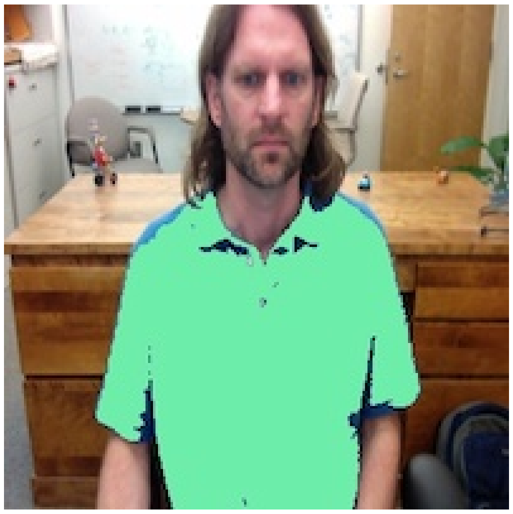
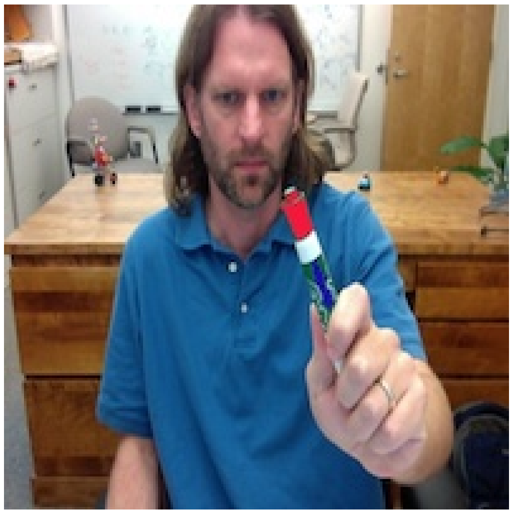
CS 10 PS-1
CamPaint

Introduction

The goal is to build a webcam-based painting program in which some portion of the image (say your hand, or a marker tip) acts as a "paintbrush".

The core computational problem is to identify the paintbursh in the webcam image. Here we base the recognition solely on color, so it should have a fairly uniform and distinct color. For example, I could wear a blue shirt and do torso-based painting, or I could use the green cap of a marker and more delicately paint something. (Both such "paintbrushes" below are recolored to distinguish them.)





To find uniformly-colored regions, we employ a "region growing" (aka "flood fill") algorithm. Region growing initializes a new region at some point that has approximately a specified target color. It then looks at each of the point's neighbors (either 4, just horizontally and vertically, or 8, also diagonally). Those that are also approximately the target color become members of the region, and their neighbors also need to be considered. The process continues until no neighbor-of-neighbor-of... points are the desired color. Thus the "flood fill" name: we expand outward from an initial point, as if a bucket of paint had been spilled there and spread to all the pixels of its same color. That detects one region; start again from another point (not already considered) to detect another region.

The basic structure of the algorithm is as follows:

```
Loop over all the pixels
  If a pixel is unvisited and of the correct color
    Start a new region
    Keep track of which pixels need to be visited, initially just that one
    As long as there's some pixel that needs to be visited
      Get one to visit, making sure it hasn't already been visited
      Add it to the region
      Mark it as visited
      Loop over all its neighbors
        If the neighbor is of the correct color
          Add it to the list of pixels to be visited
    If the region is big enough to be worth keeping, do so
```

An important point about the algorithm is the verb tense: "to visit" is different from "visited". You'll store the pixels that are still to visit in one place, and mark those that you've already visited in another. (More below.)

The target color is specified by mouse press, as in our simpler color point tracking example in class. The flood fill identifies connected regions of points that are approximately that target color. The paintbrush is the largest such region.

Implementation Notes

Provided for you are scaffolds for the two classes you need to complete: RegionFinder.java and CamPaint.java. "TODO" comments indicate what you need to fill in; you may also want to define additional helper methods, instance variables, etc. To help you develop and debug, a test driver is also provided: RegionsTest.java.

