CS324e - Elements of Graphics and Visualization

Timing Framework
Animating Swing Based Programs

• From FRC
• goal: provide framework and library to allow animation of components in a GUI
• Set of utility classes that contain the code common to handling timing issues for animation
• more functionality than the javax.swing.timer class
Example - Fading Button

• When button clicked it fades out and then back in
• fading cycle repeats until the button is pressed again
Fading Button Classes

• JFrame to hold panel and button
• checkerboard is a JPanel
• Button is a FadingButtonTF
  – class extends the JButton class
  – implements ActionListener (instead of using an anonymous inner class) and TimingTarget interfaces
FRC Timing Framework

• Handle common tasks such as determining what fraction of the animation has been completed

• Provide a simple API (Application Programming Interface)
  – a way of using existing code in a simple way
  – We have been using the Java API
Core Concepts of FRC Timing Framework

• handles timing, but *alteration* left to the programmer

• Animator
  – class that contains most of the timing functionality

• callbacks (listeners)
  – similar to timer callbacks but with more types of callbacks and more information
Core Concepts

• Duration: length of time animation runs, or can be set to indefinite (a clock for a game)
• Repetition: timer can run once and stop or repeat over and over
• Resolution: frame rate or frames per second
  — how often does timer send out notifications
  — default is 20 fps
Core Concepts

• Starting and Ending behaviors:
  – may add delay to start or begin in the middle of a repetition

• Interpolation
  – default is linear interpolation
  – fractions based on elapsed time / total time
  – possible to have other kinds of interpolation
  – start and end slowly (ease in and out)
Example Program

• Class declaration and instance variables

```java
public class FadingButtonTF extends JButton
    implements ActionListener, TimingTarget {
    // current opacity of button
    float alpha = 1.0f;
    // for start/stop actions
    Animator animator;
    // each cycle will take 2 seconds
    int animationDuration = 2000;
    BufferedImage buttonImage = null;
```
Example Program

• Constructor with creation of Animator

```java
/** Creates a new instance of FadingButtonTF */
public FadingButtonTF(String label) {
    super(label);
    setOpaque(false);
    setPreferredSize(new Dimension(150, 50));
    animator = new Animator(
        animationDuration/2,
        Animator.INFINITE,
        RepeatBehavior.REVERSE,
        this);
    animator.setStartFraction(1.0f);
    animator.setStartDirection(Direction.BACKWARD);
    addActionListener(this);
}
```
Animator Properties

- duration in milliseconds
  - can set to Animator.INFINITE to run continuously
- repeatCount
  - number of times to run, can also be INFINITE
- repeatBehavior: LOOP or REVERSE
- target: listener for timer notifications
Loop vs. Reverse

Loop

Fraction from Animator vs. Time

Reverse

Fraction from Animator vs. Time
Animator Properties

```java
Animator.setStartFraction(1.0f);
Animator.setStartDirection(Direction.BACKWARD);
```
Controlling the Animator

- methods for Animator Object
- void start()
  - callbacks to start and timingEvent methods
- void stop()
  - callback to end method
- void cancel()
  - stops Animator, but no callbacks
- void pause()
- void resume()
- boolean isRunning()
Fading Button Demo

- Respond to button clicks
- recall the FadingButtonTF class implements the ActionListener interface

```java
/**
 * This method receives click events, which start and stop the animation
 */

public void actionPerformed(ActionEvent ae) {
    if (!animator.isRunning())
        this.setText("Stop Animation");
        animator.start();
    } else {
        animator.stop();
        this.setText("Start Animation");
        // reset alpha to opaque
        alpha = 1.0f;
    }
```
Responding to Notifications

• To respond to notifications from the animator create a class that implements the timing

```java
public class FadingButtonTF extends JButton
    implements ActionListener, TimingTarget {

    /**
     * TimingTarget implementation: this method
     * sets the alpha of our button equal to
     * the current elapsed fraction of the animation.
     */
    public void timingEvent(float fraction) {
        alpha = fraction;
        // re-display our the button
        repaint();
    }
```
public void paintComponent(Graphics g) {
    // Create an image for the button graphics if necessary
    if (buttonImage == null || buttonImage.getWidth() != getWidth() ||
        buttonImage.getHeight() != getHeight()) {
        buttonImage = getGraphicsConfiguration().createCompatibleImage(getWidth(), getHeight());
    }

    Graphics gButton = buttonImage.getGraphics();
    gButton.setClip(g.getClip());

    // Have the superclass render the button for us
    super.paintComponent(gButton);

    // Make the graphics object sent to this paint() method translucent
    Graphics2D g2 = (Graphics2D)g;
    AlphaComposite newComposite =
        AlphaComposite.getInstance(AlphaComposite.SRC_OVER, alpha);
    g2.setComposite(newComposite);

