User Interfaces and OpenGL

Java offers several options for creating user interfaces – windows, button, sliders, etc. – for graphical programs. We will use the Swing package, which is fairly standard and for which there is a lot of availabe documentation. This is not a class in user interfaces. If you want to use something other than Swing you are welcome to do so, but you are own your own for that and your programs need to run on the lab machines.

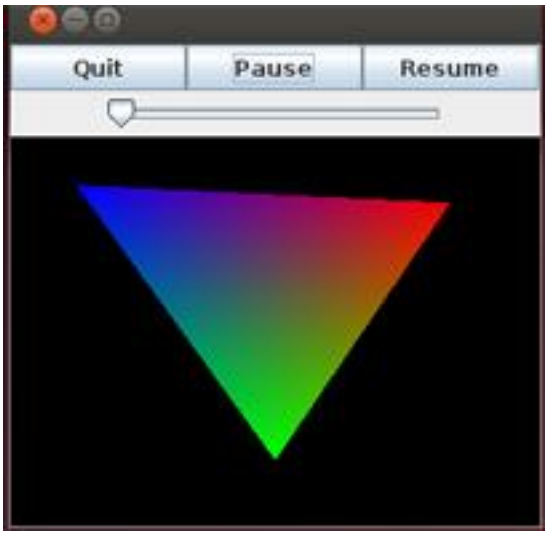
The structure of our programs is a bit complex but it is the same in every program. Here is the issue. The widgets in the Swing package expect to find a class containing the callback method for the widgets. Buttons, for example, expect a class that implements the ActionListener interface, which in turn requires a method **void ActionPerformed(ActionEvent e)**. If you make such a class, it doesn't have access to the instance variables elsewhere in the program, which you need to make the button have some effect on the program. Our solution is to make the constructor for the primary class of the program setup the entire graphical system and also to have the ActionPerformed method be part of this class. The main( ) method does nothing but construct an instance of this class.

This means a typical program looks like this:

```
public class Whatever implemenents GLEventListener, ActionListener, etc. {

        public static void main(String[] args) {
                new Whatever();
        }
        // data declarations go here so they can be accessed throughout the class
        ...
        JButton myButton;  // declare all of the user-interface widgets you will use here.
                ...
        private GLCanvas canvas;  // this is the window you will draw into

        public Whatever() {
                // The class constructor creates all of the windows and widgets
                GLProfile glp = GLProfile.getDefault();
                GLCapabilities caps = new GLCapabilities(glp);
                canvas = new GLCanvas(caps);
                canvas.addEventListener( this );
                ...
                myButton = new JButton( " ");
                myButton.addActionListener( this );
                ...
        }

        public void actionPeformed( ActionEvent event ) {
                // This is called when a button is clicked.
                if (event.getSource() == myButton) {
                        ...
                }
        }
}
```

Frames, Panels, and Layouts. A *Frame* is a top-level widget that holds the graphical elements of a program. Frames can have menu bars, though we won't talk about them at the moment. When you create a frame you specify a *Layout* Method for specifying where things go in the frame. The simplest layout method is a FlowLayout, in which widgets are added to the frame from the left to the right; when you run out of room a new row is created and it is also filled from left to right. This is a little too simple and doesn't givfe the designer enough control over the appearance of the frame. A more useful layout is a BorderLayout, in which elements are given one of 5 locations: NORTH, SOUTH, EAST, WEST and CENTER. The center is the primary element here, with everything else based around that. I like to put the drawing canvas in the center and the user interface to its north. Finally, a GridLayout specifies the numbers of rows and columns and makes all of the elements of a given row or a given column the same size. Widgets can be grouped together before being added to the frame. A *Panel* is a container widget that holds stuff – either widgets or other panels. Panels themselves can have layouts.

