CS 355 Lab #1: Simple Drawing

Overview

In this lab you will implement a simple interactive 2-D drawing program. This program is very basic—it includes only simple placement of the shapes without further interaction. You will extend it further in the next few labs.

The program *must* the overall model-view-controller structure:

- The model stores the shapes that have been drawn *independent of the interactions used to specify the shapes or the API used to drawn them on the screen*.
- The view component renders these accordingly using the Java 2D Graphics API.
- The controller handles the input events and make calls to the model to add new shapes or update them as they change. It should also maintain all state information for the user interface (selected drawing tool, selected color, etc.).

We will give you a shell that builds the basic GUI and provides a model-view-controller framework. You will need to provide all of the functionality for handling actions within the GUI. In particular, *you are responsible for all interaction and drawing within the primary drawing area.*

User Interface

The program should operate in a single window and include user-selectable menu options, a palette of user-selectable drawing tools, an area indicating the current color, and a primary drawing area. The primary drawing area should be at least 512×512. The shell program will build the menus, tools palette, and other GUI elements. Handling these are your responsibility. If you wish to, you are welcome to extend this shell to improve the interface, but that's not the aim of the program. Don't waste a lot of time making it look pretty—focus on the drawing portion.

Color Selection

Clicking on the button for the color selector should bring up the Java color selector. (See the shell program to see how this is done.) Once it returns to your program the selected color, you should update the current color indicator to show this color. All future drawing should be done using that color until the user selects a new one. Remember to set the initial indicator to whatever you set the initial color to.

Drawing Functions

Your program should implement drawing the following shapes. Each shape should be drawn (lines only) or filled with the currently selected color. When each shape is drawn, your program should add that shape to the stored model such that the most recent shape is drawn on top of the preceding ones. (You might find it

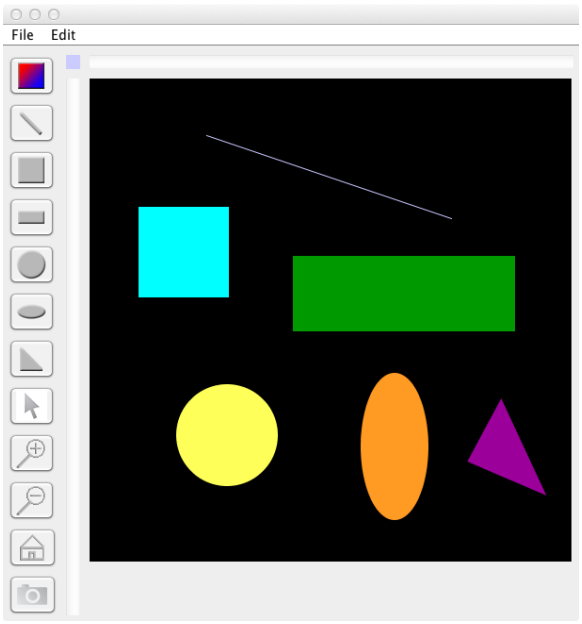

Figure 1: Lab #1 screen shot

simplest to add the shape to the model as soon as the drawing starts and then update it during the drawing, or you may add it once the user finishes drawing it.)

During the drawing you will need to refresh the display. This will also be necessary when Java's GUI system indicates that the area has been damaged and needs to be redrawn. The shell program provides the necessary functionality for double buffering: you'll draw to a new offscreen buffer and then update the drawing area with the contents of this buffer.

Lines

When the line tool is selected, a mouse-down event places one endpoint of a line. As the mouse is held down and dragged, you should show the line between the starting point and the current mouse position. Releasing the mouse button places the other end of the line.

Rectangles

When the rectangle tool is selected, a mouse down places one corner of the rectangle. As the mouse is held down and dragged, the current mouse position is used as the opposite corner. Releasing the mouse button places the other end of the opposite corner.

Please note that you should not assume which corner of the rectangle is the mouse down and which corner of the rectangle is the mouse up (or mouse move position). Just assume that they are opposite corners and figure out dynamically based on their position relative to each other what corners they actually are. This means as the user moves the mouse around the rectangle can "flip" both vertically and horizontally.

