CS 245 Homework 2

Flood Fill

due before November 18

1 Overview

In this homework, you will explore the stack abstract data type and a couple of floodfilling algorithms, which might be used to implement a image editor's paint bucket/fill tool.

The essential task of floodfilling is: 1) the user selects a fill color, 2) the user clicks on a pixel in an image, and 3) the image editor colors the clicked-upon pixel and all its like-colored neighbors with the selected fill color. Please watch `http://youtu.be/PujcpcI9Nig` to see a demonstration.

This operation may be implemented recursively, as described in this pseudocode:

```
floodfill x y colorToMatch colorToFill image =
  if colorToMatch is the same as colorToFill
    we're done

  if (x, y) is valid pixel and image[x, y] is colorToMatch
    image[x, y] = colorToFill
    floodfill right neighbor
    floodfill left neighbor
    floodfill bottom neighbor
    floodfill top neighbor
```

A recursive solution suffers from a terrible disease—StackOverflowExceptionitis—making it unsuited for coloring large swaths of pixels. The number of recursive calls grows exponentially and quickly exhausts memory allocated for the call stack. To avoid this problem, one can use an iterative algorithm instead.

The iterative algorithm makes no recursive calls but instead explicitly manages its own stack of floodfilling seed points. This algorithm, while better able to handle larger regions of filled pixels, is a bit more complex. The general idea is: 1) work down from the seedpoint until we hit the edge of the like-colored region, 2) work our way back up to the top of the like-colored region, coloring each pixel, and 3) as we do so, throw just one seedpoint from each like-colored vertical span to the left and right into a stack. The top of the stack becomes the next seedpoint and the process continues until the stack is totally empty. The algorithm is expressed by the following pseudocode:

```
floodfill x y colorToMatch colorToFill image =
  if colorToMatch is the same as colorToFill
    we're done

  make a stack for seedpoints
  push (x, y) onto seedpoints stack

  while seedpoints stack is not empty
    point = pop seedpoints stack
    lower point as long as point.y is valid and image[point] is colorToMatch
```

```
leftScanlineAlreadyPushed = false
rightScanlineAlreadyPushed = false

while point.y is valid and image[point] is colorToMatch
  image[point] = colorToFill

  if there's a pixel to the left
    -- see if there's a vertical span to the left which
    -- we need to add a seedpoint for
    if not leftScanlineAlreadyPushed and
       left neighbor is colorToMatch
      push left neighbor onto seedpoints stack
      leftScanlineAlreadyPushed = true

    -- see if the vertical span to the left has ended
    else if leftScanlineAlreadyPushed and
       left neighbor is not colorToMatch
      leftScanlineAlreadyPushed = false

  if there's a pixel to the right
    -- see if there's a vertical span to the right which
    -- we need to add a seedpoint for
    if not rightScanlineAlreadyPushed and
       right neighbor is colorToMatch
      push right neighbor onto seedpoints stack
      rightScanlineAlreadyPushed = true

    -- see if the vertical span to the right has ended
    else if rightScanlineAlreadyPushed and
       right neighbor is not colorToMatch
      rightScanlineAlreadyPushed = false

  raise point 1 pixel
```

2 Requirements

1. Write all code in package `hw2`.
2. Write a class `ImageUtilities` that has the following:
   (a) A `static` method `floodFillRecursive` that accepts two `int` parameters representing the seedpoint at which floodfilling is to start, a `Color` identifying the region to be filled, the `Color` that the region should be filled with instead, and the `BufferedImage` containing the pixels. In your implementation of this method, follow the recursive algorithm described above.

(b) A `static` method `floodFillIterative` that accepts the sam parameters as `floodFillRecursive`. In your implementation of this method, follow the iterative algorithm described above.

3. Write a class `ImageEditor` that has the following:

   (a) `JFrame` as its superclass.

   (b) A constructor that makes the frame appear and makes the application quit when the frame is closed.

   (c) A method `setImage` that accepts a `BufferedImage` parameter. The window is resized to match the size of the image and the image is drawn on the frame's canvas. (Probably you will want to create a subclass of `JPanel` to support this. This is an unspecified implementation detail.) When the user clicks on the image, the image is floodfilled starting at the clicked-upon pixel with some color of your choosing. Anytime you alter the image, you will need to issue a call to `repaint` to force the window to redraw.

   (d) At least two extra user interface features of your choosing. Some suggestions include: a menu bar and file menu that pops up an open dialog to load an image, the ability to draw shapes with the mouse or other input device, a mechanism for selecting the fill color, or a save dialog. You must indicate what features you implemented by posting a screenshot for each in a public Piazza post.

In the demo video linked above, you see the progression of the floodfilling operation. In order to show this progression, I temporarily sent my `JPanel` object to my floodfilling routines and forced an immediate repaint after I changed the color of a pixel with the following code:

```
panel.paintImmediately(0, 0, panel.getWidth(), panel.getHeight());
Thread.sleep(10); // throws an exception that you can ignore
```

3 Submission

This homework is part of a regular assignment and is graded by hand and with help from the SpecChecker. Your assignment is expected to fully meet the requirements above and the following:

1. Variable names should be meaningful and accurate.
2. Non-obvious parts of your code should be commented.
3. Code should be cleanly formatted and indented.

Your work will also be inspected for plagiarism. Please do your own work. 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. (If you find violators of this rule, please let me know.) Write your own code.

Put the SpecChecker in your Build Path. Run it as a Java Application (not a JUnit Test) and fix problems until all tests pass. Upload the resulting ZIP file to the W drive.

4 Files

- `http://www.twodee.org/teaching/cs245/2013C/homework/floodfill/speccheck_hw2.jar`