“Inaction breeds doubt and fear. Action breeds confidence and courage. If you want to conquer fear, do not sit home and think about it. Go out and get busy.” -- Dale Carnegie
Hw 1.5

• Create pathnetwork as list of lines between all pathnodes traversable by the agent
  – For all pairs of pathnodes... (possible pathlines)
  – perform a ray trace against every line in worldlines, keeping those without intersections
  – For those pathlines remaining, verify no world point is within agent radius
1. When might you precompute paths?
2. This is a single-source, multi-target shortest path algorithm for arbitrary directed graphs with non-negative weights. Question?
3. This is a all-pairs shortest path algorithm.
4. How can a designer allow static paths in a dynamic environment?
5. When will we typically use heuristic search?
6. What is an admissible heuristic?
7. When/Why might we use hierarchical pathing?
8. Does path smoothing work with hierarchical?
9. How might we combat fog-of-war?
Class N-1

1. Steering vs flocking?
2. Steering Family Tree
3. How might we combine behaviors?
4. What three steering mechanisms enable flocking?
Formations

• Coordinated Movement: M Ch 3.7
• Path plan for leader (naive)
  – All others move toward leader
• Replace team with a virtual bot
  – All members controlled by a joint animation
• Path plan for leader (alt)
  – All team members path plan to an offset
  – Flow around obstacles and through choke points
Fixed Formations

- Line
- Defensive circle
- Two abreast in cover

V or Finger
Four
Decision Making – FSMs

2016-06-02
Decision Making

• Classic AI:
  – making the optimal choice of action (given what is known or is knowable at the time) that maximizes the chance of achieving a goal or receiving a reward (or minimizes penalty/cost)

• Game AI:
  – choosing the right goal/behavior/animation to support the experience

• Decision-making must connect directly to animation so player can see the results of decision-making directly (explainable AI)
  – What animation do I play now?
  – Where should I move?
FSM theory

• A (model of a) device which has
  – a finite number of states \( (S) \)
  – an input vocabulary \( (I) \)
  – a transition function \( T(s,i) \rightarrow s' \)
  – a start state \( \in I \)
  – zero or more final states \( \subset I \)

• Behavior
  – Can only be in one state at a given moment in time
  – Can make transitions from one state to another or to cause an output (action) to take place.
 FSMs in Practice

• Each state represents some desired behavior
• Transition function often resides across states
  – Each state determines subsequent states
• Can poll the world, or respond to events
  (more on this later)
• Support actions that depend on state &
  triggering event (Mealy) as well as entry & exit
  actions associated with states (Moore)
Mealy & Moore

Mealy Output = $F(\text{state}, \text{input})$

Moore Output = $F(\text{state})$

Diagram:

- **State**
  - **Input / Output**
  - **Idle**
    - **P / Yelp**
  - **Flee**
    - **P / Yelp**
  - **Search**
    - **H / Growl**
    - **P / Yelp**

- **State**
  - **Input**
  - **OnEnter / OnExit**
  - **Search**
    - **Growl / Sigh**
  - **Flee**
    - **P**
  - **Idle**
    - **Relief / Startle**
    - **P**
  - **Yelp / Sigh**
FSM as GAI

• Character AI modeled as sequence of mental states
• World events (can) force a change in state
• Mental model easy to grasp (for all)
## State Transition Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Condition</th>
<th>State Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gather Treasure</td>
<td>Monster</td>
<td>Flee</td>
</tr>
<tr>
<td>Flee</td>
<td>Cornered</td>
<td>Fight</td>
</tr>
<tr>
<td>Flee</td>
<td>No Monster</td>
<td>Gather Treasure</td>
</tr>
<tr>
<td>Fight</td>
<td>Monster Dead</td>
<td>Gather Treasure</td>
</tr>
</tbody>
</table>

![State Transition Diagram](diagram.png)
Advantages

• Ubiquitous (not only in digital games)
• Quick and simple to code
• (can be) Easy* to debug
• Fast: Small computational overhead
• Intuitive
• Flexible
Debugging FSM’s

