Artificial Intelligence and Computer Games

John Laird and Sugih Jamin

EECS 494

University of Michigan

Also based on talks by Doug Church and Lars Lidén

What is AI?

- The term AI is broadly used in computer games
  - Behave rationally: Use available knowledge to maximize goal achievement.
    - Often leads to optimization techniques.
  - A set of capabilities: Problem solving, learning, planning, ...
- Different game genre employs different techniques

Roles of AI in Games

- Opponents
- Teammates
- Strategic Opponents
- Support Characters
- Autonomous Characters
- Commentators
- Camera Control
- Plot and Story Guides/Directors

AI Provides

- Character, Emotion
- Understanding Environments
- Solving Logic, Resolving Rules
- Decision making, w/Attitude Bias
- Not yet “virtual people”, as such

AI Roles in Games

Game Genres	**AI Roles**
Racing	Other cars
Fighting	Other fighters
Action/FPS	Enemies, allies
RPG	Monsters, party members
Adventure	
Strategy	Strategic Enemies
Simulation	Units
Sports	Commentators, camera

Obvious Examples

Situations where “AI” might be, could be, should be, and is used in Games

- Car Game – write a virtual driver
- Shooter – write a virtual player
- Sports Games – write a virtual coach
- RTS – write a virtual general

Racing Opponents

- Originally follow course “on rails”
- Next allow different speeds in curves, hills, …
- Finally, control vehicle using game physics
  - Use human play traces
  - Provide variety and skill levels with different humans
  - Transition between trace following and respond to dynamics
    - Powerups, human player, …
- Attempt to have driver “personality”
- What makes for a “fun” racing game?

Making a "fun" racing game

- As designers, we want to recreate racing, not just driving around on a track
- Competition is a crucial part of that
- Need to increase likelihood of a close race
- So we could count on players getting good or, essentially, we could cheat

How do we cheat well?

- We have to slow down the front, speed up the back
- Easiest way is just with speed
  - Cars in front slow down, in back, speed up
- Rubber banding near player makes it challenging
- However, this can be very obvious to players
  - Violate “fairness” and “consistency”
- And, worse, risks removing player’s feel of interaction

Dynamic Difficulty Adjustment

- Game monitors player behavior
- As player struggles, game changes to try and help the player through it
- If player does well, game becomes harder
- Examples?

Risks of DDA approaches

- It seems obvious adaptive models are better for tuning an experience
- However, if a player realizes they are involved, they can exploit them
- Slowing down until the end of the race, for instance

Players abuse the rules

Players learn to win at the provided rule-system, not the ideas in your head

- They don’t learn the manual
- They don’t play what you thought was cool
  - If the way to “win” is to fight, you can say “hide” all you want, but they will fight
- They don’t only do “reasonable” things
- They poke and prod the systems, and exploit any weaknesses they can find
  - If there are bugs in the rules, they will find and exploit them, even if they enjoy it less

Somewhat Real Examples

- Car Game – write AI to keep races close
- Shooter – enemies die lots, win little
- Sports – commentators, help player
- RTS – generals who work on pacing
- It is A Question of Design Purpose

Commentator Examples

- Excitement, plus reason for play result
- Finite range of possible utterances
  - “decision quality” is often less important than the “media asset quality”
- Better to be silent than stupid
- Correct isn’t good enough
  - [take a knee != loss of 2]

Requirements for Game AI

- What is the goal of the game?
  - Focus on the Player Experience
- How is the AI going to further that goal?
  - Needs to achieve design aim (and be fun)
  - Foil for the player, creates opportunity
- Dynamic challenge
- Assists in Driving the action
- Allow player to understand AI actions
- Configurable, Override-able, Testable
  - Reproducibility is vital, for test and design
- Satisfies data and speed constraints

The “Thief” AI

Design Goals

- Player is going to be a Thief
- I.e. Sneak Around, Ambush, Hide, Steal
  - AI must allow players to make plans
  - And react to player actions, provide challenge
- Game will feature a loose overall story
  - Ability to script/override behavior
  - In game actions fed back out to story control

“Watch-able” by the player

- Has to “go about its business” with intent
- Actions must make sense to player
  - “interestingly predictable”
  - present play opportunities for player
- Overemphasize thoughts
  - Telegraph all actions
  - Goals must be very explicit

Artificial Stupidity

- Intelligence != Fun
  - What makes a game entertaining and fun does not necessarily correspond to making characters smarter
  - Must be fun, not correct
- The player is, after all, supposed to win
- Lars Liden’s 11 Ways to be stupid

1. Don’t cheat

- AI should not be omniscient:
  - Knows where enemies are without seeing them
  - Know where to find resources, weapons, ammo
- Players can detect cheating and will find the game unfair

2. Don’t kill on first attempt

- It’s not fun to suddenly and unexpectedly take damage
- Player may feel cheated, particularly if attacked with a weapon that kills the player or does a lot of damage
- By missing the player the first time, it gives the player a second to react and still keeps the tension high

3. Have horrible aim

- Having abundant gun fire in the air keeps the player on the move and the tension high
- However, the player is supposed to win
- By giving NPC bad aim, one can have abundant gun fire without being too hard on the player
- "Half-Life" used a wide spread on NPC weapons (as much at 40 degrees)

4. Never shoot when first see the player

- When a player first walks into an area and is spotted by an enemy, the enemy should never attack right away
- A secondary activity, such as running for cover or finding a good shooting location is more desirable
- Gives player time to react

5. Warn the Player

- Before attacking the player, warn the player that you are about to do so
  - Make a sound (beep/click)
  - Play a quick animation
  - Say "Gotcha!", "Take this"
- This is particularly important when attacking from behind

6. Attack "kung-fu" style

- Player is usually playing the role of "Rambo" (i.e. one man taking on an army)
- Although many NPCs may be in a position to attack the player, only a couple should do so at a time
- The remaining NPCs should look busy, reloading, changing positions, etc.

7. Tell the player what you are doing

- Interpreting the actions of AIs can often be subtle
- Complex behaviors are often missed by the player. (Lot’s of work for nothing)
- AIs should tell the player what they are doing
  - “flanking!” “cover me!” “retreat!”
- Players will often intuit intelligence behavior that isn’t really there

8. Intentionally be vulnerable

- Players learn to capitalize on opponent's weaknesses
- Rather than allowing the player to discover unintentional weaknesses in the AI, vulnerability should be designed into an AI's behavior
  - Stop moving before attacking
  - Pause and prepare weapon before attacking
  - Act surprised and slow to react when attacked from behind
- Planned vulnerability makes the characters seem more realistic
- Unintentional mistakes break the realism (seems like fighting a computer program)

9. Don’t be perfect

- Human players make mistakes
- When AIs behave perfectly they seem unnatural
- If an AI knows how to avoid trip mines, run into then occasionally
- When reloading, sometimes fumble with the gun

10. Pull back last minute

Trick:

- Push the player to the limit
- Attack vigorously until the player is near death
- Then pull back. Enemy becomes easier to kill
- Makes player feel like they really accomplished something

11. React To Mistakes

- Mistakes in AI are inevitable
- Unhandled, they make make the AI look dumb
- By recognizing mistakes and reacting to them intelligently they can be turned into features

11. React To Mistakes

- Example 1:
  - Occasionally when an NPC throws a grenade, it bounces off of another object and lands back at the NPCs feet
    - (Note that the player occasionally makes this mistake too!)
  - Looks dumb as the NPC blows himself up
  - If the NPC reacts, however, the mistake turns into a feature:
    - NPC body and facial expression can show surprise, fear
    - NPC can say “Oh Shoot!” or “Doh!”

11. React To Mistakes

- Example 2:
  - Player throws a grenade at a group of NPCs. As they are crowded together not all of them are able to find a path to get away
  - Looks dumb if the NPCs that can't get away, shuffle around trying to get out
  - If we detect that the problem has arisen, can have the trapped NPC's react
    - Crouch down and put hands over head

Themes

- Player Player Player Player Player Player
- How can AI enhance player experience
- AI is facilitator of the “fun”
- Enable creative expression for player
  - Allow player to impact the world
  - Put player in interesting situations
- Entertaining game != “smarter” opponents
  - Machine opponents are babysitters, not ruthless opponents
  - Players aren’t pro players, or pro strategists
  - Give player ways to make the big play
- The illusion of intelligence is far more important than actual intelligence
  - Predictable often more important than smart
  - Clever AI decisions are no better than secret special knowledge if player can’t tell

Observations

- AI has three basic game roles
  - Replacement for human opponents and players
  - Support characters for interesting player interaction
  - Units for player management

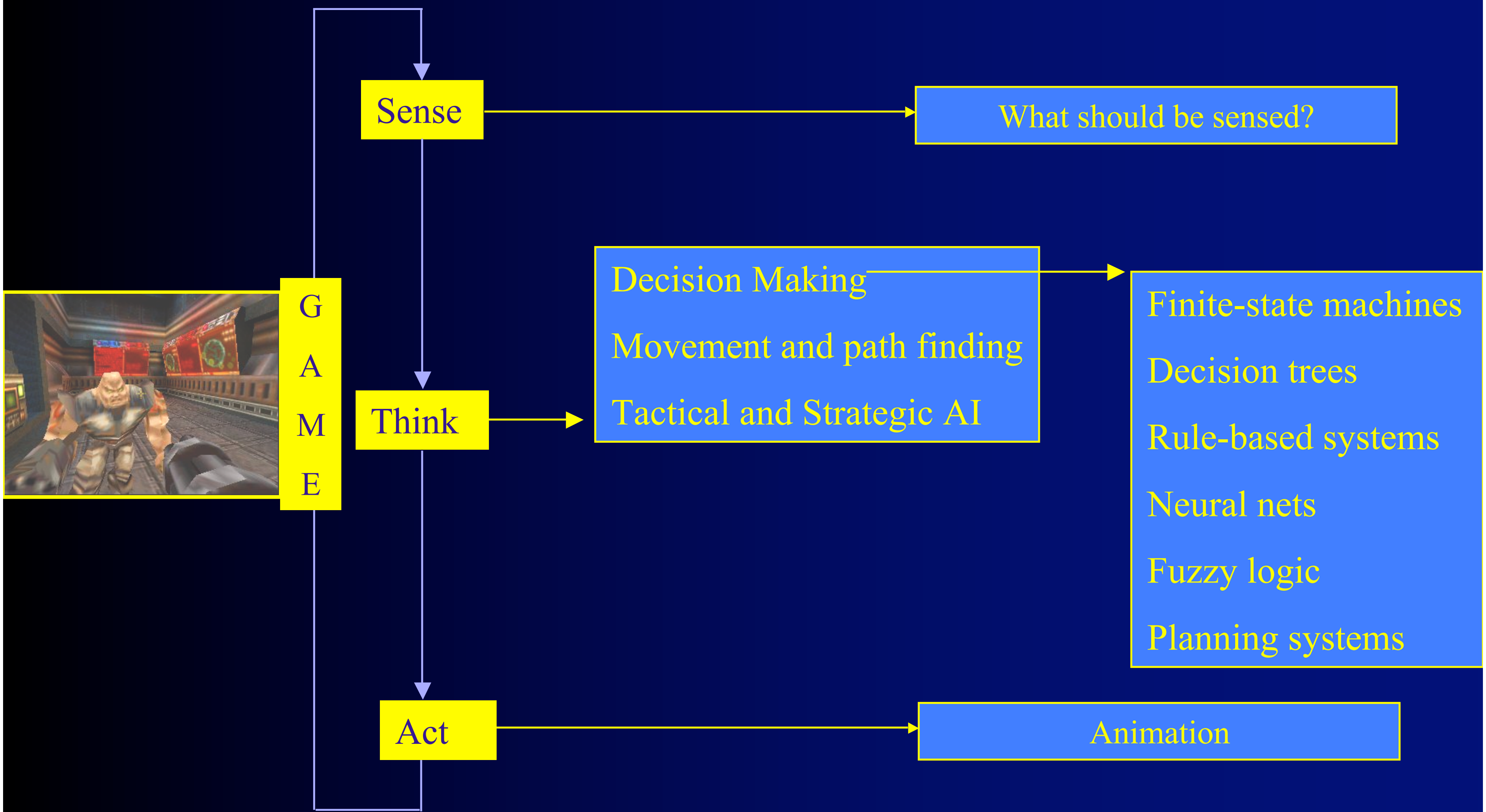
- Entertainment is much more important than realism
  - Cheating is ok if user can't detect it
  - Play to lose or at least make it challenging
  - Must include variable levels of skills

- No single type of AI is right for all games or all AI roles

AI Agent in a Game

- Each time through control loop, “tick” each agent
  - Sometimes only 1/N times through loop
  - More frequently if in view of player
- Define an API for agents: sensing and acting
- Encapsulate all agent data structures
  - And so agents can’t trash each other or the game
  - Share global data structures on maps, etc.

Execution Flow of an AI Engine

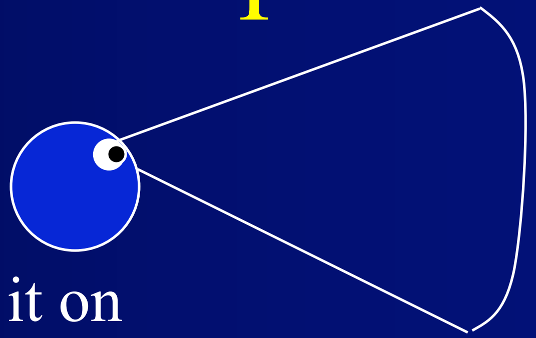


Structure of an Intelligent Agent

- Sensing: perceive features of the environment
- Thinking: decide what action to take to achieve its goals, given the current situation and its knowledge
- Acting: doing things in the world

- Thinking has to make up for limitations in sensing and acting
- The more accurate the models of sensing and acting, the more realistic the behavior

Sensing Limitations & Complexities

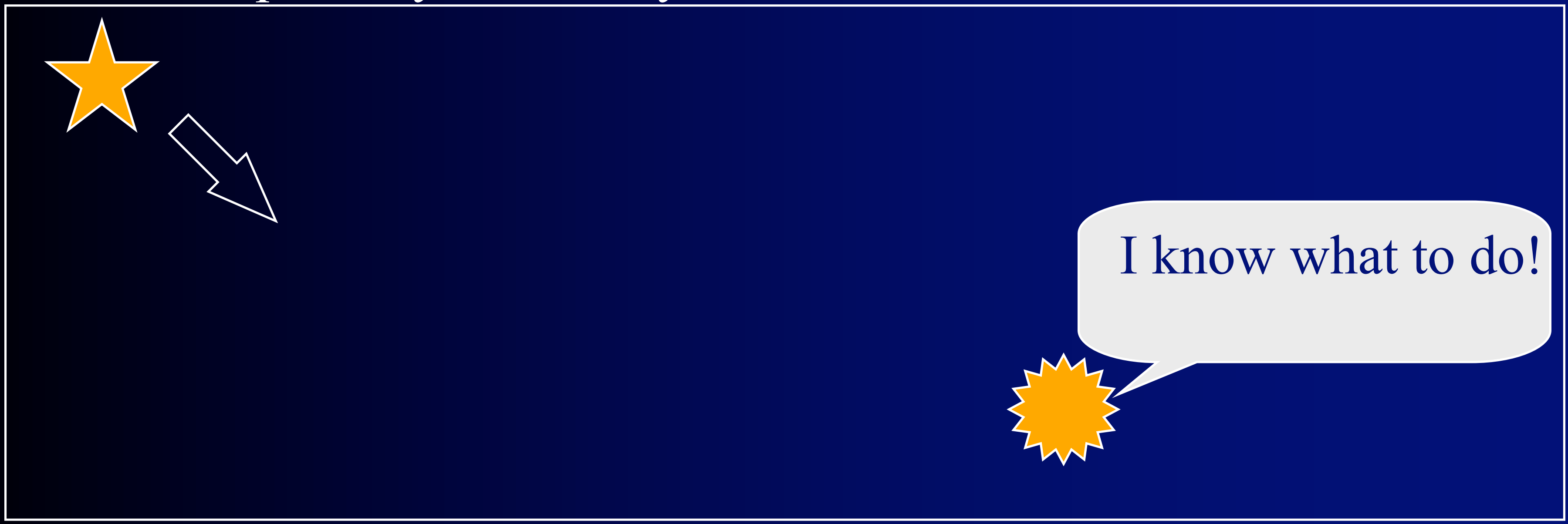


- Limited sensor distance
- Limited field of view:
  - Must point sensor at location and keep it on
- Obstacles
- Complex room structures
  - Detecting and computing paths to doors
- Different sensors give different information and have different limitations
  - Sound: omni-directional, gives direction, distances, speech, ...
  - Vision: limited field of view, 2 1/2D, color, texture, motion, ...
  - Smell: omni-directional, chemical makeup

  ➪ Need to integrate different sources to build complete picture.

