WELCOME!
Introduction

TA STAFF
A WORD FROM OUR SPONSORS
Introduction

COURSE OVERVIEW
Class goals

• Build a 2D game engine from scratch
• Build games on top of your engine
• Improve your software engineering skills
Class times

• Class: Wednesday 3:00p-5:30p
  • Lecture: ~ hour (CIT 241)
  • Playtesting: ~30 minutes (MS Lab)

• Design Checks/TA Hours:
  • Thursday 8-10
  • TBD
  • Sign up using cs1971_signup <project>

• cs.brown.edu/course/cs1971/
Introduction

THE FIVE PROJECTS
Tic

- Tic-Tac-Toe
- UI-only game
- Due next week
Alchemy

- Little Alchemy
  - Puzzle game!
  - Combining different elements to make new ones
- 2 week project
- Viewports, content management, simple collisions
Wizardry

- 2D dungeon crawler
- 2 week project
- Map generation & loading, AI, pathfinding
Ninja

- Platformer
- 2 week project
- Polygon collisions, physics, level loading, polished UI
Final Project!

- You choose everything! Engine features, game features-- the only limit is your imagination
- Groups recommended
- Public playtesting
- More details later
Introduction

GRADING
Our Grading System

• Only projects
• No homework or exams
• Every project is broken down into weekly checkpoints
  • Handins due every Tuesday at 11:59 PM
  (Sunlab at 11:59PM on most days)
Our Grading System Cont.

• For each project, you have...
• Playtesting requirements:
  – Basic, playable demo
• Primary requirements:
  – Major features
• Secondary requirements:
  – Less important features
Design Checks

• Sign up with cs1971_signup <project>
• High-level conceptual questions
• Gives one standard retry
  which brings us to…
The Retry System

• Standard Retry
  • As long as you complete a design check, you are allowed to re-hand in a checkpoint

• Extra Retries
  • You have two for the whole class
  • Used instead of or in addition to standard retries

• You have a week to use each retry
Incomplete handins

- Playtesting requirements cannot be retried
- No credit for secondary requirements until primary requirements are met
- Only your best handin will count (retries will never hurt your grade)
Final grades

• No curve!
  – Do the work, get an A

• 4 points per checkpoint:
  – Playtesting Reqs – 1 pt
  – Primary Reqs – 2 pts
  – Secondary Reqs – 1 pt

• You need to meet all engine requirements by the end of the
Please hand in on time!

• Falling behind causes a “snowball of death”
• Grading late handins puts stress on TAs
• If your handin is playable, hand it in even if you’re missing some reqs so you can be playtested
• If it isn’t, you have another week to retry
QUESTIONS?
In order to take this class, you must...

• Be comfortable with Java
  – CS15, CS18
Introduction

ABOUT REGISTRATION
Registering for cs1950N

• Courses At Brown
  – We’re “2D Game Engines”
  – May need an override code to register

• For non Brown students
  – Send us an email
  – cs1971htas@lists.brown.edu
CS1950N as a Capstone

• You can take this course as your capstone!
• Nothing extra for this week
• Extra engine features starting with Alchemy
• Broader final project requirements
Introduction

QUESTIONS?
LECTURE 0

Basic Engine Architecture

Hang in there!
Basic Engine Architecture

WHAT IS AN ENGINE?
What is an engine?

- The thing that games are built on
- Games tend to have a lot of functionality in common
- Solution: create engines that abstract out common functionality
- Implemented as a library or a framework
  - Yours is going to be a framework
What is an engine?

• Should be usable by many games
  • If you gave your engine to someone, could they write a game without modifying engine code?
• Should be general
  • No game-specific logic!!!
What does this look like?

• Sample package hierarchy:
  – src/
    • engine/
      – Screen.java
    • game/
      – TicScreen.java

• Any code in your engine package SHOULD NOT reference code in your game package.
Basic Engine Architecture

THE MOST ESSENTIAL INTERFACE
A game generally needs...

