CS 145 Homework 4
TwixPix

1 Overview

Your objective in this homework is to acquaint yourself with conditional statements and loops, which enable you to write code that switches paths or repeats sequences. You will do this in the context of writing methods that might be used in slideshow software to transition between images.

This homework involves some math. Thinking about math by hacking away at the computer usually leads to frustration. You are strongly encouraged to draw pictures on paper and develop an intuition about the problems.

Each transition effect is described in some detail below and a visual demonstration is available at `http://youtu.be/048HBqwUfLA`.

2 Requirements

Complete the two classes described below. Place all classes in package `hw4`. Make all methods `static`.

2.1 TwixPix

Write a class `TwixPix` that defines several methods for transitioning from one image to another. You are encouraged to write one effect at a time, testing with `Main` before moving on to the next.

1. Define `static`, `final`, and `int` constants `ROLL_UP`, `WIPE_LEFT`, `CROSSFADE`, `DISSOLVE`, `CLOCK`, and `BOX_OUT`. These will be used by the generic `transition` method, described later, to identify which effect to apply.

2. Write the following effect methods. All accept two `BufferedImage`s and a `double` proportion in [0, 1]. The first `BufferedImage` is the image we are transitioning from, and the second is the image we are transitioning to. All return a new `BufferedImage`, which is a mix of the two parameter images. You must visit each pixel of this transition image and assign an appropriate color.

   (a) Write a method `dissolve`. It blends the corresponding pixels of the from and to images according to the proportion. When the proportion is 0.5, 50% of a pixel's "from color" is combined with 50% of the pixel's "to color." When the proportion is 0.75, 75% of the to color is combined with 25% of the from color.

   There's no builtin method for mixing colors. We recommend the following approach:

   i. Extract two corresponding colors from the `BufferedImage`s using the `getRGB` method. The colors will be `int`s, which are not immediately useful.

   ii. Convert the colors into `Color` objects using `new Color(anInt)`. `Color` objects allow you to pull out the channel intensities separately.

   iii. Use the `Color`'s `getRed`, `getGreen`, and `getBlue` methods and mix the three channels separately.

iv. Construct a new `Color` using the three mixed intensities: `new Color(red, green, blue)`.

v. Use `Color.getRGB` and `BufferdImage.setRGB` to assign the mixed color to your transition image.

(b) Write a method `crossfade`. At proportions less than 0.5, the transition image is a mix of the from image and the color black. At proportions greater than or equal to 0.5, the transition image is a mix of the to image and the color black. The closer the proportion is to 0.5, the blacker the transition image. The closer the proportion is to 0 or 1, the more the transition image looks like the parameter images.

Fading to or from black can be accomplished by determining the proportion's distance from 0.5. This distance will be in [0, 0.5]. Multiply it by 2 to get a weight in [0, 1]. Convert the pixel you wish to fade into a `Color` object, weight each channel separately, and construct a new color as described in `dissolve`.

(c) Write a method `boxOut`. At a proportion of 0.5, the to image is revealed in a rectangular box that is 50% of the width and height of the images. At 0, only the from image is visible. At 1, only the to image is visible.

(d) Write a method `wipeLeft`. At a proportion of 0.5, the left half of the from image is shown and the right half of the to image is shown. At a proportion of 0.75, the right three-quarters of the to image is shown and the left one-quarter of the from image is shown. No colors are mixed in this effect.

(e) Write a method `rollUp`. This effect makes the to image appear to push the from image out the top of the view. At a proportion of 0.5, the bottom half of the from image is shown in the top half of transition image, and the top half of to image is shown in the bottom half of the transition image. At a proportion of 0.75, the top three-quarters of the from image is shown in the bottom three-quarters of the transition image, and the bottom one-quarter of the to image is shown in the top one-quarter of the transition image.

(f) Write a method `clock`. This effect reveals the to image through a radial sweep of a clock's hand. At a proportion of 0.25, the first quadrant of the to image appears in the first quadrant of the transition image. At proportion 0.5, quadrants 1 and 4 are revealed. At proportion 0.75, quadrants 1, 3, and 4 are revealed. To determine whether a pixel in the transition image should come from the from or to image, use the following process:

i. Determine a pixel's x- and y-distances from the center of the image by subtracting off the coordinates of the middle pixel. Call these distances Δx and Δy. These two distances form the sides of a right triangle whose hypotenuse stretches from the center of the image to the pixel.

