# *Geometric Functions* 10

This chapter discusses the XIL library's geometric functions. These functions enable you to perform such operations as:

- Translating an image (moving an image up, down, left, or right)
- Scaling an image (changing an image's width or height)
- Rotating an image
- Warping an image
- Transposing an image (flipping an image across a horizontal or vertical axis, or across a diagonal)

The following sections discuss the XIL functions that perform the geometric functions; because you must pass an interpolation option as an argument on most of them, this chapter begins with a discussion of the available interpolation options.

---

**Note –** The functions discussed in this chapter cannot be performed in place. That is, the source and destination images for a geometric operation must be different images.

---

# 10

# Interpolation Options

Geometrically transforming images centers around the notion of *point sampling*. In point sampling, each pixel in a destination image is located with integer coordinates at a distinct point *D* in the image plane. The geometric transform *T* identifies each destination pixel with a corresponding point *S* in the source image; thus, *S* is the point that *T* maps to *D*. In general, *S* doesn't correspond to a single source pixel; that is, it doesn't have integer coordinates. Therefore, the value assigned to the pixel *D* must be computed as an interpolated combination of the pixel values closest to *S* in the source image.

When performing most geometric transformations, you must specify the interpolation method to be used in calculating the destination pixel values. Table 10-1 shows the interpolation options the XIL library provides, and the strings you use to request them.

*Table 10-1* Types of Interpolation

| Interpolation Type | String | What it Assigns to *D* |
|---|---|---|
| Nearest neighbor | `nearest` | The value of the pixel nearest *S* |
| Bilinear | `bilinear` | A value that's a bilinear function of the four pixels nearest *S* |
| Bicubic | `bicubic` | A value that's a bicubic function of the sixteen pixels nearest *S* |
| General | `general` | A value that's a separable user-defined function of the pixels in a rectangular region surrounding *S* |

To see how the interpolation type can affect the result of a geometric operation, see Color Plate 1.

## Nearest Neighbor Interpolation

Nearest-neighbor interpolation, sometimes called zero-order interpolation, is the fastest interpolation method because it simply assigns to point *D* in the destination image the value of the pixel nearest S in the source image (see "Interpolation Options" on page 150"). Though it's a good choice when speed is important, nearest-neighbor interpolation can produce undesirable artifacts

in the destination image, especially near edges where there may be a big change in color or gray level between two adjacent pixels. For example, smooth lines in the source image may show up as jagged lines in the destination.

## *Bilinear Interpolation*

A routine that performs bilinear, or first-order, interpolation assigns to point *D* in the destination image a value that's a bilinear function of the four pixels nearest *S* in the source image (see "Interpolation Options" on page 150). This interpolation type yields better results than nearest-neighbor interpolation, but can itself have an undesirable smoothing effect on an image. To alleviate this problem, you can use bicubic interpolation.

## *Bicubic Interpolation*

A routine that performs bicubic interpolation assigns to point *D* in the destination image a value that's a bicubic function of the 16 pixels nearest S in the source image (see "Interpolation Options" on page 150). Using bicubic interpolation preserves fine detail present in a source image, but it takes more time than the nearest-neighbor or bilinear interpolation methods.

## *General Interpolation*

The interpolation options discussed so far base the interpolated values on relatively few pixels: nearest neighbor uses 1 pixel, bilinear uses 4 pixels, and bicubic uses 16. If these options don't provide the quality you need, you can use the general interpolation option. For example, if you're subsampling by a factor of four, bicubic interpolation would result in aliasing artifacts that can be improved by using more source pixels in the interpolation. General interpolation lets you:

- Determine how many pixels nearest point *S* in the source image are used to calculate the interpolated pixel value of point D in the destination image (see "Interpolation Options" on page 150). If desired, you could use every pixel in the source image.

- Weight the pixels used in the calculation. By ensuring that the pixels closest to point *S* in the source image have more influence on the value assigned to *D* than pixels that are further from *S*, you can reduce the contrast between adjacent pixels in the destination image, thus providing smoother line and color transitions.
- Designate the number of pixel subsamples to use for interpolating pixel values in the destination. This effectively lets you divide into fractional locations the space between adjacent source-image pixels so you can interpolate a destination pixel's value differently, depending on which location point *S* falls in.