• Timed updates ("ticks")
• Ability to render to the screen ("draws")
• Input events (in some form or another)
• Knowledge that it has been resized (more info later)
Ticks

• General contract:
  • `public void tick(long nanos)`
  • Nanos is the most precision most computers have
  • Tip: Many people prefer to convert to `double` seconds
• Notifies the engine that a given amount of time has elapsed since the previous “tick”
  • This is very important
  • Nearly all logic takes place during “ticks”
• Updates per second (UPS) is how many ticks occur in a second
  • Keeps track of how smoothly the game world is updated
  • For our engine, UPS and Frames per second (FPS) are the same
  • We require 20 FPS in all projects
Draws

• General contract:
  - `public void draw(GraphicsContext g)`
  - Convert game state into viewable form
• Must be free of side effects!
  - Two subsequent draw calls should produce identical results
• More information coming up in Graphics I section
Input Events

• Most APIs provide input events rather than making you manually poll the mouse and keyboard
• Exact contract differs depending on type, ours follows the form:
  • public void onDDDEEE(DDDEvent e)
  • DDD = device type (e.g. mouse, key)
  • EEE = event type (e.g. moved, pressed)
• Event object contains information about the event
  • Where the mouse is; what key was pressed...
• More info coming up in Input section
Putting it all together

- Basic methods of a game application:
  - (note: support code calls these, you implement them)

```java
public class Application extends FXFromtEnd {
    public void onTick(long nanos)
    public void onDraw(GraphicsContext g)
    public void onResize(Vec2d size)

    public void onKeyPressed(KeyEvent e)
    // more device and event types...
    public void onMouseDragged(MouseEvent e)
}
```
Basic Engine Architecture

QUESTIONS?
Basic Engine Architecture

APPLICATION MANAGEMENT
Application Management

- How do we build an engine around drawing/ticks/events?
- It's very different depending on what’s going on!
  - Menus
  - The actual game
  - Minigames within game
Solution: Screens within Application

• Each game mode has a dedicated Screen

• A Screen has similar methods to the Application
  • onTick
  • onDraw
  • onResize
  • Input event methods
Keeping track of Screens

• Simplest way:
  • Single Screen in Application at a time
  • Current Screen sets the screen in Application

• Alternative way:
  • List or Map of Screens maintained by the Application
  • One active Screen gets events
  • Advanced: Transparent Screens can forward calls down to other Screens
What Screens shouldn’t do

• Draw the entire game
• Handle all of the game logic
• You will have serious spaghetti code if you do this
Solution: UI System

• For Tic...
• Implement a UI toolkit for drawing common UI elements (you will use this for all of your projects)
  – Buttons
  – Text
  – Container objects (not drawn, but have children)
  – Etc.
• Screens should draw their contents using the UI system
• A robust UI system now will save you lots of work later
Recommended Design

• Have a base `UIElement` class with similar methods to `Screen`
  • `onTick`
  • `onDraw`
  • `onResize`
  • Input event methods

• Additionally
  • Has children (other `UIElements`)
  • Has a parent (other `UIElement`)
  • Has a position (relative to parent)
  • Has a size

• Extend and override methods to create more specific UI
A Note on Game Logic

• We haven’t talked about how or where to handle game logic
• This will be covered next week!
• For now, build Tic on top of your UI system
• This week, you can hardcode game logic into your Tic Screen
A Note About `main`...

- Get out of it ASAP!
- Make a dedicated game class, not in the engine
- A wholesome, healthy `main` class is < 10 lines long:

```java
public class MyGameMain {
    public static void main(String[] args) {
        Application a = new Application("Tic-Tac-Toe");
        a.setScreen(new MainMenu(a));
        a.start(); // begin processing events
        // don't put code after start()!
    }
}
```
LECTURE
Graphics I

*It’s possible that you might have a problem
Graphics I

SCREEN SIZE
Long ago...