• Offline Debugging
  – Logging
  – Verbosity Levels

• Online Debugging
  – Graphical representation is modified based on AI state
  – Command line to modify AI behavior on the fly.
EXAMPLES
* Usually animations are linked to states, transitions, or both.
Hierarchical FSM Example

- Equivalent to regular FSMs
- Easier to think about encapsulation
FSM: Quake dog monster

http://ai-depot.com/FiniteStateMachines/FSM-Framework.html
FSM Examples

- Red: Shadow, blinky
  - “pursuer” or “chaser”
- Pink: Speedy, pinky
  - “ambusher”
- Blue: Bashful, inky
  - “whimsical”
- Orange: Pokey, Clyde
  - “feigning ignorance”

http://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior
FSM Examples

• Pac-Man
• FPSs
FSM Examples

- Pac-Man
- FPSs
- Sports Simulations
FSM Examples

- Pac-Man
- FPSs
- Sports Simulations
- RTSs
UnrealScript Example

UnrealScript Example

Diagram showing state transitions for a vehicle's drivetrain.

- **off**
  - Turn key on
  - Turn key off
  - Stall

- **on**
  - Turn key on
  - Turn key off

- **idle**
  - Stall
  - Shift up
  - Shift down

- **1st**
  - Shift up
  - Shift down

- **2nd**
  - Shift up
  - Shift down

- **reverse**
  - Shift up
  - Shift down
public void runStateMachine (Event e) {
    switch (state) {
        case 0:
            if (e.isTurnOn()) { power=true; state=1;}
            break;
        case 1:
            if (e.isTurnOn()) { startEngine(); state=2;}
            else if (e.isTurnOff()) { power=false; state=0;}
            break;
        case 2:
            makeEngineSound();
            if (e.isUpShift()) { gear=1; state=3;}
            else if (e.isDownShift()) { gear=-1; state=9;}
            else if (e.isTurnOff()) { stopEngine(); state=1;}
            break;
        ...
    }
}
FSM IMPLEMENTATIONS
Impl: Centralized Conditionals

- Simplest method
- After an action, the state might change.
- Requires a recompile for changes (hard-coded)
- No pluggable AI
- Not accessible to non-programmers
- No set structure
- Can be a bottleneck.

```c
void RunLogic( int *state ) {
    switch( *state ) {
        case 0: //Wander
            Wander();
            if( SeeEnemy() )
                *state = 1;
            if( Dead() )
                *state = 2;
            break;
        case 1: //Attack
            Attack();
            *state = 0;
            if( Dead() )
                *state = 2;
            break;
        case 3: //Dead
            SlowlyRot();
            break;
    }
}
```
... in Game Loop (w/ enum)

```java
public enum State {STATE1, STATE2, STATE3};
State state = State.STATE1;
void tick ()
{
    switch (state) {
        case STATE1:
            PlayAnimation(...);
            if (...) state = newstate;
            else if (...) state = newstate;
            else if ...
            else ...
        case STATE2:
            PlayAnimation(...);
            if (...) state = newstate;
            else if ...
            else if ...
            else ...
    }
}
```
Implementation: Macros

... BeginStateMachine
  State(WANDER)
    Begin:
      Wander();
      if (SeeEnemy()) GotoState(ATTACK);
      if (Incapacitated()) GotoState(INCAPACITATED);
  State(INCAPACITATED)
    Begin:
      ...
      Moan:
        PlaySound(moan);
        goto 'Moan';
EndStateMachine
## Impl: State Transition Tables

<table>
<thead>
<tr>
<th>Current State</th>
<th>Condition</th>
<th>State Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RunAway</td>
<td>Safe</td>
<td>Patrol</td>
</tr>
<tr>
<td>Attack</td>
<td>WeakerThanEnemy</td>
<td>RunAway</td>
</tr>
<tr>
<td>Patrol</td>
<td>Threatened &amp;&amp; StrongerThanEnemy</td>
<td>Attack</td>
</tr>
<tr>
<td>Patrol</td>
<td>Threatened &amp;&amp; WeakerThanEnemy</td>
<td>RunAway</td>
</tr>
</tbody>
</table>