Here is a very typical example, from the demo program FrameTester. This sets up a frame that has a set of controls at the top and a drawing canvas below them. The control panel consists of two rows – the top row has three buttons of equal size; the bottom row has a slider as long as the three buttons put together. It looks like this:



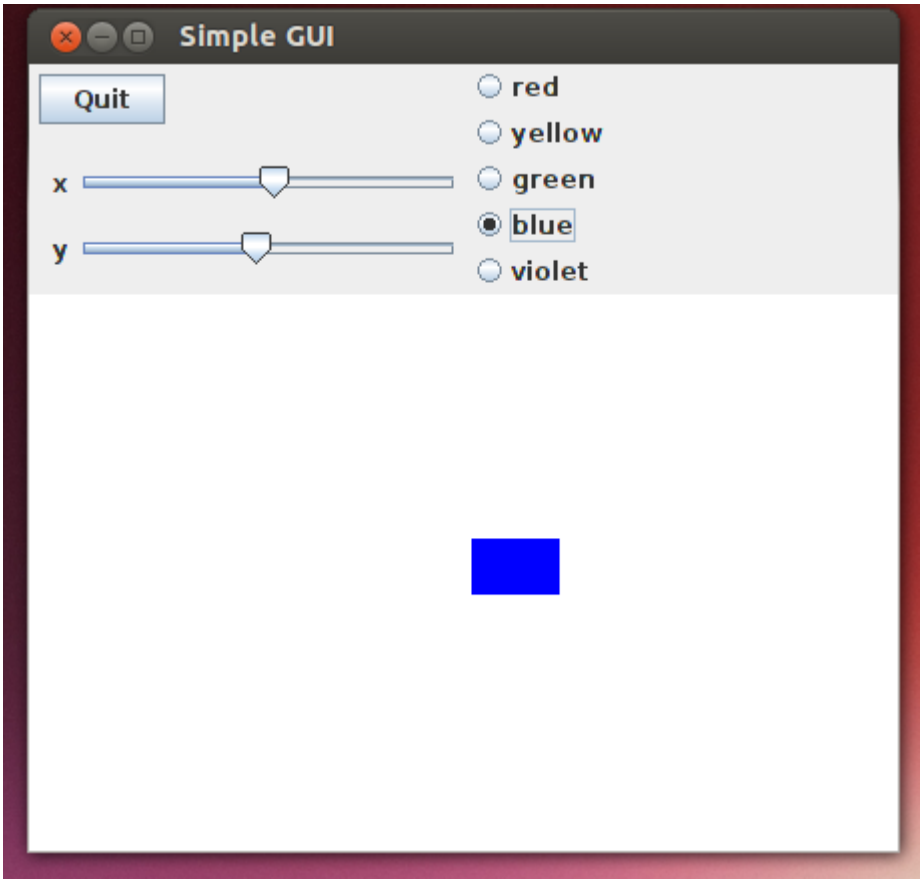
We sue a BorderLayout for the entire Frame so we can position the control panel above the canvas. We don't make the entire control panel a grid with 2 rows and 3 columns, for that would force the slider to be the same size as one of the buttons. Instead we make control panel a grid with two rows and 1 column. The top row is a grid with 1 row and 3 columns, which forces the three buttons to all have the same size. The bottom row has only 1 element, so we don't give it a layout method.

Here is the class constructor where all of this is laid out. Note that I am hiding some of the details for the widgets that we haven't discussed yet:

```
// open gl stuff to start out
canvas = new GLCanvas(...);

JFrame frame = new JFrame("");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE );
frame.setSize(300, 300);
frame.setLayout(new BorderLayout());
frame.setVisible(true);

JPanel north = new JPanel( new GridLayout(2, 1)); //  2 rows the same width and height.
JPanel topRow = new JPanel( new GridLayout(1, 3)); // 3 buttons the same size
        quitButton = new JButton( "Quit");
        topRow.add( quitButton);
        pauseButton = new JButton( "Pause");
        topRow.add(pauseButton);
        restartButton = new JButton( "Resume");
        topRow.add(restartButton)
north.add(topRow);

JPanel nextRow = new JPanel();
        speedSlider = new JSlider(1, 100);
        nextRow.add(speedSlider);
north.add(nextRow);

JPanel center = new JPanel(new GridLayout(1,1));
        center.add(canvas);

frame.add(north,  BorderLayout.NORTH);
frame.add(center, BorderLayout.CENTER);
```

We put the canvas in the center of the layout so it takes as much room as possible.

