Lecture 0
Introduction
WELCOME!

Introduction
Introduction

TA STAFF
Class goals

- Build a 2D game engine from scratch!
- Build games on top of your 2D game engine!
- Improve your software design/engineering skills
Class times

• Class: Wed 3:00p-5:20p
  • Lecture 1-1.5 hours (Lubrano)
  • Playtesting ~1 hour (Sunlab)

• Design Checks: Fri - Sat
  • Sign up using cs1971_signup <project>

• TA hours: Sun - Tue

http://cs.brown.edu/courses/cs1971/
Four projects covering a variety of topics

- **Tic**
  - UI-only game (Tic-Tac-Toe)
  - Entirely engine based
  - Due next week!

- **Tac**
  - Grid-based tactical game (StarCraft, Final Fantasy Tactics)
  - Game content/resources, AI, viewports, map loading/generation

- **Tou**
  - Shoot-'em-up (Touhou, Asteroids, Space Invaders)
  - Collision detection

- **M**
  - N-like platformer (Metroid, Sonic, Braid, Super Meat Boy)
  - Physics, data-driven game logic
One open-ended final project

• Pick engine feature(s) and gameplay of a game you want to design
  • Pitch them to the class and find teammates
  • Pitch them to the TA’s and get a mentor
• Groups allowed and strongly encouraged
• Four weeks of development culminating in final presentation
• Public playtesting required, polish recommended
• All yours afterwards
• More details later
• See previous year Showreels (under Docs & FAQ)
Game Design Mini-Course

• Starts next week!
• Supplement to 2D Game Engines
• Discuss high-level concepts
• Help create better final projects!
Simple grading system

- Only projects, no HW/exams
- Every project is broken down into weekly checkpoints
  - Handins due every Tuesday at 11:59:59 PM
- If your handin exists and meets requirements, it is “complete” otherwise it is “incomplete”

(Sunlab at 11:59PM on most weekdays)
Handin requirements include...

- **Global requirements**
- **Weekly checkpoint requirements**
- **Design check**
  - Sign up using cs1971Signup <project>
  - High level conceptual questions, but not a free pass
- **Playtesting other students’ games**
  - Help each other find bugs
  - Feedback on gameplay
Incomplete handins

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

• **Extra Retries**
  • You have two for the whole class
  • Retry a checkpoint that you retried

• You have a week to use each retry
Out of Retries

• Used the standard retry, out of extra retries, now what?

• No credit (NC) for the checkpoint

• All checkpoints must be handed in to pass the class
  - So make sure to finish it by the end of the semester
Final grades

- No curve!
  - Do the work, get an A

- Specifically:
  - 0-1 no-credit: A
  - 2 no-credit: B
  - 3-4 no-credit: C
  - >4 no-credit: NC for the course

- Remember you still need a working version of each checkpoint

<table>
<thead>
<tr>
<th># NC</th>
<th>% complete</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100%</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>93%</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>86%</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>79%</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>71%</td>
<td>C</td>
</tr>
<tr>
<td>5+</td>
<td>&lt;70%</td>
<td>NC</td>
</tr>
</tbody>
</table>
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, go to sleep! You have another week to retry
QUESTIONS?
COURSE REQUIREMENTS
In order to take this class, you must...

- Be very comfortable with object-oriented design
- Be very comfortable with the Java language
It’s helpful but not required that you…

• Have experience with large Java projects
• Are comfortable with vector arithmetic
• Have basic knowledge of high-school physics
• Have experience with version control
Introduction

DIFFICULTY
Median number of hours spent

*** These times may not apply ***
ABOUT SECTIONS

Introduction
Registering for 1971

• We NEED to know how many of you are really, really planning on taking this course
  – If you’re 75% sure, that’s enough
• PLEASE give us your CS login on the way out
• If you ARE NOT registered on banner, don’t register yet
• If you ARE registered on banner, leave yourself registered until further notice
• We will get back to you soon.
Introduction

QUESTIONS?
A WORD FROM OUR SPONSOR
Lecture 0

Basic Engine Architecture
WHAT IS AN ENGINE?

Basic Engine Architecture
What is an engine?

