lectures # week-8

Week 8: Animation & Swing Timer (Friday, February 27, 2026)

Lecture Recording

Watch on Panopto

Demo Code

Clone the lecture examples: TCSS305-Into-GUI

Related Assignment

This lecture covers concepts for Group Project Sprint 1 and Sprint 2.

Lecture Preview 0:11

Today's agenda:

1. **Sprint 1 Q&A** — Coordinate systems, code smell, and decomposition
2. **Sprint 2 introduction** — Requirements overview and new guides
3. **Animation** — The render-change-rerender loop and `repaint()`
4. **Swing Timer** — Driving animation with `javax.swing.Timer`

Admin Notes 0:11

AI Tutor Side Quest: The instructor built a full AI-powered tutoring system in ~4 hours using agentic AI coding. The system includes a React/Next.js frontend, a Node/Express API backend, and a database — all built by a team of AI agents (project manager, frontend,

backend, QA) working together overnight. The tutor has institutional knowledge of TCSS 305, its prerequisites, and assignment content. Students can ask questions about specific assignments and get guided help without receiving direct answers. A chat widget version may be embedded directly into assignment pages.

Test 2: Will be handed back Monday during group meetings. Brief review on Wednesday — for the most part, students should research what they missed on their own.

Sprint 1 Group Meetings: Monday next week. Sign-up sheet will be posted (same process as Sprint 0). One group member must have the project loaded and ready to demo — don't show up and fumble with branches. Treat it like a client demo.

Monday = No Lecture: Group meetings all day. No new content. Reading the guides independently is critical.

Sprint 1 Q&A 15:20

Coordinate System Mismatch 16:24

The tetromino data from the model uses a **Cartesian coordinate system** (Y increases upward), but you're drawing on a **screen coordinate system** (Y increases downward). The piece coordinates describe the shape — your job is to transpose those coordinates into screen pixels, centered vertically and horizontally.

Don't Hardcode Pixel Positions

You *could* hardcode pixel positions for the one tetromino in Sprint 1, but Sprint 2 introduces seven different tetrominoes with varying widths and heights. Use math (multiplication, subtraction, maybe division) to programmatically center any piece. The math isn't complex — it's basic arithmetic.

You're Not Always in Control of the Backend

The model gives you data in its own coordinate convention. You can't change the model. In the real world, you'll often build a frontend on top of a backend someone else created. It may not match your preferences — deal with it.

Code Smell & Decomposition 21:08

Code smell is when your code compiles and works, but something doesn't feel right — like cheese that smells a little funky. You can probably still eat it today, but tomorrow you won't want to.

If you're building Sprint 1 and everything is going into one class (board logic, next piece logic, controls, architecture), your intuition that something is wrong is correct. Separate by **responsibilities**, not just "panels":

- The next piece display has nothing to do with the Tetris board
- Controls have nothing to do with either
- Each responsibility should be its own class

> **Start Decomposed, Even If Classes Are Small**
>
> In Sprint 1, your controls class might just set a background color — that's fine. Having the class ready means Sprint 2's new features drop right in. A tiny class with clear responsibility is better than a god class that does everything.

Sprint 2 Introduction 27:08

Sprint 2 requirements are now posted. By the end of this sprint, you'll be **playing Tetris**:

- Pieces move down the board automatically (driven by a timer)
- Keyboard controls move and rotate pieces
- Next piece panel updates with the correct upcoming piece
- Game start functionality (game ending is not yet required)

The Event-Driven Model 28:03

The backend model fires **events** whenever its state changes (piece moves, next piece changes, etc.). Your job is to:

1. **Register as a listener** for the events your component cares about
2. **Write handlers** that respond to those events

This is the same pattern as button clicks and mouse events — just applied to game model state changes instead of UI interactions. The event system is already built into the model; you hook into it.

New Guides 29:46

Sprint 2 introduces the **heaviest conceptual guides** in the course — moving beyond "here's the code" into software design theory:

Guide	Focus
Introduction to Design Patterns	What design patterns are and why they matter
The Observer Pattern	The pattern that drives the Tetris model-view communication
Model-View-Controller (MVC)	Architectural separation of concerns
The Strategy Pattern	Behavioral flexibility through composition
Handling Key Events	Keyboard input (similar format to the Mouse Events guide)
Animation with javax.swing.Timer	Timer-driven animation (covered today in lecture)

All design pattern guides link to a shared demo project on GitHub with working code examples.

2D Graphics Review 34:52

Quick review of Wednesday's 2D drawing concepts:

Concept	Key Point
`paintComponent(Graphics g)`	Override this — not `paintComponents` (plural)

Concept	Key Point
Boilerplate	Call `super.paintComponent(g)`, cast to `Graphics2D`, enable anti-aliasing
Shape API	`Rectangle2D`, `Ellipse2D`, `Path2D` — all implement the `Shape` interface
`g2d.draw(shape)`	Renders the outline
`g2d.fill(shape)`	Renders filled
`g2d.setPaint(color)`	Changes the current color
`g2d.setStroke(stroke)`	Changes the outline thickness
Stroke caveat	Strokes wider than 1px extend **outside** the bounding box
Bounding box	Defined by x, y, width, height — x,y is the **top-left corner**
Ellipse bounding box	The top-left of the bounding box is **outside** the ellipse
Paint order	Shapes drawn later appear **on top** of shapes drawn earlier

---

Animation 38:43

Animation is a three-step loop:

1. **Render** — Display the current state
2. **Change** — Modify shape positions or properties
3. **Rerender** — Display the updated state

Repeat forever.

