CS1114 Assignment 5 Part 1

out: Friday, March 30, 2012.
due: **Friday, April 6, 2012, 9PM.**

This assignment covers two topics: upscaling pixel art and steganography. This document is organized into those two sections.

1 Upscaling pixel art

Artwork from video games in the 80s and 90s was often painstakingly drawn pixel by pixel. Every pixel in a sprite counted: important details such as an eye or an ear consisted of only a handful of pixels. When we play these classic games on our modern systems (for example, via ZSNES on a computer), it is quite natural for us to want to play at a higher resolution than the meager 512×478 (or less) that they were originally designed for. To do this, we need some way to make the original images bigger. In this assignment, you will implement and compare multiple algorithms for scaling up pixel art.

1.1 Nearest neighbor

The simplest algorithm for scaling up images is known as nearest neighbor. For each pixel in the target larger image, we inversely map it to a location in the original image. We identify the closest real pixel as the nearest neighbor and set the pixel in the target image to be the same.

$\Longrightarrow$ Implement `NN_upscale(img, scale_factor)` in `NN_upscale.m`.

1.2 Linear interpolation

Recall from lecture that interpolation involves figuring out a value at a point where you don't have data, but inside the range where you do. Let's start by considering linear interpolation in one dimension.

Your first task is to write a function called `lerp`, which takes in a 1D vector (representing samples of a function) and an x value, and outputs the estimated function value at x using linear interpolation. Linear interpolation has three main steps: finding the two neighboring data points, computing the weights for the two points, then use the formula below for doing linear interpolation.

$$v_{new} = v_{left} \cdot (x_{right} - x) + v_{right} \cdot (x - x_{left})$$

$\Longrightarrow$ Implement the `lerp` function in `lerp.m`.

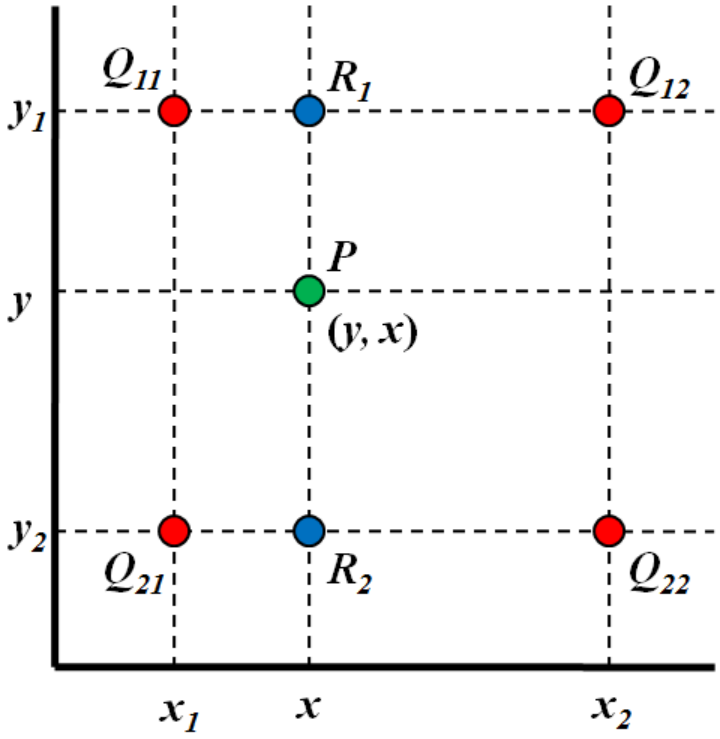

Figure 1: Bilinear interpolation.

Next, let's consider a 2D matrix of values at integer grid locations (e.g., a grayscale image). To interpolate values on a 2D grid, we can use the 2D analogue of linear interpolation: *bilinear interpolation*. In this case, there are *four* neighbors for each possible point we'd like to interpolate, and the intensity values of these four neighbors are all combined to compute the interpolated intensity, as shown in Figure 1.2. In the figure, the Q values represent intensities. To combine these intensities, we perform linear interpolation in multiple directions: we first interpolate in the x direction (to get the value at the blue points), then in the y direction (to get the value at the green points). We can implement this by calling `lerp` three times.

$\Longrightarrow$ Implement bilinear interpolation in `bilerp.m`. This function takes in a 2D matrix (an image, really), and an x, y value to compute the intensity at. You should return the computed intensity. This function must call your `lerp` function.

$\Longrightarrow$ Implement `BL_upscale(img, scale_factor)` in `BL_upscale.m`.

1.3 Cubic interpolation

We can obtain better interpolation results if we use a higher degree polynomial to interpolate values. If we use a function of degree 3, then we are using cubic interpolation. The general form of a degree 3 polynomial is $f(x) = ax^3 + bx^2 + cx + d$

with a derivative of the form $f'(x) = 3ax^2 + 2bx + c$. To obtain a formula for cubic interpolation, suppose we are interpolating in the region [0,1]. Then we have $f(0) = d, f(1) = a + b + c + d, f'(0) = c, f'(1) = 3a + 2b + c$. We can solve this system of equations for a, b, c, d in terms of $f(0), f(1), f'(0), f'(1)$. $f(0)$ and $f(1)$ are simply the actual values at the endpoints. We can approximate $f'(0)$ using the slope of the line between $x = -1$ and $x = 1$, (similarly we can approximate $f'(1)$ using the slope between $x = 0$ and $x = 2$), resulting in a polynomial approximation known as a Catmull-Rom spline.

