Lecture 4

Announcements
Wiz 1 Feedback

- Multi-directional movement
  - Probably using an if-or block for each key input
  - Use one if-or for vertical, one if-or for horizontal

- Different animations with different directions looks nicer too

- Movement should happen onTick rather than onKeyPressed/Typed
  - Set the keys that are down when they're pressed
  - onTick, look at the booleans and see which are true

- Viewport centering jank when you run into a wall?
  - Have the CenterComponent center on lateTick() instead of tick()
Get Ready for Wiz 2!

- We’re adding smart enemies!
- Actually challenging to play
- Even more fun than Wiz 1!
QUESTIONS?
Think Pair Share

- How does a game developer use your engines level generator / level loader?

- How are frames of an animation loaded and displayed.
Lecture 4

Pathfinding
Why is pathfinding important?

- NPCs need to navigate an environment that has obstructions.
- Goal: find minimum cost path from A to B
  - Cost includes factors such as distance, terrain, position of enemies.
- Typically uses a graph to represent navigable states.
Pathfinding

DIJKSTRA’S ALGORITHM
Dijkstra’s

- Process nodes in order of shortest distance from start
- To process a node, update cost to each of its neighbor and add them to a `PriorityQueue`, then never process this node again
- Each node in `PriorityQueue` keeps track of shortest distance to it and pointer to previous node
- When you process the end node, you’re done!
Why Dijkstra’s can be gross
Pathfinding

A*
General idea

- Greedy Dijkstra’s
- Dijkstra’s assumes it’s impossible to predict cost
  - This is overly pessimistic
- In pathfinding, we at least know the general direction we want to go
- A* is a graph traversal algorithm that takes advantage of this
Why A* is better
How does it work?

● Uses a “heuristic” to guess the cost from any given node to the destination node
  ○ Heuristic passed by the caller
● In addition to tracking distance from start, track heuristic value for each node
  ○ Prioritize in PriorityQueue based on \( f(node) = distance_{to}(node) + \text{heuristic}(node) \)
● Heuristic can be as simple as the Euclidean distance between the given and destination node, but also try to take other factors
  ○ Get creative – better heuristics help A* run faster!
Things to keep in mind

- You could have a bad heuristic that points you in the wrong direction.
- Experiment!
- You may use A* several times-- make it generalizable!!!
A* Pseudocode

- [https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2](https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2)
- [https://en.wikipedia.org/wiki/A*_search_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)
- Come into hours if you need help understanding the algorithm
- Please come to hours… we are so lonely…
Pathfinding

PATHFINDING IN WIZ
What do you need?

- Graph of accessible space
- A* implementation
- That’s pretty much it
The Graph

- Can’t include every point in the world
  - There are infinitely many
- Need discrete “waypoints” that units can move between
The Graph

- **One option:**
  - Build a graph of accessible waypoints, each with a position
  - Pathfind between waypoints by running A* on the graph
The Grid

- **Another option (recommended):**
  - Represent the graph as a 2D array
    - Each entry represents a position in the world
    - Each entry specifies whether or not that position is accessible
    - `getNeighbors(Vec2i pos)` function that returns accessible neighbors of a grid position
    - Literally just a graph with Cartesian edges
  - Can run A* in the same way
    - Yay for graph generalization!
The Grid

- Don’t have to manually specify all accessible positions, just inaccessible ones
- Also, easier to update from a game design perspective than waypoint graph
PathfindingComponent

- Hold onto a waypoint graph / grid of some sort
  - The graph / grid should be able to provide a path from point A to point B
- Hold onto a path
- Update position based on next location in path and current location
PathfindingSystem

- If you want a dynamic graph
  - Have a PathfindingSystem that updates the graph each tick based on game object positions and bounding boxes
Lecture 4
Decision Making
Decision Making

MOTIVATION
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
- What are some examples of good AI that you have seen?
- Any bad examples?
Decision Making

● NPCs 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…
Lecture 4

Behavior Trees
Behavior Trees

- “Recently” popularized by Halo 2
- Core functionality is engine-general!
  - Big plus!
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 or Conditions
- **Actions** do things
  - e.g. make a unit move or attack
  - Return SUCCESS or FAIL based on result of Action
  - Return RUNNING if Action is still in progress
- **Conditions** check some game state
  - Returns SUCCESS if the condition is true, or FAIL if the condition is false
The Internal Nodes