- “The things that games are built on” - zdavis
- Games tend to have a lot of functionality in common
  - Even beyond the superficial
- Solution: create “engines” that abstract out common functionality
- Implemented as a library or a framework
  - Frameworks dictate flow of control, e.g. callbacks in AWT/Swing
  - Libraries do not dictate flow of control, e.g. java.lang.Math
What is an engine?

- Should be usable by many games
  - If you gave your engine to someone, could they more easily write a game without modifying engine code?
- Should be general
  - No game-specific logic!!!
- Should be useful
  - If the logic isn’t specific to the game, put it in the engine!
What does this look like?

- **Sample package hierarchy:**
  - `src/`
    - `engine/`
      - `Screen.java`
    - `game/`
      - `TouScreen.java`

- **Any code in your game package** SHOULD NOT be referenced in your engine 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 `float` seconds

• Simply notifies the engine that a given amount of time has elapsed since the previous “tick”
  • But this is hugely 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
  • We require 20 UPS in all projects
Draws

• General contract:
  • `public void draw(Graphics2D g)`
  • Convert game state into viewable form

• Provides a “brush” object (attribute bundle) for drawing into
  • Frequently just changes pixels in a `BufferedImage`
  • Might do hardware acceleration sometimes
  • It’s an interface, you shouldn’t care or need to know

• 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 poll

• Exact contract differs depending on type, but usually of the form:
  • public void onDDDEEE(DDDEvent evt)
  • DDD = device type (e.g. mouse, key)
  • EEE = event type (e.g. moved, pressed)

• Event object contains information about the event
  • How far the mouse moved; what key was pressed...
  • Why not just use arguments?

• More info coming up in Input section
Putting it together

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

```java
public class Application {
    public void onTick(long nanos)
    public void onDraw(Graphics2D g)

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

APPLICATION MANAGEMENT
We know the external interface

- But how does one build an engine around that?
- Drawing/ticking/event handling is very different depending on what’s going on!
  - Menu system
  - The actual game
  - Minigames within game
Solution: Screens within Application

• Rather than keeping track of “modes”, separate each game screen into a dedicated Screen subclass

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

- **Simplest way:**
  - Single Screen in Application at a time
  - Current Screen calls setScreen() on Application

- **Alternative way:**
  - Stack of Screens maintained by the Application
  - Topmost Screen gets events
  - Advanced: “Transparent” Screens can forward calls down to other Screens
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) {
        MyApplication a = new MyApplication();
        a.setScreen(new MyMainMenu());
        a.startup(); // begin processing events
    }
}
```
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!
  - 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(Vec2i dim)
  • dim is the new width, height of the draw area

• Called when the size of the screen changes
  • Window resizes
  • Fullscreen is toggled
  • Storing the current size in your Application is a good idea
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
  • Acceptable, but still causes letterboxing/pillarboxing
Strategies for handling different sizes

• Best: distribute extra screen space to objects in the game as needed
  • Like a GUI toolkit
• Not always possible
  • Especially if the size of the game area has impact on the gameplay
• This is what’s required in Tic
Reacting to resizes

• Every time a resize occurs, repeatedly subdivide space according to desired layout

• In Tic, the board must remain square
  • Can fill as much space as possible with the board and center the timer in the remaining space

• When drawing, just use the computed rectangles
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)

- **Alternative: “Math-like” coordinates**
  - Origin in lower left, Y axis extends up
  - To use, just replace all Y arguments in draw calls with (height – y)
  - Better ways to do this by manipulating the Graphics2D object – experiment!

- **Use whichever is more intuitive!**
  - Don’t do standard because it’s “easier”
Actually drawing things

- Two types of draw methods in Graphics2D
  - Ones that take `java.awt.geom.Shape` objects: `draw(Shape)`, `fill(Shape)`
  - Ones that take a bunch of `int` parameters

- `int`s will not be precise enough in future assignments
  - Use the ones that take `Shapes`, but don’t store `Shapes`! Write your own package of shape classes and translate them to `java.awt.geom` shapes when drawing
  - Important for flexibility in future projects

- Only use floats for drawing
  - Otherwise, the next project might be much more difficult
Drawing text

- Use `FontMetrics.getStringBounds()` to determine how much space a piece of text will take up.
- When drawing text, the y coordinate indicates the baseline rather than the bottom.
  - Be wary of tails getting cut off.
  - Add `FontMetrics.getDescent()` to the lowest point you want the text to extend.
  - `(FontMetrics.getAscent() if using the math-like coordinate system)`

