IMPLEMENTATION OF AFFINE TRANSFORMATIONS

There are two very different approaches to implement an affine transformation on an image:

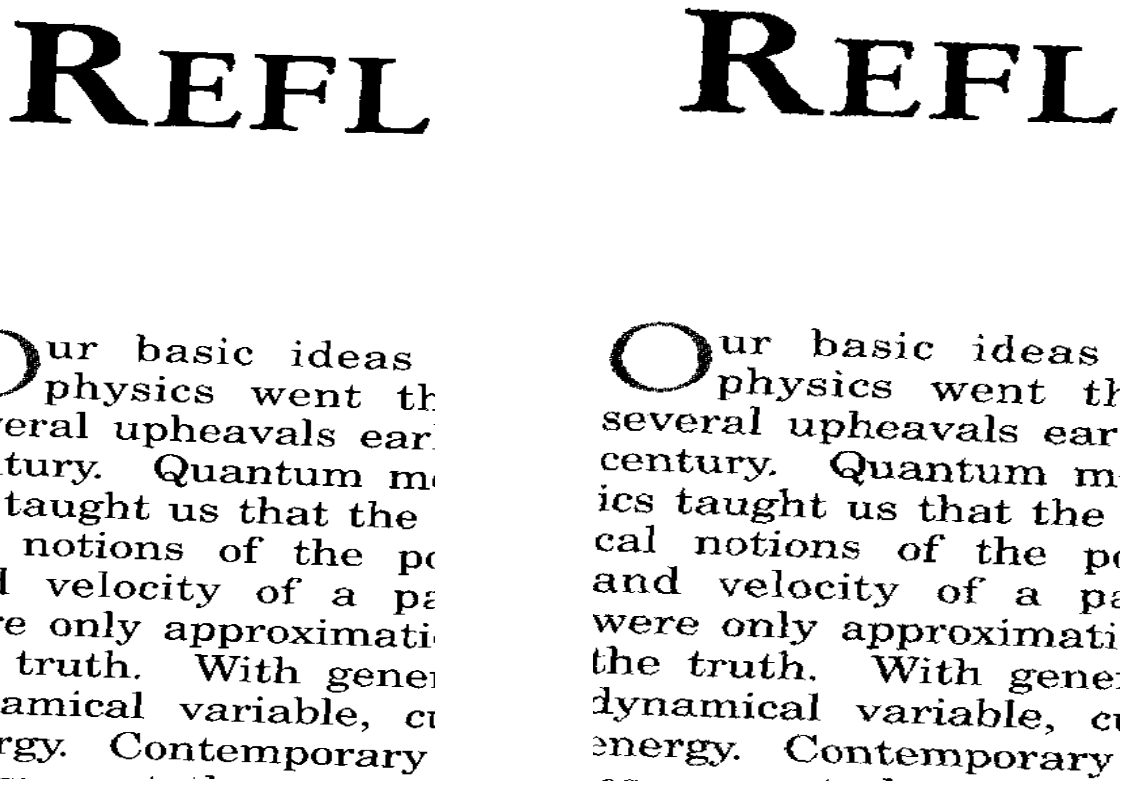
1. **Pointwise.** Each point in the dest is determined from the corresponding point in the src. We start with 3 points specifying the initial coordinate space and the 3 corresponding points that specify the transformed coordinate space, and transform an entire image pointwise by the transformation (1).
2. **Sequential.** The entire image is successively transformed by a sequence of shear, scale and translation operations.

In practice, the pointwise approach is the most useful. For each dest pixel, the corresponding src pixel (or pixels) are located, and the dest pixel value is computed.

For binary images, the *pointwise* transform is about 3 times slower than the *sequential* transform, whereas for grayscale images the pointwise transform is an order of magnitude faster! However, of more importance, the results, particularly on text, are far better with the pointwise transform.

The sequential transform involves a number of shears, which cause visible dislocation lines through characters. After such a set of sequential transforms, the edges, that were initially smooth, become irregular, significantly degrading the visual quality of the text. The pointwise transform has a minimal number of such artifacts, and any shear lines that are developed tend to have shorter spatial correlation. It is thus strongly urged that *in any situation where the quality of the resulting text image is important, the pointwise transformation should be used.*

The following image fragments show the image quality resulting from the same affine transformation. The first fragment has been done pointwise, whereas the second one was done sequentially.