- Internal nodes are **Composites** or **Wrappers**
  - Composites have multiple children nodes, and dictate traversal during an update
  - Wrappers wrap a single child node; more on this later
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
- Return SUCCESS/FAIL under other circumstances depending on the type of composite
The **Selector**

- A concrete *Composite* node
  - 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 or’s
  - If child 1 succeeds, else if child 2 succeeds, etc…

Make dinner plans

- [Denden’s](#)
- [Subway :(](#)
- [Kabob and Curry](#)
The Sequence

- Another concrete Composite
  - On update, updates each of its children in order until one *does* fail
  - If one behavior fails then the whole sequence fails, hence “sequence”
- Returns SUCCESS if the *entire* sequence completes, else FAIL
- Similar to a bunch of and’s
Other Nodes

- **Wrappers** contain a single child and modify its behavior. Examples include:
  - Invert child
  - Repeatedly update child \(X\) times until FAIL or SUCCESS
- Randomize **Selectors** to update its children in random order
  - For unpredictable behavior
  - Harder to debug though
- Not required for Wiz, but feel free to play around!
BehaviorTree Nodes

- Just needs to be updated and reset
- Sample contract:

```java
interface BehaviorTreeNode {
    Status update(float seconds);
    void reset();
}
```
Composites

- Needs a list of children
- Also should keep track of what child was running
- Sample contract:

```java
class Composite implements BehaviorTreeNode {
    List<BTNode> children;
    BehaviorTreeNode lastRunning;
}
```
Note about Composites

- **Sequences** start updating from the previously **running** child
  - Previously running child 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
  - Earlier children have priority – “I should always go back to defend my base, even if I’m in the middle of an offensive sequence”
Behavior Trees

QUESTIONS?
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

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*
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*
- Army Large Enough? *Condition*

Offense Sequence
- 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*
Example

**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
Example

Offense Sequence

Enemy Near? Condition
Setup Defense Action

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

Defend Sequence

Update

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*
Example

Offense Sequence

Defend Sequence

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

Root Selector

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

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

Offense
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

**Offense**

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

**Defend**

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

**Root Selector**
Example

- **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*
Data Persistence

- Your behavior tree nodes might need to communicate somehow
  - Finding a target and 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, wrapper for a `Map<String, ???>`
- Groups of nodes can share different Blackboards
In Summary

● Interfaces/abstract classes for:
  ○ BTNode
  ○ Composite
  ○ Condition/Action

● Full classes for:
  ○ Sequence
  ○ Selector
  ○ Other wrappers

● Game-specific classes extend Condition/Action
Behavior Trees

QUESTIONS?
Lecture 4

Goal Oriented Action Planning (GOAP)
Issues with BTs

● Behavior trees aren’t perfect
  ○ Lots of enemies
  ○ Too much work to code each
● Minor tweaks change a lot of code
● Procedurally generated enemies
  ○ Behavior trees usually aren’t expressive enough for complex behavior
What is GOAP?

- Goal oriented action planning
- e.g. What’s the fastest way to kill the player?
What is GOAP?

- GOAP is a graph of game states
- We can search over it with A*
The Nodes

- Each node is a `GameState`
- `GameState` are probably a map of string tags to booleans or integers
- The tags and their meaning are determined game-side

```java
class GameState {
    Map<String, Integer> _props;
}
```
The Edges

- Each edge is an Action the AI can take
- Each Action has a cost and a Condition
- Actions also change the GameState

```java
public abstract class Action {
    private List<Condition> _conditions;
    private double _cost;

    public abstract void changeState(GameState s);
}
```
Planning

● Goal
  ○ Generate a plan or “path” of actions
  ○ This plan should take you from start state to end state
● Just use A*!
Planning

- Start at a state
- Add neighboring states to priority queue
  - Go through all Actions (edges)
  - All Actions whose conditions are true from the current state are allowed
  - Generate a neighbor for each by applying the corresponding action to a copy of the GameState
- Pop lowest cost state from the priority queue
- Continue
- Return “path” or list of actions that took you from start to end state
Actions

