BINARY IMAGE SCALING

The high-level call pixScale() dispatches general scaling to binary, grayscale or color procedures, depending on the depth of the PIX. All these general scaling functions have two scaling parameters, one for horizontal and one for vertical. Scaling is *isotropic* when these two factors are equal and *anisotropic* otherwise.

Consider now binary scaling, and imagine that the scaling is isotropic. There are two cases: the scaling factor(s) can be either greater than 1 (*upscaling*) or less than 1 (*downscaling*). As mentioned above, with *grayscale* images one should use linear interpolation with upscaling, and for downscaling it's best to use a lowpass filter before subsampling. For *binary* images, we simply replicate pixels when upscaling. When downscaling binary images, there are three choices:

1. You can downscale by simple subsampling, and take your chances with aliasing.
2. You can apply a lowpass filter before subsampling, which for binary images could be a thresholding function implementing a rank order filter.
3. You can save some of the high resolution information by using a lowpass filter followed by subsampling to convert to a grayscale image at low resolution.

Arbitrary binary rescaling

First, we describe the process for upscaling by replication or downscaling by simple subsampling, in the case of arbitrary scale factors. For further details, consult the source code in scaleBinaryLow() in *scalelow.c*. We iterate through the destination. Two one-dimensional arrays are set up so that we can quickly find the horizontal and vertical indices of the source pixel corresponding to each pixel in the destination. For downscaling, this will be a different source pixel for each dest pixel, and we set the dest pixel if the source pixel is set. For upscaling, we only check each source pixel once, and if set, we set all the corresponding dest pixels in the row. Once a dest row is complete, we replicate it as many times as are required by the vertical index array.

The situation where the scale factor is arbitrary is relatively inefficient, because we have to read and write individual binary pixels in the source and destination images, respectively. The efficiency can be greatly improved for the special case where the scale factor is a power of 2.

Special case: power-of-2 binary expansion

Consider the situation with a power-of-2 binary replicative *expansion*. For example, with 2x expansion, we take a byte corresponding to 8 pixels, use this as an index in a 256-entry LUT of 16-bit words, each of which has each bit in the input byte repeated. After the output raster line that has a 2x horizontal expansion has been computed, it is copied again to the destination to give the 2x vertical expansion. Other power-of-2 factors (4, 8, etc.) are computed in a similar way. We use LUTs that have no more than 256 entries throughout, because they are extremely fast. If you want to guild the lily, you will find that 16-bit tables are even faster with most caches. The high-level functions are in *binexpand.c* and the low-level work on the pixels is all done in*binexpandlow.c*.

Special case: 2x binary reduction

The situation with power-of-2 binary rank-order filtered reduction is a little more complicated. Remember, we can do this for arbitrary rank order rectangular filters using the pixBlockrank() function, although, because of its generality, it is considerably slower than the special functions we describe here. The high-level functions are in *binreduce.c* and the pixel pushing is all done in *binreducelow.c*. See the source code for details and clarification of the description here.

Consider first 2x subsampling without filtering. If you want to use a LUT, there are two ways to do it. With an 8-bit LUT, you can store in the table the 4 bits (say, bits 0, 2, 4, and 6) that you are taking for each 8 bits (0 - 7) in. This is very fast, but you can get some improvement by masking out all the odd bits in a 32-bit word, and then ORing this masked word with itself, left-shifted by 7 bits. That puts the first 8 even bits (0, 2, ... 14), which will be bits (0, 1, ... 7) in the reduced image, into the first byte. There, they are in bit order (0, 4, 1, 5, 2, 6, 3, 7). An 8-bit LUT is then used to permute them to the normal order (0, 1, ... 7) in the output byte. Likewise, the second set of 8 even bits land in the third byte, and must be permuted in the same way. This is performed on even raster lines; odd raster lines are ignored in the 2x reduction.

Either of these subsampling methods can be pre-filtered. But *it is only necessary to get the filtered results at the pixels that will be subsampled*, because the other pixels are discarded at subsampling. The main reason these operations are so fast is that it is not necessary to generate a full filtered image before subsampling. Now, these are binary images, so we must use a convolution filter and apply a threshold on the result. For a 2x reduction, it makes sense only to use the 4 pixels in each 2x2 square, from which a single pixel will be chosen. The filter weights on these 4 pixels should be equal, so we simply count the ON pixels in each 2x2 square and apply a threshold on the count.

How do we efficiently get the result of this thresholded count into a specified pixel in each 2x2 square? The solution, which was found in 1991, is very simple and can be found in the appendix of the paper, Image Analysis using Threshold Reduction.

Two examples will give you the general idea. Suppose we want a threshold of 1 (rank value of 1/4), which means that if any pixels in a 2x2 square are ON, we get an ON pixel in the upper-left corner of this 2x2 square. If we do a morphological dilation using a 2x2 structuring element with center in the lower-right corner, we will get the correct pixel in the upper-left corner, and we will also get the shift-invariant filtered results in the other 3 pixels in each 2x2 square. We could then subsample any of these pixels, because they all have been filtered. Obviously, we have done too much work! Because we are only sampling the upper-left pixels in each 2x2 square, to get the information about the other 3 pixels there, OR each word with itself left-shifted by one pixel, and follow this by a vertical OR of each word on odd raster lines with the word above it, storing the result in the words on even raster lines. With three 32-bit ORs, we thus generate 16 filtered pixels for subsampling. The filtering turns out to be much faster than the subsampling! The second example implements a threshold of 4 (rank value of 4/4), which means that all pixels in a 2x2 square must be ON for the upper-left pixel to be turned on. Again this can be implemented morphologically with an erosion, this time using a 2x2 structuring element with the center in the upper-left corner.

And again, more efficiently, this can be implemented by logically performing three ANDs, in the same way that we used three ORs for threshold 1.

What about threshold levels 2 and 3? These are only slightly more complicated to implement with bit-logical operations. We need to perform both a horizontal AND and vertical OR, and a horizontal OR and a vertical AND. Then, taking the OR of these two results gives a threshold of 2, whereas taking the AND gives a threshold of 3. You can demonstrate that this works by setting up a logic table consisting of all 16 possible 2x2 binary pixels, and applying these operations. Consult the paper for a proof.

As would be expected because of the dilation, reduction using a threshold of 1 makes the image darker. Reduction using a threshold of 4 makes it much lighter. Reduction with threshold 2 appears to preserve the apparent darkness of the original image, as might be expected from a median filter (rank order of 2/4 = 0.5).

A cascade of successive 2x threshold reductions is very useful for document image analysis, because it combines morphological filtering with reduction in such a way as to select textural qualities at different scales. Texture in a binary image can be thought of as the statistical distribution of neighbor pixels of the opposite color. Although morphological operations such as the opening or hit-miss are typically considered to be shape-filtering, they can also be used to filter texture.

This is an interesting story, and you are referred to the 1991 papers Image Analysis using Threshold Reduction and Multiresolution morphological approach to document image analysis for simple examples with document images.

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