To use general interpolation, you must:

1. Create horizontal and vertical interpolation tables; these form the filters or *kernels* for the interpolation. The tables determine the kernel sizes, weighted pixel values, and number of subsamples used for a general interpolation.
2. Set the interpolation tables on the system-state object. The tables affect all general interpolation operations using images created from this system-state object.
3. Pass `general` as the interpolation string on any XIL function that requires an interpolation argument. This causes the interpolation tables to be used as the interpolation method.
4. Destroy the interpolation tables when they are no longer needed. This releases the memory that was allocated for them.

The sections that follow discuss these steps in more detail. Before proceeding to them, however, it's useful to consider a conceptual model of a general interpolation.

Figure 10-1 represents an interpolation kernel as a 5-by-5 matrix of weighted values; for simplicity, the values are labelled `a` through `y`.

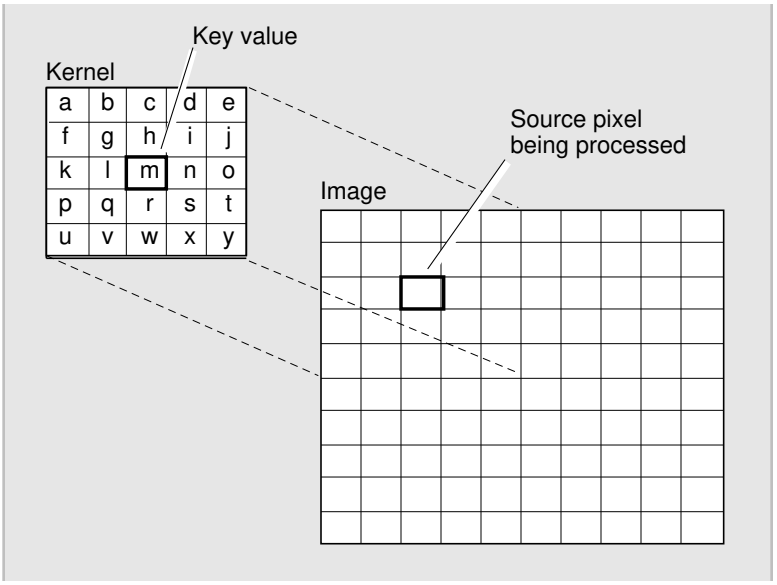


*Figure 10-1* Conceptual Model of a General Interpolation

During a general interpolation, the kernel's key value—generally its center value—is laid over the source-image pixel to be processed; this means the other kernel values lie over neighboring pixels. Each source-image pixel that is covered by the kernel is then multiplied by the kernel value that lies over it. In Figure 10-1, the source-image pixel 2,2 is multiplied by kernel value `m`; pixel 0,0 is multiplied by kernel value `a`; pixel 1,0 is multiplied by `b`; and so on. The multiplication products are then summed together, and this sum becomes the pixel value in the destination.

Though Figure 10-1 provides a useful conceptual model, the general interpolation method doesn't actually construct the two-dimensional kernel shown in the figure because it's computationally expensive. To improve efficiency, general interpolation uses separate horizontal and vertical vector arrays—the kernels in the interpolation tables you create—to calculate the same values a two-dimensional kernel would calculate. The vector arrays require you to provide fewer data elements for the kernel values; this reduction is particularly significant for large tables with many subsamples. Nonetheless, for a horizontal table with M elements and a vertical table with N elements, the number of pixels that contribute to the interpolated value is still given by M * N.

**Note –** To translate the horizontal and vertical kernels into a two-dimensional matrix as shown in Figure 10-1, the 1xn kernel values would have to be matrix-multiplied to obtain the corresponding two-dimensional kernel values.

When interpolating pixels that are at, or near, the source image edges, the general interpolation method temporarily reduces the kernel height and width so the edges fit within the defined kernel size. This strategy lets the method interpolate pixels that originally formed the image edges. Pixels outside of the image don't contribute to any interpolation values.

## *Creating Vertical and Horizontal Interpolation Tables*

To support general interpolation, XIL has an `XilInterpolationTable` object, which is an array of 1xn kernels representing either a horizontal or vertical interpolation filter. General interpolation can be performed using two interpolation tables, one to represent the horizontal filter and one to represent the vertical filter. Or it can be performed using a single interpolation table that represents either the horizontal or the vertical filter, in which case the missing matrix dimension is 1. For example, if the horizontal table defines 1-by-7 kernels and the vertical table defines 1-by-5 kernels, the effective matrix is 7-by-5; if only the 1-by-7 horizontal table is defined, the effective matrix is 7-by-1. If both the horizontal and vertical interpolation tables are `NULL`, nearest-neighbor interpolation is performed.

