Playtesting

• Hand playtesting sheets to mentor TA
• Retries for projects apply to playtesting as well
3 down, 2 to go!

• Final2: “pre minimum viable product”
• Final3: minimum viable product
  – Engines are definitely done
  – Core gameplay implemented
• Results look great!
Final4 rubric meetings

• Final4 will depend on state of final3
• Goal of final4 is a game that is mostly complete
  – Ironing out kinds from last week
  – Polish
  – Incorporate playtesting feedback
QUESTIONS?
Lecture 13
AI
Game A.I.

- Usually used to make computer controlled units behave reasonably
- Can also be used to support the human player
- Essential for a good gameplay experience
Decision Making

• NPC’s should do something, but what?
• Could hardcode the logic
  – Game-specific
  – Likely involves copied code
• We want a structured way for NPC’s to make decisions
  – Based on game state, unit state, random values, etc…
Engine-Specific AI

• Ridiculously difficult to make an AI that plays all games
  – Might as well make a universal AI

• Silly for each game to re-implement certain models
  – Engines support basic AI concepts
    • Organizational data structures
  – Games populate these with game-specific information
AI

FSM’S
Finite State Machines

• “Classic” game AI construct
  – See this article for specifics
• Current state specifies:
  – What to do during this update?
  – What transitions to check for?
• Transitions specify:
  – What can change the current state?
  – What state is transitioned to if conditions pass?
A potential problem...

- **Problem**: what if a state transitions 2 things under the same condition?
- **Ex**: always “run away” from the mouse cursor
- **Return to “find leaf” or “go home”?**
- **Depends on how you got to “run away”...**
Solution: Stack-based FSM’s

- Current state is the top of a stack of states
- Transitions specify what to do to the stack:
  - Pop
  - Push(new)
  - Pop + Push(new)
- Gives a generic “return to previous” transition

![Diagram showing stack operations]
FSM’s: Pro/Con

- **Pros:**
  - Allows for high specificity
  - Pretty lightweight engine implementation
  - Stack-based allows for “remembering” where the AI currently is

- **Cons:**
  - Typically require a ridiculous number of low-detail states - combinatorial explosion!
  - Requires a ton of planning by game code
AI – FSM’s

QUESTIONS?
AI

BEHAVIOR TREES
Behavior Trees

• “Recently” popularized by the Halo series
• Based on a rigorous, hierarchical structure
  – As a result, both flexible and stable
• Core functionality is engine-general!
Structure

• It’s a tree!
• Every tick, the root node is updated
• Each node returns a status when it’s updated
  – SUCCESS, FAIL, RUNNING
• Nodes will update their children and return a status based on responses
The Leaves

- Leaf nodes of the tree are Actions and Conditions
- Actions do things
  - Make a unit move
  - Make a unit attack
- Conditions check some game state
  - Returns SUCCESS if the condition is true, or FAIL if the condition is false

- Eat Action
- Sleep Action
- Party! Action
- Enemy near? Condition
- Is it daytime? Condition
The Others

- Internal nodes are Composites and Wrappers/Decorators
- Composites have multiple children nodes
- Wrappers wrap a single child node
- These dictate the traversal of the tree on an update
The Composites

- Maintain a list of children nodes
- Update by updating the children nodes (usually in a particular order)
- Return RUNNING if a child returns RUNNING
  - Need to remember what child was running
- Return SUCCESS/FAIL under other circumstances depending on the type of composite
The Selector

• On update, updates each of its children in order until one of them *doesn’t* fail
  – Hence “select”, as this child has been “selected”
• Returns FAIL only if all children fail
• Kind of like an if else statement or block of ||’s
  – If child 1 succeeds, else if child 2 succeeds, etc…
The Sequence

• On update, updates each of its children in order until one *does* fail
• Returns SUCCESS if the entire sequence completes, else FAIL
• If one behavior fails then the whole sequence fails, hence “sequence”
Note about Composites

- **Sequences** start updating from the previously **RUNNING** child
  - Children state should be left intact after returning, unless the entire sequence was completed
  - Goal is to complete the entire sequence — “I was in the middle of something and should continue where I left off”

- **Selectors** should always update from the first child
  - Should reset the previously running child if a child before it starts running
  - Children have priority — “I should always go back to defend my base, even if I’m in the middle of an offensive sequence”
Other Nodes

• Wrappers contain a single child and modify its behavior. Examples include:
  – Invert condition
  – Repeatedly update child X times until FAIL or SUCCESS

