Lecture 0
Course Intro, Basic Architecture, Graphics I
Introduction

WELCOME!
Introduction

COURSE STAFF
Alex

- Senior CS
- Running, hiking, and soccer
- HTA
- Design Check Hours:
  - Tuesdays 5-7 remote
- TA Hours:
  - Wednesday 5-7 in CIT 205
  - Friday 3-6 remote
Sal

● Senior concentrating in software / visual pathways
● I like interfaces and HCI too
● TA hours
  ○ Monday 5:00 - 7:00 in CIT 205
  ○ Yes, class doesn’t officially end until 5:30, but often it will end before then
● Design check hours
  ○ Remote, Tuesdays 7:00 - 9:00
Carlos

- Senior studying CS / Game Design
  - GD is an independent concentration, reach out if you wanna make your own!
- Lowkey am probably not in class rn
  - Shopping a class, we’ll see if I take it
- UTA
- Hours are probably gonna be on Tuesday or Wednesday
Class Goals

- Build a 2D game engine from scratch
- Build cool games on top of your engine
- Improve your software engineering skills
INTRODUCTION

COURSE OVERVIEW
Some Logistics

● **Class**
  ○ Mondays from 3:00–5:30 pm
  ○ Same Zoom link each time; recordings will be linked on the website

● **Hours**
  ○ SignMeUp + Zoom
  ○ Timezone form coming via Slack today

● **Slack**
  ○ We’ll send out Slack invites later today
  ○ Used for announcements, talking to your classmates, uploading your demos, and questions
  ○ (Please don’t send project-specific code; come to hours or message the TAs if you have a more complex question!)

● **Website**
  ○ [http://cs.brown.edu/courses/csci1950n/](http://cs.brown.edu/courses/csci1950n/)
Some Logistics

● Projects
  ○ Handouts released on Mondays (yes, including today)
  ○ Projects due on Mondays at 11:59pm
  ○ We use GitHub Classroom—more details on setup and handin will be in the first handout

● Design Checks
  ○ Mandatory for each project: meet with a TA for 15 minutes
  ○ Take place on Tuesday-Wednesday after a handout comes out (first ones start tomorrow!)

● Demos
  ○ At the end of each project, record a quick, informal demo of your final product
  ○ Upload it to the #demos channel on Slack!
Course Goals

● Accessibility
  ○ We want this class to be easy to access, regardless of your personal situation.
  ○ Asynchronous
  ○ Reach out if you need any accommodations

● Responsiveness
  ○ This is a student run course: We need your feedback!
  ○ Fill out the anonymous feedback form, which you can find on the website
  ○ Survey at some point in the semester.
THE FIVE PROJECTS
Tic

- Tic-Tac-Toe
- Start building a UI kit! (You can’t use JavaFX)
- No engine yet; game logic coded directly into the game screen
- Due next week
Alchemy (Alc)

- Little Alchemy
- Puzzle game! Combine different elements to make new ones
- 2 week project
- Viewports, content management, simple collisions
**Wizardry** (Wiz)

- 2D dungeon crawler
- 2 week project
- Map generation & tilesystem, simple AI, pathfinding
- Lots of room for creativity and game design
**Ninja (Nin)**

- Platformer
- 2 week project
- Polygon collisions, complex physics, level loading, polished UI
  - Don’t worry, we’ll give you all the equations
Final Project!

- You choose everything! Engine features, game features—the only limit is your imagination
- Groups recommended, but you can do it solo as well
- Maybe?? Playtesting?? We’ll see...
- More details later
Introduction

GRADING
Our Grading System

- Only projects
- No homework or exams
- Multi-phase projects are broken down into weekly assignments
- Projects due every Monday at 11:59 PM on GitHub Classroom

The Sunlab at 11:59 pm (2019, colorized)
Our Grading System

- For each project, you have…
- Global & Primary requirements
  - Major features
- Secondary requirements
  - Less important features
- We’ll give you a rubric, and you fill it out with where to find these reqs in your project. Turn this file in with your code.
Design Checks

- Mandatory, from Tuesday-Wednesday after each handout release
- High-level conceptual questions
- You aren’t expected to have 100% correct answers; we just want to see that you’ve thought about it and tried to come up with something by yourself
- Encourages you to think about the project very early so you can budget your time well.
Final Grades

- No curve!
- Do the work, get an A
- 4 points per weekly project:
  - Design Check – 1 pt
  - Global & Primary Reqs – 2 pts
  - Secondary Reqs – 1 pt
  - **Note**: no credit for secondary reqs until primary reqs are met.
- You need to meet all **engine** requirements by the end of the semester.