To create an `XilInterpolationTable` object, you call the `xil_interpolation_table_create()` function, whose prototype is shown below.

```
XilInterpolationTable xil_interpolation_table_create(
     XilSystemState state, unsigned int kernel_size,
     unsigned int subsamples, float* data);
```

The table's data type is `XIL_FLOAT`. The parameter `state` is the handle to the system-state structure created when you initialized the XIL library. The parameter `kernel_size` specifies the number of elements in each kernel. The parameter `subsamples` indicates the number of subsamples or fractional locations between source-image pixels; each subsample requires its own kernel data. The parameter `data` specifies the data values for each kernel. There is no limit or restriction on the `kernel_size` or number of `subsamples`.

The following code fragment creates a horizontal and a vertical interpolation table. The horizontal table has seven kernel elements, and the vertical table has three; each table has only one subsample.

```
XilSystemState state;
XilInterpolationTable horizTable;
XilInterpolationTable vertTable;

int subsamples = 1,
    horizSize  = 7,
    vertSize   = 3;

float horizData[] = {.005, .005, .04, .9, .04, .005, .005};
float  vertData[] = {.12, .8, .08};

horizTable = xil_interpolation_table_create(state,
                horizSize, subsamples, horizData);

vertTable = xil_interpolation_table_create(state,
                vertSize, subsamples, vertData);
```

### *Kernel Size*

The `xil_interpolation_table_create()` function's `kernel_size` parameter determines the number of data elements in each subsample of the kernel defined by the interpolation table. There is no limit on the kernel size. The horizontal and vertical tables can have different kernel sizes, as well as different subsampling values.

The key element in a kernel is its center element; for an even-numbered kernel size, the key element is offset to the next-lowest index element. Thus, the key element for a 5-element table is the center element, which has array index 2; the key element for an 8-element table is the fourth element, which has index 3. The key element's array index can be computed as an integer calculation:

```
int array_index = (kernel_size - 1) / 2
```

### *Subsamples*

The `xil_interpolation_table_create()` function's `subsamples` parameter determines the number of subsamples used for a general interpolation. Increasing the number of subsamples lets you specify different data values to be used for the interpolation kernel, depending on where point

*S* in the source image falls between pixels. There is still only one point sample taken to determine each destination pixel value, but the data used to interpolate the value depends on which subsample's kernel is used, and that, in turn, depends on the exact location of point *S*.

There is no limit on the number of subsamples you can use. For each subsample, you must define separate kernel data. Thus, for *n* subsamples, an interpolation table must have *n * kernel_size* data elements. For example, if you create a 7-element horizontal interpolation table with 4 subsamples, you must define 28 data elements. The first 7 elements define the first subsample's kernel, the second 7 elements define the second subsample's kernel, and so on.

Figure 10-2 shows how the interpolation tables are used to determine which kernel applies to a particular subsample location. In the figure, the subsampling is 4 in both the horizontal and vertical directions.

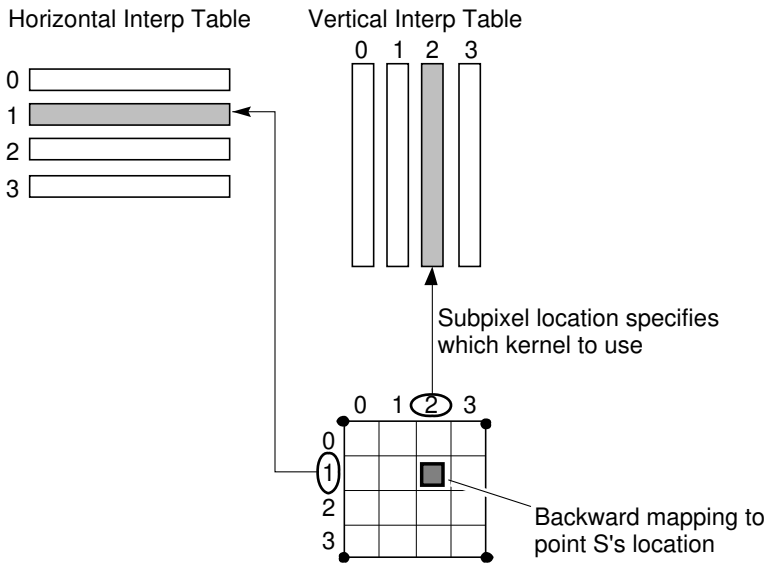


*Figure 10-2* Determining the Kernel to Use for a General Interpolation

Typically, the kernel values for each subsample are weighted according to the subsample location's proximity to the pixels used in the calculation: the closer a pixel is to the subsample location, the more weight it carries in the kernel.

