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 your boolean array and see which are true
Clipping

• At this point, all of you have commented on FPS issues resulting from g.clip()
• We will have a solution for you later this week
• Once the solution is released, you are REQUIRED to clip
Course Administration

• Things sometimes go wrong
  – Our email script is particularly temperamental

• Grades are released by Thursday
  – Email us if you don’t get a grade report by then
Get Ready for Wiz 2!

• We’re adding smart enemies!
• Actually challenging to play
• Even more fun than Wiz 1!
QUESTIONS?
Lecture 4

Pathfinding

Hang in there!
Pathfinding

MOTIVATION
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.
Dijkstra’s

- Basic idea:
  - 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

- 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
    • \[ f(node) = \text{distance\_to}(node) + \text{heuristic}(node) \]

• 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


• 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-star 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 contd.

- 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
  - Can run A* in the same way
    - Add neighbors to queue
    - Pop lowest cost neighbor from queue
    - Continue
The Grid contd.

- Don’t have to manually specify all accessible positions, just inaccessible ones
- Easier to update than waypoint graph
PathfindingBehavior

• 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 / grid
• Have a PathfindingSystem that
  – Updates the waypoint graph / grid each tick based on game object positions and bounding boxes
Pathfinding

QUESTIONS?
LECTURE 4
Decision Making

*It's possible that you might have a problem*
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
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…
Lecture 4
Behavior Trees
Behavior Trees

• “Recently” popularized by Halo 2
• 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 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 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
- 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 or’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”
- Similar to a bunch of and’s

Implement Wiz

- Implement viewports
- Implement the rest
- Implement better viewports
Other Nodes

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

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

• Not required for Wiz, but feel free to play around!
Behavior Tree Node

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

```java
interface BTNode{
    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 BTNode {
    List<BTNodel> children;
    BTNodel 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
  - 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

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*
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
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

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

**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

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
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
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

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
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**

(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

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

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

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
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, ???>`
• Certain 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 extending Condition/Action
Behavior Trees

QUESTIONS?
LECTURE 4

Greater Dog.

Goal Oriented

Action Planning
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
What is GOAP?

• Goal oriented action planning
• What’s the fastest way to kill the player?
What is GOAP?

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

- Each node is a GameState
- GameStates 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 float _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 contd.

• Start at a state
• Add neighboring states to priority queue
  – Go through all actions
  – 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 game state
• Pop lowest cost state from 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
Conditions

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
Goal Oriented Action Planning

QUESTIONS?
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.
Goal Oriented Action Planning

QUESTIONS?
Lecture 4

Tips for Wiz 2
Floating Elements

• “Floating” elements are common

• Make sure your viewport supports these!
  – Determine where on the screen a point in the game is
  – Draw element at screen scale
  – Consider a second onDraw() call (maybe “onUI”) responsible for floating elements
  – How can you “unproject” a unit’s location?
lateTick

• Expanded contract
  - Systems, objects, behaviors!
• Used for actions that need to be executed once all game objects have been ticked and collided
• Example: updating the position of the viewport
  - Avoids viewport center lagging behind player

public void lateTick(float seconds);
Tips for Wiz

QUESTIONS?
Lecture 4
Final Project Overview
Overview

• Can be any 2D game
• You should work in groups!
• 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 4-8: Code, playtest, polish, present
Week 1: Idea (this week!)

• A ½ page document
• Describe basic gameplay idea
  – How is your game fun?
  – Why should someone want to help make it?
• Describe engine feature(s) you plan on implementing
• Give a 60-second “elevator pitch” of your game in class next week
• Everyone must give a pitch, even if you already know your group and which project you’re working on
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
- Meetings will be next week
- If this time doesn’t work, please email us
Week 3: Design

• Research new engine features
• Design the engine and game
• Exact breakdown of member responsibilities
• Choose someone’s engine to use or integrate engines
• Explain how you will use version control
Weeks 4-8

• Week 4:
  – Engine should be mostly done

• Week 5:
  – Engine should be done
  – Game is playable
  – 3 playtests per member from people not in CS1971
Weeks 4-8

• Week 6:
  – Game should be more done
  – 3 more playtests per member from outsiders

• Week 7:
  – Game should be done
  – 3 playtests per member
  – Powerpoint slideshow for postmortem presentation
Weeks 4-8

• Week 8:
  – Finish/polish up your game, bug fixes, etc
  – Create a jar and script, and put it in /contrib
  – Make a video demo of your game from gameplay footage

• And then you’re done!
Final Project Overview

QUESTIONS?
Wiz 1 Playtesting
The most exciting part of the day!