<table>
<thead>
<tr>
<th>Points</th>
<th>Missing</th>
<th>Percent</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>35-43</td>
<td>0-8</td>
<td>81-100</td>
<td>A</td>
</tr>
<tr>
<td>27-34</td>
<td>9-16</td>
<td>63-81</td>
<td>B</td>
</tr>
<tr>
<td>20-26</td>
<td>17-23</td>
<td>47-63</td>
<td>C</td>
</tr>
<tr>
<td>&lt;20</td>
<td>&gt;23</td>
<td>&gt;-47</td>
<td>NC</td>
</tr>
</tbody>
</table>
The Retry System

- You have three retries for the whole semester
- Only your best handin will count (retries will never hurt your grade)
- Retries must be submitted within a week of getting your grade back; email the HTA if you intend on using one.
Please handin on time!

- This course isn’t a few sprints; it’s a marathon.
- Falling behind causes a “snowball of death”
- Handin whatever you have by the deadline
- Only use retries if you absolutely have to! The assignments will keep piling on otherwise.
Introduction

COURSE REQUIREMENTS
In order to take this class, you must...

- Be comfortable with Java
  - CS15 or CS18 at minimum
- CS32 is also recommended, but not required
Introduction

REGISTRATION
Registering for CS1950N

- Courses At Brown (CAB)
  - We’re “2D Game Engines”
  - May need an override code to register

- For non-Brown students:
  - Send us an email
  - cs1950nheadtas@lists.brown.edu

- We’re also sometimes known as CS1971
  - Don’t ask why…
    - (Pretty sure it depends on who is listed as the professor of the course)
      - ((( Could be wrong though, if this is wrong pretend we never said this o_o )))}
Capstone

- You can take this course as your capstone!
- Implement more features for your final project
Introduction

QUESTIONS?
Lecture 0

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?

- Stencil package hierarchy:
  - src/
    - engine/
      - Screen.java
    - tic/
      - TicScreen.java

- Any code in your engine package SHOULD NOT reference code in your game package.
- The use of abstract classes may make this clearer, if Screen is abstract and can’t be instantiated, then TicScreen, a subclass of Screen, can be the screen object that tic references
Basic Engine Architecture

THE MOST ESSENTIAL INTERFACE
A game generally needs...

- Timed updates ("tick")
- Ability to render to the screen ("draw")
- Input events (in some form or another)
- Knowledge that it has been resized (more info later)
Tick

- **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
Why Tick

● Given that a game needs to be rendered on a screen, it is vitally important that the screen displays the **most up to date state of the game**, and the state of the game will not change **as the game is rendering it on screen**

● Ticks are a way to make sure all game logic is completed before the game is rendered
  ○ It would be bad if the game logic determined player is in position x, began drawing that, and then updated player’s position to position y, now there’s two players on screen?
Graphics I

DRAWING THINGS
Draw

● General contract:
  ○ public void draw(GraphicsContext g)
  ○ Convert game state into viewable form

● Each time draw is called…
  ○ Clear everything that’s on-screen
  ○ Redraw the canvas based on your game’s current state
  ○ You should use the JavaFX GraphicsContext object to do this
    ■ You can use JavaFX here; you just can’t use its UI kit.

● 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 on<DeviceType><EventType>(<Device>Event e)`
  - `DeviceType`: mouse, key, etc
  - `EventType`: moved, pressed, etc
- 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: the support code calls these, you implement them)

```java
public class Application extends FXFrontEnd {
    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 the 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 itself in the 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 without delegating
- 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 (another `UIElement`)
  - Has a position (relative to parent)
  - Has a size
- Extend and override methods to create more specific UI (e.g. to go from Rectangle to Square)
TLDR, game engines are all about delegation

- Screen is the root of UIElements
- This architecture is how calls like tick, onDraw, onResize, etc. can be passed down to all UIElements
A note on UIElement children and GraphicsContext

- Drawing with GraphicsContext is like paint layering
  - GraphicsContext will initially draw whatever element you want on a blank canvas
  - Its next draw call will draw on top of what already has been drawn
- Say we have two UIElements: Rectangle, and Circle, where Circle is a child of Rectangle
- If you want Circle to appear on top of Rectangle, Rectangle needs to draw itself first, and then let Circle, its child, to draw afterwards
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
- For this week only, you can hardcode game logic into your Tic Screen
- For Tic, game logic refers to things like which player’s turn it is, did someone win with the last move, etc.
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
class MyGameMain {
    public static void main(String[] args) {
        FXApplication application = new FXApplication();
        application.begin(new App("Tic-Tac-Toe"));
        // don't put any code after begin()
        // this has all been filled in for you in the stencil anyway!
    }
}
```
Basic Engine Architecture

QUESTIONS?
Project Demos

You can find project demos on the Tic handout, if you want to see what a completed project might look like.