- Just like behavior trees, GOAP has actions
- **Actions** are much simpler in GOAP
  - Change one or more of the tags in the game state
Conditions

- Just like behavior trees, GOAP has conditions
- **Conditions** are also much simpler
  - Return true or false
  - Determined entirely by `GameState`

```java
public abstract class Condition {
    public abstract boolean isMet(GameState s);
}
```
GOAP

- The game defines a start state based on the current game world
- The game also defines a goal (Condition)
- Once the search is done, you need to map the list of actions to some real game effect
- Usually only the first action is executed before GOAP is run again
  - The action might not be completed before a new plan is generated
  - E.g., following the player
Problems

- Depending on the *Actions* available, GOAP can generate an infinite graph without any goal states.
- This can be handled by any of the following:
  - Allow each *Action* to be used once/max # of times.
  - Specify a maximum cost.
Problems

- With lots of actions and a distant goal, GOAP can be really slow
- GOAP is best used to solve small problems
Problems

● GOAP optimizes over a single parameter (time, cost, etc.)
● GOAP is good for short, discrete problems:
  ○ Which combo should I use?
  ○ Which route should I take?
● GOAP is bad for long-term, strategic problems:
  ○ How do I optimize my economy?
  ○ Which item will maximize my options next level?
Mix and Match

- Behavior trees and GOAP don’t have to be mutually exclusive
- Behavior tree can determine the strategy (setting up which actions are available, how much each is weighted, what the goal is, etc.)
- GOAP can determine the plan to execute that strategy
- Behavior tree turns that plan into concrete actions
- e.g., sequence
Lecture 4

Tips for Wiz II
Floating Elements

- “Floating” elements are common
- We recommend having your viewport support these!
  - Determine where on the screen a point in the game is
  - Draw element at screen scale
  - Consider a second `draw()` call responsible for floating elements
lateTick

- Remember this method from Lecture 1?
  - Call on Systems, GameObjects, or Components!
- Used for actions that need to be executed once all GameObjects have been ticked and collided
- Example: updating the position of the viewport
  - Avoids viewport center lagging behind player
Good things to include...

● Easy mode/cheat settings
  ○ TAs need to be able to beat your game to grade it
  ○ Also, easier for you to debug…
● Diagonal movement is cool
● Wide doorways: make sure you can easily traverse your dungeon
● Use tiled sprites for your dungeon!
Tips for Wiz II

QUESTIONS?
Overview

● Can be any 2D game
● You should work in groups!
  ○ But feel free to work alone.
● Each person is responsible for 10 “points” worth of new engine features
● More members in a group means more engine features
● More details in the final project handout
Timeline

- 4 main parts:
  - **Week 1**: Idea
  - **Week 2**: Form groups and get approved
  - **Week 3**: Design
  - **Weeks 4-7**: Code, demos, polish, present
Week 1: Idea (this week!)

- A few paragraphs submitted via Google Forms (link in Wiz II handout)
- Describe basic gameplay idea
  - How is your game fun?
  - Describe engine feature(s) you plan on implementing
  - Give a 60-second “elevator pitch” of your game in class next week
- Everyone should submit an idea!
Week 2: Groups

- Form a group (or decide to work alone)
- Finalize game and engine features
- Each group must meet with the TA’s to present their ideas
Week 3: Design

- Research new engine features
- Design the engine and game
- Decide exact breakdown of member responsibilities
- Choose someone’s engine to use
  - We don’t recommend merging engines, but you can try if you want
- We’ll set up GitHub repositories for you
Weeks 4-5

- **Week 4:**
  - Engine should be mostly done

- **Week 5:**
  - Engine should be done
  - Game is playable (barebones is fine)
  - 2 playtests per member from people not in CS1971
    - Let us know if this will be difficult for you
Weeks 6-7

● Week 6:
  ○ Game should be more done
  ○ 2 more playtests per member from outsiders

● Week 7:
  ○ Game should be done
  ○ Fix bugs, polish up gameplay
  ○ 3 playtests per member
  ○ Get ready to record a demo video!
Final Project Overview

QUESTIONS?
‘Til Next Week!

- Wiz II & Extras released today!
- See handout for FP proposals
- Remember to upload your demos