Squares

Interaction is the same as for placing rectangles, but the sides of the square are constrained to be the same length, which should be the smaller side of the rectangle indicated by the initial mouse click and the point where the mouse button is released.

Ellipses

Same interaction as for rectangles, except that an ellipse should be drawn such that it exactly fills this bounding box.

Circles

Same interaction as for squares, but a circle should be placed at the center of this bounding box (the equivalent square) such that it fills the square.

Triangles

The method for placing a triangle is different from the usual click-and-drag placement of the other shapes since it involves three points. The first click should place the first corner, the second click should place the second corner, and the third click should place the third corner and close off the triangle. After the third point is placed, the triangle should be added to the model and drawn.

Model

There is no interface provided for the model, so you can design it as you choose subject to the following specifications.

The model must extend the `java.util.Observable` class, and when anything changes it should use its own `notifyObservers` method to notify all registered `Observers`.

Your model should conceptually store an ordered list of shapes in back-to-front order. It should provide methods that allow the controller to add new shapes and to modify existing shapes. It should also provide a method for the viewer to query the model and retrieve the shapes in order.

The Shape Class

You should have an abstract `Shape` class that for now stores only the shape's color. You should have accessor methods for the color. Each shape is stored with the following information, each of which should also have accessor methods.

Lines

The Line class should store the two endpoints for the line.

Rectangles

The Rectangle class should store the location of the upper left corner, the height, and the width.

Squares

The Square class should store the location of the upper left corner and the size of the square.

Ellipses

The Ellipse class should store the center of the ellipse, the height, and the width.

Circles

The Circle class should store the center of the circle and its radius.

Triangles

The Triangle class should store the location of each of the three corner points.

Notes

The ways you represent the shapes in the model are not the same as the ways you draw the different shapes, nor are they the same as the parameters used to draw the shape using Java's 2D Graphics API. This is intentional—the model should be as general as possible, *independent of the choice of interface used by the controller or the means of drawing used by the viewer.* These representations will change in future labs.

Contrary to what you learned in CS 142, these classes in the model should **not** do any drawing of the shapes. Think of the model as data to be manipulated, not the contents of the screen (though obviously for a drawing program the two are the same). Drawing things on the screen is the role of the viewer and should only happen there.

Once the viewer gets the list of shapes from the model, it will need to draw each one. Since the objects in the model do not contain drawing methods, it's tempting to do a big `instanceof` switch when drawing, but this is a bit ugly. One way to deal with this is to create a `DrawableShape` class in the viewer with its own subclasses (`DrawableSquare`, etc.), then create a factory that takes a `Shape` and internally does a `instanceof` switch to create and return the necessary `DrawableShape` subclass. Depending on what you want to do with it, you could have these `DrawableShape` subclasses extend the corresponding `Shape` subclasses (which carries along their full functionality), simply store a reference to the corresponding `Shape` and reference it as needed, or remove any other extraneous information and store only what is needed for drawing. Notice that this isolates in one place the "impedance mismatch" across the abstraction between the model and viewer layers and contains all of the logic for converting the program's stored data to the external visual display. You can then simply iterate through the list of `DrawableShapes` and draw each one using polymorphism if you wish. Note that using this approach is optional: you can use other ways, including a brute-force if-elseif-chain and `instanceof` a single draw routine.

Program Shell

Our program shell handles the basic GUI elements and lets you focus on event handling and object drawing. It also provides a basic MVC architecture. You will write and plug in your own classes that implement the

controller and the viewer. The shell knows nothing about your model, which you should create separately. Your controller and viewer then interact with your model.

First, download and unzip the provided `src.zip` file. This will create a source directory with a subdirectory for the `cs355` package. Within this you will find a `solutions` folder with a stub `CS355.java` main file.