### *Kernel Data*

The kernel data for each table is an array of floating point numbers. As mentioned in "Subsamples" on page 155, for *n* subsamples, there must be *n* * *kernel_size* data elements. Thus, for a 3-element kernel size with 2 subsamples, you must define an array of 6 floating point numbers; the first 3 numbers form the first subsample's kernel, and the second 3 numbers form the second subsample's kernel.

**Caution –** Providing too few kernel values to complete an interpolation table results in bad values.

To preserve the source image's intensity in the destination image, the sum of the data values in each interpolation kernel should equal one. Kernel values whose sum is greater than one tend to increase the destination image's intensity, and those whose sum is less than one tend to diminish the intensity. The results of a given kernel depend on the image it is used on.

To sharpen an image, the kernel values should be weighted heavily toward the center, with approximately 90% of the kernel value being concentrated there. To blur an image, the kernel values can be weighted away from the center.

## *Setting the Interpolation Tables on the System-State Object*

After creating the vertical and horizontal tables, you must set them on the system-state object by calling the `xil_set_interpolation_tables()` function, whose prototype is shown below.

```
void xil_state_set_interpolation_tables(
     XilSystemState state,
     XilInterpolationTable horiz_kernel,
     XilInterpolationTable vertical_kernel);
```

The parameter `state` is a system-state data structure; the parameters `horiz_kernel` and `vertical_kernel` are the horizontal and vertical interpolation tables you created for general interpolation (see "Creating Vertical and Horizontal Interpolation Tables" on page 154).

The following code fragment creates horizontal and vertical interpolation tables, sets them on the system-state object, and uses them to perform a general interpolation for translating an image (see "Translating Images" on page 160).

```
XilSystemState state;
XilInterpolationTable horizTable;
XilInterpolationTable vertTable;
XilImage src, dst;

int subsamples = 8,   /* subsample and kernel sizes */
    horizSize  = 2,
    vertSize   = 2;

float kernelData[] = {1.0,  0.0,
                      .875, .125,    /* 7/8,  1/8 */
                      .75,  .25,     /* 6/8,  2/8 */
                      .625, .375,    /* 5/8,  3/8 */
                      .5,   .5,      /* 4/8,  4/8 */
                      .375, .625,    /* 3/8,  5/8 */
                      .25,  .75,     /* 2/8,  6/8 */
                      .125, .875 };  /* 1/8,  7/8 */

horizTable = xil_interpolation_table_create(state,
                 horizSize, subsamples, kernelData);

vertTable = xil_interpolation_table_create(state,
                 vertSize, subsamples, kernelData);

xil_state_set_interpolation_tables(
    state, horizTable, vertTable);

xil_translate(src, dst, "general", 50.0, 50.0);
```

In this example, the general interpolation effectively performs a bilinear interpolation with 8 subsamples. The kernel values indicate how much influence the source-image pixels have on the destination value; the kernel value 1 indicates that a source pixel completely determines the value for the destination pixel, and 0 indicates a source pixel has no influence on the destination value. If *S* is a source-image point that maps to a given destination pixel *D* (*S* and *D* are described in "Interpolation Options" on page 150), the

closer *S* is to a source pixel used to interpolate *D*'s value, the more influence that source pixel has on the computation. Conversely, the further *S* is from the source pixel, the less influence the source pixel has on *D*'s computed value.

### *Additional Kernel-Related Functions*

The following XIL functions are available for getting an interpolation table's kernel and subsample sizes, and for getting the kernel data values.

```
unsigned int xil_interpolation_table_get_kernel_size(
     XilInterpolationTable table);
```

```
unsigned int xil_interpolation_table_get_subsamples(
     XilInterpolationTable table);
```

```
float* xil_interpolation_table_get_data(
     XilInterpolationTable table);
```

To retrieve an interpolation table, call the `xil_state_get_interpolation_tables()` function, whose prototype is shown below.

```
void xil_state_get_interpolation_tables(
    XilSystemState state,
    XilInterpolationTable* horiz_kernel,
    XilInterpolationTable* vertical_kernel);
```

### *Destroying an Interpolation Table*

After performing a general interpolation, you should destroy the vertical and horizontal interpolation tables if you aren't going to use them again. Destroying the tables releases the memory that was allocated to them.

To destroy an interpolation table, call the `xil_interpolation_table_destroy()` function, whose prototype is shown below.

```
void xil_interpolation_table_destroy(
     XilInterpolationTable table);
```

The only parameter to this function is a handle to the kernel you want to destroy.

