CS 245 Preassignment 1
Interpolant

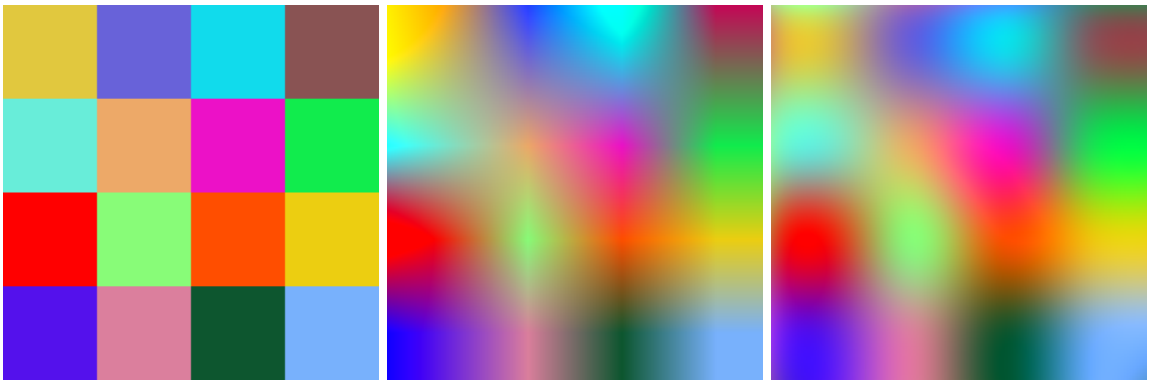
1 Overview

If something is *digital* it can be counted with our fingers. (The Latin also admits toes.) When you take a digital image, it is comprised of a finite collection of splotches of color, which we really could count if we wanted to. The music our generation listens to is also digital. We don't capture exactly every moment of every guitar lick. We only sample; we retain just a subset of readings of the pressure waves that hit our microphone.

Our grandparents lived in an *analog* world. Their vinyl records and photographic film continuously captured the world. If they claim their music was more authentic than yours, you must concede. However, digital reigns today because digital media is much cheaper to produce and transmit, and hey, if we take enough digital samples, we never even know that our media is not continuous.

What happens if we don't take enough samples? Suppose you take a 320×240 photo and then try to print an $10" \times 8"$ portrait[1], you will end up with a sorry-looking picture. Each of the original pixels will cover a $\frac{10}{320} \times \frac{8}{240} = 0.03125" \times 0.03333"$ area. This is a bit less than a square millimeter. Pixels will be jarringly visible. We are trying to project too little information on too large an area.

This preassignment deals with the problem of scaling digital images. Since the source image and the scaled destination image are not the same size, pixels will not match up perfectly. We must define how to compute color values for pixels that didn't actually exist in the source image. This definition is formalized in an *interpolant*. We will look at three popular interpolants: nearest neighbor, linear, and cubic, which span the tension between quality and complexity. The effect of these interpolants on an 8×8 image scaled to 256×256 can be seen in figure 1.



(a) Nearest Neighbor (b) Linear (c) Cubic

Figure 1: An 8×8 image scaled using three different interpolants.

Nearest neighbor interpolation is the simplest but its blocky result may not be aesthetically pleasing. The intuition behind nearest neighbor is that we virtually align the source and destination images. The color of a pixel in the destination image is the nearest pixel in the source image. Unlike

[1] Yes, these two areas have different aspect ratios. Your forehead will be abnormally short.

the other two interpolants, no pixels are blended together. We can compute the location of the source pixel nearest to the destination pixel with following equations:

$$\text{src}_x = \text{round}(\frac{\text{dst}_x + 0.5}{\text{dst.width}} \cdot \text{src.width} - 0.5)$$
$$\text{src}_y = \text{round}(\frac{\text{dst}_y + 0.5}{\text{dst.height}} \cdot \text{src.height} - 0.5)$$

You should question the semantic meaning behind these equations. Let's ignore the 0.5s in our discussion. First, we determine the proportion of the destination pixel within the destination image. This gives a unitless value, which we can freely apply to the source image dimensions to get a possibly fractional pixel—a subpixel—in the source image's bounds. We then round to get the nearest integral pixel location.

Linear interpolation is smoother and more complex. It uses the fractional portion of the source subpixel value to weight and blend the four source pixels surrounding the subpixel. We first find the pixel at the least corner of the four-pixel region:

$$f_x = \frac{\text{dst}_x + 0.5}{\text{dst.width}} \cdot \text{src.width} - 0.5$$
$$f_y = \frac{\text{dst}_y + 0.5}{\text{dst.height}} \cdot \text{src.height} - 0.5$$
$$\text{src}_x = \lfloor f_x \rfloor$$
$$\text{src}_y = \lfloor f_y \rfloor$$

Next, we find how "into" the four-pixel region we are by computing the difference between the fractional pixel and the integral corner:

$$w_x = f_x - \text{src}_x$$
$$w_y = f_y - \text{src}_y$$

These values are in [0, 1] and represent how close we are to each corner. Think of them as percentages—"I'm 80% of the way to the right edge, but only 10% away from the bottom." We'll now use them to compute a weighted blended of the four neighbors:

$$\text{bottom blended color} = (1 - w_x) \cdot \text{color at } (\text{src}_x, \text{src}_y) + w_x \cdot \text{color at } (\text{src}_{x+1}, \text{src}_y)$$
$$\text{top blended color} = (1 - w_x) \cdot \text{color at } (\text{src}_x, \text{src}_{y+1}) + w_x \cdot \text{color at } (\text{src}_{x+1}, \text{src}_{y+1})$$
$$\text{color at } (\text{dst}_x, \text{dst}_y) = (1 - w_y) \cdot \text{bottom blended color} + w_y \cdot \text{top blended color}$$

These are the famous linear interpolation equations which drive most of your computer games.