See the documentation in affine.c, where it is shown that the pointwise transformation should be performed *backwards*, so that for every point in the (primed) dest, you find the corresponding point in the (unprimed) src. For binary images, the implementation is slower in the backwards direction, because you have to find the corresponding point in the src for *every* point in the dest, whereas going in the forward direction, you only write to the dest when the src pixel is ON. However, the forward direction implementation can miss some pixels entirely in the dest. When a solid black region is transformed, this will in general result in a regular pattern of white pixels within it, even when no overall scaling is done.

With the affine transform, as with the projective and bilinear transforms (see below), the pointwise transform can be done for all pixel depths by *sampling*. For 8 bpp and RGB images, better results are obtained, particularly on document images that have sharp edges, using *interpolation*. Interpolation involves subdividing the src pixels into subpixels (we divide each into 16 x 16 subpixels), and using the subpixel location to generate weighting factors for the four neighboring src pixels. Interpolation is about 5x slower than sampling the nearest src pixel. When there is relatively little scale change, interpolation is nearly equivalent to area mapping, as we have shown with rotation.

Sequential affine transform

We now describe the situation where the affine transform is performed as a sequence of translations, scalings, shears and, optionally, rotations. Because a rotation is equivalent to three shears), we need in principle only translations, scaling and shear. We have already demonstrated that there are 6 independent parameters in the affine transformation. This can also be seen from the transformation from one coordinate space to another. There are 2 parameters to align the origins, 2 parameters for the scaling of the two axes, and 2 parameters describing the *change* in angle of each axis. Now, suppose you are given the two coordinate spaces, and want to transform from one to the other. How do we do this with the image operations in ***Leptonica***?

These operations are as follows:

- Arbitrary translation, implemented by *pixRasterop()*. We also have an in-place version *pixRasteropIP()*.
- Anisotropic scaling in x and y, implemented by *pixScale()*.
- Horizontal shear about an arbitrary horizontal line *pixHShear()*, with an in-place version *pixHShearIP()*.
- Vertical shear about an arbitrary vertical line *pixVShear()*, with an in-place version *pixVShearIP()*.

- Rotation about an arbitrary point, implemented by *pixRotateShear()*, with an in-place version *pixRotateShearIP()*.

To make use of these transforms to align our two coordinate spaces, we use the following construction, described also in affine.c. The problem to be solved is to take an image with one coordinate space (unprimed) and transform it by an affine transformation to coincide with another coordinate space (primed). We use only shears parallel to the x and y axes, orthogonal scaling with scaling factors in x and y, and translation.

A typical application is to align the image with a second image in another coordinate space, related by the affine transformation. This invites the following model for making the affine transform. Imagine that the unprimed coordinate space is in our original image (image 1) and the corresponding primed coordinate space is in a second image (image 2). We can transform *both* images so that the coordinate spaces coincide and are aligned with the x and y axes. After both the unprimed and primed coordinate spaces are coincident, we finally shear the unprimed coordinate space to coincide with the original primed space. Thus, all operations really happen on a single image. Specifically, we do the following:

1. *Horizontal shear transform* about point 1 to align point 3 with the y axis.
2. *Vertical shear transform* about point 1 to align point 2 with the x axis.

3. *Compute the horizontal shear angle and the final location of point 3'* required to put point 3' on the y axis.
4. *Compute the vertical shear angle and the final location of point 2'* required to put point 2' on the x axis.
5. *Scale* x and y axes anisotropically so that points 2 and 3, which are on the x and y axes, move out to a distance equal to the distance of the newly sheared points 2' and 3' from their origin 1'. This scaling operation will translate the origin (point 1).
6. *Translate* the origin 1, after it has been moved by scaling, to coincide with the origin 1'.
7. *Vertical shear transform* about the new origin (point 1, which is now aligned with 1'), using the negative of the angle computed in step 3 above.
8. *Horizontal shear transform* about the new origin (point 1), using the negative of the angle computed in step 4 above.

In all this, it is only necessary to keep track of the shear angles and translations of points during the shears. What has been accomplished is a general affine transformation on the image. See affine.c for the specifics of the implementation.

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