• The screen size of a game was hardcoded at a fixed resolution
  • Especially in consoles
• This allowed absolute sizing and positioning!
  • Ugly but effective!
• Modern games must support many resolutions
How do we know the screen size?

• There’s another method in `Application`...
  • `public void onResize(Vec2d newSize)`
  • `newSize` is the new width and height of the draw area

• Called when the size of the screen changes
  • Window resizes
  • Fullscreen is toggled

• You should store the current window size in your `Application`
Strategies for handling different sizes

• Blindly draw at fixed size anyway
  • Unacceptable, even if centered
• Better?: blindly scale to fit
  • Uses all space, but really gross
• Much better: scale up maintaining aspect ratio
  • This is what we expect you to do
Strategies for handling different sizes

• Best (not what you'll implement): distribute extra screen space to objects in the game as needed

• Not always possible
  • Especially if the size of the game area has impact on the
Reacting to resizes

• In Tic, the board must remain square
  • Scale up the board
  • Then scale up the other UI Elements
Graphics I

DRAWING THINGS
Window coordinate system

• Standard window coordinates:
  • Origin in upper left
  • X axis extends right
  • Y axis extends down
  • Convention initially set up to follow English text (left-right, top-bottom)
Actually drawing things

- **Draw methods in** GraphicsContext
  - Fill (fills in the entire shape)
  - Stroke (draws the outline of the shape)
    - fillRect(), strokeRect(), fillOval(), strokeOval(), etc.
- **Only use doubles for drawing**
  - Otherwise, future projects may be more difficult!
- **Colors**
  - There should be a constant in your UIElement which holds its color
  - Create colors with Color.rgb(int, int, int)
  - Change color for drawing with setStroke(), setFill()
Drawing text

- Set font with `g.setFont()`
- Set font color with `g.setColor()`
- Draw text with `g.fillText()`
- You can use our `FontMetrics` class to get the width and height of the text
QUESTIONS?

Graphics I
LECTURE 0

Greater Dog.

Input
Input

THE KEYBOARD
JavaFX KeyEvents

• Our support code listens for three key events:
  • onPressed
  • onReleased
  • onTyped

• What do those actually mean?
Then what is keyTyped?!  

- Fired when a character has logically been typed  
  - E.g. Shift+A results in one keyTyped event for a capital ‘A’ while A without shift results in ‘a’  
  - Especially nice for non-Latin characters  
- Ultimately only useful if implementing text input  
  - NOT useful for detecting key repeat!
Input

THE MOUSE
JavaFX MouseEvents

• Button events
  • onPressed, onReleased actually do what they advertise!
  • onClicked is when a “click” occurs—a press quickly followed by a release
    • Includes clickCount (2 for double click, 3 for triple etc.)
JavaFX MouseEvents

- Cursor position events
  - onMouseMoved when the cursor moves and no button is held
  - onMouseDragged when the cursor moves and at least one button is held
- Note that JavaFX only gives you one at a time, so if keeping track of the cursor position, listen to both.
General Input Advice

- Mess with `println`ing events for a while to get a sense of them
- Better understand their contracts
LECTURE 0
Tips for Tic and Beyond
Tips for Tic and Beyond

SOFTWARE ENGINEERING TIPS
Plan.

• You are about to embark on a large software adventure!
  • So make a map
• You will have to maintain the code you write, or rewrite it
  • Find weaknesses in your design before they ever become code
Program abstractly.

• Split your code into black boxes defined by contracts (interfaces)
  • For example, have a concept of a UI element that can resize and draw itself
• Separate capability
  • For example, don’t draw your entire board in the screen’s draw method, have UI elements for the board, X’s and O’s
• Really bad code = incomplete
Use good practices.

• Comment your code!
  • For yourself as much as us!
• Use packages to separate your engine code from your game code
  • This is a requirement!
• Copy your engine code into each new project rather than making a dependent project
Test often and incrementally.