RegionFinder

- The built-in Java Point class holds x and y coordinates (here's an example of a class where we access instance variables directly, no getter). We can package up a list of them and think of it as a region. A list of these lists is then our set of regions.
- There are many ways we could test color similarity. In my solution, I simply compared the absolute value of each channel, and made sure each was less than a threshold. Note that the choice of threshold will affect region sizes, and how much a region can "bleed" into others. Euclidean distance is also fine. I set a value for a threshold; you should adjust this based on your implementation and how tight you want the region colors to be (e.g., in playing with the gradient images)
- We need to keep track of which points we've visited, so that we don't revisit them. An additional image actually provides a convenient way to do that, as it's exactly the same structure of the one we're looking at. It starts off as all black (getRGB is 0), and we can change the color of an (x,y) position to something else when we visit it. Thus we don't keep going there again and again.

```
visited = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
...
if (visited.getRGB(x, y) == 0) {
  ...
   visited.setRGB(x, y, 1);
}
```

- Consider 8-adjacency (NE, N, NW, W, E, SE, S, SW) neighbors using nested loops as in some of our image processing code.
- We also need to keep track of the neighbors (and neighbors of neighbors...) that need to be visited in the region we are growing. An ArrayList can do that; initialize it with the point itself. Then in the loop, remove the last point from the list, and handle it by adding its neighbors to the list (if they are the target color). Two other classes that we'll use more extensively soon have the same ability: Stack lets us push and pop objects, while Queue lets us enqueue and dequeue. Any of these approaches is fine.
- Again, note the difference in verbs: toVisit holds the pixels we still need to check for inclusion in the current region, while visited marks the places we've already been so we don't go back to them. Make sure you don't keep visiting the same pixels again and again! If you find the program freezing up in an infinite loop, that could be the issue.
- There's a nested loop structure for recoloring: loop over all the regions; for each, loop over all the points.

CamPaint

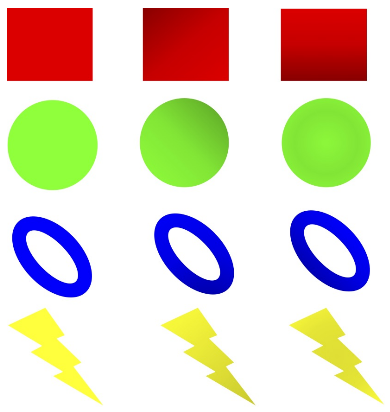
- The basic idea is that the region finder gives regions for each frame of the camera, the largest of which is considered the paintbrush. The paintbrush leaves a trail over time as it moves around. That's the painting. Ex:

- The basic structure is like other webcam code. I've provided some instance variables and key commands to set the tracking color, save snapshots, and control which image is being shown (in the draw method). Here, live webcam is just the streaming video with no processing; recolored image is from region finder (as in the static version and the example regions at the top); and the painting is the tracks left by the paintbrush over time (as in the blue smiley above).
- You need to plug in calls to your region finder within `processImage`, in order to find the largest region. Be sure to give the region finder the image and the color (and that those aren't `null`).
- Once the region finder gives the location of the brush (the largest region), update the painting accordingly. That is, the pixels that are in the largest region should be colored in the painting, thereby leaving a trail. You can color them however you like; my sample solution is monochromatic, but you could transfer colors from the webcam, allow the user to set colors, etc.
- Also handle the mouse press to set the tracking color, and the draw method to display the appropriate image as mentioned above.
- In general, webcam processing can be flakey. Work in a well-lit room. Do all the core development with static images first via the test code (i.e., no webcam required). If you have problems with your webcam, do everything else and have your partner (if you have one) test that part. If you don't have a partner, contact your TA.