Cubic interpolation is a bit wild. This interpolant has already been implemented for you and is available in the homework repository.

I'm happy to discuss these equations with you further if you are interested. For this homework, it's enough to be able to apply them. If you want a better sense of what's happening, try the equations with some made-up image sizes and pixel coordinates.

Your task is two write some code to scale an image using these three interpolants. Along the way, you will see how Java's `interface`s help you write reusable code. Instead of writing three different scaling methods, you'll write just one—which happens to accept an interpolant as a parameter.

2 Requirements

Specifications do not tell you how to solve a problem—just what pieces may be used. The classes and methods described below will need to be thought about and pieced together using your own good mind. You will likely need to read this section many times.

Your solution is to meet the following specification:

1. Write all code in your fork of the class Bitbucket project, in package `pre1`.
2. Write a class `Utilities` that holds a hodge-podge of methods that will make writing your other classes easier. It has the following specification:
   (a) Has nothing but `static` methods.
   (b) Has a method `clamp` that accepts three `int` parameters: a lower bound, an upper bound, and a value. It returns the value clamped between the two bounds. That is, if the value is within the bounds inclusively, the value itself is returned. If less than the lower bound, the lower bound is returned. If greater than the upper bound, the upper bound is returned.
   (c) Has a method `getColorClamped` that accepts a `BufferedImage` parameter and `int` parameters for a xy pixel coordinate. It returns the color of the specified pixel in the image. However, the x and y coordinates are clamped to the image's valid pixel range. For instance, if one tries to access pixel (-1, 72) in a 100×60 image, the color for (0, 59) will be returned. The color is returned as a `Color`. (Hint: `BufferedImage.getRGB` yields an `int` representation of a color, not `Color`, but a `Color` can be made from an `int`.)
   (d) Has a method `scale` that accepts four parameters: a `BufferedImage` to scale, an `int` target width, an `int` target height, and an `Interpolant`. It creates a brand new `BufferedImage` of the target dimensions. The color of each pixel in the scaled image is determined by proportionalizing the pixel's coordinate and looking up these proportionalized coordinates in the original image using the interpolant.
3. Write an `interface Interpolant` that represents the abstract notion of approximating colors that appear between others. It has the following specification:
   (a) Imposes a method `interpolatePixel` that accepts two `float` parameters and a `BufferedImage` parameter. The two float parameters are a proportionalized x-coordinate and a proportionalized y-coordinate, both in [0, 1]. These coordinates are applied to the image by this interface's subtypes, and the color at the applied location is retrieved according to the rules of the interpolant. It returns a `Color` object.
4. Write a class of `NearestNeighborInterpolant` with the following specification:
   (a) Conforms to `Interpolant`.
   (b) Has a method `interpolate1D` that accepts three `float` parameters: a proportion and two values. It performs nearest neighbor interpolation. If the proportion is less than 0.5, the first value is returned. Otherwise, the second value is returned.

(c) Implements `interpolatePixel` according to the nearest neighbor interpolation equations described above. Clamp the source pixel location that you compute to be in bounds.

5. Write a class `LinearInterpolant` with the following specification:

   (a) Conforms to `Interpolant`.

   (b) Has a method `interpolate1D` that accepts three `float` parameters: a proportion and two values. It performs linear interpolation, which blends the two values together linearly. For example, if the proportion is 70%, then we are 70% of the way between values a and b. We want to take 70% of b and mix it with 30% of a. In general, we can express this as: $(1 - p) \cdot a + p \cdot b$.

   (c) Has a method `interpolate1D` that accepts three parameters: a `float` proportion and two `Color`s. It performs linear interpolation, which blends the two `Color`s together linearly. This can be done by linearly interpolating each of the three color channels separately. It returns the interpolated color as a `Color`. (Hint: there are several ways to do this. If you want to do the least amount of casting, check out `Color.getRGBComponents`.)

   (d) Implements `interpolatePixel` according to the linear interpolation equations described above. Clamp all source pixel locations that you compute to be in bounds.

6. Include the provided `CubicInterpolant` class as a further example of an `Interpolant`.

7. Write a class `Main` with the following specification:

   (a) Has a `main` method. What it does is not specified, but you are suggested to use it to test your code. Relying exclusively on the SpecChecker to test things will rob your brain of some neurons that are in your best interest to grow. You may be interested in `ImageIO.read` and `ImageIO.write` to load and save images.

3 Submission

This homework is a preassignment and is graded with help from the SpecChecker. To submit your work for grading:

1. Put the SpecChecker for this homework in your Build Path.

2. Run the SpecChecker as a Java Application (not a JUnit Test) and fix problems until all tests pass.

3. Commit and push your work to your repository. If you are resubmitting an earlier assignment, email me. The time of your email will determine the submission week.

The SpecChecker cannot check everything. Your assignment is also expected to fully meet the requirements above and the following:

- 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. (If you find violators of this rule, please let me know.)

- Work must be submitted according to course policies on deadlines. To be eligible for later-week submission, you must have at minimum the skeletons for all specified classes and methods in your repository by the homework deadline.