Perfect Agent: Unrealistic

- Sensing: Have perfect information of opponent
- Thinking: Have enough time to do any calculation
  - Know everything relevant about the world
- Action: Flawless, limitless action
  - Teleport anywhere, anytime



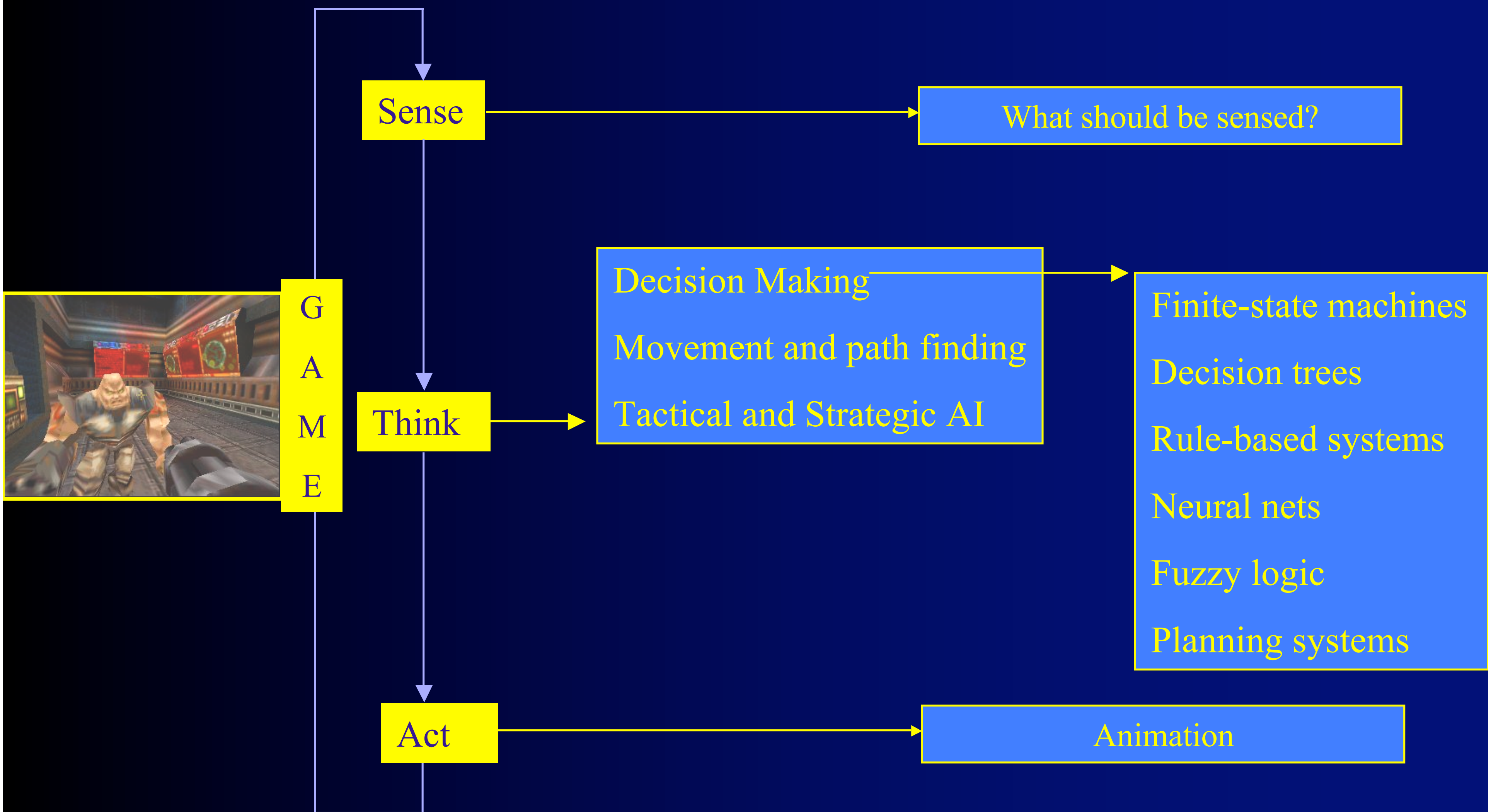
Conflicting Goals for AI in Games

- Goal Driven
- Reactive
- Knowledge Intensive
- Human Characteristics
- Low CPU & Memory Usage
- Fast & Easy Development

Complexity

- Complexity of Execution
  - How fast does it run as more knowledge is added?
  - How much memory is required as more knowledge is added?
- Complexity of Specification
  - How hard is it to write the code?
  - As more "knowledge" is added, how much more code needs to be added?
- Memory of prior events
  - Can it remember prior events?
  - For how long?
  - How does it forget?

Execution Flow of an AI Engine

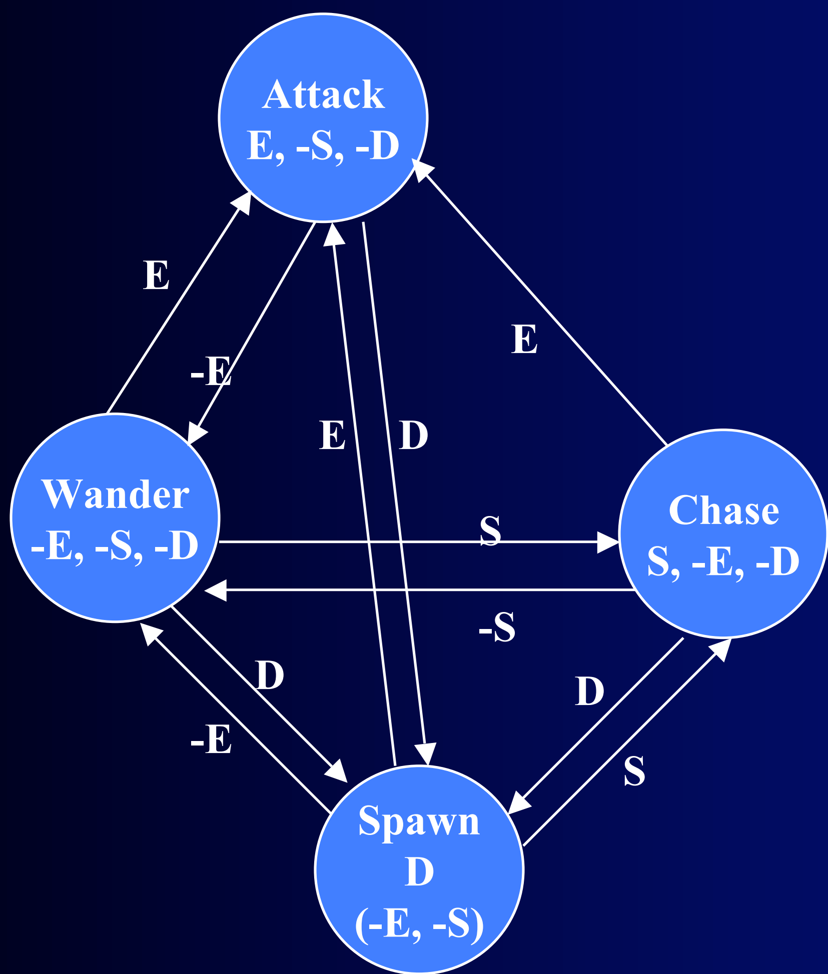


Types of Behavior to Capture

- Wander randomly if don't see or hear an enemy
- When see enemy, attack
- When hear an enemy, chase enemy
- When die, respawn
- When health is low and see an enemy, retreat

- Extensions:
  - When see power-ups during wandering, collect them

Finite State Machine



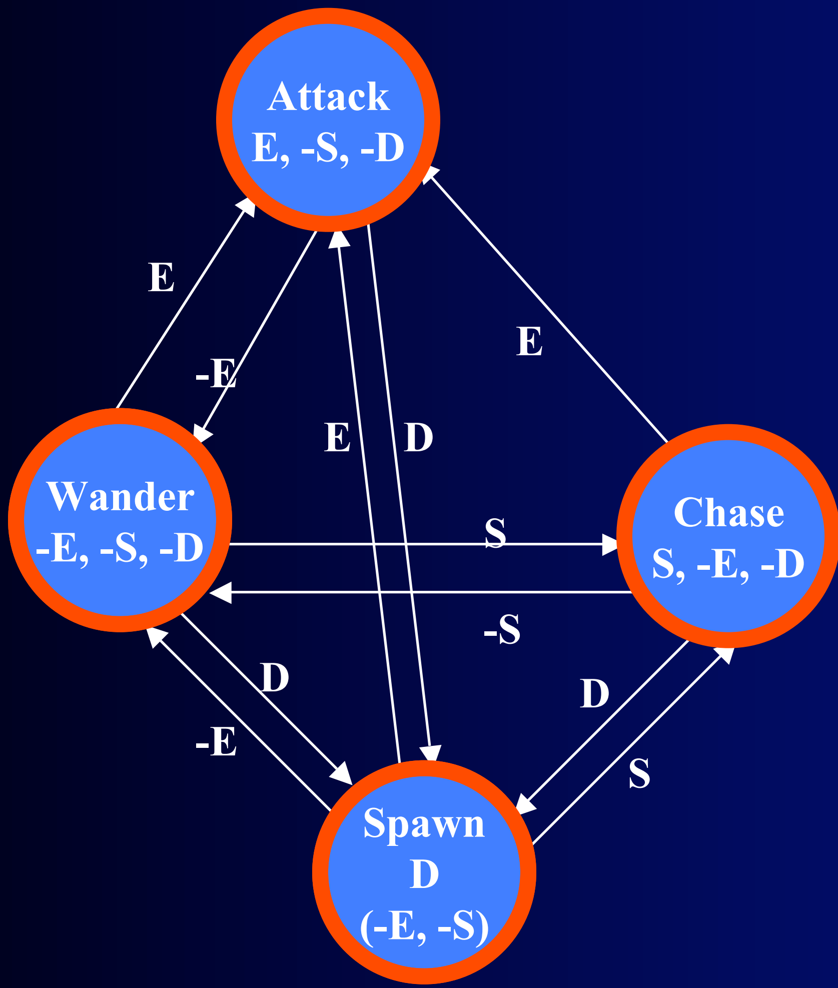
Events:

E=Enemy Seen

S=Sound Heard

D=Die

Example FSM



Events:

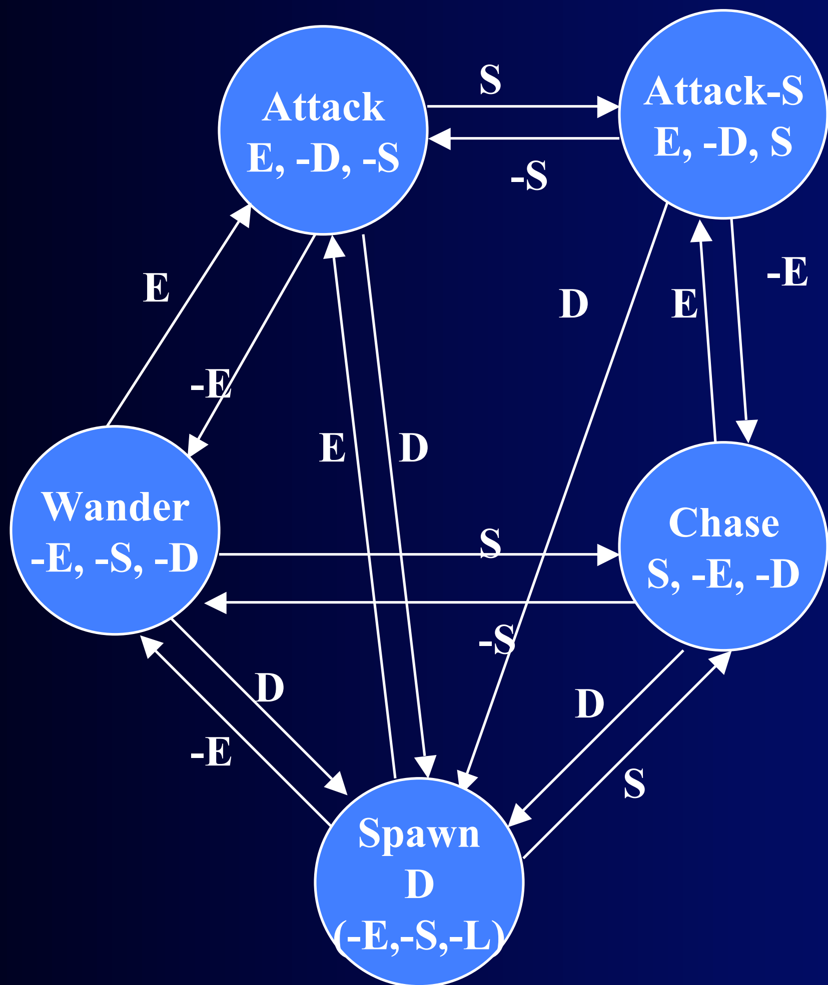
E=Enemy Seen

S=Sound Heard

D=Die

Problem: No transition from attack to chase

Example FSM - Better



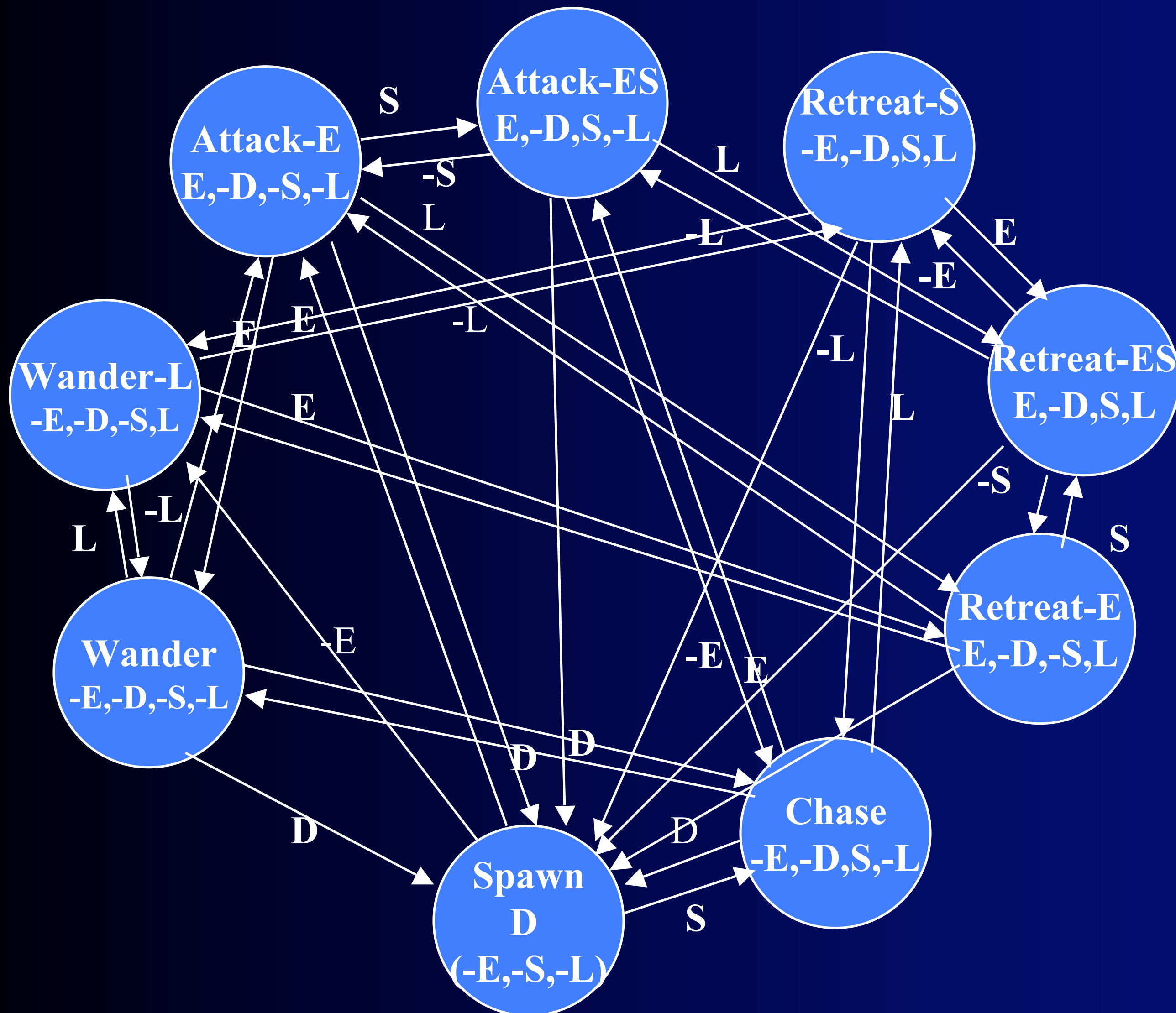
Events:

