Tac 2 Feedback

• Email your grader when you want your retries graded!
  – We won’t look at any retry handins until you email us

• Even if you’re not done, hand in something!
  – Knowing exactly what you’re missing makes getting the standard retry correct a lot easier
Tac 2 Feedback

• Zooming sensitivity
  – Most of you need to make it more sensitive
  – Remember to multiply your scale value instead of adding to it (refer to the Viewport lecture)

• Panning should be relative to your zoom level
  – More zoomed out means panning should be faster

• Looking good!
  – Playtesting should be a bit more fun today!
Get Ready for Tac3!

• Real gameplay!
• Your own gameplay ideas!
• Actually fun!
• Not tic tac tou!
• Weird requirements!
LECTURE 3
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
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…
Decision Making

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”
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

• Not required for Tac, but feel free to play around!
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

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
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
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**

**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
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

Army Large Enough? Condition
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
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

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

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
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
Behavior Tree Node

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

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

• Why do we pass in the time?
Composites

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

```java
class Composite implements BTNode {
    List<BTNode> children;
    BTNode lastRunning;
}
```
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”
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>`
  – Could be more cleanly done with generics

• 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
In Summary

- BTree internal interfaces/abstracts classes for:
  - BTNodel
  - Composite
- BTree abstract classes for:
  - Condition
  - Action
- BTree full classes for:
  - Sequence
  - Selector
  - Other wrappers
- Game-specific inner classes extending Condition/Action
  - Automatically have access to object state
Can they be used for more?

- Sprite drawing/animations?
- Handling input events based on unit selections
- Basically any place you would have a giant if else block
QUESTIONS?

Behavior Tree AI
UI: Part 2

• “Floating” elements are common
• Make sure your viewport supports these!
  – Determine where on the screen a point in the game is
  – Consider a second onDraw() call responsible for floating elements and triggered by your viewport
  – How can you “unproject” a unit’s location?
Constants: The “Old” Way

• Each value is assigned a constant int

• A few issues...
  – No typechecking
  – No namespace
  – No printout

```java
public static int STATE_STARTUP = 0;
public static int STATE_ACTIVE = 1;
public static int STATE_TRANSITION = 2;
public static int STATE_SHUTDOWN = 3;
public static int STATE_DEAD = 4;
```
All About Enums

- An **Enum** enumerates all possible values for a type
- Uses labels instead of arbitrary indices to make code clearer

```java
public enum SEASON {
    WINTER,
    SPRING,
    SUMMER,
    AUTUMN
}
```
Enums as a Class

• Acts like any other class, with fields, constructors, etc
• Switch statement shortcuts
  – Warnings when you forget a case
• `EnumSet` and `EnumMap` for speedy operations
• Good tutorials on all features available from Java.com
public enum Action {
    OPEN_PROJECT ("Open...", KeyEvent.VK_O),
    QUIT_APP ("Quit", KeyEvent.VK_Q),
    SAVE ("Save", KeyEvent.VK_S),
    CLOSE_PROJECT ("Close", KeyEvent.VK_C),
    NEW_ENTRY ("New Entry...", KeyEvent.VK_N),
    DELETE_ENTRY ("Delete Entry", KeyEvent.VK_D);

    private String displayName;
    private int vk;

    Action(String displayName, int vk) {
        this.displayName = displayName;
        this.vk = vk;
    }

    public String toString() { return displayName; }
    public int getVK() { return vk; }
}
Functions aren’t objects...

• In some languages, functions are first class objects
• In Java this is not the case (except Java 8)
• So functions (and arbitrary code) can’t be stored so easily

```java
void map(List list, Function lambda) {
    for (Object o : list) {
        lambda(o);
    }
}

// in your code...
map(numbers, f(x) = { print(x); });
```
Introducing Anonymous Classes

- Essentially an in-line subclass
  - Looks exactly like a subclass in another file, minus the name

- All anonymous classes are inner classes
  - And therefore have a reference to their superclass

```java
interface Mappable {
    abstract void run(Object o);
}

void map(List list, Mappable lambda) {
    for (Object o : list) {
        lambda.run(o);
    }
}

map(myList, new Mappable() {
    void run(Object o) {
        doCoolStuff(o);
    }
});
```
Use: Adapter Classes

• Adapters pass some code or function to another object
  – Hey, this sounds a lot like a behavior tree node…
• Subclass them in-line to avoid having to write new classes each time

```java
public class Tank {
    public ActionNode shootNode = new ActionNode() {
        public State run() {
            shoot();
            return SUCCESS;
        }
    };

    private void shoot() {
        // … code to shoot a bullet
        board.addBullet(...);
    }
}
```
Use: Listener Classes

- Listeners wait for events from other objects
- Subclass them in-line to avoid writing new designated listeners
- A lot more elegant than having a logic class implement ten interfaces!

```
// inside, say, a JPanel somewhere
addActionListener(new ActionListener(){
  actionPerformed(ActionEvent e) {
    // handle the action as normal
  }
});
```
Some Restrictions

• No way to use extends or implements
  – You’re implicitly doing this anyway!