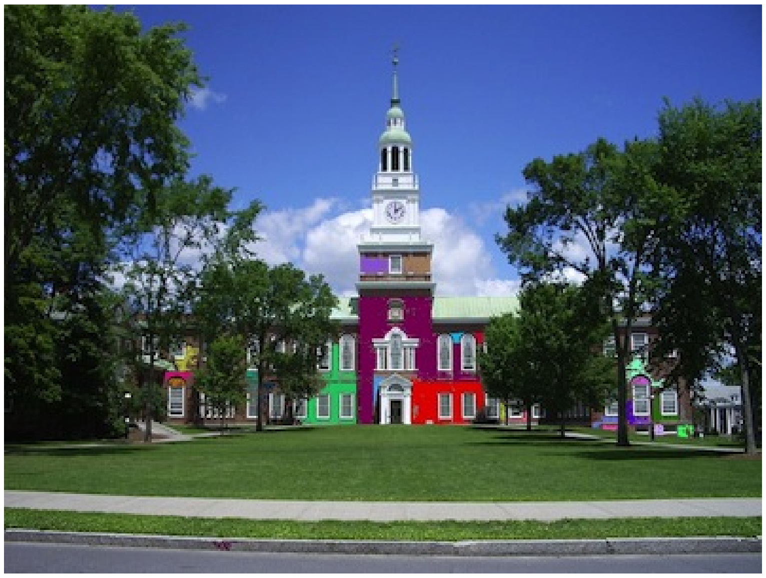
Exercises

For this problem set, you are allowed to work with exactly one partner. Note that you do not have to work with a partner, and if you do, you will both be given the same grade; there is no penalty (or bonus). Re-read the course policies. You should weigh whether you will get more out of this assignment working alone or with someone else; there are advantages to both. If you want us to set you up with a partner, please fill out the form on Canvas.

1. First implement `RegionFinder`, including the region growing algorithm, a method to recolor the image so that regions show up distinctly, and a method to find the largest region.
2. Do not even think about the webcam stuff until region finding works well on static images. Test it in the provided RegionsTest scaffold. I have provided on Canvas an image "shapes.jpg" with a number of shapes of fairly uniform colors. (I also provided the powerpoint file in case you want to modify for additional test cases.)



The rightmost column is solid, while the other two columns have some gradients, so you can evaluate the effect of your color similarity test. The top row should be identified by the first case in the provided RegionsTest (more or less, according to the gradient); make test cases yourself for the others, along with any other images you care to test. For a bigger challenge, try Baker. For example, here's what I get for the brick-ish regions (R=130, G=100, B=100) with at least 50 pixels in the usual Baker image, recolored to random colors:



Submit a corresponding image from your region growing algorithm (and feel free to submit additional ones). Depending on choices of parameters, it may be somewhat different from mine; that's okay. Briefly describe the implementation and parameter choices you made and their impact on the detected regions.

3. Now plug that into the provided webcam-based GUI scaffold. Set the target color by mouse press, and paint according to the detected brush (largest region). Submit screenshots of you / your partner's work, both an image of webcam with recolored regions and a resulting painting. Briefly describe the utility and limitations of region growing in this context.

You may obtain extra credit for extending and enhancing the app. Only do this once you are completely finished with the specified version. Make a different file for the extra credit version, and document what you did and how it works. Some ideas:

- Allow multiple brushes
- Allow pause/restart of the painting, easy change of colors and brushes, etc.
- Account for the size, shape, and trajectory of the specified paintbrush to filter out some of the spurious ones

Submission Instructions

Submit a single zipfile holding the following:

- Your completed versions of the two classes: RegionFinder and CamPaint.
- If you want to show off more extensive testing, you can also provide an updated RegionsTest.java and input images, but that is not required.
- Screenshots of static region finding test cases.
- Saved images from the webcam in action, both recolored images and resulting painting.
- A document with your brief dicussion of region growing (both static and streamed).

Grading Rubric

Total of 100 points

Correctness (70 points)

5 Matching color
5 Starting region growing at appropriate pixels
5 Keeping track of visited pixels
5 Keeping track of to-visit pixels
5 Visiting correctly colored neighbors
5 Keeping big-enough groups of points as the regions

10 Recoloring image based on detected regions
10 Finding the largest region
5 Drawing the appropriate image
5 Setting tracking color
10 Updating the painting according to the paintbrush

Structure (10 points)

4 Good decomposition of and within methods
3 Proper used of instance and local variables
3 Proper use of parameters

Style (10 points)

3 Comments within methods and for new methods
4 Good names for variables, methods, parameters
3 Layout (blank lines, indentation, no line wraps, etc.)

Testing (10 points)

5 Static image (Baker, and any others you want) region detection and discussion
5 Webcam region detection, painting, and discussion