## Translating Images

Translating an image means moving it up, down, left, or right. The XIL function you use to translate an image is called `xil_translate()`. The prototype for this function is shown below.

```
void xil_translate(XilImage src, XilImage dst,
    char *interpolation, float xoffset, float yoffset);
```

The parameters `src` and `dst` are handles to the source and destination images. The parameter `interpolation` is a string that specifies an interpolation type. For information on this parameter, see "Interpolation Options" on page 150. The parameters `xoffset` and `yoffset` are floating-point numbers that represent the number of pixels the image should be moved horizontally and vertically. If `xoffset` is positive, the image is moved to the right, and if it is negative, the image is moved to the left. If `yoffset` is positive, the image is moved down, and if it is negative, the image is moved up.

If you have already read the section "Origin" on page 46, which discusses image origins, you may think that translating an image is the equivalent of setting the destination image's origin and then copying the source image to the destination. For the most part, this is true. That is, the two code fragments shown below have the same effect.

```
XilImage src, dst;

xil_set_origin(dst, 50.0, 50.0);
xil_copy(src, dst);
```

```
XilImage src, dst;

xil_translate(src, dst, "nearest", 50.0, 50.0);
```

Both fragments move the source image 50 pixels to the right and 50 pixels down.

However, what if the 50.0's in the fragments above are changed to 49.5's? In the first case, the coordinates that make up the destination image's origin (49.5, 49.5) will be rounded to integers (50, 50) before the copy occurs, so the result will not change. In the second case, though, `xil_translate()` actually moves the image 49.5 pixels down and 49.5 pixels to the right so that pixels in the destination image map to noninteger coordinates in the source image. Since there is no longer a one-to-one correspondence between the pixels in the source and destination images, the values of the pixels in the destination image must be estimated or *interpolated*. The different types of interpolation the XIL library provides are discussed in "Interpolation Options" on page 150.

## *Scaling and Subsampling Images*

The XIL library includes three functions that enable you to change the width and height of an image:

- `xil_scale()`
- `xil_subsample_adaptive()`
- `xil_subsample_binary_to_gray()`

The first function, `xil_scale()`, is the most general of the routines: it increases or decreases either the width or height of an image. The second, `xil_subsample_adaptive()`, is useful only for reducing an image's size; it is not useful for enlarging an image. It is available for reducing images because it often produces better results than using `xil_scale()` to reduce images. The last function, `xil_subsample_binary_to_gray()` is a special function used only for reducing the size of `XIL_BIT` images.

# 10

## xil_scale()

The XIL library's general scaling routine is `xil_scale()`, whose function prototype is shown below.

```
void xil_scale(XilImage src, XilImage dst, char *interpolation,
    float xscale, float yscale);
```

The parameters `src` and `dst` are handles to the image to be scaled and the destination image. The parameter `interpolation` is a string that specifies the type of interpolation to be used for the operation: nearest neighbor, bilinear, bicubic, or general. (For information about these interpolation methods and how to request one of them, see "Interpolation Options" on page 150.) The parameters `xscale` and `yscale` are the horizontal and vertical scale factors. If `xscale` is greater than 1, the width of the source image will be increased, and if it is between 0 and 1 (exclusive), the width will decrease. Similarly, if `yscale` is greater than 1, the height of the source image will be increased, and if it is between 0 and 1 (exclusive), the height will decrease.

One thing to bear in mind when scaling images is the way image origins can affect the scaling operation. (For information about image origins, see "Origin" on page 46.) For instance, assume the source and destination images are the same size and that you want to scale the source image by a factor of 2 both horizontally and vertically. Figure 10-3 shows what will happen if, when you scale the image, the origins of both the source and destination images are set to 0.0, 0.0 (the default).

