Software Development (CS2500)

Lecture 39: **Combining Sound and Graphics**

M.R.C. van Dongen

January 28, 2011

Contents

1 Outline

In this lecture we shall start by implementing a computer-based animation. Next we shall implement an application that combines graphics and sound. As in Lecture 38 our final implementation relies on inner classes.

2 An Animation

In this section we shall create a computer-based animation. As with the previous two lectures we shall base our solution on event handlers.

The animation consists of a circle which moves along a line for a number of iterations. The following is what we'll do.

- We create a main class that creates an animation object.

- The object has a panel and the circle's x and y coordinates.
- The object uses a `for` loop.
- In each iteration the loop does the following:
  - It increases the x and y coordinates.
  - It repaints the panel attribute of the animation object.
  - It *sleeps* for a short time. Here sleeping for a number of milliseconds means that the program suspends itself for that number of milliseconds. After the suspension, the program resumes as normal.
- An inner class represents the animation's pannel.
- The inner class extends `JPanel` and overrides `paintComponent( )`. This lets it draw the circle at different positions.

2.1 First Attempt

The following is the start of a first attempt to implementing the application. We've seen all of this several times before, so you should be able to understand this without too much effort. The `import` statements have been omitted for simplicity.

```Java
public class SimpleAnimation {
    private static final Random rand = new Random( );

    private JPanel drawPanel;
    private int x;
    private int y;

    public static void main( String args[] ) {
        SimpleAnimation animation = new SimpleAnimation( );
        animation.draw( );
    }

    // Rest in next listings.
}
```

The variable `drawPanel` is an instance variable of the `SimpleAnimation` class, which is shown in the next listing. Remember that instance variables of a class can be seen by the inner classes of that class. This explains why `drawPanel` can be seen int the inner class `MyDrawPanel`, which extends `JPanel`.

The following is the constructor of the class `SimpleAnimation`. All it does is setting up the `JFame`, construct the instance of the inner class `MyDrawPanel`, and add the `MyDrawPanel` to the `JFrame`.

```java
private SimpleAnimation( ) {
    JFrame frame = new JFrame( "Moving Dot" );
    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    drawPanel = new MyDrawPanel( );
    frame.getContentPane( ).add( drawPanel );
    frame.setSize( 300, 300 );
    frame.setVisible( true );
}
```

The following are the remaining methods, except for the method `handleException( )` which has been omitted. The method `Thread.sleep( )` does the sleeping. The sleeping is implemented by a call to the class method `Thread.sleep( )`, which may throw an exception. This explains the `try-catch` block because the exception should be caught and handled. The inner class is presented in the next listing.

```java
private void draw( ) {
    final int MAX_DOTS = 80;
    final int MILLI_SECONDS = 50;
    for ( int circleCount = 0;
          circleCount != MAX_DOTS;
          circleCount ++ ) {
        x ++;
        y ++;
        drawPanel.repaint( );
        sleep( MILLI_SECONDS );
    }
}

private static void sleep( int milliSeconds ) {
    try {
        Thread.sleep( milliSeconds );
    } catch (Exception exception) {
        handleException( exception );
    }
}
```

The following is the inner class. As you see, it is pretty simple.

```java
private class MyDrawPanel extends JPanel {
    @Override
    public void paintComponent( Graphics g ) {
        final int RADIUS = 40;
        g.setColor( Color.GREEN );
        g.fillOval( x, y, RADIUS, RADIUS );
    }
}
```

But, horror of horrors.... When we run the application it draws a sequence of circles which appear as a thick line on the screen.

2.2 Fixing the Problem

The source of the problem in the previous section is easy to find. The line is a sequence of partially superimposed circles which appears as a line on the screen. We forgot to erase the previous circle in the method `paintComponent( )` in the inner class. To solve the problem we erase the current circle by filling the `JPanel` with white. Next we draw a new circle in green. The following shows the implementation.

```java
private class MyDrawPanel extends JPanel {
    @Override
    public void paintComponent( Graphics g ) {
        final int RADIUS = 40;
        g.setColor( Color.WHITE );
        g.fillRect( 0, 0,  this.getWidth( ), this.getHeight( ) );
        g.setColor( Color.GREEN );
        g.fillOval( x, y, RADIUS, RADIUS );
    }
}
```

Note that the `this.` notation isn't really needed. However, it is included here to make explicit that the methods `getwidth( )` and `getHeight( )` are instance methods which return the width and height of the current `MyDrawPanel` instance. The `MyDrawPanel` inherits these methods from the `JPanel` class.

3 Listening to Non-GUI Events

In this section we shall implement an application that listens to non-GUI-based events. The following is what we do.

- We create a sequence of notes using `MIDI`.
- Each time there's a new note we print some text.
- This is implemented using an event listener.
- The listener listens to "`MIDI`" events but the mechanism is the same as for GUI events.

- The kind of event we're looking for are `Controller` events.