When you open up `CS355.java` you'll see a line like this:

```
GUIFunctions.createCS355Frame(null,null,null,null);
```

This line builds the Frame through which the program operates. The four `null`s are placeholders for classes you will pass to it. This is where you plug in the parts of your viewer and controller:

- A class that implements the `CS355Controller` interface — this is the main part of your controller
- A class that implements the `ViewRefresher` interface – this is the viewer and is responsible for all drawing
- A `MouseListener`
- A `MouseMotionListener`

The flow of information between the pieces is illustrated in Figure 2.

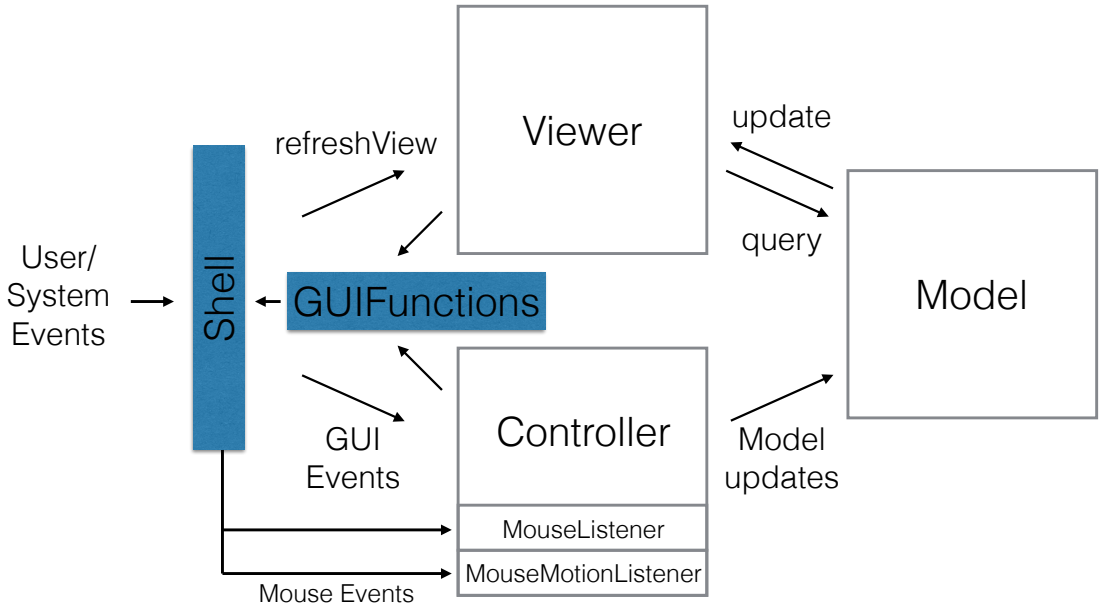

Figure 2: Flow of information between the shell and the M-V-C components

The Controller

This controller takes the form of a `CS355Controller` object. The interface for this object is included with the framework. Simply make an object that implements the `CS355Controller` interface, create a new instance of it, and pass it in as the first argument in the `CS355Frame` constructor. By doing so, you will be able to tell the framework how to handle events that take place, such as button presses and menu selections.

In response to events, your controller object may need to change settings or or otherwise affect display of the GUI. To do this, use the methods provided by the `GUIFunctions` class.

The View Refresher

The framework also has a drawing area. It is up to you to control what is drawn there, but we have simplified the process by allowing you to draw to a `Graphics2D` object instead of requiring you to figure out how to manage buffer strategies and redraw images yourself. You do this by creating a `ViewRefresher` and passing it as the second argument into the framework constructor.

Each time it redraws the screen, the framework will create a `Graphics2D` object that it then passed to the `ViewRefresher` you provide it. After your `ViewRefresher` has had a chance to draw on the `Graphics2D`, it displays the `Graphics2D` onto the canvas using double buffering. Simply create a new object that implements `ViewRefresher` and pass a new instance of it to the constructor of the framework. (This is very much like the "strategy" design pattern you have learned about/will learn about in CS 360). The `ViewRefresher` you implement should query the model and use what it learns to draw on the `Graphics2D`.