E=Enemy Seen

S=Sound Heard

D=Die

Example FSM with Retreat

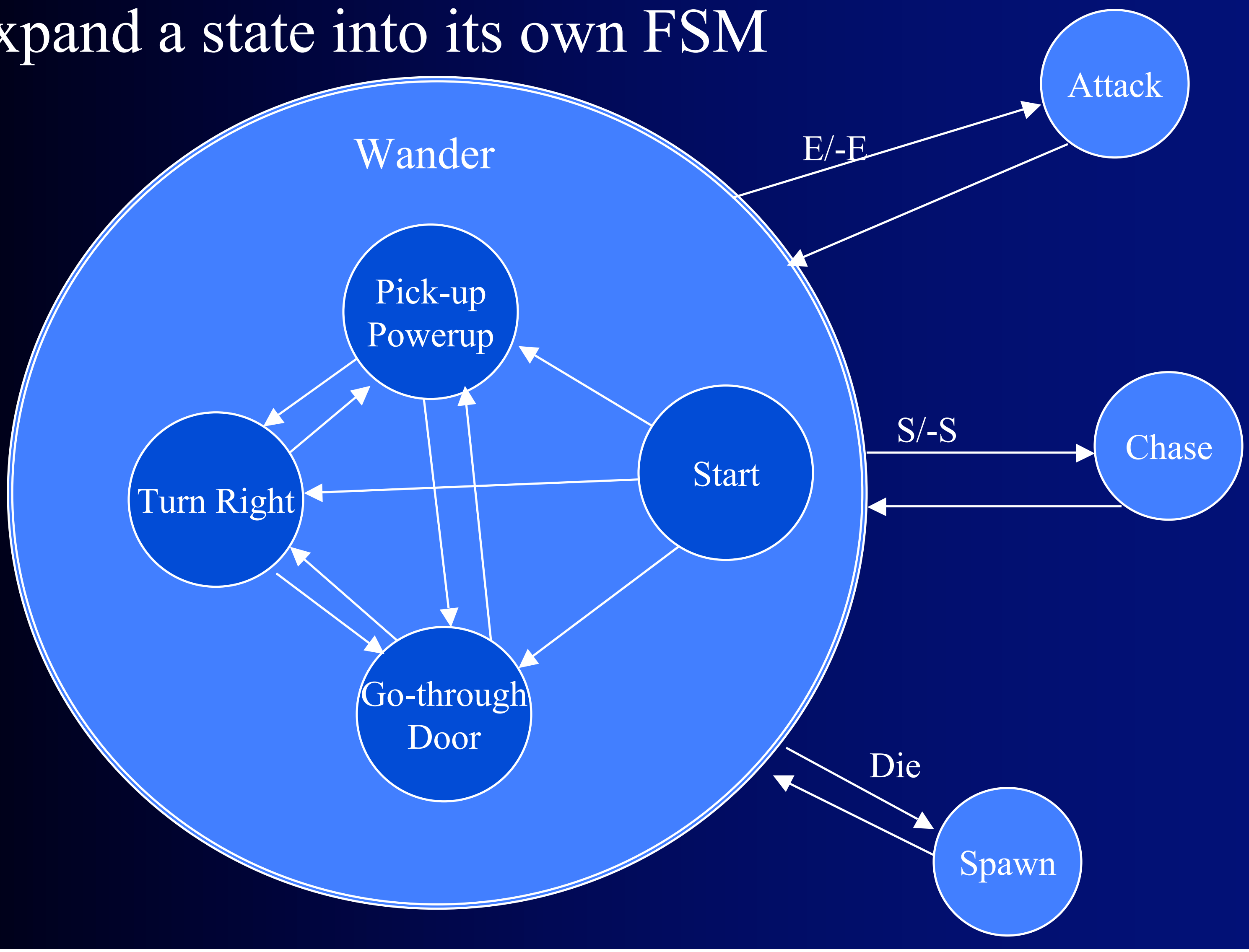


Events:
E=Enemy Seen
S=Sound Heard
D=Die
L=Low Health

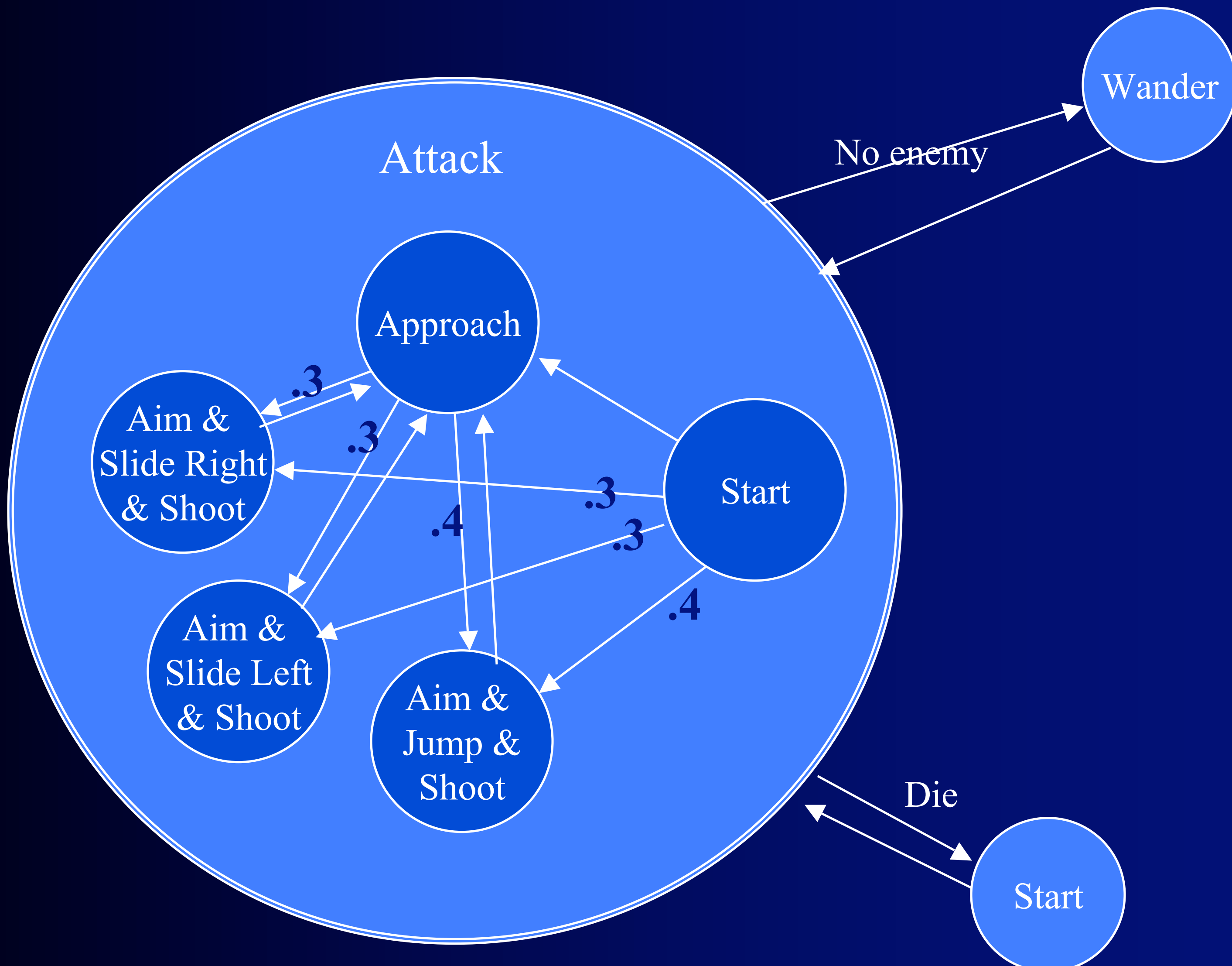
Each feature with N values can require N times as many states

Hierarchical FSM

- Expand a state into its own FSM



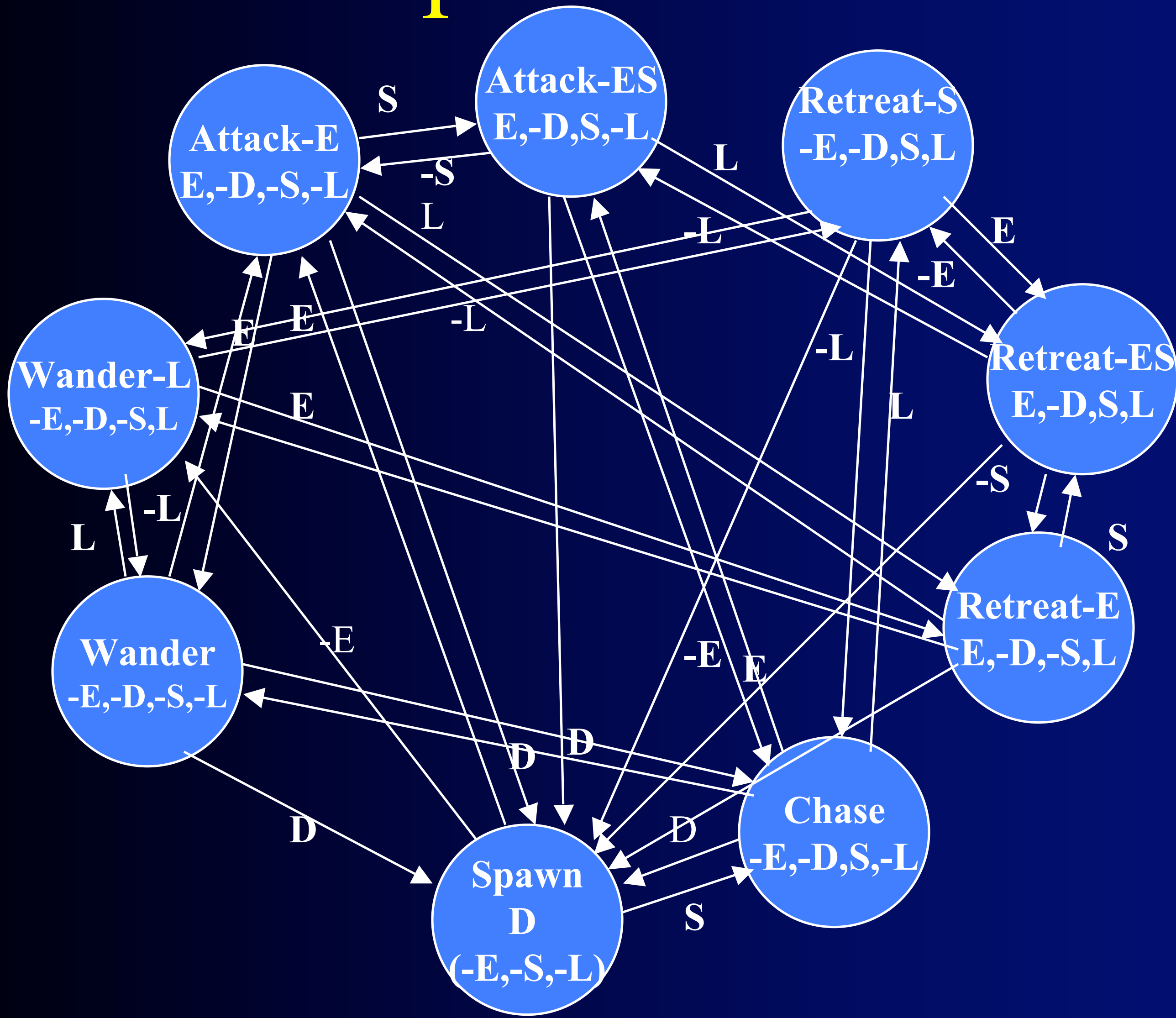
Non-Deterministic Hierarchical FSM (Markov Model)



Decision Trees

- Tree nodes represent attribute tests
  - One child for each possible value of the attribute
- Leaves represent classifications
- Classify by descending from root to a leaf
  - At root test attribute associated with root attribute test
  - Descend the branch corresponding to the instance's value
  - Repeat for subtree rooted at the new node
  - When a leaf is reached return the classification of that leaf

Example FSM with Retreat



Events:

E=Enemy

S=Sound

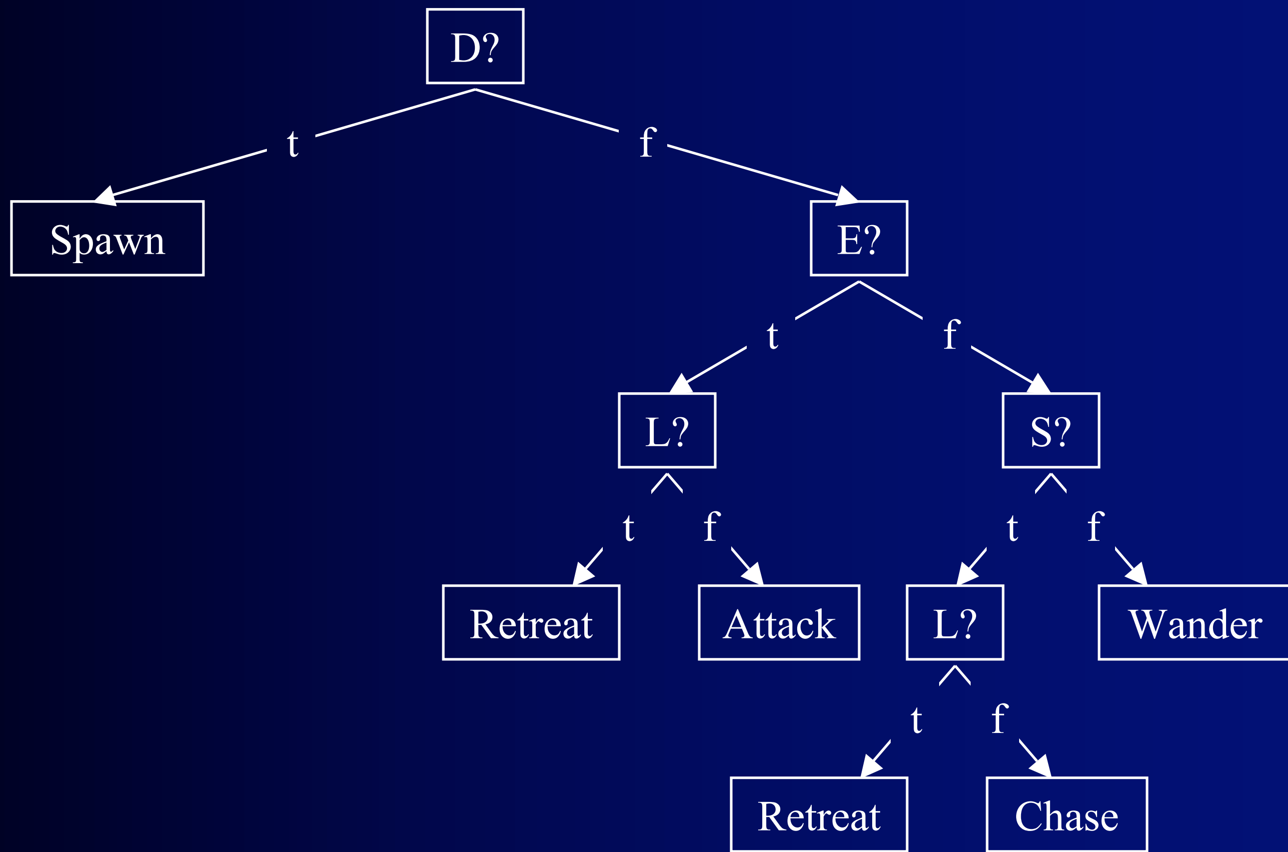
D=Die

L=Low Health

Each new feature can double number of states

Decision Tree for Quake

- Input Sensors: E=<t,f> L=<t,f> S=<t,f> D=<t,f>
- Categories (actions): Attack, Retreat, Chase, Spawn, Wander



Learning Decision Trees

- Decision trees are usually learned by induction
  - Generalize from examples
  - Induction doesn’t guarantee correct decision trees
- Learning is non-incremental
  - Need to store all the examples
- If X is true in every example X must always be true
  - More examples are better
  - Errors in examples cause difficulty
  - Note that induction can result in errors

Entropy

- Entropy: how "mixed" is a set of examples
  - All one category: Entropy = 0
  - Evenly divided: Entropy = $\log_2$(# of examples)
- Given S examples Entropy(S) = $\Sigma -p_i \log_2 p_i$ where p_i is the proportion of S belonging to class i
  - 14 days with 9 in play-tennis and 5 in no-tennis
    - Entropy([9,5]) = 0.940
  - 14 examples with 14 in play-tennis and 0 in no-tennis
    - Entropy ([14,0]) = 0