• No non-final static fields
  – Changed in Java 8 for no good reason!

• No constructors
  – Instance initializers exist for exactly this reason!
Tips for Tac

QUESTIONS?
GAME DESIGN 2

Controls
Why are controls important?

• In games, the player interacts with some game world
• Controls are the player’s interface for interaction with the game world
• Can make or break your game!
A Brief History of Game Controllers

• Atari 2600 (1977)
  – 1 Button, 1 Stick
• NES (1983)
  – 4 Buttons, 1 D-Pad
• SNES (1990)
  – 8 Buttons, 1 D-Pad
• Playstation Dualshock (1997)
  – 10 Buttons, 1 D-Pad, 2 Sticks
  – Still the basic standard today
The Keyboard and Mouse

• Primary controller for PC games
• Strong for games with complicated controls (RTS, MMO, etc.)
• Core challenges:
  – Tons of buttons: easy to overwhelm the player
  – Some genres don’t make sense: flight simulators, fighting games
Good Controls: 3 Core Principles

Good controls must be:
1. Intuitive
2. Ergonomic
3. Agentive
Principle 1: Intuitive

• Intuitive controls:
  – Are easy to pick up and learn
  – Make sense for the game being played
  – Require little cognitive effort from the player
  – Follow pre-existing conventions when possible
Principle 1: Intuitive

General conventions:

<table>
<thead>
<tr>
<th>Button</th>
<th>RTS</th>
<th>MMO</th>
<th>Shooter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left Click</td>
<td>Select units</td>
<td>Select target</td>
<td>Fire</td>
</tr>
<tr>
<td>Right Click</td>
<td>Move to/attack units</td>
<td>Attack target</td>
<td>Secondary fire/hold aim</td>
</tr>
<tr>
<td>WASD</td>
<td>Pan camera</td>
<td>Movement</td>
<td>Movement</td>
</tr>
<tr>
<td>Space</td>
<td>Focus on event</td>
<td>Jump</td>
<td>Jump</td>
</tr>
<tr>
<td>Mouse Wheel</td>
<td>Zoom camera</td>
<td>Zoom camera</td>
<td>Cycle weapons</td>
</tr>
<tr>
<td>Number keys</td>
<td>Hotkeyed units</td>
<td>Special skills</td>
<td>Select weapon</td>
</tr>
<tr>
<td>Shift (held)</td>
<td>Modify click actions</td>
<td>Use second skill set</td>
<td>Sprint</td>
</tr>
</tbody>
</table>
Principle 2: Ergonomic

• Ergonomic controls:
  – Have a consistent “home-base” position
    • e.g. one hand on mouse, one hand on WASD
  – Place most used buttons at or around home-base
  – Demand as little movement as possible from the player
Principle 2: Ergonomic
Principle 2: Ergonomic

Blue: Home Base

(No Movement Necessary)
Principle 2: Ergonomic

Cyan: Most Common Actions

(Minimal Finger Movement)
Principle 2: Ergonomic

Green: Common Actions

(Some Finger Movement)
Principle 2: Ergonomic

Purple: Held Actions

(Pinky movement for Shift and Ctrl, Thumb for Alt)
Principle 2: Ergonomic

Yellow: Infrequent Actions
(Some Hand Movement)

*Most players will already have learned muscle memory for ESC*
Principle 2: Ergonomic

Red: Use With Caution

(Full Hand Movement Required)
Principle 3: Agentive

- Agentive controls:
  - Give the player as much control as possible
  - Produce consistently predictable results
  - Respond as quickly as possible to player input
Principle 3: Agentive

- Common player complaints about controls that are not agentive:
  - “It feels laggy.”
  - “I can’t get it to do what I want.”
  - “Why doesn’t anything happen when I press the button?”
Principle 3: Agentive

- **Solutions:**
  - Make response time on every button press as quick as possible
  - Provide feedback when a particular action is not available
3 Principles: Recap

• Good controls are:
  – Intuitive (the player can pick them up quickly)
  – Ergonomic (the player can perform them easily)
  – Agentive (the player feels in control)

• These need to be kept in balance!
  – Ergonomic controls may not be intuitive and vice versa
Common CS1971 Control Pitfalls

• Not following conventions for no discernible reason
  – Movement on IJKL

• Assigning actions to the keys of the first letter they start with
  – J for jump, L for laser, F for fire

• Using modifier keys unnecessarily
  – Left click to select unit, Shift + left click to move, Ctrl + left click to attack
Exceptions?

• It’s acceptable to break these rules when the controls are the intended source of difficulty for the game.

• Popular examples:
  – QWOP
  – Surgeon Simulator 2013
Demo: Surgeon Simulator
Game Design Tips for Tac 3

• Follow the conventions of the genre!
  – Typically, left click to select and right click to move and attack
  – If you’re not sure, steal from well known games
• For TBS games: visualize how far a unit can move
• For RTS games: smooth panning instead of click+drag
  – Either edge of screen or WASD/arrow keys
• Think from the player’s perspective!
  – (recap from last lecture)
TAC 2 PLAYTESTING!
The most exciting part of the day!