeredTransition = transition
                break
        if triggeredTransition != None:
            targetState = triggeredTransition.getTargetState()
            actions = currentState.getExitAction()
            actions += triggeredTransition.getAction()
            actions += targetState.getEntryAction()
            currentState = targetState
            return actions
        else: return currentState.getAction()
Implementing Actions

- FSMs may be called every frame
  - They must return quickly
  - Action execution must not block the FSM

- In a typical game engine with a main game loop, all the action code takes a small incremental action per game tick

- So the AI is returning a discrete decision about which incremental update code to run
Asteroids Action Example

Consider the Approach

Asteroid state

GameObj* asteroid = parent->m_nearestAsteroid;
Ship* ship = parent->m_ship;
Point3f deltaPos = asteroid->m_position – ship->m_position;
deltaPos.Normalize();

// bunch of math to correct deltaPos for current ship velocity and compute angles
If (fabsf(angDelta) < 2 || fabsf(angDelta) > 172) {
    ship->stopTurn();
    if (speed < AI_MAX_SPEED_TRY || parent->m_nearestAsteroidDist > 40)
        fabsf(angDelta)<2? ship->ThrustOn() : ship->ThrustReverse();
}
else if (fabsf(angDelta) <= 90) {
    if (angDelta > 0) ship->TurnRight();
    else ship->TurnLeft();
...
Example of Implementing Sensors

- Similarly to actions, lower-level code will implement the sensor abstractions available for transition testing.

```cpp
m_nearestPowerup = Game.GetClosestGameObj(m_Ship, GameObj::OBJ_POWERUP);
...

m_powerUpNear = false;
if (m_nearestPowerup) {
    m_nearestPowerupDist =
        m_nearestPowerup->m_position.Distance(m_ship->m_position);
    if (m_nearestPowerupDist <= POWERUP_SCAN_DIST) m_powerupNear = true;
}
```

From *AI Game Engine Programming* by Brian Schwab
Working Game Example
Working Game Example

DIY: FSM for Pac-Man Ghost States

- Chasing
- Scattering
- In Ghost House
- Frightened
- Dead

Alex Champandard - http://ai-depot.com/FiniteStateMachines/FSM_Framework.html
DIY: FSM for Pac-Man Ghost States

- **chasing**
  - 7 seconds passed
  - 20 seconds passed
- **scattering**
- **In Ghost House**
  - Resurrected
  - Power Pill
  - Power Pill effect off
- **frightened**
  - Power Pill
- **Dead**
  - Collision with Pac-Man

Alex Champandard - http://ai-depot.com/FiniteStateMachines/FSM-Framework.html
Combining Decision Trees and FSMs

- Duplicate tests on FSM transitions can be expensive

Diagram:
- On Guard
  - Player in sight AND far -> Alarm
  - Player in sight AND near -> Defend
Motivating Hierarchical FSMs

- Search
  - Seen trash
  - Trash disposed

- Head for Trash
  - Got item

- Head for Compactor
Adding Interruptions

Adding interruptions doubles the number of states!
Adding a Second Interruption

Double again!
Hierarchical FSM with One Interruption

- **Clean**
  - **Search**
    - Seen trash ➔ **Head for Trash**
    - Trash disposed ➔ **Head for Compactor**
  - **Head for Trash**
    - Got item ➔ **Clean**
  - **Head for Compactor**
  - **Clean**

- **Recharge**
  - Low power ➔ Recharged ➔ **Clean**
Adding Second Interruption

Clean

Search

Head for Trash

Head for Compactor

Got item

Seen trash

Trash disposed

Low power

All clear

Battle

Recharge

Hide (recharge)

Hide (clean)

All clear

Battle

Recharged

Clean
Cross Hierarchy Decisions

No trash and < 75 power

Clean

Search

Head for Trash

Got item

Head for Compactor

Trash disposed

Seen trash

Add more complex recursive update

Got item

Repository

Clean

Search

Head for Trash

Head for Compactor

Trash disposed

Seen trash

No trash and < 75 power
Implementation Sketch

class HSMBase:
    struct UpdateResult… // holds actions, transition, level
    def getAction() return: []
    def update():
        // fill UpdateResult with action, no trans, level 0

class State (HSMBase):
    def getStates() return [this]
    # the rest same as before

class SubMachineState(State, HSM):
    def getAction: return State::getAction()
    def update(): return HSMF::update()
    def getStates():
        if currentState:
            return [this] + currentState.getStates()
        else:
            return [this]

class HSM (HSMBase):
    def getStates():
        if currentState: return currentState.getStates()
    def update():
        if not currentState:
            currentState = initialState
            return currentState.getEntryAction()
        triggeredTransition = None
        for transition in currentState.getTransitions():
            // search for triggered transition…
            if triggeredTransition:
                // create tran struct and assign to result
            else:
                result = currentState.update()
                if result
                    // fill transition structure with actions
                    return result
        return result
class HSMBase:
    struct UpdateResult... // holds actions, transition, level
    def getAction() return: []
    def update():
        // fill UpdateResult with action, no trans, level 0

class State (HSMBase):
    # stack of return states
    def getStates() return [this]
    # the rest same as before

class SubMachineState(State, HSM):
    def getAction: return State::getAction()
    def update(): return HSFM::update()
    def getStates():
        if currentState:
            return [this] + currentState.getStates()
        else:
            return [this]

class HSM (HSMBase):
    def getStates():
        if currentState: return currentState.getStates()
    def update():
        if not currentState:
            currentState = initialState
            return currentState.getEntryAction()
        triggeredTransition = None
        for transition in currentState.getTransitions():
            // search for triggered transition...
            if triggeredTransition:
                // create tran struct and assign to result
            else:
                result = currentState.update()
                if result
                    // fill transition structure with actions
                    return result