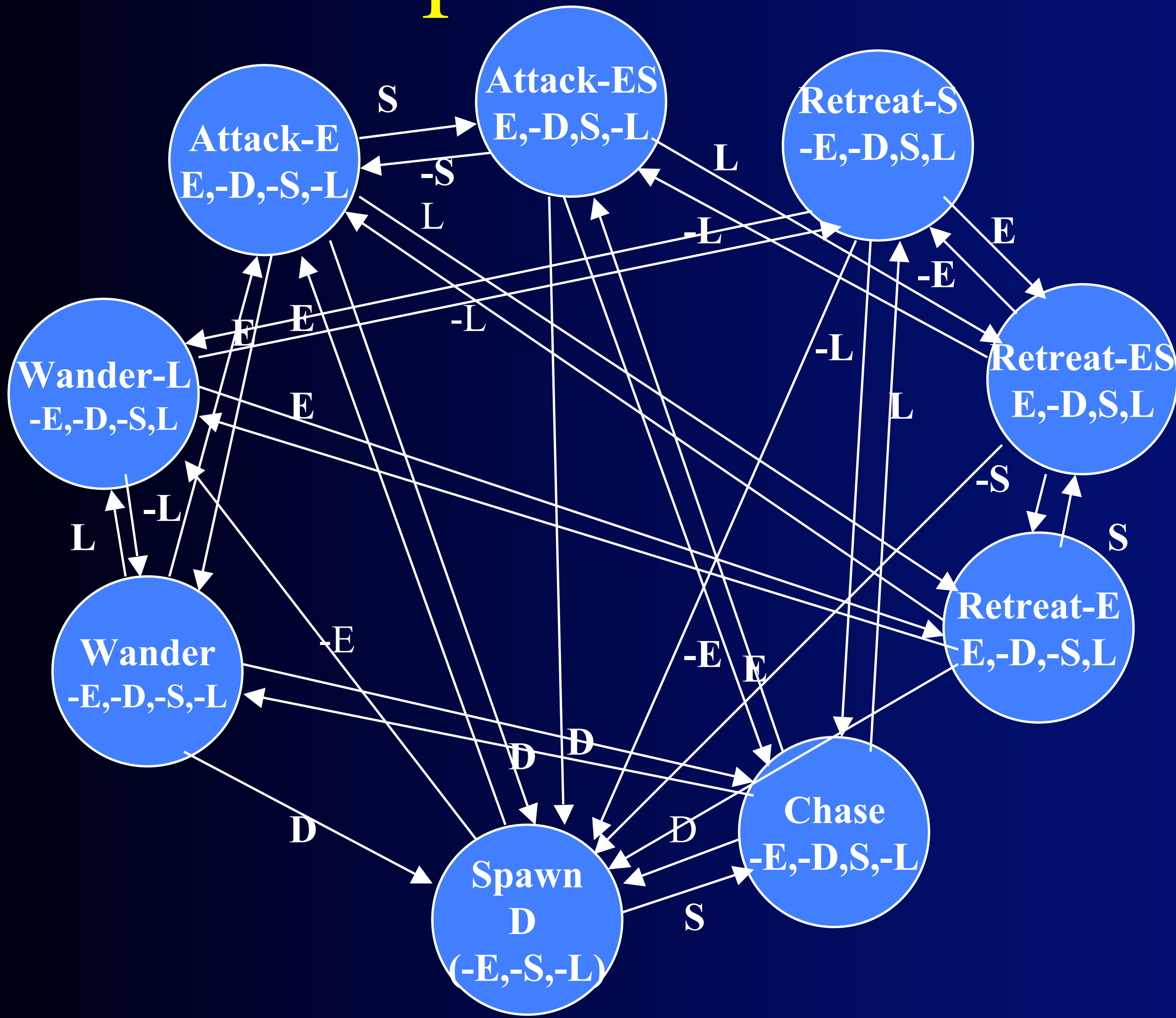
Information Gain

- Information Gain measures the reduction in Entropy
  - Gain(S, A) = Entropy(S) – Σ A/S Entropy(A)
- Example: 14 days: Entropy([9,5]) = 0.940
  - Measure information gain of Wind=<weak,strong>
    - Wind=weak for 8 days: [6,2] out of [9,5]
    - Wind=strong for 6 days: [3,3] out of [9,5]
    - Gain(S,Wind) = 0.048
  - Measure information gain of Humidity=<high,normal>
    - 7 days with high humidity: [3,4] out of [9,5]
    - 7 days with normal humidity: [6,1] out of [9,5]
    - Gain(S,Humidity) = 0.151
  - Humidity has a higher information gain than Wind
    - So choose humidity as the next attribute to be tested

Learning Example

- Learn a decision tree to replace the FSM
- Four attributes: Enemy, Die, Sound, Low Health
  - Each with two values: true, false
- Five categories: Attack, Retreat, Chase, Wander, Spawn
- Use all 16 possible states as examples
  - Attack(2), Retreat(3), Chase(1) Wander(2), Spawn(8)
- Entropy of first 16 examples (max entropy = 4)
  - Entropy([2,3,1,2,8]) = 1.953

Example FSM with Retreat



Events:

E=Enemy

S=Sound

D=Die

L=Low Health

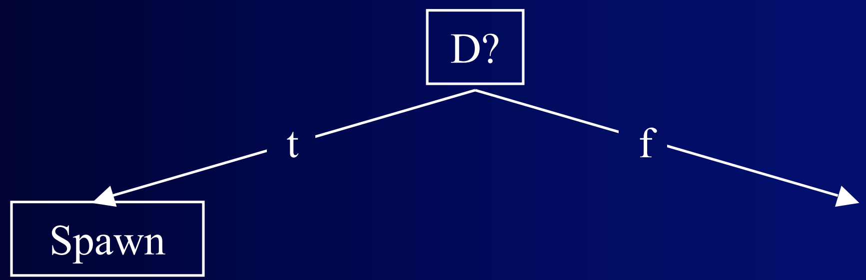
Each new feature can double number of states

Learning Example (2)

- Information gain of Enemy
  - 0.328
- Information gain of Die
  - 1.0
- Information gain of Sound
  - 0.203
- Information gain of Low Health
  - 0.375

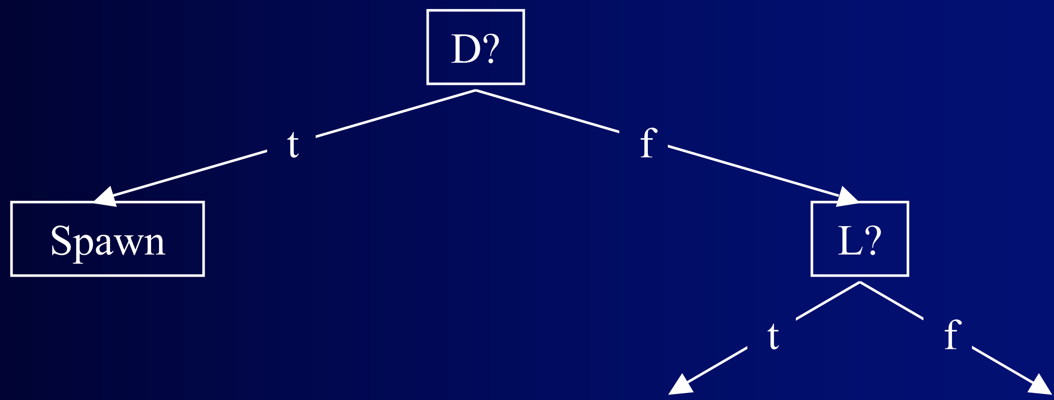
- So Die should be the root test

Learned Decision Tree



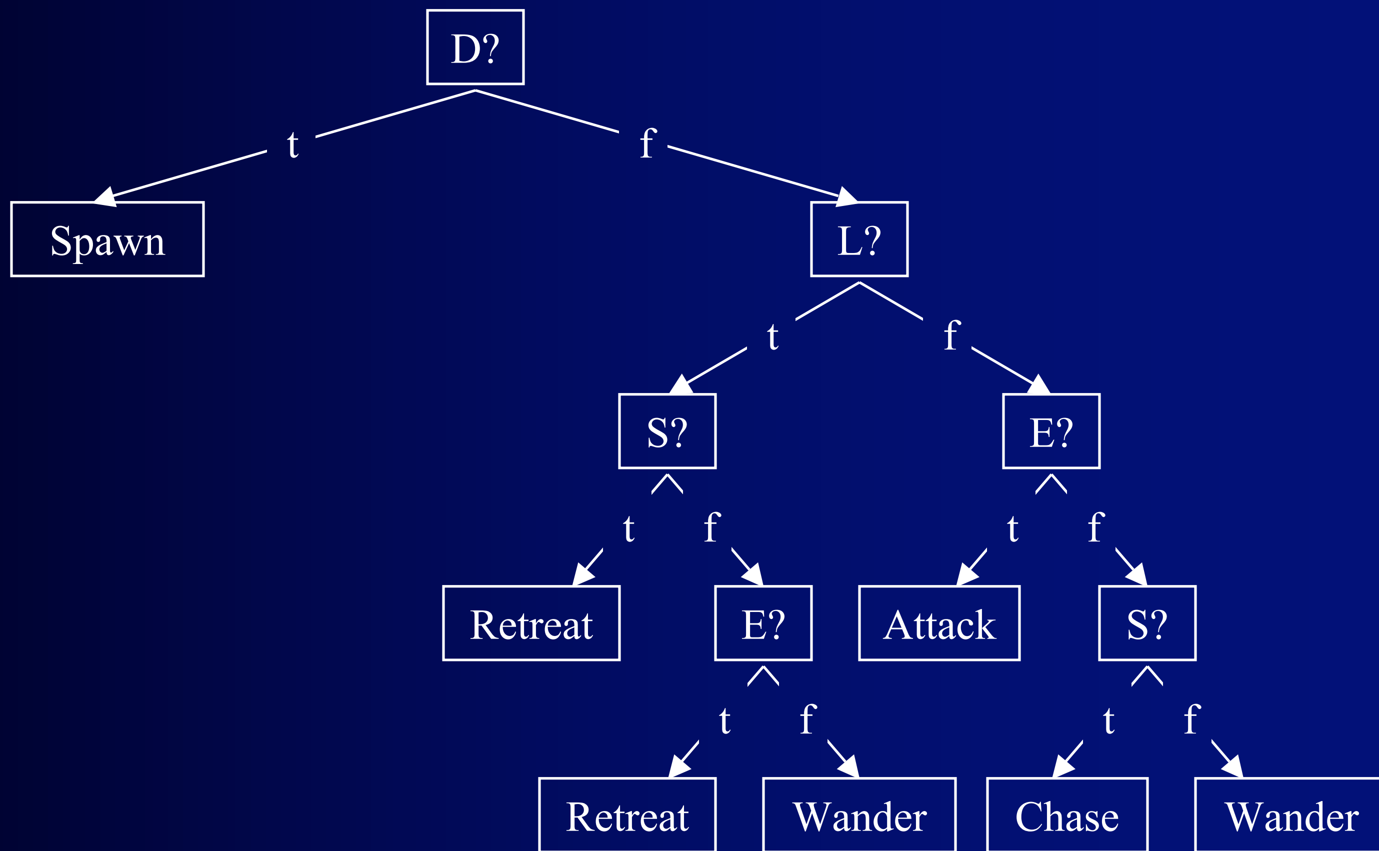
- 8 examples left [2,3,1,2] = 1.906
- 3 attributes remaining: Enemy, Sound, Low Health
- Information gain of Enemy
  - 0.656
- Information gain of Sound
  - 0.406
- Information gain of Low Health
  - 0.75

Learned Decision Tree (2)



- 4 examples on each side: t = 0.811; f = 1.50
- 2 attributes remaining: Enemy, Sound
- Information gain of Enemy (L = f)
  - 1.406
- Information gain of Sound (L = t)
  - .906

Learned Decision Tree (3)



Decision Tree Evaluation

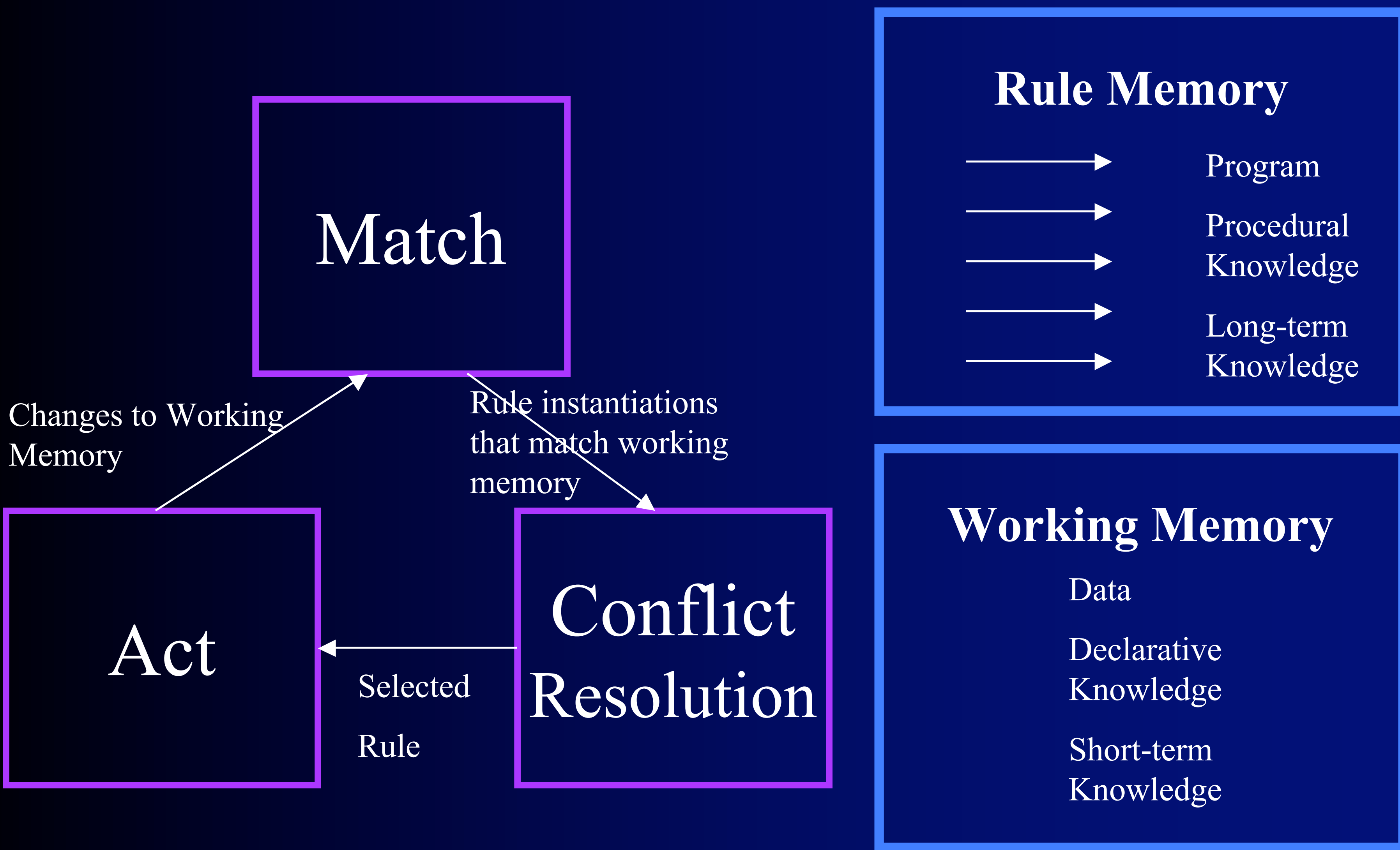
- Advantages
  - Simpler, more compact representation
  - State = Memory
    - Create “internal sensors” – Enemy-Recently-Sensed
  - Easy to create and understand
    - Can also be represented as rules
  - Decision trees can be learned
- Disadvantages
  - Decision tree engine requires more coding than FSM
  - Need as many examples as possible
  - Higher CPU cost
  - Learned decision trees may contain errors