• Random Selectors update its children in random order
  – For unpredictable behavior
  – Harder to debug though
Example

**Root Selector**

**Defend Sequence**
- Enemy Near? *Condition*
- Setup Defense *Action*

**Offense Sequence**
- Army Large Enough? *Condition*
- Go to enemy base *Action*
- Siege Base *Action*
Example

**Defend Sequence**
- Enemy Near? *Condition*
- Setup Defense *Action*

**Offense Sequence**
- Army Large Enough? *Condition*
- Go to enemy base *Action*
- Siege Base *Action*

Root Selector

update
Example

Root Selector

Defend Sequence
- Enemy Near? Condition
- Setup Defense Action

Offense Sequence
- Army Large Enough? Condition
- Go to enemy base Action
- Siege Base Action
Example

Defend Sequence
- Enemy Near? *Condition*
  - Setup Defense *Action*

Offense Sequence
- Army Large Enough? *Condition*
- Go to enemy base *Action*
- Siege Base *Action*

Root *Selector*

(update)
Example

- **Defend Sequence**
  - **Enemy Near? Condition**
  - **Setup Defense Action**

- **Offense Sequence**
  - **Army Large Enough? Condition**
  - **Go to enemy base Action**
  - **Siege Base Action**

Root Selector

update
Example

Defend Sequence
- Enemy Near? Condition
- Setup Defense Action

Offense Sequence
- Army Large Enough? Condition
- Go to enemy base Action
- Siege Base Action

Root Selector

update
Example

Defend Sequence

- Enemy Near? Condition
- Setup Defense Action

Offense Sequence

- Army Large Enough? Condition
- Go to enemy base Action
- Siege Base Action

Root Selector

update
Example

Defend Sequence

Offense Sequence

Enemy Near? Condition
Setup Defense Action
Army Large Enough? Condition
Go to enemy base Action
Siege Base Action
Example

Defend Sequence

- Enemy Near? Condition
- Setup Defense Action

Offense Sequence

- Army Large Enough? Condition
- Go to enemy base Action
- Siege Base Action

Root Selector

update
Example

[Diagram showing a tree structure with nodes labeled 'Root Selector', 'Defend Sequence', 'Offense Sequence', 'Enemy Near? Condition', 'Setup Defense Action', 'Army Large Enough? Condition', 'Go to enemy base Action', 'Siege Base Action']

- **Defend Sequence**
  - Enemy Near? Condition
  - Setup Defense Action

- **Offense Sequence**
  - Army Large Enough? Condition
  - Go to enemy base Action
  - Siege Base Action

**Root Selector**

(update arrow)

The diagram illustrates a decision-making process with conditions and actions for defense and offense sequences.
Example

Defend Sequence
- Enemy Near? Condition
- Setup Defense Action

Offense Sequence
- Army Large Enough? Condition
- Go to enemy base Action
- Siege Base Action
Example

Root Selector

Defend Sequence
- Enemy Near? Condition
- Setup Defense Action

Offense Sequence
- Army Large Enough? Condition
- Go to enemy base Action
- Siege Base Action
Example

Defend Sequence
- Enemy Near? Condition
- Setup Defense Action

Offense Sequence
- Army Large Enough? Condition
- Go to enemy base Action
- Siege Base Action

Root Selector
- update

update
Example

Root Selector

Defend Sequence
- Enemy Near? Condition
- Setup Defense Action

Offense Sequence
- Army Large Enough? Condition
- Go to enemy base Action
- Siege Base Action

(update)

(update)
Example

Update

Root Selector

Defend Sequence
- Enemy Near? Condition
- Setup Defense Action

Offense Sequence
- Army Large Enough? Condition
- Go to enemy base Action
- Siege Base Action
Example

Root
Selector

Defend
Sequence

Enemy Near?
Condition

Setup Defense
Action

Army Large Enough?
Condition

Go to enemy base
Action

Siege Base
Action

Offense
Sequence

update

update
Example

Root Selector

Defend Sequence
- Enemy Near? Condition
- Setup Defense Action

Offense Sequence
- Army Large Enough? Condition
- Go to enemy base Action
- Siege Base Action
Example

Root Selector

Defend Sequence
- Enemy Near? Condition
- Setup Defense Action

Offense Sequence
- Army Large Enough? Condition
- Go to enemy base Action
- Siege Base Action
Example

Defend Sequence
- Enemy Near? (Condition)
- Setup Defense (Action)

Offense Sequence
- Army Large Enough? (Condition)
- Go to enemy base (Action)
- Siege Base (Action)

Root Selector
Data Persistence

• Your behavior tree nodes might need to communicate somehow
  – Finding a target, going to the target are separate nodes

• How to share data?