![Diagram showing ascent, descent, and baseline of text]
QUESTIONS?
LECTURE 0

Input
Input

THE KEYBOARD
AWT KeyEvents

- AWT would have you believe that there are three key event types
  - onPressed
  - onReleased
  - onTyped
- What do those actually mean?
AWT KeyEvents

• Pressed gets fired once when you press the key
  • NOPE! Fired multiple times when held (key repeat)

• Released gets fired once when you release the key
  • Usually yes
  • But on X-based unix systems such as the department machines, fired multiple times when held (key repeat)
  • Support code handles this case
Then what is `keyTyped`?! 

- Actually a very important distinction that was correctly implemented
- 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 manually implementing text input (not required in this course)
  - NOT useful for detecting key repeat!
Better key events

- Four events:
  - `onDown`
  - `onUp`
  - `onRepeated`
  - `onTyped`
- `onTyped` same as AWT
- `onDown`/`onUp` only fired once per key press
- `onRepeated` for key repeats
THE MOUSE
• **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.)
AWT 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 AWT only gives you one at a time, so if keeping track of the cursor position, listen to both.
Better mouse events

- May want to unify mouse button + key presses and releases
- DragStart and DragEnd events are not difficult to implement and are nice to have
**General Input Advice**

- Mess with `println`ing events for a while to get a sense of them
  - Better understand their contracts
- Wrap AWT events in your own event classes
  - Extend functionality and prevent AWT references from polluting your code
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 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 separate drawCell, drawX, drawO...
- 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 actually a global req
- 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
  • If you never make a bad design decision, you didn’t need to take this course!

• Don’t be afraid to redesign/refactor your code
  • It will only get worse if you try to hack around your old design
Four support code classes

- **SwingFrontEnd**
  - Class which you will extend and implement `onTick`, `onDraw`, etc.

- **CS1971FrontEnd**
  - Base class of SwingFrontEnd

- **Vec2f, Vec2i**
  - Contain nearly all basic vector operations you will need.
  - Familiarize yourself!
  - DON’T ADD FIELDS

- CS1971FrontEnd → SwingFrontEnd → Your Application → Vec2f → Vec2i
No Swing

- SwingFrontEnd is used to set up the frame and events, but that’s all we are using Swing for.
- Never use JPanels, JLabels, JButtons, etc.
- Make it all yourself.
- You're welcome 😊
CS1971FrontEnd “Debug” mode

- Enabled by default
- Displays screen size, aspect ratio, and UPS in title bar
- F11 toggles fullscreen
- F12 allows resizing
Development Environment

- **eclipse**
  - Brand new, shiny version of Eclipse (Mars)
  - (only) TA-supported IDE
  - Is pretty much just a swell program all around

- Build instructions required for anything else
  - Lets us debug your program if we think we know a quick fix
You can run demos!

cs1971_demo tac3 ebirenba

cs1971_demo tic gtrousda
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!
  - If you use raw types we may give you an incomplete for poorly designed code
- Be particularly careful of `instanceof` – it is a sign of poor design

```java
package trin;

import java.util.List;

public class Generics {
    List badlist;
    List<Integer> goodlist;
}
```
Java Math Tips

• Use float literals instead of casting
  • (1.0f) or (1f) is better than ((float)1.0)

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

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

QUESTIONS?
Tips for Tic and Beyond

GAME DESIGN TIPS FOR TIC
Weekly Game Design Tips!

• Playtesting is less enjoyable when the games are boring, ugly, or hard to figure out.
• Quick easy ways to make your games more fun for others to playtest.
• Specific to each project, as opposed to the game design mini-course.
Tic-Tac-Tou

- No real gameplay design this week — it’s tic-tac-toe
- Instead, let’s focus on making tic-tac-toe look good!
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 http://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
• Show ghost pieces on the tic-tac-toe board
Recap

• Use a good color scheme
• Add juice with mouse hover effects
• Also, turn on anti-aliasing
  – g.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
• **Do these things last – finish requirements first!**
Game Design Tips for Tic

QUESTIONS?
REMEMBER TO GET US YOUR LOGIN!
GOOD LUCK!

‘Till next week!