    // Copy the button's image to the destination graphics
    g2.drawImage(buttonImage, 0, 0, null);
}
Non Linear Interpolation

• Animator objects have acceleration and deceleration properties
• by default these are not used
• can set so animation eases in and / or out
• instead of fraction of animation being linear with respect to time elapsed
Set Acceleration and Deceleration

• represented as fraction of animation to accelerate to average speed and decelerate to stop

• sum of fractions must be <= 1

// sample to show ease in and out
animator.setAcceleration(.2f);
animator.setDeceleration(.4f);
Linear

- Horizontal axis is time
- Vertical axis is fraction
Acceleration and Deceleration

- .3 of duration for acceleration
- .3 of duration for Deceleration

- note, at 1 second (1/4 of time) fraction is still below 0.2
Acceleration and Deceleration

- .7 of duration for acceleration
- .2 of duration for Deceleration
Triggers

• Part of FRC Timing Framework
• Start animation based on a specified event
  — for example:
    — user presses a button
    — user clicks on a door in a game
    — car in a game goes off the track in a game
Triggers

• Respond to
  – GUI events
  – time events
  – custom events created by the programmer

• Use triggers by:
  – create trigger including information about the Animator the trigger will run
  – add listeners to respond to trigger when it goes off
Triggers in Action

• Firing: when trigger event occurs the Animator objects start

• Disarming: canceling a trigger

• Auto-reverse: trigger has ability to run Animator forwards and then backwards
  —like the button fading in and out
Trigger Classes

• Trigger: base case for more complicated triggers

• TriggerEvent: part of framework to make it easy to add different kinds of Triggers

• ActionTrigger: simplest kind of Trigger
  — responds to java.awt.event.ActionEvent (like button clicks)
Trigger Demo

• Spheres fall and bounce based on various triggers
• The spheres are in their own panels, placed side by side

@Override
// SpherePanel paintComponent method
protected void paintComponent(Graphics g) {
    g.setColor(Color.white);
    g.fillRect(0, 0, getWidth(), getHeight());
    g.setColor(Color.BLACK);
    g.drawRect(0, 0, getWidth(), getHeight());
    g.drawImage(sphereImage, sphereX, sphereY, null);
}
Trigger Types

• Program demos 5 different kinds of triggers
• Yellow Sphere - ActionTrigger
• when the Trigger button is clicked the yellow ball's animation runs
• restarting if button clicked again before animation complete

```java
ActionTrigger.addTrigger(triggerButton, action.getAnimator());
```

• action is the SpherePanel for the Yellow sphere
Animator for SpherePanel

• The FRC timing framework includes a class to automate alteration aspect of animations
  – PropertySetter

• From SpherePanel

• Creates animator

```java
bouncer = PropertySetter.createAnimator(2000, this, "sphereY",
0, (PANEL_HEIGHT - sphereImage.getHeight()), 0);
bouncer.setAcceleration(.5f);
bouncer.setDeceleration(.5f);
```
More on PropertySetter

• Note the PropertySetter.createAnimator method

```javascript
bouncer = PropertySetter.createAnimator(2000,
this,
"sphereY",
0,
(PANEL_HEIGHT - sphereImage.getHeight(),
0);
```

• duration, object that has property animated, name of property (must have set... method), values property takes (y coordinate -> top, bottom, top)
Focus Trigger

• Blue Sphere - A FocusTrigger

```java
FocusTrigger.addTrigger(triggerButton,
    focus.getAnimator(), FocusTriggerEvent.IN);
```

• When the Trigger Button gains the focus (not pressed) the Blue Sphere bounces

• demo: use tab to change focus between Trigger button and Other Button
MouseTriggers

• Two MouseTriggers in the demo
• Red Sphere - armed trigger

MouseTrigger.addTrigger(triggerButton, armed.getAnimator(), MouseTriggerEvent.PRESS);

• When the mouse is pressed on the button the red sphere bounces
• press and hold button
• change from triggerButton to action panel
MouseTrigger

• Green Sphere - MouseTrigger
• Activated when the mouse enters the Trigger button region
  — does not need to be pressed

```javascript
MouseTrigger.addTrigger(triggerButton, over.getAnimator(), MouseTriggerEvent.ENTER);
```
Timing Trigger

• Gray Sphere - TimingTrigger
• When the action trigger (yellow sphere) stops then the timing trigger for the silver sphere starts
• useful for chaining animations

TimingTrigger.addTrigger(action.getAnimator(),
    timing.getAnimator(), TimingTriggerEvent.STOP);

• Add triggers so when gray sphere stops, red, green, and blue bounce