• Blackboard: shared object that holds information, that nodes can write and read from
  – Minimally, a map<string, string>

• Certain groups of nodes can share different blackboards
Game State

• Your behavior tree also needs to interact with the game state
  – Game world, or a unit, or a group of units

• Make your nodes inner classes of your units
  – Your unit can be used as the blackboard
  – You can only affect the game world as much as the unit can

• More complex or global behavior might need a world reference or be in the world
Can they be used for more?

- Sprite drawing/animations?
- Handling input events based on current game state
- Basically any place you would have a giant nested if else block
Behavior Trees: Pro/Con

• Pros:
  – Natural structure for decision making
  – Somewhat lightweight engine implementation
  – Node variety can make things easier for the game code

• Cons:
  – Still some combinatorial explosion
  – How fine grained should an individual node be?
    • Tons of individual conditions/actions?
    • General “leaf” nodes that do both?
  – Can degenerate into a strange FSM
QUESTIONS?
AI

GOAP
Planning challenges

• Combinatorial explosion of world states
  – Need to simplify representation of world
  – Only model what is relevant to a specific agent

• Efficiently searching through possible actions
  – Can model as graph search problem

• Biasing towards certain actions
  – Weights on graph
Case study: F.E.A.R

- 2005 horror FPS
- Ranked #2 Most influential AI Game
F.E.A.R. AI

• Uses GOAP (Goal-Oriented Action Planning)
  – Consists of goals and actions
  – Goal: certain state of the world we want to reach
  – Action: set of preconditions and effects

• Solved using graph search
  – Nodes are world states and edges are actions
  – Search from current state toward goal
  – Edges are directional (directed graph)
  – Analogy to chess AI: nodes are board states and edges are moves
GOAP example

• World state:
  – bool enemyHasGun
  – bool playerAlive

• Only models states relevant to agent

<table>
<thead>
<tr>
<th>Current state</th>
<th>Goal state</th>
</tr>
</thead>
<tbody>
<tr>
<td>enemyHasGun = false playerAlive = true</td>
<td>enemyHasGun = true playerAlive = false</td>
</tr>
<tr>
<td>Drop gun</td>
<td>Shoot player</td>
</tr>
<tr>
<td>Pick up gun</td>
<td></td>
</tr>
</tbody>
</table>

Pick up gun
Shoot player
Drop gun
Pick up gun
GOAP

• Different from FSMs
  – FSM: Spend all time in nodes (actions), transitions are instant
  – GOAP: Spend all time in transitions (actions), nodes are instant

• Advantages over FSMs
  – Don't need to encode all possible transitions from each action to each other action
  – Simpler to specify and easier to scale
  – Just specify preconditions and effects for each action
GOAP in F.E.A.R.

• Graph is dynamic
  – Edges are actions with preconditions and effects
  – Edges come and go based on game state
• Procedural preconditions
  – Taking an action may be currently impossible
  – Example: Escape through door requires door unlocked
• Procedural effects
  – Effects take time to execute and may fail
  – Example: Firing a projectile is blocked by something
GOAP in F.E.A.R.

• Don't want all agents to act the same
  – Vary edge weights based on individual preference (aggressive vs careful individual)
  – Need variety in available actions (run, crouch, dodge, roll, slide)

• Hard for player to understand AI's behavior
  – Add audio cues for player
  – "Send in reinforcements!"
 QUESTIONS?
AI

OTHER STUFF
Influence maps

- Goal: decide which team controls an area
  - Used as an input to higher level decision algorithms
Influence maps

• Details
  – Numeric value that varies throughout the world
  – Positive for team A, negative for team B
  – Magnitude indicates strength of influence
  – Can also encode other information: safety, congestion, etc...