Rule-based System

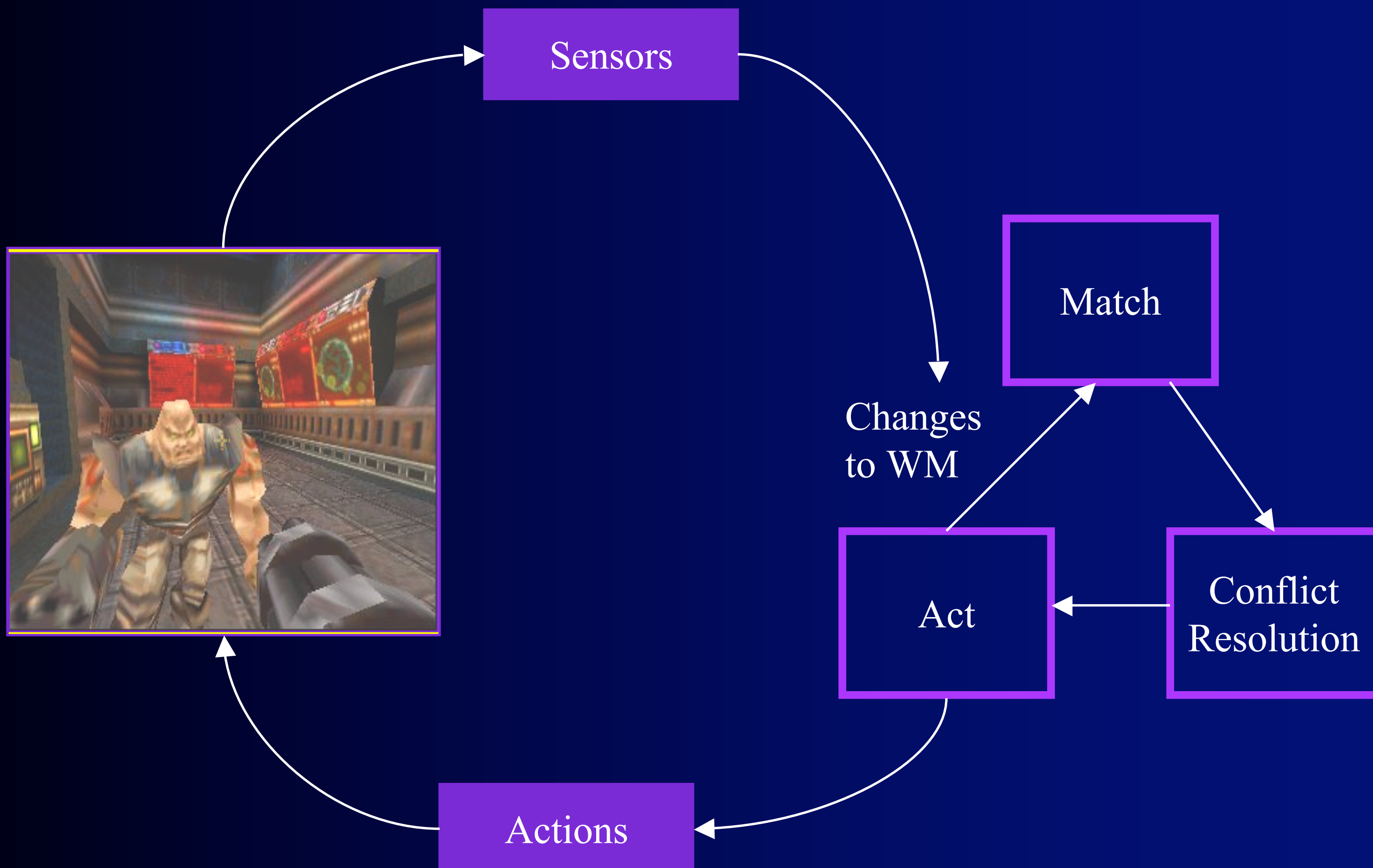
```
; The AI will attack once at 1100 seconds and then again
; every 1400 sec, provided it has enough defense soldiers.

(defrule
      (game-time > 1100)
=>
      (attack-now)
      (enable-timer 7 1100))

(defrule
      (timer-triggered 7)
      (defend-soldier-count >= 12)
=>
      (attack-now)
      (disable-timer 7)
      (enable-timer 7 1400))
```

Age of
Kings
Microsoft

Rule-Based Systems Structure



Complete Picture



Simple Approach

- No rules with same variable in multiple conditions
- Restricts what you can write, but might be ok for simple systems

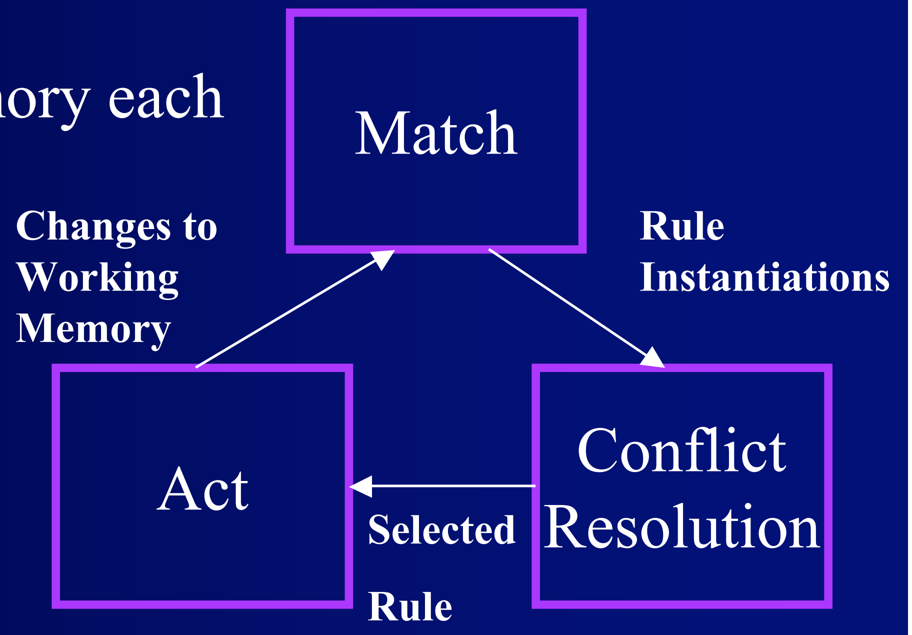
Picking the rule to fire

Simple approach

- Run through rules one at a time and test conditions
- Pick the first one that matches
- Time complexity depends on:
  1. Number of rules
  2. Complexity of conditions
  3. Number of rules that don’t match

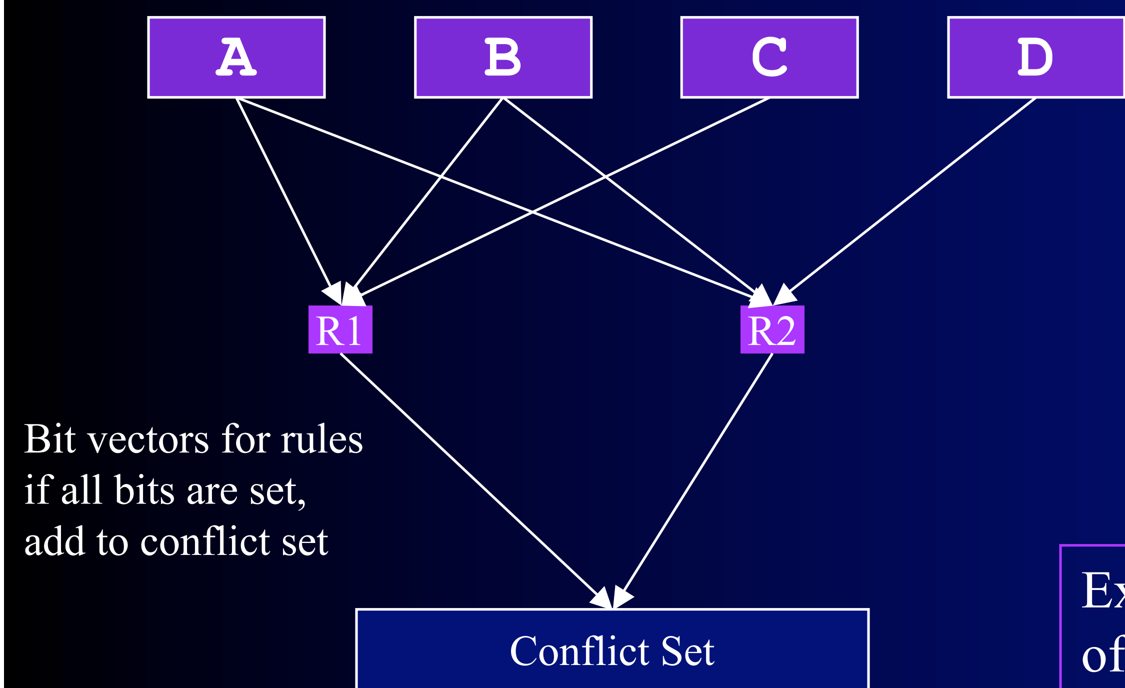
Creating Efficient Rule-based Systems

- Where does the time go?
  - 90-95% goes to Match
- Matching all rules against all of working memory each cycle is way too slow
- Key observation
  - # of changes to working memory each cycle is small



Picking the next rule to fire

- If only simple tests in conditions, compile rules into a match net
- Process *changes* to working memory: hash into tests



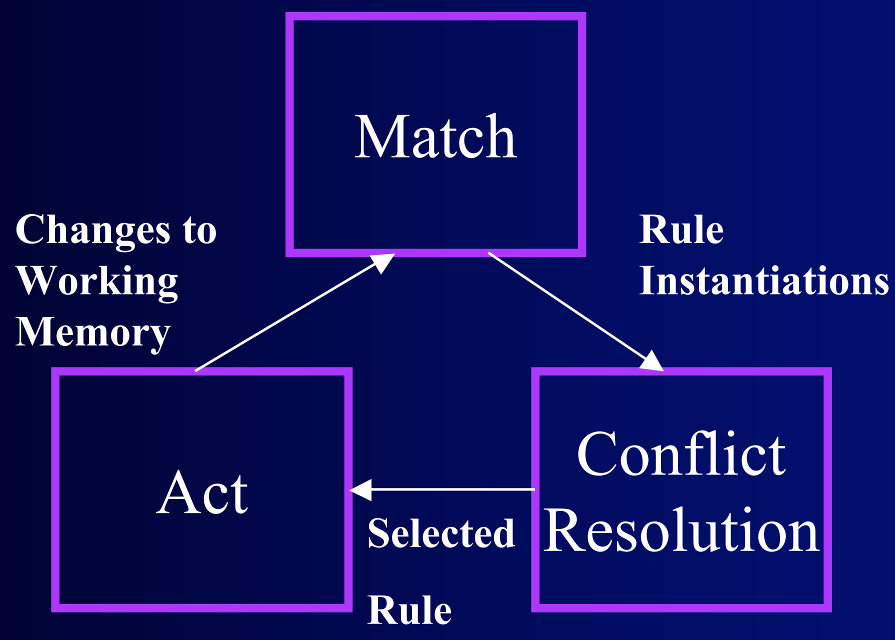
R1: If A, B, C, then …

R2: If A, B, D, then …

Expected cost: Linear in the number of changes to working memory

Conflict Resolution

- Which matched rule should fire?
- Which *instantiation* of a rule should fire?
  - Separate instantiation for every match of variables in rules



Conflict Resolution Filters

Select between instantiations based on filters:

1. Refractory Inhibition:
   - Don’t fire *same instantiation* that has already fired
2. Data Recency:
   - Select instantiations that match most recent data
3. Specificity:
   - Select instantiations that match more working memory elements
4. Random
   - Select randomly between the remaining instantiations

Other Conflict Resolution Strategies

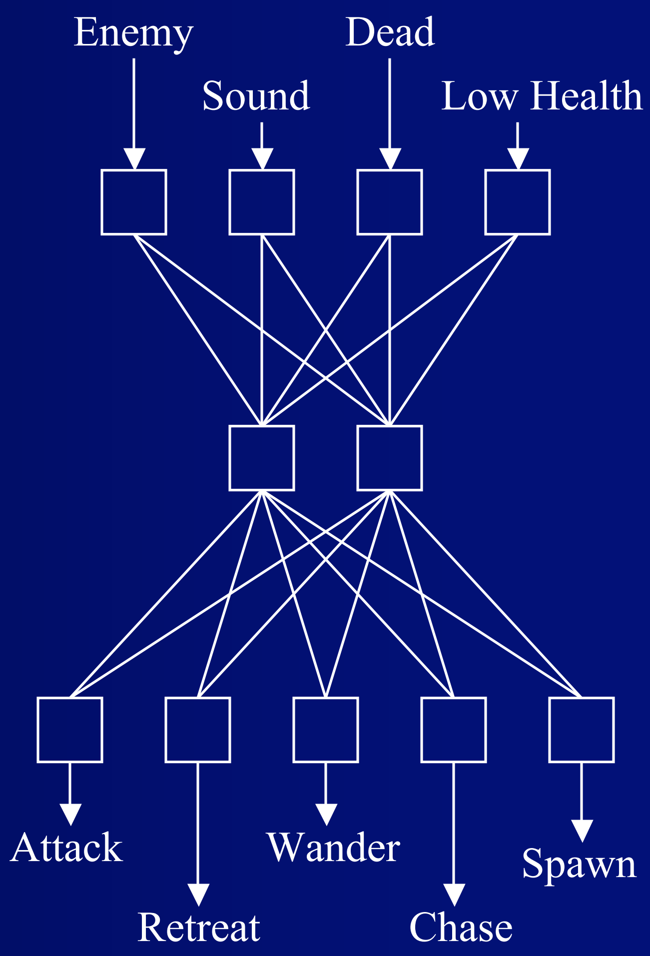
- Rule order – pick the first rule that matches
  - Makes order of loading important – not good for big systems
- Rule importance – pick rule with highest priority
  - When a rule is defined, give it a priority number
  - Forces a total order on the rules – is right 80% of the time
  - Decide Rule 4 [80] is better than Rule 7 [70]
  - Decide Rule 6 [85] is better than Rule 5 [75]
  - Now have ordering between all of them – even if wrong

Rule-based System Evaluation

- Advantages
  - Corresponds to way people often think of knowledge
  - Very expressive
  - Modular knowledge
    - Easy to write and debug compared to decision trees
    - More concise than FSM
- Disadvantages
  - Can be memory intensive
  - Can be computationally intensive
  - Sometimes difficult to debug

Neural Network for Quake

- Four input neuron
  - One input for each condition
- Two neuron hidden layer
  - Fully connected
  - Forces generalization
- Five output neuron
  - One output for each action
  - Choose action with highest output
  - Probabilistic action selection



Back Propagation

- Learning from examples
  - Examples consist of input and correct output
- Learn if network's output doesn't match correct output
  - Adjust weights to reduce difference
  - Only change weights a small amount (η)
- Basic neuron learning
  - $W_{i,j} = W_{i,j} + \Delta W_{i,j}$
  - $W_{i,j} = W_{i,j} + \eta(t-o)a_j$
  - If output is too high (t-o) is negative so $W_{i,j}$ will be reduced
  - If output is too low (t-o) is positive so $W_{i,j}$ will be increased
  - If a_j is negative the opposite happens

Neural Networks Evaluation

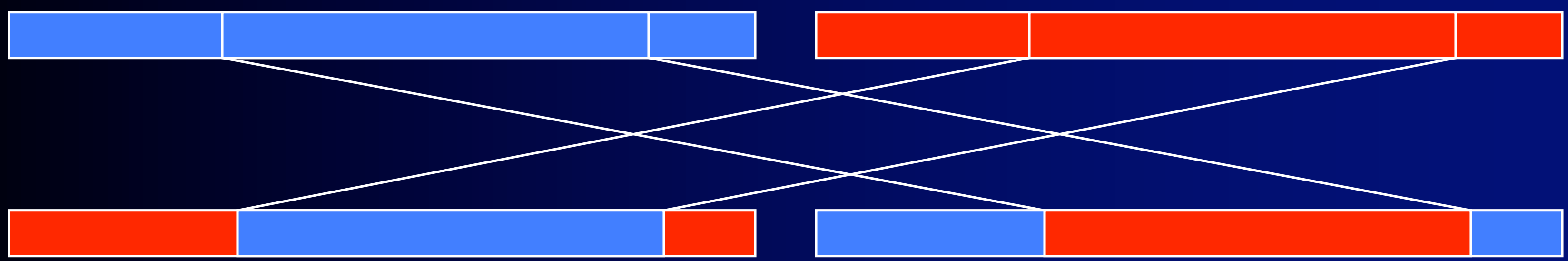
- Advantages
  - Handle errors well
  - Graceful degradation
  - Can learn novel solutions
- Disadvantages
  - Can’t understand how or why the learned network works
  - Examples must match real problems
  - Need as many examples as possible
  - Learning takes lots of processing
    - Incremental so learning during play might be possible

Genetic Algorithm: Inspiration

- Evolution creates individuals with higher fitness
  - Population of individuals
    - Each individual has a genetic code
  - Successful individuals (higher fitness) more likely to breed
    - Certain codes result in higher fitness
    - Very hard to know ahead which combination of genes = high fitness
  - Children combine traits of parents
    - Crossover
    - Mutation
- Optimize through artificial evolution
  - Define fitness according to the function to be optimized
  - Encode possible solutions as individual genetic codes
  - Evolve better solutions through simulated evolution