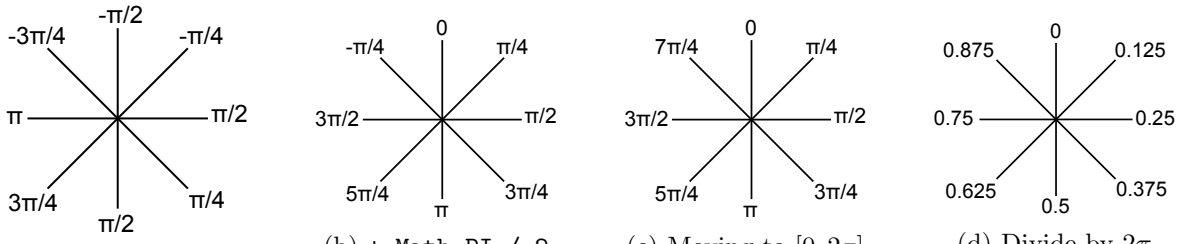
ii. Calculate the central angle θ of the right triangle. First we observe that the tangent relates these two side lengths:

$$\tan\theta = \frac{\Delta y}{\Delta x}$$

Now we solve for θ:

$$\theta = \tan^{-1}\frac{\Delta y}{\Delta x}$$

Java provides an inverse tangent method in `Math.atan2`. This method yields radians in $[-\pi, \pi]$. Figure 1a shows how the radian values change across the image.



(a) `Math.atan2` (b) `+ Math.PI / 2` (c) Moving to $[0, 2\pi]$ (d) Divide by 2π

Figure 1: Converting the result of `Math.atan2` to a more useful result can be accomplished with a little arithmetic manipulation.

iii. The result of `atan2` is a little awkward to work with. It'd be really nice if we could convert these radians into numbers that correspond to a real clock, with 0 radians at 12 o'clock. Let's start by adding $\frac{\pi}{2}$ to the radians, yielding the values shown in figure 1b.

iv. The radians values are looking mostly nice, but there's a discontinuity in quadrant 2. The radians dip below 0, which will make determining whether a pixel is before or after the sweeping hand of the clock more complicated. We can fix this by checking if the radians are negative. If so, add 2π to make an equivalent but positive angle. The new radian values are shown in figure 1d.

v. Finally, we can divide the radians by 2π to get a value in $[0, 1]$. Let's call this the *normalized angle*. If a pixel's normalized angle is greater than the proportion, the hand of the clock has not yet reached it and the from image's pixel is shown. Otherwise, the hand has already swept by and the to image's pixel is shown.

3. Write a method transition that accepts a `BufferedImage` from image, a `BufferedImage` to image, a `double` proportion in [0, 1], and an `int` transition type. The transition type will be one of the constants you defined earlier. This method returns the transition image produced by the effect identified by the type parameter. Since the actual generation of the transmission image is done by the methods described above, this method should do little but call the appropriate helper. Compare against the constants by name, not literal value.

2.2 Main

Write a class `Main`. It has a `main` method, which you are encouraged to use to test your code. Nothing in particular is required of it, but it must exist.

Your instructor has provided a graphical user interface (GUI) that allows you to interactively transition between images with a slider. The `SplittinImage` class relies heavily on your `TwixPix` class. To test, write enough of each specific effect method to allow the code to compile, and write the `transition` method in full. Then in `main` construct a `SplittinImage` with two images of the same size:

```
BufferedImage pre = ImageIO.read(new File("path/to/pre.jpg"));
BufferedImage post = ImageIO.read(new File("path/to/post.jpg"));
new SplittinImage(pre, post, TwixPix.CLOCK);
```

The `SplittinImage` class may be downloaded from `http://www.twodee.org/teaching/cs145x/homework/hw4/SplittinImage.java`.

3 Submission

To submit your work for grading:

1. Put the SpecChecker for this homework in your Build Path. Run the SpecChecker as a Java Application and fix problems until all tests pass.
2. Commit and push your work to your repository. Verify that your solution is on Bitbucket.

A passing SpecChecker does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The SpecChecker checks some of them, but not all.
- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, CS 145 moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don't want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.
- Your code must be submitted correctly and on time. Most excuses devolve into, "I started too late." The fix for this problem is not an extension.