CS 116x Winter 2015
Craig S. Kaplan

Assignment 08: Image Processing

Due date: Wednesday, 25 March, 12:00pm

Question 1: Miniature Faking

Potentially useful functions:

- Global: `brightness()`, `color()`, `colorMode()`, `hue()`, `loadImage()`, `map()`, `saturation()`
- Methods and fields of `PImage`: `copy()`, `filter()`, `get()`, `loadPixels()`, `pixels`, `updatePixels()`

Miniature Faking is a trick that can be applied to digital photographs, in which an image (usually a high-angle photo of a relatively flat landscape) is processed to look like a close-up of a tiny model. Most of the time, miniature faking relies on two main visual tricks to achieve its characteristic look:

- **Saturation:** the saturation of the colours in the source image is boosted to give an unnatural, toy-like appearance.
- **Blurring:** The main secret ingredient in miniature faking is the simulation of a lens with a narrow depth of field (a narrow range of depths in which things are in focus). Assume that depths change gradually with vertical position in the image, so that the bottom of the image is close to the viewer and the top is far away (as is usually the case with aerial photographs). As you get farther from the focal depth, objects in the photograph should be progressively blurrier. So we blur the image, but vary the amount of blur to be maximal at the top and bottom of the image, and zero in them middle.

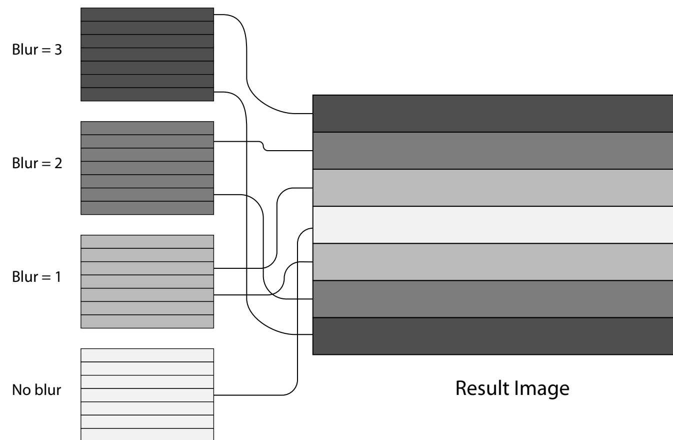
Your task is to create a Processing sketch that can perform miniature faking on an input image. This will require a couple of steps, but the whole algorithm can fit into a single `setup()` function and doesn't need more than about 50 lines of code. I recommend proceeding as follows:

- First, write a short sketch that loads an input image, sets the size of the sketch window to the size of the image, and then displays the image on the screen. (Eventually you'll display the result image and not the input image, but it never hurts to start by writing a complete sketch that solves a much simpler problem.)

- Next, add some code to boost the saturation. Use the method of looping over all the pixels in the image's `pixels` array, not forgetting to bracket the loop by sending the `loadPixels()` and `updatePixels()` messages to the image. (If you prefer, you can use the `get()` and `set()` methods of `PImage`.) If you work in the HSB colour mode, then it's easy to read each pixel, add a fixed amount to its saturation, and write the resulting colour back into the image (or into a second image). As a test, display the resulting image to see that it really does look super-saturated. You might even want to prepare some low-saturation test images (in Photoshop) to use as inputs to make sure this step does what you expect.

- Now it's time to add the blurring. Unfortunately, Processing doesn't have the kind of blur filter that we want (i.e., one that blurs by different amounts in different places) built in. But we can simulate one in a neat way.

  1. Create an array of, say, 20 images. Set each element of the array to a new copy of the saturated image. Then blur each copy by a different amount, without altering the source image. Choose blur amounts that grow progressively up to a maximum. For example, the nth image might be blurred using radius $n/2$.
  2. Now create a new image to hold the miniature faking result. I recommend doing this using the *zero*-argument `get()` method of `PImage`, in order to create an exact copy of the image from the saturation step.
  3. Next, you're ready to fill in the pixels of the result image row by row. Each row of the result image is taken from the corresponding row of one of the blurred array images, or from the unblurred but saturated source. The trick is to compute, for every row of the result image, which blurred image to take a row of pixels from. Rows should be unblurred in a narrow strip near the middle of the result image, and progressively more blurred towards the top and bottom. If you were starting with a really small image, the organization of rows might look something like this:



> Because your input image is likely to have many more rows than the number of blur levels you construct, each blurred image is likely to contribute a number of rows to the final result image.

Note that you don't need to write nested loops in order to complete this assignment. The saturation step can just iterate over all the pixels in the flat array of an image's pixels. And you can copy a whole row of pixels at a time from one image to another by using the `copy()` method in `PImage`.

Put all the code into a single `setup()` function. You don't need a `draw()` function, or any other global functions or global variables, though you can move some of the code into helper functions if you want. Upon startup, your sketch should think for a while (my implementation takes 5–10 seconds to create the final image), and then display the result in a window of the right size to accommodate it. You can also save the result image to a file if you want, but that isn't required. If you get stuck on the last part of the blurring step above, for part marks generate any result image that mixes multiple blur levels together.

You can download the test image of London from the course webpage, or substitute your own image. But your implementation should be able to work on any input image just by changing the filename in the source code. You can also choose to vary the number of blur levels you calculate, but keep in mind that the number you choose affects how long it takes for the algorithm to run.

What to submit: On LEARN, you should submit the sketch `A08TiltShift`. Submit the entire sketch folder.