Genetic Operators

- Crossover
  - Select two points at random
  - Swap genes between two points

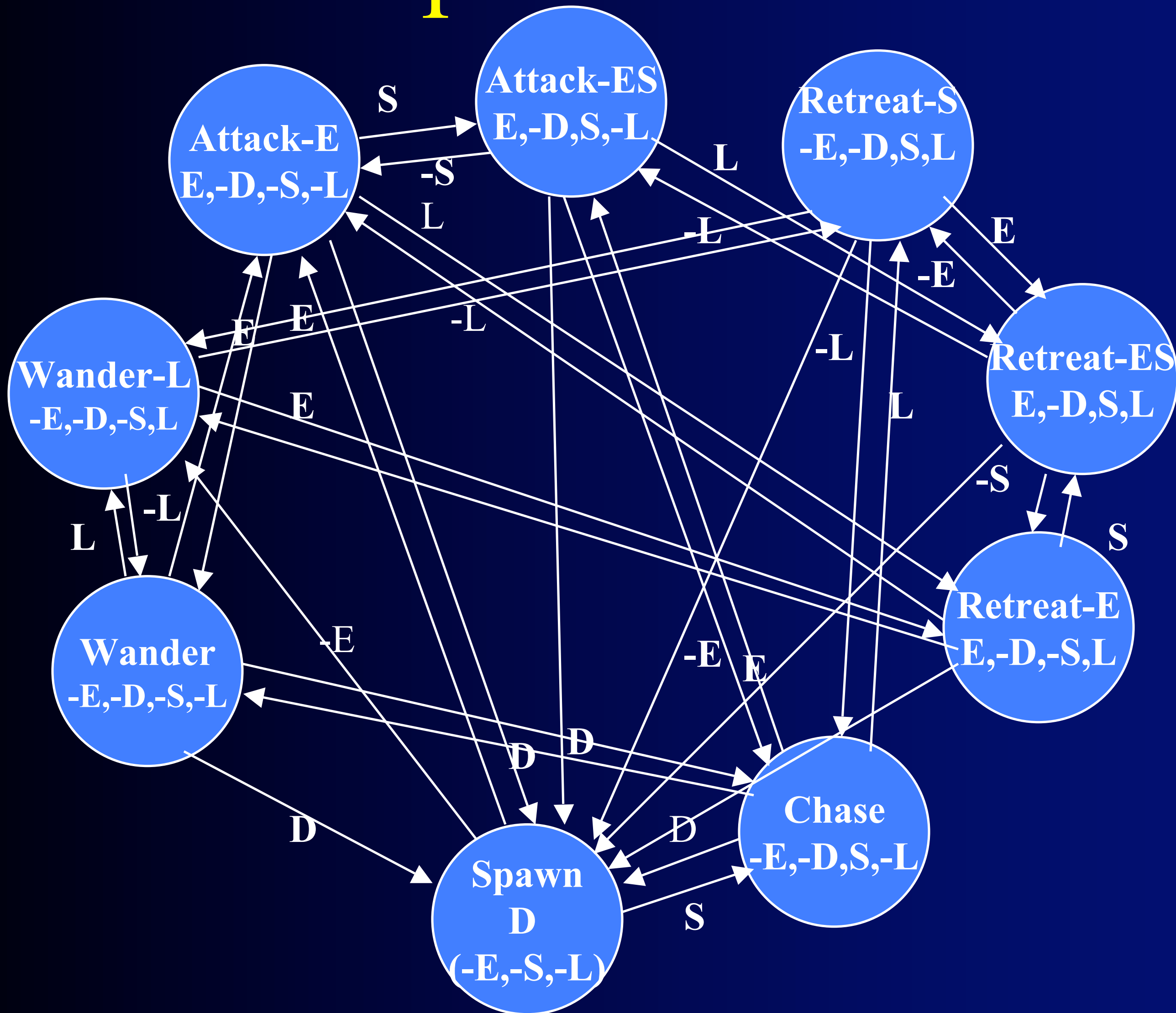


- Mutate
  - Small probably of randomly changing each part of a gene

Representation

- Gene is typically a string of symbols
  - Frequently a bit string
  - Gene can be a simple function or program
    - Evolutionary programming
- Every possible gene must encode a valid solution
  - Crossover should result in valid genes
  - Mutation should result in valid genes

Example FSM with Retreat



Events:

E=Enemy

S=Sound

D=Die

L=Low Health

Each new feature can double number of states

Representing rules as bit strings

- Conditions
  - Enemy = <t,f>: bits 1 and 2
    - 10: Enemy = t; 01: Enemy = f; 11: Enemy = t or f; 00: Enemy has no value
  - Sound = <t,f>: bits 3 and 4
  - Die = <t,f>: bits 5 and 6
  - Low Health = <t,f>: bits 7 and 8
- Classification
  - Action = <attack,retreat,chase,wander,spawn>
  - Bits 9-13: 10000: Action = attack
- 1111101100001: If dead=t then action=spawn
- Encode 1 rule per gene or many rules per gene
- Fitness function: % of examples classified correctly

Genetic Algorithm Example

- Initial Population

```
10 11 11 11 11010: E => Attack or Retreat or Wander
11 10 10 11 10100: S D => Attack or Chase
01 00 01 10 01100: -E -D L => Retreat or Chase
10 10 10 11 00010: E S D => Wander
...
```

- Parent Selection

```
10 11 11 11 11010: Sometimes correct
11 10 10 11 10100: Never correct
01 00 01 10 01100: Sometimes correct
10 10 10 11 00010: Never correct
...
```

Genetic Algorithm Example

- Crossover

```
 10 11 11 11 11010: Sometimes correct
 01 00 01 10 01100: Sometimes correct

 10 10 01 10 01010: E S -D L => Retreat or Wander
 01 01 11 11 11100: -E -S => Attack or Retreat or Chase
```

- Mutate

```
 10 10 01 10 01010: E S -D L => Retreat or Wander
 10 10 01 10 01000: E S -D L => Retreat
```

- Add to next generation

```
 10 10 01 10 01000: Always correct
 01 01 11 11 11100: Never correct
 ...
```

Genetic Algorithm Evaluation

- Advantages
  - Powerful optimization technique
  - Can learn novel solutions
- Disadvantages
  - Finding correct representation can be tricky
    - The richer the representation, the bigger the search space
  - Fitness function must be carefully chosen
  - Evolution takes lots of processing
    - Can’t really run a GA during game play
  - Solutions may or may not be understandable

Fuzzy Logic

- Philosophical approach
  - Ontological commitment based on "degree of truth"
  - Is ***not*** a method for reasoning under uncertainty
    - See probability theory and Bayesian inference
- Crisp Facts – distinct boundaries
- Fuzzy Facts– imprecise boundaries
- Example – Scout reporting an enemy
  - "Two to three tanks at grid NV 123456" (Crisp)
  - "A few tanks at grid NV 123456" (Fuzzy)
  - "The water is warm." (Fuzzy)
  - "There might be 2 tanks at grid NV 54 (Probabilistic)

Fuzzy Rules

- If the water temperature is cold and water flow is low then make a positive bold adjustment to the hot water valve.
- If position is unobservable, threat is somewhat low, and visibility is high then risk is low.

Fuzzy Variable

Fuzzy Value represented as a fuzzy set

Fuzzy Modifier or Hedge

Fuzzy Sets

- Classical set theory
  - An object is either in or not in the set
- Sets with smooth boundary
  - Not completely in or out – somebody 6" is 80% tall
- Fuzzy set theory
  - An object is in a set by matter of degree
  - 1.0 => in the set
  - 0.0 => not in the set
  - 0.0 < object < 1.0 => partially in the set
- Provides a way to write symbolic rules but "add numbers" in a principled way

Apply to Computer Game

- Can have different characteristics of entities
  - Strength: strong, medium, weak
  - Aggressiveness: meek, medium, nasty
  - If *meek* and attacked, run away fast.
  - If *medium* and attacked, run away slowly.
  - If *nasty* and *strong* and attacked, attack back.

- Control of a vehicle
  - Should slow down when *close* to car in front
  - Should speed up when *far* behind car in front

- Provides smoother transitions – not a sharp boundary

Evaluation of Fuzzy Logic

- Does not necessarily lead to non-determinism
- Advantages
  - Allows use of numbers while still writing “crisp” rules
  - Allows use of “fuzzy” concepts such as medium
  - Biggest impact is for control problems
    - Help avoid discontinuities in behavior
- Disadvantages
  - Sometimes results are unexpected and hard to debug
  - Additional computational overhead
  - Change in behavior may or may not be significant

What is Planning?

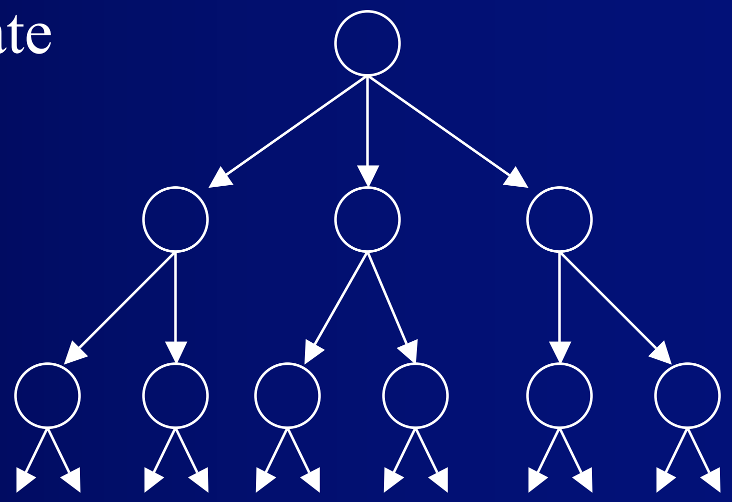
- Plan: sequence of actions to get from the current situation to a goal situation
  - Higher level mission planning
  - Goal-oriented behavior (GOB)
- Planning: generate a plan
  - Initial state: the state the agent starts in or is currently in
  - Goal test: is this state a goal state
  - Operators: every action the agent can perform
    - Also need to know how the action changes the current state
- Note: at this level planning doesn’t take opposition into account

Two Approaches

State-space search

- Search through the possible future states that can be reached by applying different sequences of operators
  - Initial state = current state of the world
  - Operators = actions that modify the world state
  - Goal test = is this state a goal state

State-space Search

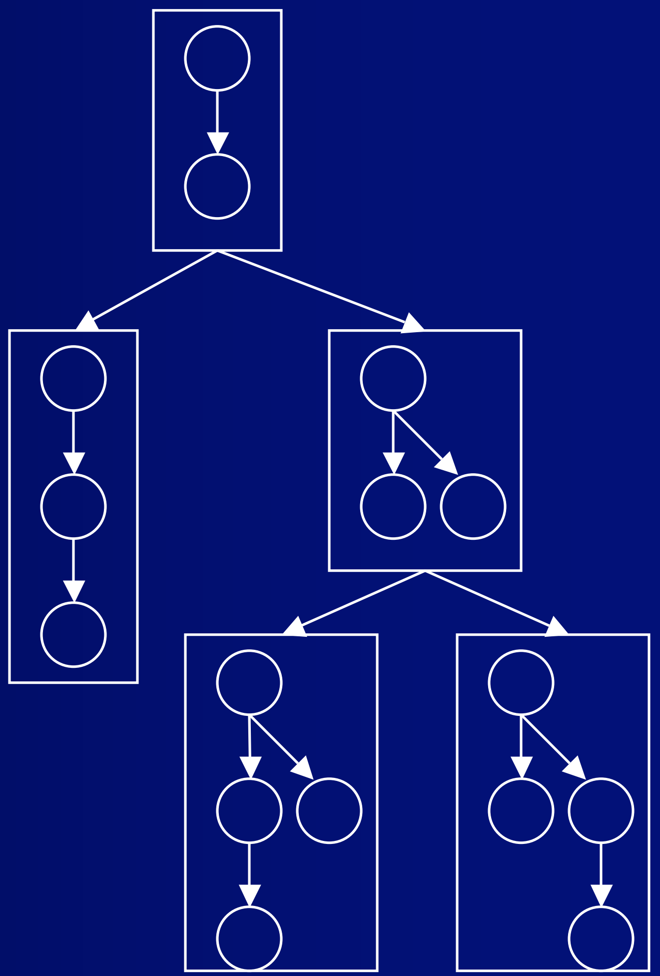


Two Approaches

Plan-space search

- Search through possible plans by applying operators that modify plans
  - Initial state = empty plan (do nothing)
  - Operators = add an action, remove an action, rearrange actions
  - Goal test = does this plan achieve the goal state

Plan-space Search

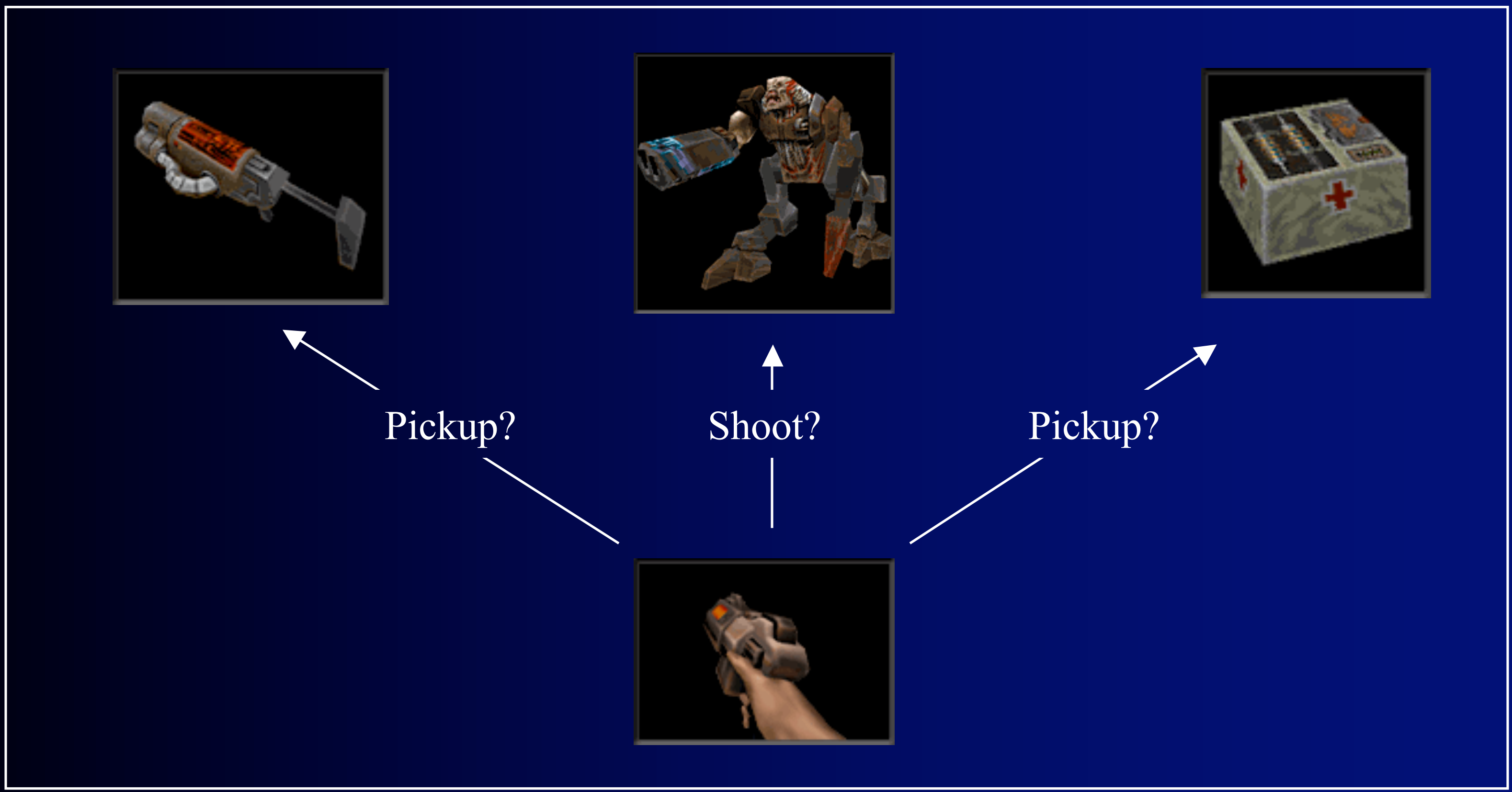


AI for Strategic Games

- Possible way to do Warcraft
  - Define top goals such as kill enemy, mine gold, build buildings
  - Weight the goals and pick most important one that is not achieved and we are not working on
  - Determine what needs to be done to achieve this goal
    - Get more gold, get men closer to enemy, ...
  - Select operators to achieve goal
    - Some operators may involve complex actions like walking to the goal mine -- use specialized planning approaches for these: A*
    - Some operators may manipulate multiple pieces at once: teams

  - Once resources assigned to a goal, go to next goal in list and see if any resources available to achieve it

What should I do?