3.1 `Controller` Events

`Controller` events are generated by `Sequencers`. The event is triggered when a `Sequencer` encounters and processes a control-change event. The control-change event corresponds to the message type ShortMessage.CONTROL_CHANGE. The messages are added to the `Track` each time we start a new note. When the `Track` is played the relevant listeners are informed about the `Controller` event.

Registering `Controller` event listeners is as usual. However, this time we need to register a `ControllerEventListener` and these listeners have special needs which are passed as an additional argument to the method that does the registering. The method `addControllerEventListener` registers `ControllerEventListeners`. It is the equivalent of the method `addActionListener` for `ActionEvents`.

```java
int[] addControllerEventListener( ControllerEventListener listener,
                                  int[] controllers )
```

This registers a `ControllerEvent` listener. The listener is notified of control-change events of certain types. The types are specified by the argument `controllers`; this should be an array of `MIDI` controller numbers. Each controller number should be between `0` and `127`, inclusive. The returned array consists of the possible `MIDI` controller numbers for which the listener may now receive events. The listener class should override the method `controlChange( )`. Overriding `controlChange( )` is similar to overriding `actionPerformed( )`.

The following shows the registering of the event listener part.

```java
private static final int CONTROL_CHANGE = ShortMessage.CONTROL_CHANGE;
private static final int CONTROLLER_TYPE = 127;

int[] ints = new int[] {CONTROLLER_TYPE};
sequencer.addControllerEventListener( panel, ints );
```

Each time we play a new note, we also add a control-change message to the track. The following shows how it's done. The method `addMidiEvent( )` is the same as the one we implemented last Friday. You may look it up in the lecture notes of Lecture 36.

```java
addMidiEvent( track, CONTROL_CHANGE, 1, CONTROLLER_TYPE, 0, tick );
```

The `ControlEvent` listener should override the method `controlChange( )`.

```java
@Override
public void controlChange( ShortMessage event ) {
    System.out.println( "Do, re, me, fa, so, la, ti, do" );
}
```

3.2 Implementing the Application

The remainder of this section lists the details of the remaining methods.

```java
public class SingingMusicPlayer implements ControllerEventListener {
    private static final int VELOCITY = 100;
    private static final int ON = ShortMessage.NOTE_ON;
    private static final int OFF = ShortMessage.NOTE_OFF;
    private static final int CONTROL_CHANGE = ShortMessage.CONTROL_CHANGE;
    private static final int CONTROLLER_TYPE = 127;
    private static final int END_OF_TRACK = 47;
    private static final String lines[]
      = { "Doe, a deer, a femal deer",
          "Ray, a drop of golden sun",
          "Me, a name I call myself",
          "Far, a long, long way to run",
          "Sew, a needle pulling thread",
          "La, a note to follow Sew",
          "Tea, a drink with jam and bread",
          "That will bring us back to Do (oh-oh-oh)\n" };
    private int line = 0;

    // Rest in next listings.
}
```

```java
public static void main( String[] args ) {
    SingingMusicPlayer singer = new SingingMusicPlayer( );
    singer.play( );
}

private void play( ) {
    try {
        Sequencer sequencer = newSequencer( );
        sequencer.open( );
        int[] ints = new int[] {CONTROLLER_TYPE};
        sequencer.addControllerEventListener( this, ints );
        Sequence seq = new Sequence( Sequence.PPQ, 4 );
        singSong( seq );
        sequencer.setSequence( seq );
        sequencer.start( );
        sequencer.setTempoInBPM( 120 );
    } catch( Exception exception ) {
        handleException( exception );
    }
}
```

```java
private static void singSong( Sequence seq )
             throws InvalidMidiDataException {
    Track track = seq.createTrack( );
    int note = 64;
    for (int tick = 0; tick < 60; tick += 4) {
        addMidiEvent( track, ON,  1, note, VELOCITY, tick );
        addMidiEvent( track, CONTROL_CHANGE, 1, CONTROLLER_TYPE, 0, tick );
        addMidiEvent( track, OFF, 1, note, VELOCITY, tick + 2 );
        note ++;
    }
}
```

```java
@Override
public void controlChange( ShortMessage event ) {
    System.out.println( lines[ line ] );
    line = (line + 1) % lines.length;
}
```

4 Combining Sound and Graphics

In this section we shall make some changes to the program from the previous section. Instead of printing out text to the `MIDI` beats we shall generate a random rectangle which is coloured in a random colour.