Here is another example. This has buttons, radio buttons and sliders for the user interface:



The entire cotnrol panel is called *north*; this gets a Border layout. Its western portion, which is panel NorthWest, also gets a Border layout, with the Quit button to its North, the x-slider at its Center, and the y-slider at its South. The right portion of the control panel is a panel I call NorthEast. This has a grid layout with 5 rows and 1 column, to hold the radio buttons. NorthEast is installed as the Center of the north panel – you should always have something in the center of a Border layout.

The code for this follows:

```
canvas = new GLCanvas( ...);

JFrame frame = new JFrame("Simple GUI");  // This name appears in the window bar.
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE );
frame.setSize(800, 800);
frame.setLayout(new BorderLayout());
frame.setVisible(true);

JPanel north = new JPanel( new BorderLayout());
JPanel northWest = new JPanel( new BorderLayout());
JPanel topRow = new JPanel( new FlowLayout(FlowLayout.LEADING));
        quitButton = new JButton( "Quit");
        topRow.add(quitButton);
        northWest.add(topRow, BorderLayout.NORTH);

JPanel secondRow = new JPanel(new BorderLayout());
        xSlider = new JSlider(0, 100);
        JLabel xSliderName = new JLabel("   x");
        secondRow.add(xSliderName, BorderLayout.WEST);
        secondRow.add(xSlider, BorderLayout.EAST);
        northWest.add(secondRow, BorderLayout.CENTER);

JPanel thirdRow = new JPanel( new BorderLayout());
        ySlider = new JSlider(0, 100);
        JLabel ySliderName = new JLabel( "   y");
        thirdRow.add(ySliderName, BorderLayout.WEST);
        thirdRow.add(ySlider, BorderLayout.EAST);
        JLabel spacer = new JLabel( "   ");
        thirdRow.add(spacer, BorderLayout.SOUTH);
        northWest.add(thirdRow, BorderLayout.SOUTH);
north.add(northWest, BorderLayout.WEST);

JPanel northEast = new JPanel(new GridLayout(5,1));
        redButton = new JRadioButton( "red", true);
        yellowButton = new JRadioButton( "yellow");
        greenButton = new JRadioButton( "green");
        blueButton = new JRadioButton( "blue");
        violetButton = new JRadioButton( "violet");
        ButtonGroup radioButtons = new ButtonGroup();
        radioButtons.add(redButton);
        radioButtons.add(yellowButton);
        radioButtons.add(greenButton);
        radioButtons.add(blueButton);
        radioButtons.add(violetButton);
        northEast.add(redButton);
        northEast.add(yellowButton);
        northEast.add(greenButton);
        northEast.add(blueButton);
        northEast.add(violetButton);
        north.add(northEast, BorderLayout.CENTER);
frame.add(north, BorderLayout.NORTH);
```

```
JPanel center = new JPanel(new GridLayout(1,1));
center.add(canvas);
frame.add(center, BorderLayout.CENTER);
```

Widgets

Buttons are the most common interface widget. They are objects of the class JButton, so they are declared with code such as

```
JButton quitButton;
```

The JButton constructor takes one argument, the string to print on the face of the button:

```
quitButton = new JButton( "quit" );
```

Buttons need to provide a class that implements the ActionListener interface. This interface requires one method:

```
public void actionPerformed( ActionEvent event );
```

The only interesting method of the ActionEvent class is **getSource( )**, which returns the button that was clicked as the source of the event. The typical actionPerformed( ) method looks like this:

```
public void actionPerformed( ActionEvent event) {
        if (event.getSource( ) == quitButton )
                System.exit(0);
}
```