If Kitty_Hungry AND NOT Kitty_Playful SWITCH_CARTRIDGE eat_fish
### Impl: Tables Alt

<table>
<thead>
<tr>
<th>State</th>
<th>Event</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td></td>
<td>----</td>
<td>A1/S2</td>
<td>A3/S1</td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

S: state, E: event, A: action, ----: illegal transition
## Implementation: Virtual FSM

<table>
<thead>
<tr>
<th>State Name</th>
<th>Conditions</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current state name</td>
<td>Entry</td>
<td>Outputs...</td>
</tr>
<tr>
<td></td>
<td>Exit</td>
<td>Outputs...</td>
</tr>
<tr>
<td></td>
<td>Condition 1...</td>
<td>Outputs...</td>
</tr>
<tr>
<td></td>
<td>Condition 2...</td>
<td>Outputs...</td>
</tr>
<tr>
<td>Next state name</td>
<td>Condition X</td>
<td>Outputs...</td>
</tr>
<tr>
<td>Next state name</td>
<td>Condition Y</td>
<td>Outputs...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

## Implementation: Virtual FSM

<table>
<thead>
<tr>
<th>State Name</th>
<th>Conditions</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patrol</td>
<td>Entry</td>
<td>SwingKeys()</td>
</tr>
<tr>
<td></td>
<td>Exit</td>
<td>DropClipboard()</td>
</tr>
<tr>
<td></td>
<td>Happy()</td>
<td>Whistle()</td>
</tr>
<tr>
<td></td>
<td>NearDog()</td>
<td>PetDog()</td>
</tr>
<tr>
<td>Flee</td>
<td>Overwhelmed()</td>
<td>Scream()</td>
</tr>
<tr>
<td>Attack</td>
<td>!Overwhelmed()</td>
<td>TakeOutGun()</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Impl: Distributed

- Rules for transition contained within state
- Good encapsulation
- Can swap in/out states easier
- AKA
  - “State Design Pattern” (Buckland italics)
  - “Embedded rules” (Buckland subheading)

Eat_fish cartridge knows when to switch to Use_litterbox
## Impl: Distributed

```java
interface Entity {
    void update (); // Where “thinking” happens.

    //void changeState (State newstate);
}

interface State {
    void execute (Entity thing);

    void onEnter (Entity thing);

    void onExit (Entity thing);
}
```
class Troll implements Entity
{
    int liveTime=0;
    State currentState, previousState;
    @Override
    void update () {
        liveTime++;
        currentState.execute( this );
    }
    //@Override
    void changeState (State newstate) {
        previousState = currentState;
        currentState.onExit( this );
        currentState = newstate;
        currentState.onEnter( this );
    }
}

Class CoolState implements State
{
    @Override
    void execute (Entity thing) {}
    void execute (Troll thing) {
        if ( thing.liveTime = 0 ) {
            thing.playAnimation(ani1);
            thing.changeState(new st);
        }
        else thing.doSomething();
    }
    @Override
    void onEnter (Entity thing) {...}
    @Override
    void onExit (Entity thing) {...}
}
class StateMachine //implements Entity?
{  State currSt, prevSt, globalSt;
    Entity owner;