We shall use an inner class for the event listener. The remainder of this section lists the main methods. The techniques are different from the techniques listed in the book.

```java
import java.util.Random;
import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;

public class MiniMusicPlayer {
    private static final int VELOCITY = 100;
    private static final int ON = ShortMessage.NOTE_ON;
    private static final int OFF = ShortMessage.NOTE_OFF;
    private static final int CONTROL_CHANGE = ShortMessage.CONTROL_CHANGE;
    private static final int CONTROLLER_TYPE = 127;
    private static final int END_OF_TRACK = 47;
    private final JFrame frame;

    public static void main( String[] args ) {
        MiniMusicPlayer mini = new MiniMusicPlayer( );
        mini.play( );
    }

    // Rest in remaining listings.
}
```

The constructor is pretty much straightforward. The method `play( )` is similar to the implementation from the previous section. However, this time it's the instance of the inner class which is the listener. We can get a reference to the instance of the inner class by applying the method `frame.getContentPane( )`. This works because we set the content pane of the `JFrame` attribute `frame` using the call `frame.setContentPane( panel )` in the constructor. In the book they have both a `JFrame` attribute *and* a `MyDrawPanel` attribute in the outer class, but this is a bit redundant. Rather than *explicitly* representing the `MyDrawPanel` as an attribute, our current solution represents it *implicitly*. To get the `MyDrawPanel` out application *computes* the `MyDrawPanel` from the `JFrame` object.

```java
private MiniMusicPlayer( ) {
    frame = new JFrame( "My First Music Video" );
    // Create inner class listener object.
    MyDrawPanel panel = new MyDrawPanel( );
    frame.setContentPane( panel );
    frame.setBounds( 30, 30, 300, 300 );
    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    frame.setVisible( true );
}

private void play( ) {
    try {
        Sequencer sequencer = newSequencer( );
        sequencer.open( );
        MyDrawPanel panel = (MyDrawPanel)frame.getContentPane( );
        int[] ints = new int[] {CONTROLLER_TYPE};
        sequencer.addControllerEventListener( panel, ints );
        Sequence seq = new Sequence( Sequence.PPQ, 4 );
        singSong( seq );
        sequencer.setSequence( seq );
        sequencer.start( );
        sequencer.setTempoInBPM( 120 );
    } catch( Exception exception ) {
        handleException( exception );
    }
}
```

The following is the inner class.

```java
private class MyDrawPanel extends JPanel
                       implements ControllerEventListener {
    // Class constants and instance attributes

    private MyDrawPanel( ) {
        random = new Random( );
        setRandomColour( );
        setRandomSizeAndPosition( );
    }

    // Instance methods.
}
```

The main reason why the inner class is so large is that it extends `JPanel` *and* implements the `ControllerEventListener` interface. It needs to override the method `paintComponent( )` from the `JPanel` class and the method `controlChange` from the `ControllerEventListener` interface.

Note that the class `MyDrawPanel` extends a class and implements an interface.

The following are the class constants and instance attributes of the inner class. The class constants are used to generate the random colours, the random width and height of the rectangle, and the random position of the rectangle. The instance attributes determine the current colour, the current width and height, and the current position of the rectangle.

```java
private static final int MAX_COLOUR_PART = 249;
private static final int MIN_SIZE =  10;
private static final int MAX_SIZE = 120;
private static final int MIN_POSITION = 10;
private static final int MAX_POSITION = 40;
private final Random random;
private Color colour;
private int width;
private int height;
private int xPosition;
private int yPosition;
```

Note that our solution uses a `Random` object whereas the book uses the static method `random( )` from the `Math` class. Using the `Random` object is much easier than using `Math.random( )`.

The instance methods `setRandomColour( )` and `setRandomSizeAndPosition( )` compute new random values for the colour, the width and height, and the position of the rectangle. They do this by using the auxiliary instance method `randomSize( int min, int max )` which computes a random `int` in the range `min–max`.

```java
private void setRandomColour( ) {
    int redPart   = random.nextInt( MAX_COLOUR_PART + 1 );
    int greenPart = random.nextInt( MAX_COLOUR_PART + 1 );
    int bluePart  = random.nextInt( MAX_COLOUR_PART + 1 );
    colour = new Color( redPart, greenPart, bluePart );
}

private void setRandomSizeAndPosition( ) {
    width     = randomSize( MIN_SIZE, MAX_SIZE );
    height    = randomSize( MIN_SIZE, MAX_SIZE );
    xPosition = randomSize( MIN_POSITION, MAX_POSITION );
    yPosition = randomSize( MIN_POSITION, MAX_POSITION );
}

private int randomSize( int min, int max ) {
    return random.nextInt( max - min + 1 ) + min;
}
```

The first of the following two method listens to `ControllerEvents`. When this method is called, we compute a new randomly coloured rectangle and draw it. The other method is called when the method `repaint( )` is called. It simply draws the current rectangle using the rectangle's current instance attribute values.

```java
@Override
public void controlChange( ShortMessage event ) {
    setRandomColour( );
    setRandomSizeAndPosition( );
    repaint( );
}

@Override
public void paintComponent( Graphics g ) {
    g.setColor( colour );
    g.fillRect( xPosition, yPosition, width, height );
}
```

5 For Monday

Study the lecture notes, and study Chapter 11.