THE RASTEROP IMPLEMENTATION OF BINARY MORPHOLOGY

A general rasterop is an image operation that combines a rectangle of pixels of the source image with a rectangle of pixels in the destination image, where the combination operation can be any of 12 binary logical pixel-wise operations. There are also special case unary rasterops that only affect a rectangle within the destination image.

As described in the previous section, a morphological operation can be implemented by performing a sequence of logical ORs (for the dilation) or logical ANDs (for the erosion) of the source image, translated each time according to one of the hits in the *Sel*. The rasterop can be used, where the rectangle chosen is the entire source image. For all rasterops, the source image is clipped so that it does not try to write beyond any edge of the destination. There are other details, of course, such as the initial conditions on the destination image and the handling of the boundary conditions.

We have already introduced the issue of different conventions for choosing boundary conditions for erosion. We can imagine implementing *erosion* by rasterops in two ways:

1. Start with a dest image that is cleared (all pixels have value 0). Copy the source image to it with a displacement given by (an arbitrary) one of the hits in the *Sel*. For simplicity, imagine that this copy is made without any shift because the first hit in the *Sel* is at the *Sel* origin. The rest of the hits each result in an AND of the shifted and clipped source with the destination.

2. Start with a dest image where all pixels are set to a value 1. For each hit in the *Sel*, AND the shifted source with the dest. Because we initialize to 1, you can use either a copy operation or and AND for the first hit. This is dual to the implementation of the dilation, where we initialize all pixels to 0 and do a sequence of ORs for each hit in the *Sel*.

We use the second method because of its simplicity, and because it can be used directly to implement erosion with both SBC and ABC.

Now, when an image is shifted to the right by 2 pixels, the leftmost 2 columns of pixels in the destination are unaffected by the AND. What should happen to these 2 columns? It depends on whether you are using SBC, which logically brings in ON pixels from the frame, or ABC, which logically brings in OFF pixels from the frame. (But if we do full image rasterops without a frame, nothing is actually brought in -- the 2 columns of pixels are simply not altered.) If you are using SBC and initialize the image pixels to ON, these 2

columns remain ON, just as if you really pulled ON pixels in from the frame. If you are using ABC, regardless of image content you must clear these columns out later.

Suppose we are using asymmetric boundary conditions, where we require that the result of erosion using rasterops is equivalent to a situation where the source is surrounded by enough OFF pixels to cover the dest for every translation specified by the *Sel*. Details on this choice of boundary condition convention can be found in a recent paper on the *Implementation Efficiency of Binary Morphology* . We have two methods for insuring that the pixels near the boundary are properly cleared. We can *extend* the source image with OFF pixels all around, and clip each rasterop to the destination image. Then those columns of OFF pixels would automatically clear the destination because of the AND operation. Or equivalently, without embedding the source image in a larger image of OFF pixels, we can perform all the rasterops in the erosion with usual clipping to the translated source image and, as a final step, clear the (up to) four rectangles of the appropriate boundary pixels in the destination. For each such rectangle, the size in a direction normal to the boundary is equal to the maximum shift (relative to the *Sel* origin) in the corresponding one of the four directions. For example, if the maximum shift to the right in the *Sel* is 4 pixels, the rectangle consisting of the 4 left-most columns of the destination must be cleared. (Remember that for erosion, the vector giving the image shift for each hit in the *Sel* is *from* the hit location *to* the*Sel* origin.)

We must also clear the edge pixels for the hit-miss transform, because it is a near relative of the erosion. The size of the rectangles to be cleared is determined from the hits in the *Sel*, exactly as in the erosion. The location of the misses in the *Sel* are not important for the clearing operation because the pixels that would be brought in from beyond the image boundary (if it were extended) would be ON. This is because for each miss, a translated version of the *negative* image is ANDed.

Suppose on the contrary that you wish to adopt the symmetric boundary conditions; namely, that the image is surrounded with ON pixels for an erosion, as discussed above. Then you simply use the second method for the erosion given above: *set the dest image pixels (to 1); then AND each raster operation, properly shifted and clipped. This operation is dual to the dilation, and you stop there -- you don't clear the edge pixels.* For closing, with the example given in the previous section, we do not need to add any surrounding frame pixels for the rasterop implementation. After the dilation, we set all destination pixels ON for the erosion and do the set of ANDs from the source (which is the destination of the previous dilation). Because of rasterop clipping, it's easy to see that the line of pixels from the original to the left boundary will be preserved by the erosion.

As mentioned in the previous section, you can determine the boundary conditions using the function `resetMorphBoundaryCondition()`. The effects of the boundary are most striking with the closing operation. Closing is formally extensive, but with the ABC

convention without added frame pixels, ON pixels near the image boundary can be removed because the dilation does not extend past the boundary. If you are using the ABC convention, the function `pixCloseSafe()` will remove this anomaly by adding a frame of minimal required size to every image before closing, and removing the frame pixels at the end. The routines for adding and removing a such a frame are in *pix2.c*. (In the libarary, we refer to this as adding or removing a *border*). Note again that if you are using SBC, closing will be extensive without the addition of actual pixels around the boundary, and the anomaly will be avoided automatically.

The source code for the rasterop implementation of binary morphology is in *morph.c*. A few things to note are:

- **Call arguments**. The dilation is typical:

  ```
  PIX * pixDilate(PIX *pixd, PIX *pixs, SEL *se);
  ```

  The first argument is the dest; if it is NULL, a new dest pix is allocated. The function returns the dest in all circumstances. If pixs and pixd are pointers to the same *Pix*, the operation is in-place, and pixs will be changed; otherwise, pixs is unaltered.

- **Data structure definitions**. The *Sel* is defined in *morph.h* and the *Pix* is defined in *pix.h*. All *Pix* fields should only be changed using the "get" and "set" accessors provided in *pix1.c*.

- **Construction of *Sel*s**. There are many functions for creating a *Sel*.
  1. `selCreate()` makes a *Sel* initialized to don't-care entries.
  2. `selCreateBrick()` makes a *Sel* initialized to either don't-care, hit or miss entries.
  3. The created *Sel*s can be packaged up into an array (*Sela*) using `selaAddSel()` repeatedly.
  4. *Sel*s can be extracted from *Sela* (arrays) by name using `selaFindSelByName()` or by index using `selaGetSel()`.
  5. *Sel*s can be created from either *Pix* or an array of pts (a *Pta*).
  6. *Sel*s can be created from a compiled string that, when displayed, shows the hits, misses and origin in correct geometrical arrangement.
  7. *Sel*s can be created from a simple file format that uses the same format as for the compiled string.
  8. *Sel*s can be created from a colored pix generated with a pixmap editor.

- **Visualization and archiving of *Sel*s**. There are several ways to save or display a *Sel* or a *Sela* array.
    1. Both *Sel*s and *Sela* can be written in ascii, either to a stream or to a file (*sel1.c*).
    2. *Sel*s can be written to a string that, when displayed, shows the hits, misses and origin in correct geometrical arrangement (*sel1.c*).
    3. *Sel*s can be displayed in a binary pix in a form suitable for publishing (`selDisplayInPix()`).
    4. A color image of a *Sel* (in general, a hit-miss *Sel*) can be generated using `pixDisplayHitMissSel()` in *selgen.c*.

Source : http://www.leptonica.com/binary-morphology.html