    StateMachine( Entity e ){ owner = e; }

    void update () {
        if( globalSt != null)
            globalSt.execute( owner);
        currentState.execute( owner);
    }

    void changeState (State newstate) {
        previousState = currentState;
        currentState = newstate;
        currentState.onEnter( owner );
    }

    void revertToPrev(){ changeState( prevSt ); }

    boolean isInState( State st ) { ...}
}

class Troll implements Entity
{  StateMachine fsm;
    Troll(){
        fsm = new StateMachine( this );
        fsm.setGlobalState(
            TrollGlobalState.singleton() );
        fsm.setLocalState(
            TrollSleepInCave.singleton() );
    }

    void update(){
        liveTime++;
        fsm.update();
    }

    StateMachine getFSM(){ return fsm; }
}
class StateMachine:
    states #list of states
    initST
    curST = initST

def update():
    triggeredT = None
    for t in curST.transitions():
        if t.isTriggered():
            triggeredT = t
            break

if triggeredT:
    targetST = triggeredT.getTargetState()
    actions = curST.getExitAction()
    actions += triggeredT.getAction()
    actions += targetST.getEntryAction()
    curST = targetST
    return actions
else: return curST.getAction()
Global States

• May have multiple states that could happen at any time
• Want to avoid authoring many transitions from every other state to these
• Create a global state that is called every update cycle
• State “blips” (return to previous after global)
FSM Extensions

• Extending States
  – Adding onEnter() and onExit() states can help handle state changes gracefully.

• Stack Based FSM’s
  – Allows an AI to switch states, then return to a previous state.
  – Gives the AI ‘memory’
  – More realistic behavior
  – Subtype: Hierarchical FSM’s
Motivating FSM Stacks

• Original version doesn’t remember what the previous state was.
• One solution is to add another state to remember if you heard a sound before attacking.

E: Enemy in sight; S: hear a sound; D: dead
Motivating FSM Stacks (2)

Worst case:
Each extra state variable can add $2^n$ extra states
$n = \text{number of existing states}$

Using a stack would allow much of this behavior without the extra states.

E: Enemy in sight; S: hear a sound; D: dead
Stack FSM – Thief 3

Stack allows AI to move back and forth between states.

Leads to more realistic behavior without increasing FSM complexity.
Hierarchical FSMs

• Expand a state into its own sub-FSM
• Some events move you around the same level in the hierarchy, some move you up a level
• When entering a state, have to choose a state for it’s child in the hierarchy
  – Set a default, and always go to that
  – Random choice
  – Depends on the nature of the behavior
Hierarchical FSM Example

E: Enemy in sight; S: hear a sound; D: dead
Non-Deterministic Hierarchical FSM

Diagram:
- Attack
  - Approach
  - Start
  - Go through Door
  - Aim, Slide left, and Shoot
  - Aim, Slide right, and Shoot
- Wander
- Start
Hierarchical FSMs in
*Destroy All Humans 2*

http://www.gamasutra.com/view/feature/130279/creating_all_humans_a_datadriven_.php
Hierarchical FSMs in *Destroy All Humans 2*

- Active (blue), pending (orange)
- Only active behaviors update
- Only active behaviors have children
- If *children* startable, rank
- States can be marked as non-interruptable or non-blocking
Hierarchical FSMs in *Destroy All Humans 2*

- Self-contained behaviors
  - When to activate
  - What activates it, interrupts it
  - What to do on start, exit
  - What children it starts

- Code-supported behaviors exist for complex, non-generalizable cases
Hierarchical FSMs in *Destroy All Humans 2*

**Figure 3:** Representations of three behaviors used in the protect HFSM are shown.

- **Protect behavior**
  - Accepts parameter: protect target
  - Assign parameter to hostile target
  - Override behavior alias:
    - Mapalas ("taunt", "protect_taunt")

- **Approach behavior**
  - Accepts parameter: target
  - Activate movement system:
    - Move to position of parameter "target"
  - Precondition: target farther than 20 meters

- **Combat behavior**
  - Accepts parameter: target
  - Precondition: target closer than 40 meters
  - Start child prioritized: taunt
  - Start child prioritized: melee
  - Start child prioritized: fireweapon
  - Start child prioritized: patrol around target
  - Pass parameter: protect target

- **Start child prioritized**:
  - Approach
  - Pass parameter: protect target
More FSM Extensions

• Fuzzy State Machines
  – Degrees of truth allow multiple FSM’s to contribute to character actions.