Shapes as Instance Fields 46:36

For animation, shapes **cannot** be local variables inside `paintComponent`. They must be **instance fields** so external code can modify their position between renders.

```
// Shape defined as instance field — position can change externally
private final Ellipse2D myMovingShape;
```

`paintComponent` becomes simple: set graphics properties, fill/draw the shape. No shape creation, no position calculation — just render whatever the shape currently is.

Shapes Are Mutable 49:25

Shape objects like `Ellipse2D` and `Rectangle2D` are **mutable** — you can call `setFrame()` to change their position and size. This is convenient for animation but be careful: something else could mutate your shapes unexpectedly.

The `repaint()` Method 40:21

> **Never Call `paintComponent` Directly**
>
> You should **never** call `paintComponent(g)` in your own code. Instead, call `repaint()`.

`repaint()` tells the Event Dispatch Thread: "whenever you get around to it, rerender this component." Key behaviors:

- **Not immediate** — the EDT processes it when it can, but it's fast enough that humans can't perceive the delay
- **Coalescing** — if you call `repaint()` 100 times before the EDT gets to it, only one actual repaint happens
- **Overloaded** — `repaint(x, y, width, height)` repaints only a clipping region for performance

> **For Tetris**
>
> Just call `repaint()` on the whole panel. Don't bother with clipping regions — your Tetris game isn't stressing the graphics system.

Button-Driven Animation Example 44:26

The first demo shows a ball that moves each time you click a button:

```
// In the button's action listener:
final Rectangle2D bounds = myShape.getBounds2D();
myShape.setFrame(bounds.getX() + MOVE_DISTANCE,
                 bounds.getY() + MOVE_DISTANCE,
                 bounds.getWidth(),
                 bounds.getHeight());
repaint();  // Tell EDT to rerender
```

Line by line:

1. Get the shape's current bounding box
2. Set a new frame shifted by `MOVE_DISTANCE` pixels
3. Call `repaint()` to trigger a rerender

This is the **change → rerender** part of the animation loop. The initial render happens automatically when the component is first displayed.

Swing Timer 51:31

Instead of clicking a button for every frame, a **timer** drives the animation loop automatically.

Concept 52:54

At a specified interval, the timer "ticks." On each tick, you:

1. Change what needs to be rendered
2. Call `repaint()`

Uses beyond animation: AI behavior (every tick, check player position and move toward them), game state updates, periodic checks.

Why `javax.swing.Timer` ? 55:06

Java has multiple classes named `Timer`. **Always use `javax.swing.Timer`** for Swing applications because:

- It ticks **on the Event Dispatch Thread**, integrating cleanly with the Swing rendering system
- Other timers (e.g., `java.util.Timer`) may not interact properly with Swing

Import the Right Timer

Always import `javax.swing.Timer`, not `java.util.Timer`. IntelliJ may suggest the wrong one.

Key API 56:35

Method	Purpose
`new Timer(delay, listener)`	Create timer with delay in ms and an `ActionListener`
`start()`	Begin ticking
`stop()`	Stop ticking (pause!)
`setDelay(ms)`	Change tick frequency (useful for speed-up on level changes)

The constructor requires a **delay** (milliseconds between ticks) and an **ActionListener** — the same `ActionListener` interface you use for buttons. When the timer ticks, it fires an `ActionEvent`.

```
// Timer that calls myGame.step() every 1000ms
final Timer timer = new Timer(1000, e -> myGame.step());
timer.start();
```

Tetris Speed-Up

As the player levels up, call `setDelay()` with a smaller value to make pieces fall faster.

Pause/Unpause

Pause = `timer.stop()`. Unpause = `timer.start()`. That's it.

Timer Accuracy & Drift 59:26

The Swing Timer does **not** have CPU priority. If the CPU is busy, the timer waits — leading to **time drift**:

Tick Rate	Drift Impact
10 ticks/sec	Negligible (~5ms/sec)
1000 ticks/sec	Significant (~500ms/sec)

For Tetris, this is a non-issue. For real-time systems (self-driving cars, missile defense), don't use Java Swing — use a language with direct hardware access.

> **Negative Delay Values**
>
> Passing a negative delay to the timer constructor doesn't throw an exception (despite what you might expect). Negative values are silently treated as zero, meaning the timer ticks as fast as possible.

Controlling Speed 1:10:31

Two ways to make animation faster:

1. **Decrease the timer delay** — tick more frequently
2. **Increase the step size** — move more pixels per tick

There's a sweet spot: too large a step size causes choppy, jumpy animation. Too small a delay taxes the system for minimal visual gain.

Already Seen a Timer? 54:31

You've already used a Swing Timer — the Road Rage assignment (A2) uses one internally. When you adjusted the speed slider, you were changing the timer's delay. Go look at the Road Rage GUI code for a working example.

This lecture outline is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.