• NEVER write a whole week from scratch and then run it
  • There will be a problem, and it can be anywhere

• Write one part at a time, stubbing out or putting println calls where necessary
  • Bug source is now bounded

• E.g. implement and test input and drawing separately
Deal with bad design decisions.

• At some point you will make a bad design decision
• Don’t be afraid to redesign/refactor your code
  • It will only get worse if you try to hack around your old design
• Each new project gives you an opportunity to refactor
Tips for Tic and Beyond

SUPPORT CODE

OVERVIEW
Six support code classes

- FXFrontEnd
  - Class which you will extend Application from and implement onTick, onDraw, etc.
- CS1971FrontEnd
  - Base class of FXFrontEnd
- FXApplication
  - Gets around a certain limitation
  - Nothing you need to worry about!
- All three of these must be in the same package
Six support code classes cont.

- Vec2d, Vec2i
  - Contain nearly all basic vector operations you will need.
  - Familiarize yourself!
  - DON’T ADD FIELDS
- FontMetrics
  - Gives width and height of drawn text
JavaFX

- FXFrontEnd is used to set up the frame and events, but that’s all we are using JavaFX for
- Never use javafx panes, buttons, scenes, etc.
- Make it all yourself
CS1971 FrontEnd “Debug” mode

- Enabled by default
- Displays screen size and FPS in title bar
Development Environment

- **eclipse**
- (only) TA-supported IDE
- Is pretty much just a swell program all around
You can run demos!

cs1971_demo wiz2
bgabinet

cs1971_demo tic tademo
Tips for Tic and Beyond

JAVA TIP OF THE WEEK
Use the standard Java collections!

- Need an easy way to clump objects of some type?
  - Use a `List<E>`
  - Note: Almost no reason to use `LinkedList<E>` over `ArrayList<E>`

- Need a mapping from one class of objects to another?
  - Use a `Map<K,V>`
  - Usually `HashMap<K,V>`

- Avoid synchronized counterparts `Vector<E>` and `Hashtable<K, V>`
  - Unnecessary overhead
Use generics!

- Use the *generified* versions of the standard Java collections!
- This means don’t use raw types!
- Be particularly careful of instanceof – it is a sign of poor design
Java Math Tips

• Avoid Math.pow() if possible
  • \(x^x\) is WAY better than Math.pow(x, 2)

• Don’t pass around pairs of numbers
  • Use Vec2i/d to represent sizes or coordinates
Tips for Tic and Beyond

QUESTIONS?
Tips for Tic and Beyond

GAME DESIGN TIPS FOR TIC
Tic Game Design Tips!

• Playtesting is less enjoyable when the games are boring, ugly, or hard to figure out
Color Schemes

- Players will judge your game immediately based on how it looks.
- Bad color schemes are an easy way to lose your player’s favor...
- But good color schemes will draw them in!
Here’s an ugly Tic...

• Board sort of fades into the background
• Colors don’t feel like they go together at all
• X and O are the same color
And here’s a better one!

• Board pops from the background and is clearly the focus
• Colors feel more cohesive
• X and O are different colors
How to pick a color scheme

• Less tends to be more
• Easiest: white on black with a few accent colors
  – Just like these slides!
• Use www.colorschemedesigner.com/ to get colors that go well together
  – Plenty of similar tools are out there
Juice

• “A juicy game feels alive and responds to everything you do”
  – From *How to Prototype a Game in Under 7 Days*

• How can we make Tic juicy?
Basic Juice: Mouse Hover

- Mouse hover effects make software feel much more responsive
- Have your buttons change slightly when hovered
- This is required for Tic!
- Show ghost pieces on the tic-tac-toe board
Recap

• Use a good color scheme
• Add juice with mouse hover effects
• Start early, start now, start yesterday! -Andy
Game Design Tips for Tic

QUESTIONS?
REMEMBER TO SIGN UP FOR A DESIGN CHECK

‘Til next week!