Comp 401 - Assignment 9: Toolkits and Graphics

Date Assigned: Thu Oct 24, 2013
Completion Date: Fri Nov 1, 2013
Early Submission Date: Wed Oct 30, 2013

In Part 1, you will learn to use a toolkit (Swing/AWT) to create a widget tree and a controller. Extra credit will also let you create widget trees that change dynamically and implement views and create menu items, buttons, and progress bars.

In Part 2, you will learn to draw objects on the screen. You have two alternatives for this part – one is the simpler, it gives you regular credit; the other is conceptually cleaner but more difficulty, it gives you regular and extra credit.

For additional extra credit, you can embellish these user interfaces in many ways.

The following new material is relevant to this assignment. Again, the key is understanding the relevant material. Once you do so, it should be straightforward.

The class material did not give details of the Graphics API. The shape parameters it assumes are slightly different from those ObjectEditor uses – for lines, images, and string. So you will have to do some translation for these. Look at the online Java api documentation for Graphics to learn its details.

<table>
<thead>
<tr>
<th>MVC and Toolkits (10/22, 10/24)</th>
<th>PowerPoint</th>
<th>PDF</th>
<th>MVC Chapter</th>
<th>lectures.mvc.toolkit Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVC and Graphics (10/24)</td>
<td>PowerPoint</td>
<td>PDF</td>
<td>MVC Chapter</td>
<td>lectures.mvc.toolkit Package</td>
</tr>
</tbody>
</table>

Part 1: Toolkit Command Interpreter User Interface
Use a Java toolkit to implement a user interface to manipulate the command interpreter. Use the model view controller design pattern to do so. This means you must write a controller object that calls the setter of the command interpreter. How the manually implemented user-interface looks like is entirely up to you. You can use Swing or AWT Toolkit widgets discussed in class or other widgets to create your user interface. The minimum requirement is that you provide a
way to set the string manipulated by the interpreter. If you use JTextField, look at the class material on how the manual user-interface of a BMI Spreadsheet was implemented, which I quickly went over in class.

**Regular Credit Part 2: Inheriting Bridge-Scene Painter**
Create a view class, tagged “Inheriting Bridge Scene Painter,” that displays the bridge scene and reacts to changes to models in the scene. This scene will be very similar to the console view you implemented in the last assignment, except that it will call a paint method in response to receiving a property change event. This means that the view class is a (direct or indirect) subclass of Component and implements the standard paint() method to draw all graphical objects in the logical structure of the scene model. This implies that the view object must register itself as a listener of all the model objects it paints. It also means that the view object will repaint the entire scene even if only part of it (say the arm) changes. This is inefficient but no worse than what OE and most applications do. (There is a way to paint only part of the scene; you override the update rather than paint method of a component. The advantage of overriding paint() is that you do not have to erase the previous contents of the component before drawing – the whole components is automatically cleared before paint() is called. Overriding update would require you to clear the area you are redrawing before doing the redraw, which is tedious to program.)

Your view should ignore the OE annotations in the models it displays, but should render each of the required properties of a shape correctly. You are free to consider also the optional properties of the avatar such as color, stroke, and font.

If you sailed through the previous assignment, you may want to create the alternative extra credit implementation given below.

**Alternative (Extra Credit) Part 2: Observing Bridge-Scene Painter**
Instead of using the above approach, illustrated in class, to create the view, fix a problem in AWT/Swing – there is no notion of a paint listener. Define a paint listener interface, tagged “Paint Listener,” which should include a method with the following signature:

```java
void paint (Graphics2D g)
```

Now create a subclass of Component, tagged “Observable Painter,” that provides a method to registers instances of this interface. Whenever the Java paint(Graphics) method is called (by repaint()) in this subclass, it calls the paint(Graphics2D) method in each of the registered paint listeners.

You will no longer have a single monolithic view object that paints all of the objects in the scene. Instead, you will create multiple view objects – one for each avatar and one for the background. (You are free to create even finer-grained view objects). A view object will no longer be a subclass of some window class. Instead it will be an observer or listener to the observable painter subclass you created. The order in which the view objects get registered with the...
observable painter will determine whether a drawing is on top or bottom of another because it will determine the order in which the paint methods are called. A view object will receive property change events from the model object it paints and thus will also register itself as a listener with these models. It will call the repaint() method in the observable painter, which in turn will call the paint methods in all of the paint listeners. Tag the class of this view object as “Observing Bridge Scene Painter”.

**Animating Demoing Main Class**

To demonstrate your observable and observables, write a main class that creates a scene object and displays an animation of it using both the painting scene view and ObjectEditor. Specifically, the main class:

1. Instantiates a scene object.
2. Displays it using your inheriting or observing view object.
3. Displays it using ObjectEditor.
4. Instantiates a command interpreter object.
5. Displays the controller user interface for the command interpreter.
6. Creates an animation that moves an avatar, sets its text, and rotates each of its rotatable parts.

Your main code should call methods only in the models. This means that it will not cause any changes to the controller widgets such as the text field you created for entering commands. The TAs will test this text field manually. Changes made from the application program to a widget do not cause listener events to be fired. For example, calling setText() on a JTextField does not cause the listener event to be fired. Thus calling widget methods in the main program does not help you demonstrate your widget listeners.

Similarly, The TAs will manually interact with your scene windows to test your mouse and key controller.

**Extra Credit: Command-Interpreter View**

If your interpreter has a read-only error property, then for extra credit, display it in your command interpreter user interface. This means you must now make the command interpreter an observable also and include a view that observes this object. Add the additional tag, “Observable” to the command interpreter if you do this part.

**Extra Credit: Action Listeners**

Add to your command interpreter user interface at least two “action components”. An action component is a menu item or a button. Each of them should perform some canned (pre-programmed) action on the simulation. For instance a menu item could move all of the avatars by some distance in the x direction and a button could do so in the y direction.
**Extra Credit: Animation Progress Bars/Sliders**
Connect a progress bar or slider to your main method that shows to what extent your animation in the main method has completed. You do not have to use MVC to display the progress bar as it will be too much work for the TAs to check that. However, if you have spare time, you are encouraged to use it.

**Extra Credit: Bridge Scene Controller**
Define a controller that listens to the mouse and key events of the window displaying the bridge scene. (This window will be the inheriting bridge scene object if you did not do the extra credit and the observable painter if you did.) The controller should keep track of the position of the last mouse click. Let us refer to this location as the last click point. (You can get this location by calling the `getPoint()`, `getX()`, `getY()` methods on a `MouseEvent`.) If the user types the letter ‘a’, ‘g’, ‘l’, or “r” in this window, then the Arthur, Galahad, Lancelot, or Robin avatar, respectively, should move to the last click point. If the user types the letter ‘o’ then all avatars should return to their original positions. Tag this class as “Bridge Scene Controller.”

In case your key listener does not receive key events, then call `setFocusable(true)` in the constructor of your component subclass. If this also does not work, make this class a subclass of `Panel` instead of `Component`. My experience is that these two steps are not necessary if a frame has only one input component, but I may be wrong.

**Constraints**
1. You should MVC for all user-interfaces except the progress bar.
2. As before, there should be no warnings from `ObjectEditor` – if there are spurious warnings, let me know.

*Be sure to follow the constraints of the previous assignments.*

**Submission Instructions**
- These are the same as in the previous assignment. The TAs will run the main method to see the test cases animate.
- Be sure to follow the conventions for the name and package of the main class.

Good luck!