$\Longrightarrow$ Implement 1D cubic interpolation in `cerp.m`. When interpolating in the first or last interval of your input, make a reasonable decision on what to do for the missing left-most and right-most data points and note it in a comment.

$\Longrightarrow$ Implement bicubic interpolation in `bicerp.m`. This function takes in a 2D matrix (an image, really), and an x, y value to compute the intensity at. You should return the computed intensity. This function must call your `cerp` function.

$\Longrightarrow$ Implement `BC_upscale(img, scale_factor)` in `BC_upscale.m`.

1.4 Scale2x/AdvMAME2x

Suppose we wanted to scale up an image by a factor of two. For every source pixel, there will be four destination pixels. First, we assign the four destination pixels to be the same value as the source pixel. Then, if the pixel to the left of the source and the pixel above the source are the same value X, and if neither the pixel below the source nor the pixel to the right of the source are this value X, then the top left destination pixel becomes this value X. We compute the other three destination pixels in a symmetric fashion.

$\Longrightarrow$ Implement `SC2_upscale.m`.

1.5 Scale3x/AdvMAME3x

This is similar to the previous filter, though we generate nine destination pixels for each source pixels, as we are scaling up an image by a factor of three. Again, we start by assigning the nine destination pixels to be the same color as the source pixel. We then assign the four corner pixels of the output using the exact same method as in the previous scaling algorithm. The four edge pixels are a bit more complicated. We use the top edge pixel as an example, and the other four cases are symmetric. We set the top edge pixel to be the same color as the pixel above the source if either of the following conditions is satisfied:

- The condition for setting the top left pixel of the output was satisfied, and the source pixel is not the same color as the pixel to its upper-right

- The condition for setting the top right pixel of the output was satisfied, and the source pixel is not the same color as the pixel to its upper-left

$\Longrightarrow$ Implement `SC3_upscale.m`.

1.6 Eagle

Eagle is another filter which scales up the image by a factor of two, resulting in four destination pixels. We start by assigning the four destination pixels to be the same color as the source pixel. If the top, left, and top-left pixels (relative to the source pixel) are the same color, then we set the top-left destination pixel to be this color. The other three pixels are done in a symmetric fashion.

$\Longrightarrow$ Implement `EAG_upscale.m`.

1.7 Comparison of algorithms

We have included sample sprites in a zip file for you to test your scaling algorithms on. Feel free to use sprites of your favorite characters and see how well your algorithms do on them.

$\Longrightarrow$ Implement `compare_upscale2.m`, which will accept as input a single RGB image (a 3-dimensional matrix), and then display titled figures on the screen: the original, and an image twice the size of the original produced by the applicable previously mentioned algorithms

$\Longrightarrow$ Implement `compare_upscale3.m`, which will accept as input a single RGB image (a 3-dimensional matrix), and then display titled figures on the screen: the original, and an image triple the size of the original produced by the applicable previously mentioned algorithms

2 Steganography

If you have a message to transmit that you want to ensure only the intended recipient can read, you would be correct in wanting cryptography to make sure that any prying eyes can't decipher the message. However, what if you have a message to transmit that you don't want anyone to know you sent? This is where steganography, the process of writing hidden messages such that no one even suspects its existence except the sender and recipient, comes in. In this assignment, you will implement functions for hiding and deciphering text in RGB images. Your implementations should be compatible with other students' implementations - a good way to test your code would be to send each other hidden text. NOTE: In this part of the assignment, we will be

using images in the uint8 format, so make sure you are using `im2uint8` for converting your images.

As a prerequisite for hiding and deciphering text, we need a numerical representation of the alphabet that we can embed into the image. We will use 0 through 25 to represent a-z, 26 to represent a space, 27 to represent a period, and 28 to represent the end of the message. You should note that these are not the corresponding integer values for the alphabet in MATLAB strings.

To hide a text of length n, it might be tempting to choose a color channel in the image and overwrite the first n pixel values with our alphabet and be done with it. However, this will likely result in significant irregularities in the output image and raise the suspicions of anyone looking at the image. To be a bit more clever about this, we will convert our alphabet to binary numbers and replace only the least significant bit (LSB) of each pixel value: this requires 5 pixels for every character in our text, will alter each pixel value by at most 1. For example, if a pixel in an image has the value 5, which is 101 in binary, and we need to embed a 0, it becomes 100 in binary, which is 4.

$\Longrightarrow$ Implement `hide_msg.m`, which accepts as input an RGB image, an integer indicating which channel to hide the message in, and a string, and outputs an image with the string hidden in the appropriate channel.

$\Longrightarrow$ Implement `decipher_msg.m`, which accepts as input an RBG image and an integer indicating a channel, and outputs any text hidden in the specified channel.

For information about string manipulation in MATLAB, you should consult this link.