Look-ahead search

- Try out everything I could do and see what works best
  - Looking ahead into the future
  - As opposed to hard-coded behavior rules
- Can’t look-ahead in real world
  - Don’t have time to try everything
  - Can’t undo actions
- Look-ahead in an internal version of the world
  - Internal state representation
  - Internal action representation
  - State evaluation function

Internal State Representation

- Store a model of the world inside your head
  - Simplified, abstracted version
- Experiment with different actions internally
  - Simple planning
- Additional uses of internal state
  - Notice changes
    - My health is dropping, I must be getting shot in the back
  - Remember recent events
    - There was a weak enemy ahead, I should chase through that door
  - Remember less recent events
    - I picked up that health pack 30 seconds ago, it should respawn soon

Internal State for Quake II

- Self
  - Current-health
    - Last-health
  - Current-weapon
    - Ammo-left
  - Current-room
    - Last-room
  - Current-armor
    - Last-armor
  - Available-weapons
- Enemy
  - Current-weapon
  - Current-room
  - Last-seen-time
  - Estimated-health
- Powerup
  - Type
  - Room
  - Available
  - Estimated-spawn-time
- Map
  - Rooms
  - Halls
  - Paths
- Current-time
- Random-number

- Parameters
  - Full-health
  - Health-powerup-amount
  - Ammo-powerup-amount
  - Respawn-rate

Internal Action Representation

- How will each action change the internal state?
  - Simplified, abstracted also
- Necessary for internal experiments
  - Experiments are as accurate as the internal representation
- Internal actions are called operators
  - Pre-conditions: what must be true so I can take this action
  - Effects: how action changes internal state
- Additional uses of internal actions
  - Update internal opponent model

Example: Pick-up-health operator

- Preconditions:
  - Self.current-room = x
  - Self.current-health < full-health
  - Powerup.current-room = x
  - Powerup.type = health
  - Powerup.available = yes
- Effects:
  - Self.last-health = self.current-health
  - Self.current-health = current-health + health-powerup-amount
  - Powerup.available = no
  - Powerup.estimated-spawn-time = current-time + respawn-rate

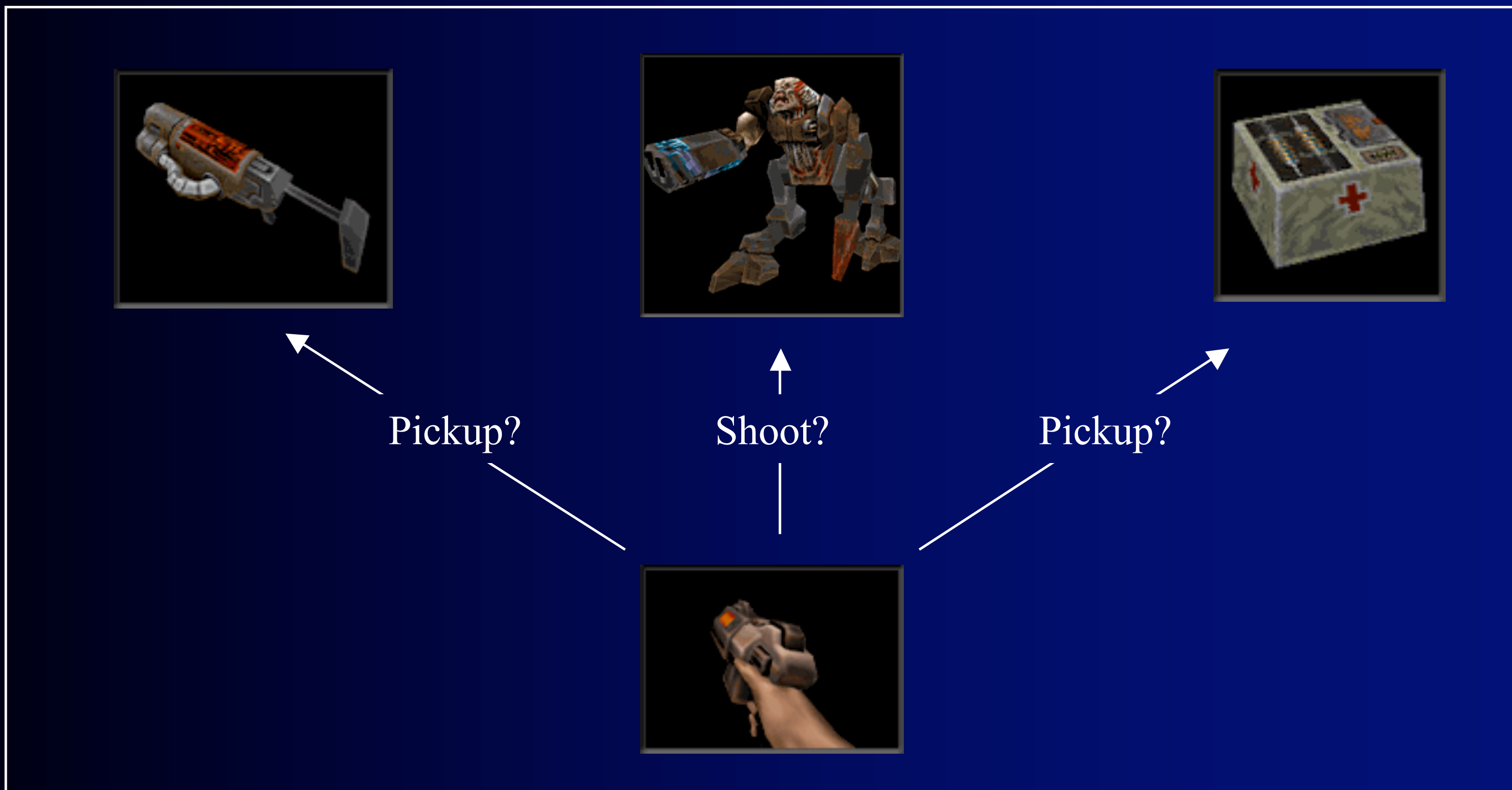
State Evaluation Function

- What internal states are good and bad?
  - Way to compare states and decide which is better
  - Traditional planning talks about goal states
  - Desirable state properties
- Internal experiments find good states and avoid bad ones

State Evaluation for Quake II

- Example 1: Prefer states with higher self.current-health
  - Always pick up health powerup
  - Counter example: Self.current-health = 99% and Enemy.current-health = 1%
- Example 2: Prefer lower enemy.current-health
  - Always shoot enemy
  - Counter example: Self.current-health = 1% and Enemy.current-health = 99%
- Example 3: Prefer higher self.health – enemy.health
- More complex evaluations
  - If self.health > 50% prefer lower enemy.health else higher self.health
  - If self.health > low-health prefer lower enemy.health else higher self.health

What should I do?

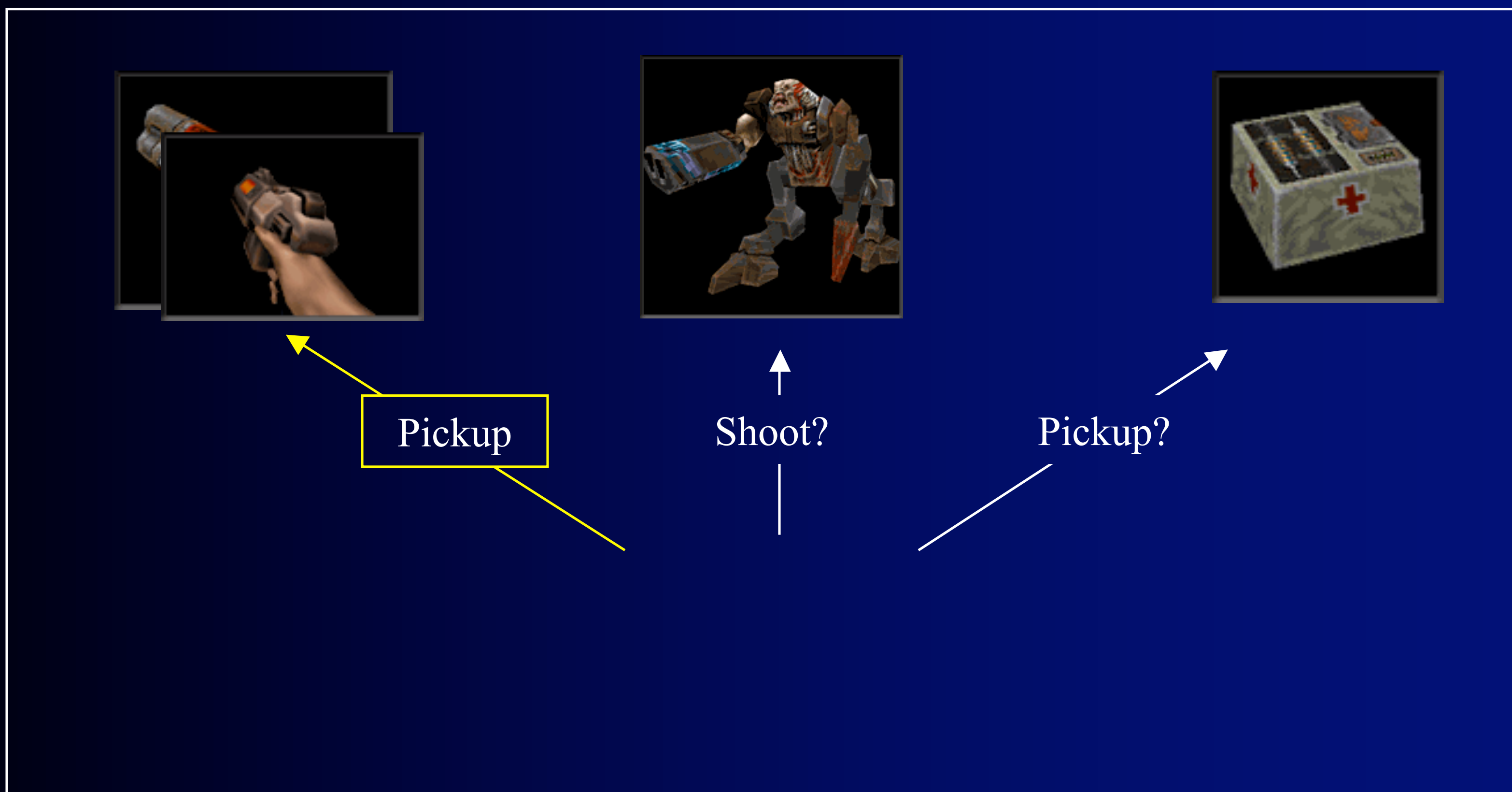


Self.current-health = 20
Self.current-weapon = blaster

Enemy.estimated-health = 50

Powerup.type = health-pak
Powerup.available = yes
Powerup.type = Railgun
Powerup.available = yes

One Step: Pickup Railgun

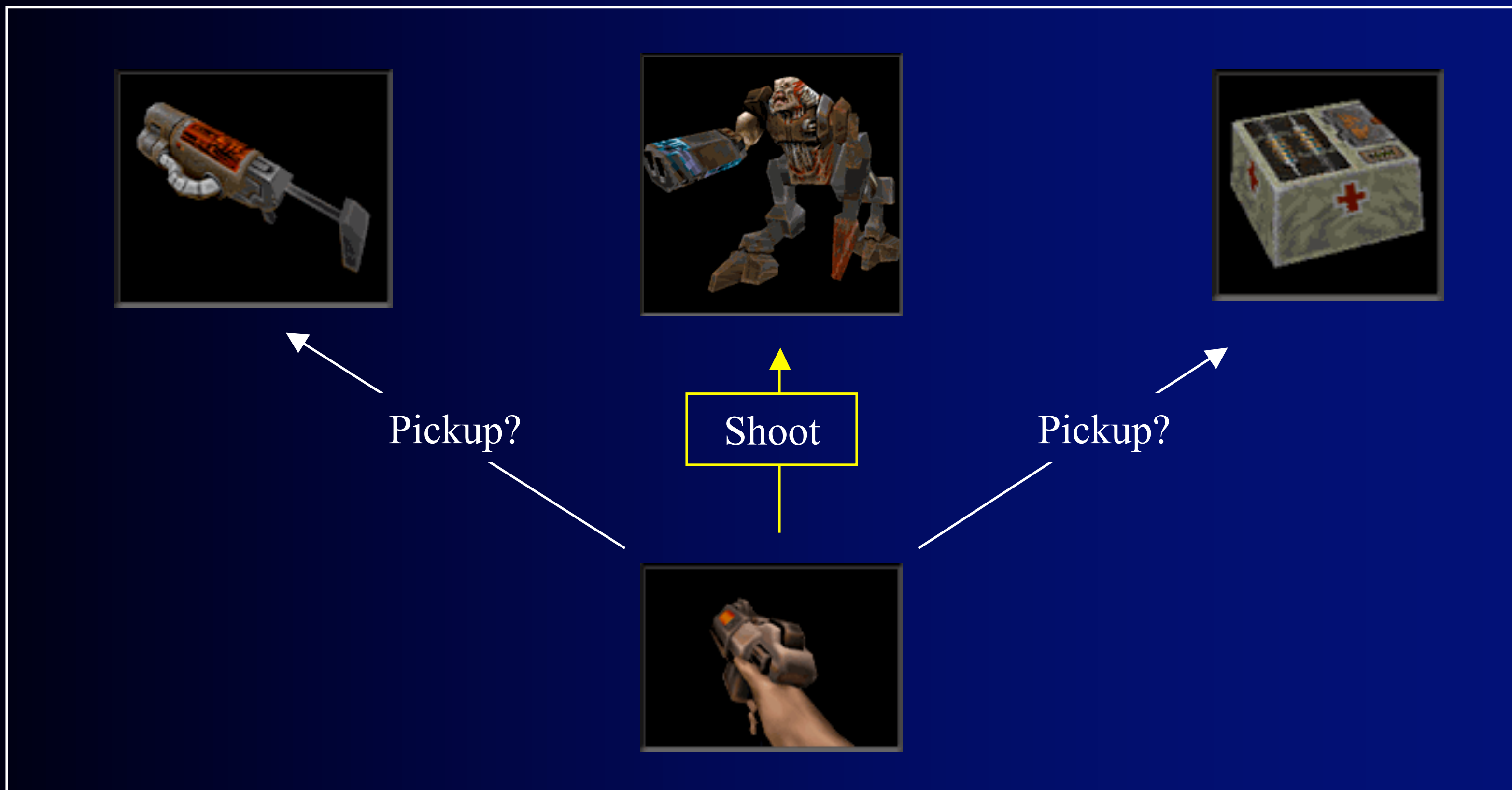


Self.current-health = 10
Self.current-weapon = Railgun

Enemy.estimated-health = 50

Powerup.type = health-pak
Powerup.available = yes
Powerup.type = Railgun
Powerup.available = no

One Step: Shoot

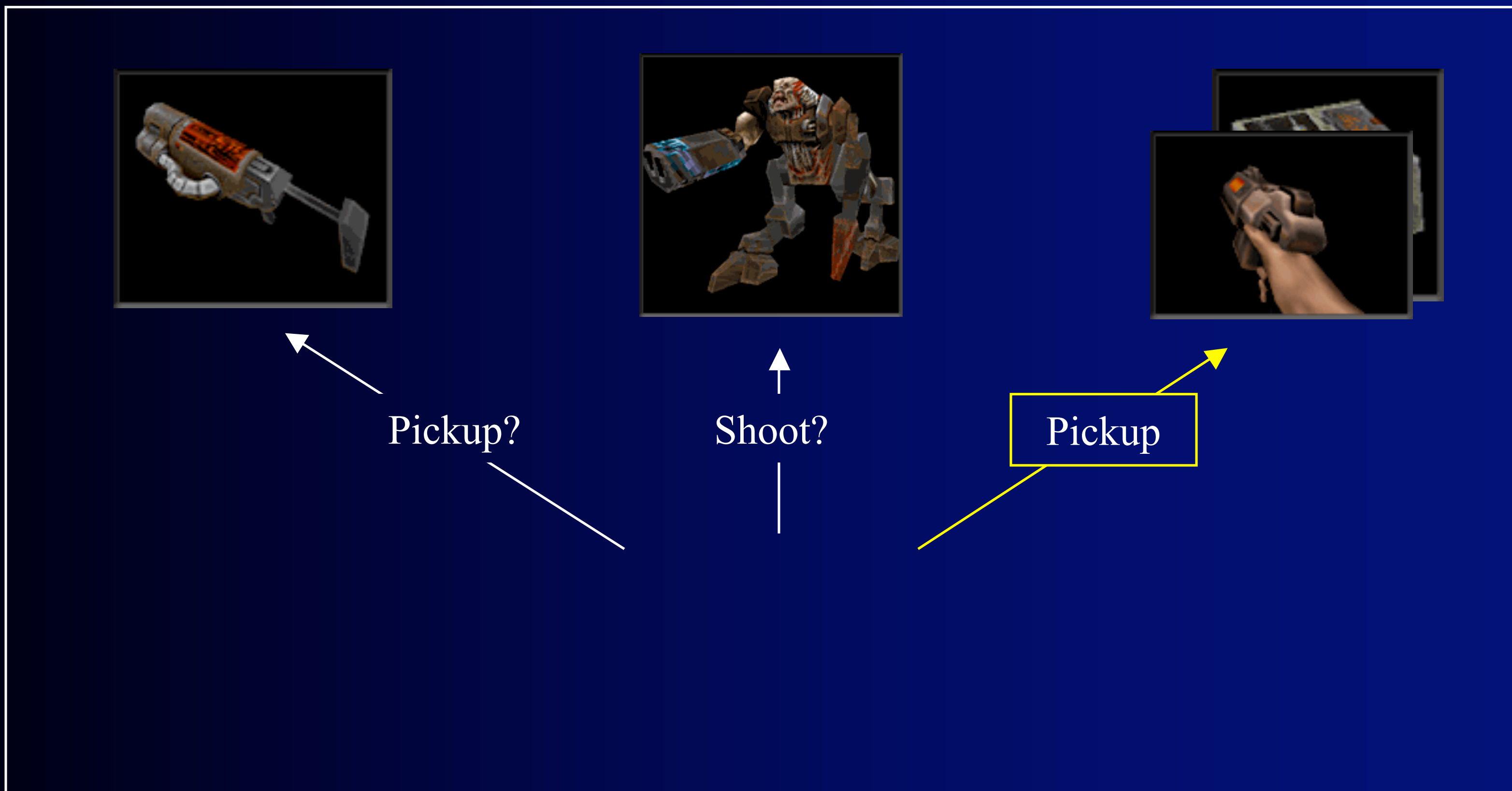


Self.current-health = 10
Self.current-weapon = blaster

Enemy.estimated-health = 40

Powerup.type = health-pak
Powerup.available = yes
Powerup.type = Railgun
Powerup.available = yes