Also, let's take a short break
Lecture 0

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!
  - Hard to maintain, 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

- **Bad**: Blindly draw at fixed size anyway
  - Please don’t do this
- **Bit better**: Rescale everything
  - Loses aspect ratio
- **Much better**: scale up maintaining aspect ratio
  - Extraneous space in the background can be black
- **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 gameplay
Reacting to resizing

- In Tic, the board must remain square
  - How you handle the other UI elements is up to you, as long as they resize in some fashion
- Make sure to test your mouse events too
  - The new windows size might affect where you are clicking, logically
How to think about resizing

- Your game window opens in a default size (defined in CS1971FrontEnd)
- Increasing or decreasing the size of the window is incr/decr the ‘view’ of the canvas by some scale or factor
- Resizing can change the relative position of an object on the screen
- How to deal: scale your object’s position and size with the resize to maintain its visual aspect.
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)
How to actually draw things

- **Draw methods in GraphicsContext**
  - Fill (fills in the entire shape)
  - Stroke (draws the outline of the shape)
  - fillRect(), strokeRect(), fillOval(), strokeOval(), etc.

- **Each UIElement's draw method should invoke these methods to draw itself**

- **Only use doubles for drawing**
  - Otherwise, future projects may be more difficult!

- **Colors**
  - There should be a constant in your UIElement which holds its colors
  - Create colors with Color.rgb(int r, int g, int b)
  - 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
  - Comes with the support code
QUESTIONS?
Lecture 0

Input
THE KEYBOARD
JavaFX KeyEvents

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

- What do those actually mean?
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!
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
QUESTIONS
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 `UIElement` 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!
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
  - Better yet, use a debugger or assert statements!
- 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
Seven support code classes

- **FXFrontEnd**
  - Class that Application extends from (see below)

- **CS1971FrontEnd**
  - Base class of FXFrontEnd

- **FXApplication**
  - Gets around a certain limitation of JavaFX
  - Nothing you need to worry about!

- **Application**
  - The only class you’ll have to edit
  - Found in the root engine directory
Seven support code classes

● **Vec2d, Vec2i**
  ○ One for *doubles*, another for *ints*
  ○ Contain nearly all basic vector operations you will need.
  ○ Familiarize yourself!
  ○ DON'T ADD NEW FIELDS, especially game-specific ones!

● **FontMetrics**
  ○ Gives width and height of drawn text
JavaFX

- FXFrontEnd is used to set up the frame and events, and GraphicsContext is used to draw, but that’s all we are using JavaFX for
- Never use JavaFX panes, buttons, scenes, etc.
- Make it all yourself
CS1971FrontEnd “Debug” mode

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

- IntelliJ IDEA
- (only) TA-supported IDE
- Is pretty much just a swell program all around

rip eclipse
Anonymous Methods

- Essentially an in-line class/interface
- All anonymous methods are inner classes
- And therefore have a reference to the instance that creates them
- Keeps engine classes abstract; their game-specific functionality is delegated game-side

```java
button.addKeyPressedHandler(new SaveGameHandler(e));

private class SaveGameHandler implements KeyHandler {
    public void onKeyDown(KeyEvent e) {
        // Implementation here
    }
}

// Syntactic sugar: the above example is functionally the same as this:
button.addKeyPressedHandler(e -> {
    // Implementation here
});
```
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 a `HashMap<K,V>`
- Always declare with the (usually abstract) superclass and initialize with the specific class
- Avoid synchronized counterparts `Vector<E>` and `HashTable<K,E>`
  - Unnecessary overhead

<table>
<thead>
<tr>
<th></th>
<th>Standard (non-threadsafe)</th>
<th>Synchronized (threadsafe)</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td><code>ArrayList</code></td>
<td><code>Vector</code></td>
</tr>
<tr>
<td>Map</td>
<td><code>HashMap</code></td>
<td><code>Hashtable</code></td>
</tr>
</tbody>
</table>
Use generics!

- Use the generified versions of the standard Java collections
  - This means don’t use raw types!
- Note: once you parameterize during type declaration, you don’t need to do it again during initialization
  - List<Integer> myList = new ArrayList<>();
- 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/Vec2d` to represent sizes or coordinates
QUESTIONS?
Tips for Tic and Beyond

GAME DESIGN TIPS FOR TIC
Tic Game Design Tips!

- Games are 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...

- Colors don’t feel like they go together at all
  - The blue at the bottom is hard to read against the pink and yellow
- Board is similar color to the background
- X and O are the same color
And here’s a better one!

- Board pops from the background and is clearly the focus
- X and O are different colors
- You can still do better, but this is a good baseline
How to pick a color scheme

- Less tends to be more
- Easiest: offwhite on black with a few bright accent colors
- Play with https://coolors.co/generate until you find something you like
- 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 van Dam
QUESTIONS?
‘Til Next Week!

● Remember, coming out today…
  ○ Tic handout
  ○ Slack invites

● Action items (will be sent via Slack)
  ○ Timezone form, ASAP
  ○ Design check signups, by today
  ○ Collaboration Policy form, by Monday
  ○ Start working on Tic!