I find it convenient to have the main class implement the ActionListener interface, so all buttons have the same actionPerformed callback method. This means that my typical code for creating a button is

```
quitButton = new JButton( "quit" );
quitButton.addActionListener( this );
```

Sliders allow the user to enter values in a range of possible values. Sliders provide only integer values, so if you want a float you will need to transform the value given by the scale.. Sliders can be set by the user and also by your program, which allows them to act as output widgets for showing time remaining or other values generated by your program. There are several constructors for sliders I generally use the constructor that specifies the minimum and maximum values of the range, as in

```
speedSider = new JSlider( 1, 100);
```

There are **set( )** and **get( )** methods that allow you to set the current value of the slider.
The callbacks for sliders must implement the **ChangeListener** interface, which requires a **stateChanged** method**.** As with buttons, I usually have the main class implement this interface. Altogether, here is typical code for creating a slider:

```
speedSlider = new JSLider(0, 100);
speedSlider.setValue(20);
speedSlider.addChangeListener(this);
```

The stateChanged( ) method for the callback looks like this:

```
public void stateChanged (ChangeEvent e) {
        speed = speedSlider.getValue( )/100.0;
}
```

As with buttons, the ChangeEvent class has a getSource( ) method that will tell you which widget was the source of the input. If you have several sliders the stateChanged method might look like

```
public void stateChanged (ChangeEvent e) {
        if (e.getSource( ) == xSlider)
                x = xSlider.getValue( )/10.0;
        else if (e.getSource( ) == ySlider)
                y = ySlider.getValue( )/10.0;
}
```

Sliders by default are horizontal; if you want a vertical slider you can get it with the constuctor

new JSlider(int orientation, int min, int max)

where "orientation" is **SwingConstants.HORIZONTAL** or **SwingConstants.VERTICAL**.

In an elaborate layout you will want labels attached to your sliders. The slider class has a label facility, but I find it overlay complicated and inflexible; it seems easier to me to use Label widgets.

Radio Buttons are useful if you want the user to choose one of a small set of options. Only one of the buttons can be "on" at a time. You make radio buttons in the same way as regular buttons, including giving each an ActionListener class. Radio buttons are also added to a ButtonGroup, which turns the current button off when a new one is turned on. Here is code for creating a simple group of 3 radio buttons:

```
ButtonGroup radioButtons = new ButtonGroup( );
redButton = new JRadioButton( "red" );
redButton.addActionListener(this);
radioButtons.add(redButton);
greenButton = new JRadioButton( "green" );
greenButton.addActionListener(this);
radioButtons.add(greenButton);
blueButton = new JRadioButton( "blue" );
blueButton.addActionListener(this);
radioButtons.add(blueButton);
```

Note that the ButtonGroup doesn't need to be a class variable because it is never referenced after the buttons are added to it.

You can choose which of the buttons is currently selected with button.setSelected(true), as in

```
greeenButton.setSelected(true);
```

The actionPerformed( ) method is just like any actionPerformed method for multiple buttons:

```
public void actionPerformed( ActionEvent even ) {
        if (event.getSource( ) == redButton ) {
                colorRed = 1.0;
                colorGreen = 0.0;
                colorBlue = 0.0;
        }
        if (event.getSource( ) == greenButton ) {
                colorRed = 0.0;
                colorGreen = 1.0;
                colorBlue = 0.0;
        }
        etc.
}
```

CheckBoxes are generally used in groups like radio buttons, only multiple checkboxes can be "on" concurrently. Your program can select checkboxes to be on or off, and can query individual boxes to see if they are currently on. Here is code for creating two checkboxes:

```
catBox = new JCheckBox("cat" );
catBox.addActionListener(this);
catBox.setSelected(true);

dogBox = new JCheckBox( "dog" );
dogBox.addActionListener(this);
dogBox.setSelected(false);
```

and code for the actionPerformed( ) callback. This assumes we have methods drawCat( ) and drawDog( ) that we want to call when the appropriate boxes are checked, and methods eraseCat( ) and eraseDog( ) to call when the boxes are unchecked.

```
public void actionPerformed( ActionEvent event) {
        if (event.getSource( ) == catBox ) {
                if (catBox.isSelected( ))
                        drawCat( );
                else
                        eraseCat( );
        }
        else if (event.getSource( ) == dogBox ) {
                if (dogBox.isSelected( ) )
                        drawDog( );
                else
                        eraseDog( );
        }
}
```

