HOW DO YOU SCALE A GRAYSCALE IMAGE?

In the following, we are assuming that a scaling factor *S* is used, and that when *S* is larger than 1.0, the image is magnified (and the dest pixels are smaller than the source pixels), and when *S* is smaller than 1.0, the image is reduced (and the dest pixels are larger than the source pixels). In the actual implementations, we do not assume isotropic scaling, and provide two scaling factors: S_x and S_y.

Sampling.

The simplest scaling method uses *sampling*. If the image is reduced, we call it *subsampling*; if the image is enlarged it is *replicative sampling*. While this method is fast, it gives relatively poor results for both situations.

If the image is reduced, subsampling produces aliasing by the Nyquist theorem. Here's a simple illustration. Suppose the image consists of alternating light and dark pixel columns, and you subsample with a scale factor of 0.25. You are choosing pixels from every fourth pixel column. Every pixel you choose will be either light or dark, depending on where you start. The high frequency signal has entirely disappeared in the subsampled version.

Now suppose the scale factor is 0.24, the first column you pick is dark, and you choose the column integer by truncating the floating point value. The second column you pick will be at 1/0.24 = 4.17 pixels, which is rounded to 4, so it will also be dark. The third will be at 8.33; rounded to 8 is a dark column. And so it continues, with the fourth at 12.5, the fifth at 16.67, and the sixth at 20.83, all dark columns. But the seventh, at 25, will be a light column. The next 5 columns will be also be light, until you get to the thirteenth, at 50.0, which is another dark column. Etc. You have a signal with a periodicity of 12 in the subsampled image. Nothing like this existed in the original. The low-frequency signal in the subsampled image, which corresponds to a periodicty of 50 pixels in the original, appeared through *aliasing*. To remove the aliasing, it is necessary to remove frequencies in the image whose periodicity is less than twice your sample spacing, because signals from those higher frequencies are *aliased down into your reduced bandwidth*. So here's the general rule to remove those high frequencies and avoid aliasing: ***Use a lowpass filter before subsampling***.

If the image is enlarged by a large factor, pixel replication will give a blocky appearance. Each pixel will be magnified into a large block of pixels. Sharp edges will be enlarged, and lines that are nearly horizontal or vertical will have a stair-step appearance at their edges.

Area mapping (or area averaging) and lowpass filtering.

When an image is scaled, each *dest* pixel will partially or completely cover one or more *source* pixels. In *area mapping*, the dest pixel *value* is found by assigning it the average of the source pixels it corresponds to, weighting by the fractional area of each source pixel that it partially (or wholly) covers.

Strict area mapping is fairly expensive, compared to using a lowpass filter that averages entire pixels followed by downsampling. With large upscaling, it gives a result not significantly better than inexpensive replicative sampling. Consequently, it should only be used when downsampling. (Our rule of thumb is to use area mapping for downsampling with a scale factor less than 0.7). With very large downscaling (e.g., a factor less than 0.2), scaling by area mapping (pixScaleAreaMap()) does not give results much better than low-pass filtering followed by subsampling for antialiasing (pixScaleSmooth()). However, for scale factors between about 0.2 and 0.7, it gives significantly better visual results than pixScaleSmooth().

Let's consider scaling in more detail. Suppose you are enlarging the image, where the scaling factor is much larger than 1. Then most *dest* pixels cover just a small amount of a single *source* pixel. Relatively few dest pixels will overlap source pixel boundaries.

In this case, the result of area mapping will be nearly identical to that of replicative sampling. Most dest pixels will be assigned the value of a single source pixel, so in this limit, area mapping reduces to subsampling except for the small number of dest pixels that lie across source pixel boundaries, where you get a weighted average of the source pixel values. The appearance, like the case for sampling alone, will be blocky, because you will see each source pixel magnified and with only a little fuzziness around the edge.

At the other limit, where you are doing a large reduction, each dest pixel covers many source pixels. If you take an average of the source pixels, including only the area of each source pixel that is actually covered by the dest pixel, you will get an unaliased and very good estimate of the value each subsampled pixel should have. However, doing this sum is computationally expensive, and the result is not appreciably better than simply doing an average (using an appropriate a lowpass filter) followed by subsampling.

As a good rule of thumb, for downscaling with a scale factor less than 0.7, you should use a lowpass filter and then subsample. To prevent aliasing, the size of the lowpass filter kernel on the source image should be approximately the area of the source that corresponds to a single pixel of the dest. The implementation of this approximate filter is straightforward; see pixScaleSmooth().

Better antialiased results are obtained by using strict area mapping, with a filter that is of constant (normalized) height over a rectangular region of the source image that corresponds closely to each dest pixel; see pixScaleAreaMap().

Mip-mapping.

Mip-mapping is the name given to the texture mapping method used in graphics (in particular, in games), where a multiresolution pyramid is constructed at power-of-2 scales, and each pixel at an intermediate scale is evaluated by interpolating between the corresponding pixels on the pyramid at resolutions above and below. It is widely used to display textures in graphics for two reasons: (1) the appearance of a texture (a set of neighboring pixels) varies smoothly as the resolution is changed, and (2) hardware to support the pyramids and perform the interpolations in real time is available in high-end graphics cards.

Mipmap scaling can be performed efficiently in the CPU, and we provide an implementation that is used in conjuction with scale-to-gray. For document images with text and sharp edges, mipmap scaling suffers from severe aliasing and should not be used if the quality of the images is important.

Min-max

Morphological transforms are the most important class of non-linear image transforms. Grayscale morphology, even using the van Herk/Gil-Werman method, is relatively expensive if you just want the min or max for each tile in a subsampled image. Rather than performing grayscale morphology on the full resolution image and throwing most of the results away with subsampling, for integer reduction it makes more sense to identify the tiles in the src image that correspond to each dest pixel and compute the min or max (for erosion or dilation) directly.

The function pixScaleGrayMinMax() performs this function for arbitrary (isotropic) downscaling. The speed is about 70 MPix/sec/GHz of source data. For the special case of downscaling by 2x, the function pixScaleGrayMinMax2() is about twice as fast again.

Min-max is a special case of the general rank order downscaler. The problem with the general rank order downscalers is that, except for the min and max values, they require sorting of pixel values. This is relatively expensive: sorting n elements is computationally of order n*log(n). For a 2x2 kernel (which we use here for filtering before a 2x reduction), this isn't too expensive.

A general 2x rank order function pixScaleGrayRank2() allows a choice of the two intermediate rank values in addition to the min and max, but at about 3x the computational cost. For grayscale rank reductions greater than 2x, a sequence of 2x reductions with different ranks can be used to approximate an intermediate rank over the entire (power-of-2) reduction.

Linear interpolation

We describe the implementation(s) of *linear interpolation* in some detail because it is useful and conceptually simple, and because there exist very efficient implementations for the special cases of 2x and 4x upscaling. The general scaling implementations for linear interpolation of grayscale and RGB images are in the functions pixScaleGrayLI() and pixScaleColorLI(), respectively. They are appropriate for upscaling with a scale factor larger than 1, or for downscaling with a scale factor larger than about 0.7. For scale factors between 0.7 and 2.0, better results are achieved, particularly on orthographically generated images (as opposed to natural scenes) by a small amount of sharpening after scaling.

Source: http://www.leptonica.com/scaling.html