• Multiple FSM’s
  – High level FSM coordinates several smaller FSM’s.

• Polymorphic FSM’s
  – Allows common behavior to be shared.
  – Soldier -> German -> Machine Gunner
Polymorphic FSMs

• Small changes to low level behaviors may be needed for different types of entities
• Polymorphism allows multiple versions of a single FSM to be executed on NPC state
Polymorphic FSM Example

- Soldier
  - Rifleman
    - American
    - German
    - British
    - Soviet
  - Machine Gunner
    - American
    - German
    - British
    - Soviet
  - Officer
    - American
    - German
    - British
    - Soviet
Impl: Data Driven

- Developer creates scripting language to control AI.
- Script is translated to C++ or bytecode.
- Requires a vocabulary for interacting with the game engine.
- A ‘glue layer’ must connect scripting vocabulary to game engine internals.
- Allows pluggable AI modules, even after the game has been released.
Scripted AI

• Many game engines are virtual machines
• Script is a program written in a programming language that makes calls into the game engine
• AI is the script
• Examples: Lua, Ruby, UnrealScript
• Powerful when paired with trigger systems
Game Engine Interfacing

- **Simple hard coded approach**
  - Allows arbitrary parameterization
  - Requires full recompile

- **Function pointers**
  - Pointers are stored in a singleton or global
  - Implementation in DLL
    * Allows for pluggable AI.

- **Data Driven**
  - An interface must provide glue from engine to script engine.
Processing Paradigms

• Polling
  – Simple and easy to debug.
  – Inefficient since FSM’s are always evaluated.

• Event Driven Model
  – FSM registers which events it is interested in.
  – Requires Observer model in engine.
  – Hard to balance granularity of event model.

• Multithreaded
  – Each FSM assigned its own thread.
  – Requires thread-safe communication.
  – Conceptually elegant.
  – Difficult to debug.
  – Can be made more efficient using microthreads.
Single-threaded execution
Multi-threaded execution
Messaging/Triggers vs Polling

• Well-designed games tend to be event driven
• Examples (broadcast to relevant objs)
  – Wizard throws fireball at orc
  – Football player passes to teammate
  – Character lights a match (delayed dispatch match)
• Events / callbacks, publish / subscribe, Observers (GoF)
  – See Buckland Ch 2: Adding Messaging (pp69)
Time Management

• Helps manage time spent in processing FSM’s.

• Scheduled Processing
  – Assigns a priority that decides how often that particular FSM is evaluated.
  – Results in uneven (unpredictable) CPU usage by the AI subsystem.
    • Can be mitigated using a load balancing algorithm.

• Time Bounded
  – Places a hard time bound on CPU usage.
  – More complex: interruptible FSM’s
FSM Pros and Cons

• Advantages:
  – Very fast – One array access
  – Can be compiled into compact data structure
    • Dynamic memory: Current state
    • Static memory: State diagram – Array implementation
  – Can create tools so non-programmer can build behavior
  – Non-deterministic FSM can make behavior unpredictable

• Disadvantages:
  – Number of states can grow very fast
    • Exponentially with number of events: $s=2^e$
  – Number of arcs can grow even faster: $a=s^2$
  – Hard to encode complex memories or sequences of action
  – Propositional representation
    • Difficult to put in “pick up the better weapon,” attack the closest enemy
References / See Also

- AI Game Programming Wisdom 2
- Web
  - http://ai-depot.com/FiniteStateMachines
- Buckland Ch 2
- Millington Ch 5
- Jarret Raim’s slides (Dr. Munoz-Avila’s GAI class 2005)
  - http://www.cse.lehigh.edu/~munoz/CSE497/classes/FSM_In_Games.ppt
- Mark Riedl, Brian O’Neill, and Brian Magerko
Trajectory Update

• Start next homework, ASAP!
• To come: More decision making
  – Planning
  – Decision trees
  – Behavior trees
  – Rule based systems
  – Fuzzy Logic
  – Markov Systems