One Step: Pickup Health-pak

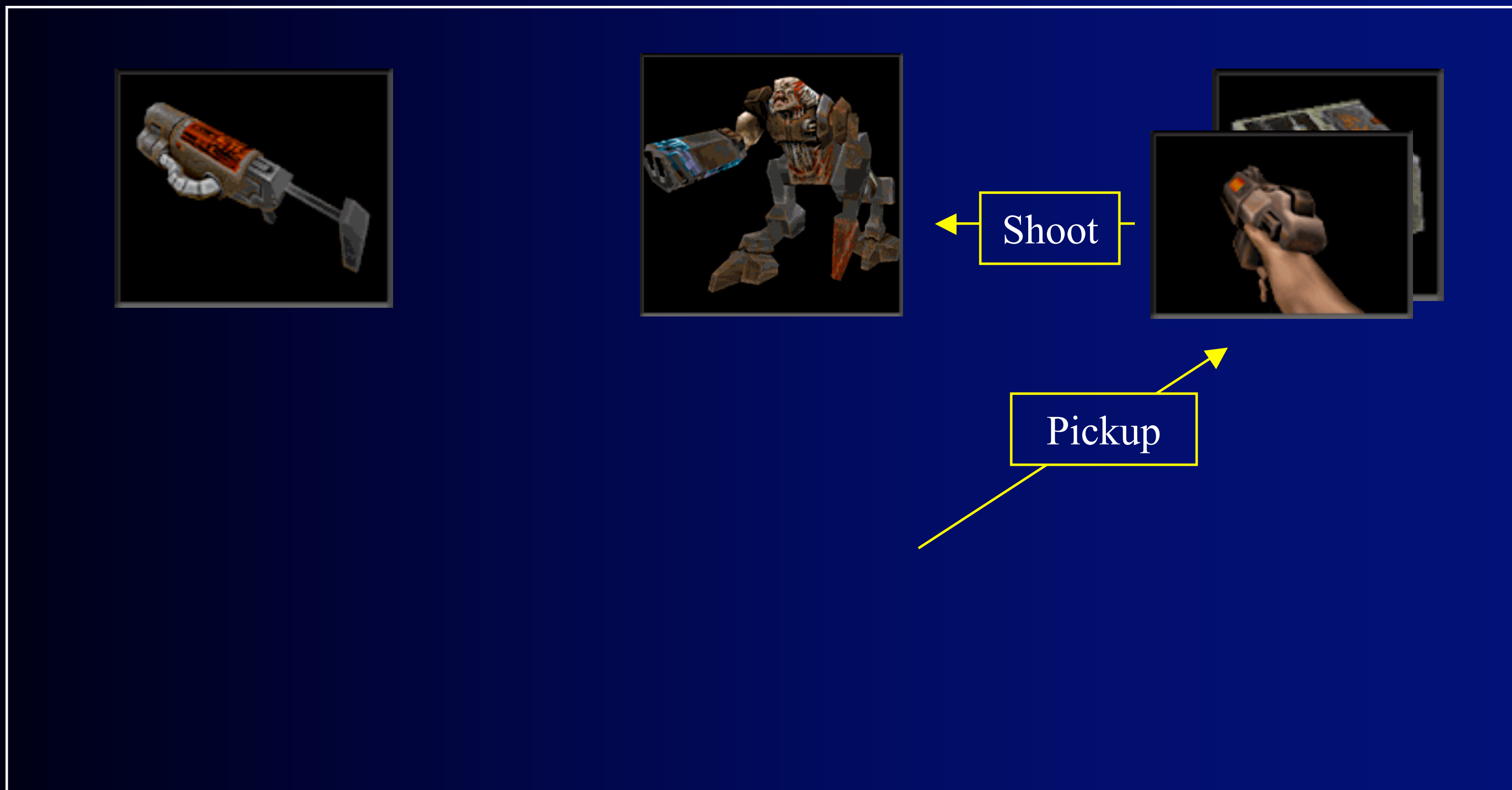


Self.current-health = 90
Self.current-weapon = blaster

Enemy.estimated-health = 50

Powerup.type = health-pak
Powerup.available = no
Powerup.type = Railgun
Powerup.available = yes

Two Step

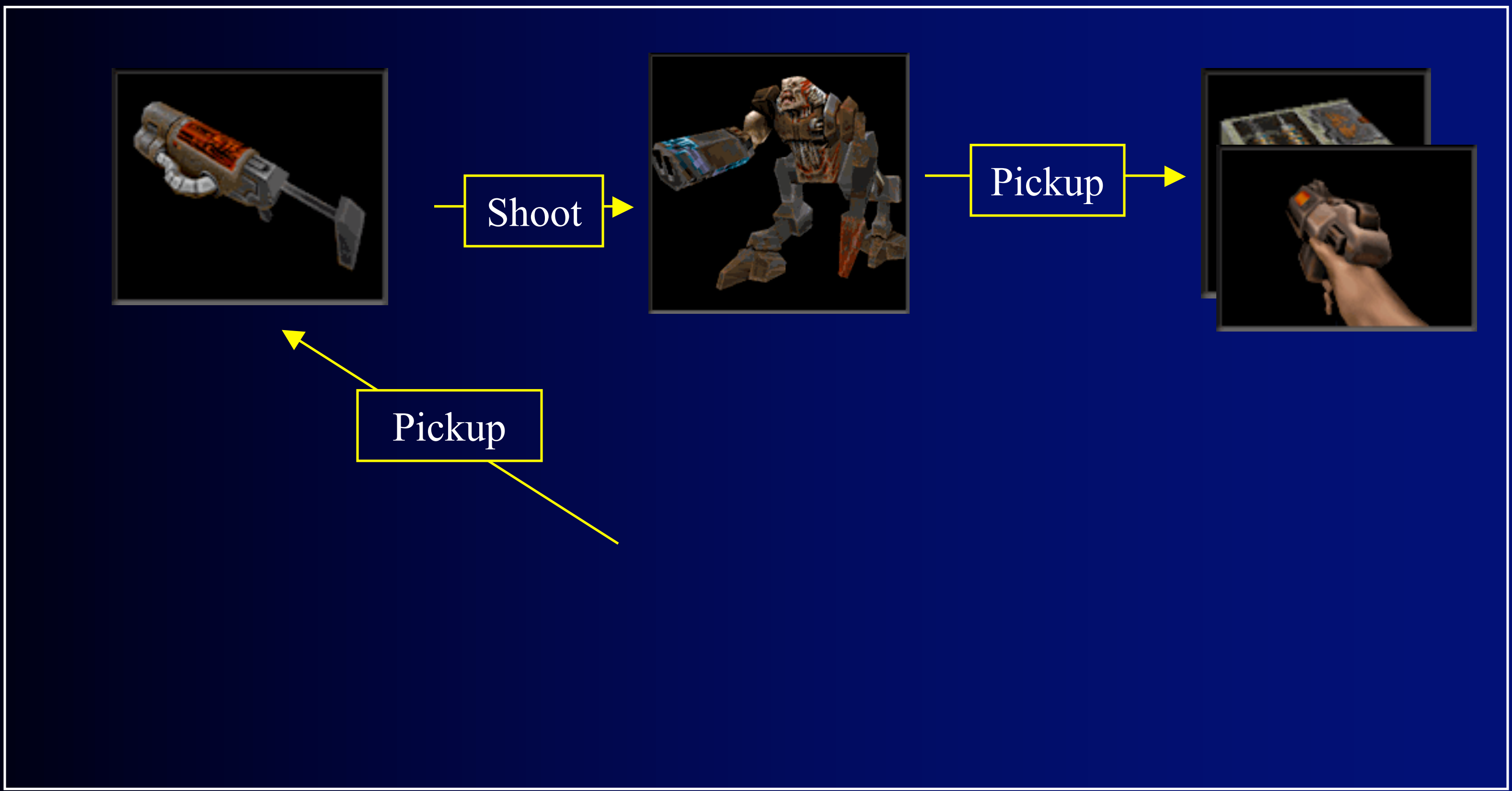


Self.current-health = 80
Self.current-weapon = blaster

Enemy.estimated-health = 40

Powerup.type = health-pak
Powerup.available = no
Powerup.type = Railgun
Powerup.available = yes

Three Step Look-ahead



Self.current-health = 100
Self.current-weapon = Railgun

Enemy.estimated-health = 0

Powerup.type = health-pak
Powerup.available = no
Powerup.type = Railgun
Powerup.available = no

Look-ahead Search

- Simple limited depth state-space search
  - One step look-ahead search

    for each operator with matching pre-conditions

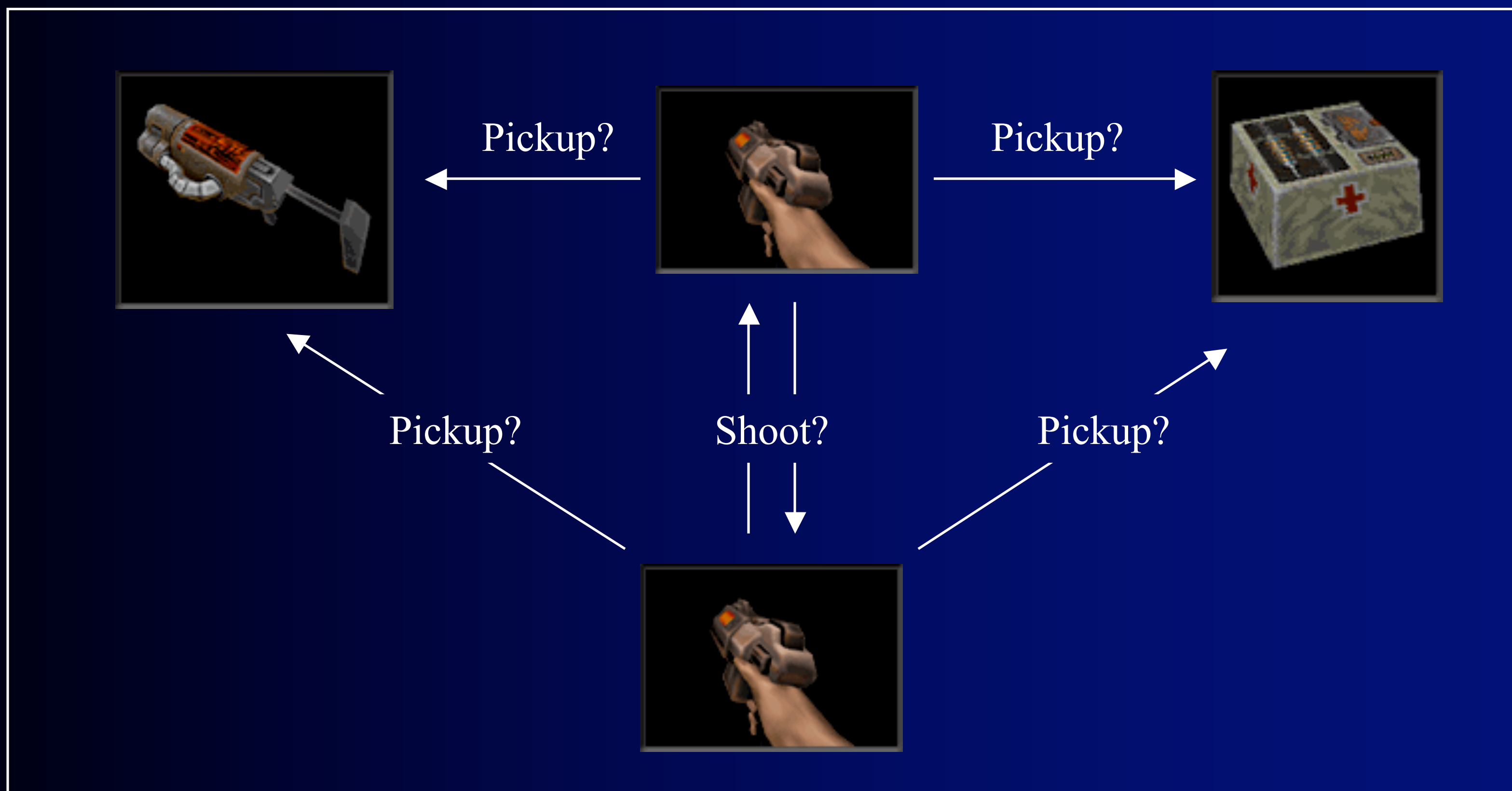
        apply operator to current state

        evaluate resulting state

    choose "real" action that looked best internally

- Searching deeper
  - Longer, more elaborate plans
  - More time consuming
  - More space consuming
  - More chances for opponent or environment to mess up plan
  - Simplicity of internal model more likely to cause problems

Opponent: New problems



Self.current-health = 20
Self.current-weapon = blaster

Enemy.estimated-health = 50
Enemy.current-weapon = blaster

Powerup.type = health-pak
Powerup.available = yes
Powerup.type = Railgun
Powerup.available = yes

Opponent Model

- Need to know what opponent will do
  - Accurate internal state update
  - Actions can interfere
- Solution 1: Assume the worst
  - Opponent does what would be worst for you
  - Game tree search
  - Exponential increase in number of state evaluations
- Solution 2: What would I do?
  - Opponent does what you would in the same situation
- Solution 3: Internal model of opponent
  - Remember what they did last time or like to do

Means-ends Analysis

- What’s the difference between the current situation and the goal?
  - Goal is represented by a target state or target conditions
  - Compare current state and target state
- What reduces the differences?
  - Match differences up with operator effects
- Can I perform these operators?
  - Try to achieve operator pre-conditions
  - Match pre-conditions up with other operator’s effects
- Searching backwards from the goal is sometimes cheaper
  - Many operators to perform at any time
  - Few operators achieve the goal conditions

Planning Evaluation

- Advantages
  - Less predictable behavior
  - Can handle unexpected situations
- Disadvantages
  - Less predictable behavior (harder to debug)
  - Planning takes processor time
  - Planning takes memory
  - Need simple but accurate internal representations

Traditional planning and combat simulations

- Combat simulations are difficult for traditional AI planning
  - Opponent messes up the plan
  - Environment changes mess up the plan
  - “Goal state” is hard to define and subject to change
  - Lots of necessary information is unavailable
  - Too many steps between start and finish of mission
- Some applications of traditional AI planning
  - Path planning
    - State-space search algorithms like A*
  - Game theoretical search
    - State-space search algorithms with opponents like min-max and alpha-beta

Why aren't AI's better?

- Don't have realistic models of sensing
- Not enough processing time to plan ahead
- Space of possible actions too large to search efficiently (too high of branching factor)
- Single evaluation function or predefined subgoals makes them predictable
  - Only have one way of doing things
- Too hard to encode all the relevant knowledge
- Too hard to get them to learn

Fighting Opponents

- Must select between different attacks, blocks, etc.
- Could easily overwhelm human
  - Reaction-time
- Rely heavily on motion-capture for animation
- Varying amounts of AI
- State machines
- Some learning/adaptation

Tactical Enemies

- Early days
  - Run and shoot: no navigation
  - Sometimes see through walls
- Next step
  - “Invisible” nodes for navigation
  - Pick up powerups on the fly
  - Still little or no obstacle avoidance
  - Variability in skill
- Key issues
  - Challenging but not overwhelming opponents
- Example games
  - Deus Ex, Return to Wolfenstein, Max Payne, Metal Gear Solid, Halo
- Standard technology
  - Scripting languages, hierarchical finite-state machines

Action/FPS Game Opponent

- Provide a challenging opponent
  - Not always as challenging as a human -- Quake monsters
  - What ways should it be subhuman?
- Not too challenging
  - Should not be superhuman in accuracy, precision, sensing, ...
- Should not be too predictable
  - Through randomness
  - Through multiple, fine-grained responses
  - Through adaptation and learning


Tactical Opponent
Run
Attack
Pickup
Die
Soldier of Fortune
Raven Software

Strategic Enemies

- Decide on overall strategy
  - Aggressive, defensive
  - Throw a lot, run a lot, dump and chase, …
- Resource Allocation
  - Decide what to build, mine, grow, … with available resources
- Control Units
  - To build, mine, grow, attack, defend, …
  - Use special abilities of units
- Sometimes cheat to overcome weaknesses
- Play to lose?
- Example games
  - Football, Age of Kings, Starcraft, Warcraft, …
- Standard technology
  - Predefined scripts/plans
  - Simple rule-based systems

Units

- Military units, team sport players, …
- Path planning and route following are very important!
  - Efficient, flexible A*
- Formations, collision detection, motion capture
- Standard technology:
  - Scripting languages
  - Finite-state machines
  - Simple rule-based systems

Support Characters

- Scripted behavior
  - Small set of behavior routines
  - Small set of responses to predefined set of questions
  - Navigate via nodes
- Example Games
  - Blade Runner, Diablo II, Monkey Island series, …