Labels are static widgets that hold a piece of text. The Swing collections also provide TextField widgets that allow the user to input text, but you probably won't have a use for those in this class. Labels are quite useful and simple to use.

JLabel xSliderLabel = new JLabel(" x ");

creates a label, which you can add in an appropriate place in your layout. Labels aren't interactive, though your program can use the **setText( )** method to change the text of the label.

Separators are static horiozontal or vertical lines that can be used to group widgets in a layout.

JSeparator sep = new JSeparator( ) creates a horoizontal line

JSeparator sep1 = new JSeparator(SwingConstants.VERTICAL) creates a vertical line

If you want an invisible widget use a label with an empty text string. For example, if you want a layout with one button at the left side of the control panel and another at the right side, you might use a GridLayout(1, 4), adding the left button, then twice adding new JLabel(" "), then the right button.

JOGL code. Every OpenGL program you write should have the following class variables for its main class:

```
private GLCanvas canvas;
private GL2 gl;
private GLU glu;
```

The constructor for this class should assign values to some of these. All of my constructors start with the code

```
GLProfile glp = GLProfile.getDefault();
GLCapabilities caps = new GLCapabilities(glp);
canvas = new GLCanvas(caps);
```

At the end of the constructor I make three calls that allow you to draw and modify the canvas:

```
canvas.addGLEventListener(this);
FPSAnimator animator = new FPSAnimator(canvas, 30);
animator.start();
```

The animator is responsible for updating the canvas. The number given in the animator constructor, 30 in this example, is the frame rate OpenGL attempts to maintain.

OpenGL calls an **init( )** method after the user interface is created. At a minimum this should assign to the **gl** and **glu** class variables:

```
public void init(GLAutoDrawable drawable) {
    gl = drawable.getGL( ).getGL2( );
    glu = new GLU( );
    ...
}
```

Usually init( ) also sets up the coordinate system for the world. We'll discuss this in class, but here is typical code that makes a 2D coordinate system scaled from 0 to 10 in both x and y:

```
gl.glMatrixMode(GL2.GL_PROJECTION);
```

```
gl.glLoadIdentity( );
glu.gluOrtho2D(0f, 10f, 0f, 10f);
gl.glClearColor(1, 1,, 0, 1); // making a yellow background
```

There are 4 other standard OpenGL methods:

public void display(GLAutoDrawable drtawable); which draws the current scene

public void reshape(GLAutoDrawable, int x, int y, int width, int height), which is called if the window is resized to shape (x, y, width, height)

public void dispose(GLAutoDrawable drawable); which is called when you exit from the program. I usually leave this empty.

Further, the display( ) method is usually broken down into calls to 2 other methods:

update( ), which handles changes in variables for animation

render( ),which does the actual drawing with OpenGL commands.

We will discuss all of these methods in class.

References

1. Oracle has a nice document about using Swing components to set up user interfaces: http://docs.oracle.com/javase/tutorial/uiswing/components/index.html
   Some of this is very useful and some of it is more convoluted than we will need this term, but it is documentation from the horse's mouth.

2. In addition, you can find the Oracle documentation for each of the component classes. Rather than looking through the documentation I usually just Google the name of the class; the Oracle documentation is almost always the first hit.

3. The Weiss book we often use as a text for CSCI 151 has an appendix devoted to Swing:
   Mark Allen Weiss: Data Structures & Problem Solving Using Java (Addison Wesley) Appendix B.