Sometimes the controller might need to initiate a screen refresh because it changed something in the interface (the selected color indicator, the current tool indicator, toggling on or off various layers that will be used in later labs, etc.). To do this, it should call the `GUIFunctions.refresh` method, which will initiate calling the `ViewRefresher`'s `refreshView` method with the appropriate `Graphic2D`.

Sometimes the screen will need to be refreshed because the data in the model changed. We will handle this using an Observer pattern, which is very useful for allowing potentially multiple view/controller pairs to collaboratively share the same model. The viewer should implement the `java.util.Observer` interface, register itself with the model using `addObserver`, and implement an `update` method that calls `GUIFunctions.refresh` to trigger a screen refresh sequence. When anything in the model changes, the model should use the `notifyObservers` method. (Remember to use `setChanged` first.) This will trigger the viewer's `update` method, which then begins the refresh sequence. The model should not do any drawing, nor should it directly call `GUIFunctions.refresh`. The controller should not use `GUIFunctions.refresh` when it changes things in the model—the model itself should notify all of its observers. (Though you might find it useful to have a way to let the controller notify the model that it changed something, depending on how you encapsulate things in the model.)

The Mouse Listener

In addition to processing button events and displaying your model, you must also be able to respond to mouse-clicks on the canvas. The framework will attach a `MouseListener`, which you provide, to the canvas. We don't provide an interface for this; a perfectly good one already exists in the default Java libraries. Just create an object that implements `java.awt.event.MouseListener` and pass it in as the third argument, and your listener will be notified at the appropriate times whenever the canvas is clicked.

Don't forget that you can get more specifics about the nature of the click (such as its position or which mouse buttons were involved) by using the methods of the `MouseEvent` object provided by the interface.

The Mouse Motion Listener

Since the program involves dragging the mouse while the button is pressed, you will also need to attach your own `MouseMotionListener`. Create an object that implements `java.awt.event.MouseMotionListener` and pass it in as the fourth argument, and your listener will be notified at the appropriate times whenever the mouse is moved.

Implementation Notes

You are welcome to implement your own framework if you wish, as long as it has the same functionality. We give you this framework in an attempt to make your life easier, not to restrict you in any way.

There are many ways to handle updating the shape continuously while it is being drawn. For lines, you might find it easiest to store one endpoint for the line where the mouse-down event occurs, then immediately place the other endpoint at the same place (essentially adding a 0-length line to the model). Subsequent mouse-drag events would result in updating the second endpoint and initiating a screen refresh. Similar strategies can be used for the other shapes.

Submitting Your Lab

To submit this lab, zip up your src directory to create a single new `src.zip` file, then submit that through Learning Suite. We will then drop that into a NetBeans project, do a clean build, and run it there. If you need to add any special instructions, please do so in the notes when you submit it. If you use any external source files, please make sure to include those as well.

Rubric

- Basic drawing (50 points total)
  - lines (10 points)
  - rectangles
    - drawing (5 points)
    - arbitrary corner placement / dragging (10 points total)
  - squares (5 points - correctly constrained rectangles)
  - ovals (5 points)
  - circles (5 points - correctly constrained ovals)
  - triangles (10 points)

- Refreshing correctly to show the shapes while being drawn (10 points)
- Correct handling of color (5 points total)
- Correct drawing order (5 points)
  - back-to-front in the same order as initially drawn
- Generally correct behavior (15 points)
  - this is a catch-all for overall correct or reasonable handling of everything else
- Correct model-view-controller structure (15 points)
  - model is independent of user clicks or drawing parameters
  - viewer is the only one that draws
  - controller (including mouse and mouse motion listeners) is the only one that handles user events
  - proper use of the observer pattern between the model and viewer

TOTAL: 100 points

Change Log

- August 25: Initial version for Fall 2014
- September 4: Corrected to specify that the model should extend the `Observable` *class* and the viewer should implement the `Observer` *interface*. (The initial version had class/interface backwards.)
- September 8:
  - Added figure for the flow of information between the components.
  - Clarified use of `DrawableShape` and that it is optional.