• Advantages
  – Entire influence map not updated immediately, will "remember" recent history
  – Helps reason about strategy in complex worlds (open areas, choke points)
  – Commonly used in RTS games
Influence maps

- Algorithm
  - Influence map starts off at zero
  - Sources of influence are marked
    - Entities (friends and foes)
    - Events like grenade explosions, taking damage
  - Influence is propagated (diffused) to neighbors
- Parameters
  - Momentum (m): How fast do new values overwrite old values?
  - Decay (d): How fast do values decrease from the source?
Influence maps

- Influence maps can be combined with graphs
Genetic algorithms in games

• Hardly ever used
  – Developers want fine control over behavior, often just enumerate all the cases
  – Difficult to test
  – Take thousands of iterations to converge on “interesting” behavior

• Creating faster genetic algorithms is an active area of research
Case study: Learning bots in Quake

- Remco Bonse et. al. in 2004
- Use the Q-learning algorithm to train a neural network for an AI bot in Quake III
  - One-on-one game against preprogrammed AI
  - Replaced combat movement subsystem with NN
  - Reward is given for avoiding damage
- State:
  - Distance and angle of opponent
  - Distance and angle of nearest projectile
- Actions:
  - All 18 combinations of WASD + jump
Case study: Learning bots in Quake

- **Q-learning algorithm:**
  - Agent is in some state $s \in S$
  - Agent can take some action $a \in A$
  - State/action quality function: $Q : S \times A \rightarrow \mathbb{R}$

- Agent gets reward for each state change
  - Make a correction to $Q$ based on new information
  - Learning rate: 0 = learn nothing, 1 = ignore past
  - Discount factor: 0 = short-term, 1 = long-term

\[
Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t(s_t, a_t) \times \left[ R_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]
\]
Case study: Learning bots in Quake

• Best parameters were to play to 100,000 frags on a network of 15 neurons (10 hours training time)
• Learned bot wins twice as often against a preprogrammed bot
• Human players didn't notice a difference
• Limited interaction time in a FPS
Randomness and AI

• One motivation for AI is variability
• Adding randomness can help
  – Easy way of varying AI behaviors
  – Need to be careful about how users perceive randomness
Randomness and human perception

• Which is more random?

50% 50%

or

50% 50%
Randomness and human perception

• Humans are illogical
  – Conditioned to see patterns
  – Patterns are more memorable

• Gambler's Fallacy
  – We think outcomes depend on past events for independent events with fixed probabilities
  – Not just gamblers, everyone does this
  – "His luck will run out sooner or later"
Law of Large Numbers

- Average approaches the expected value as the number of trials increases

Unfair in the short term
Law of Small Numbers?

• Humans expect random binary choices will alternate up to 40% more than is mathematically reasonable
  – Example: If a player loses a game involving randomness 4 times in a row, is that a good experience?

• Make randomness easier to understand
  – Random choices without replacement
  – Keep track of past choices and prevent them from being chosen again too soon
QUESTIONS
LECTURE 13

Final4 Tips
Playtesting

• Our form is minimal
  – Read: “not great”
  – Basically just demoing — no critical feedback
  – Real playtesting is more thorough

• Impossible to make a general playtest procedure
  – Cater it to your game!

• Extra Credits!
Playtesting this week

• 5 playtest signatures per person
• We recommend trying to find a few strangers
  – Friends-of-friends in the Sunlab are a good bet
  – Don’t be too pushy – it’s near finals period!
The procedure

• Totally dependent on what your game is, what state it’s in next week, etc…

• Main purpose:
  – So you can learn specific things about your game
  – So each player gets the same *controlled* experience
    • Like running an experiment
    • Limit confounding variables
  – So both of you take things more seriously

• Goal: one playtest session lasts ~15 minutes

• We require a few specific sections…
“The Speech”

• Introduce the game!
  – *Very* quick description of the game
  – Put the player in the right state of mind
  – Try not to give too many comparisons – let them form their own expectations!

• Like explaining a research experiment to a new subject
“The Manual”

• Your job is to watch and listen
  – Not to tell them exactly what to do
  – Let your game speak for itself

• Controls should be either
  – Canonical (WASD movement on keyboard)
  – Very easy to learn

• Either way, written instructions should be enough
  – If they aren’t, you probably have too many/complicated controls

• OK to answer questions during play, but breaks the player’s immersion
“The Experience”

• The gameplay you expect the player to go through
  – Series of levels
  – Series of actions
  – Like a tutorial!

• Don’t explicitly share this with them!
  – “You’ll be playing a few levels” at most
  – Take notes as they progress through these parts
  – Think of it like the checklist for their play session
“The Survey”

- **Written survey questions**
  - Make a google form!
  - Write detailed questions about the play experience
  - Try to get some numbers!
- **Conversational questions**
  - Do these after the written survey!
  - Helps avoid bias by talking to them
Final 3 Playtesting