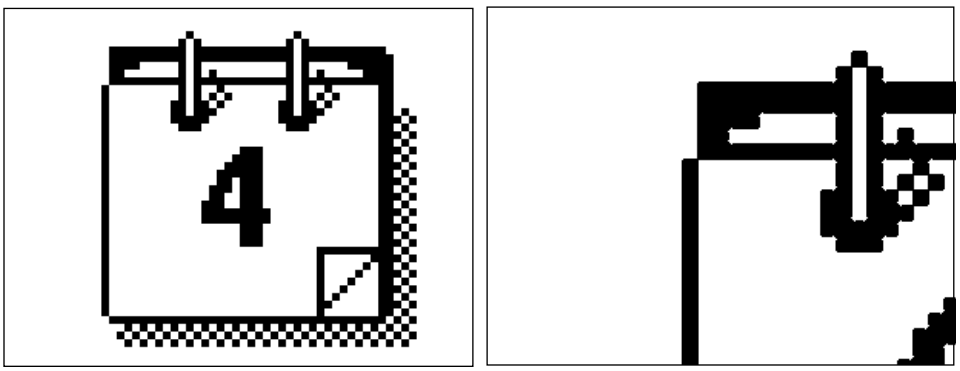


*Figure 10-3* Zooming the Upper-Left Corner of an Image

The image on the left is the original image, and the image on the right is the scaled image. Because the origins are in the upper-left corner of the images, the destination image contains a zoomed version of the upper-left quadrant of the source image.

If, on the other hand, the origins of the source and destination images are set to the pixels at the centers of the images, the scale operation will produce the results shown in Figure 10-4.

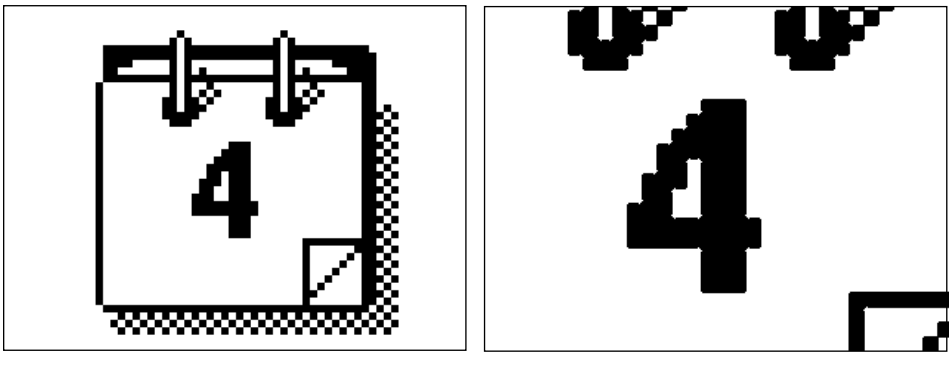


*Figure 10-4* Zooming the Center of an Image

This time the destination image contains a zoomed version of a block taken from the center of the source image.

---

**Note –** The function `xil_scale()` has been optimized for the case where the *x* and *y* scale factors are .5.

---

## `xil_subsample_adaptive()`

The `xil_scale()` function discussed in the last section is the only function XIL provides for enlarging images. However, you can reduce an image's size using either `xil_scale()` or `xil_subsample_adaptive()`.

The library includes the second function because when you scale down an image using `xil_scale()`, you must request one of the interpolation methods discussed earlier: nearest neighbor, bilinear, bicubic, or general. The first three of these interpolation methods look at relatively few source-image pixels when calculating the value of a destination pixel. Therefore, if you use one of these options on `xil_scale()` to reduce the size of an image, there are going to be many cases where some pixels in the source image make no contribution to the value of a pixel in the destination. For example, say that you scale down a 512-by-512 image using an *x* scale factor of .25, a *y* scale factor of .25, and bilinear interpolation. The source image contains 262,144 pixels, and the part of the destination image that represents the scaled-down source image contains 16,384 pixels. Since the operation used bilinear interpolation, the maximum number of source-image pixels that could have contributed to the scaled-down image is 4 times 16,384, or 65,536. In other words, at least three-fourths of the pixels in the source image had no effect on the values in the scaled-down image.

To increase the number of source-image pixels used in the calculation with `xil_scale()`, you could specify the general interpolation method and define your own interpolation tables. However, the function `xil_subsample_adaptive()` circumvents this problem for you because it guarantees that no matter what scale factors you use, every pixel in the source image will contribute to the value of one (and only one) pixel in the scaled-down image. For this reason, scaling down images using `xil_subsample_adaptive()` is generally preferable to using `xil_scale()`.

The function prototype for `xil_subsample_adaptive()` is shown below.

```
void xil_subsample_adaptive(XilImage src, XilImage dst,
    float xscale, float yscale);
```

The parameters to this function look much like those used by the `xil_scale()` routine. Note, however, that you don't specify an interpolation type. The reason for this is that `xil_subsample_adaptive()` has an interpolation scheme built in.

Because `xil_subsample_adaptive()` is useful only for reducing the size of images, `xscale` and `yscale` must be less than or equal to 1.

## xil_subsample_binary_to_gray()

The function `xil_subsample_binary_to_gray()` is similar to `xil_subsample_adaptive()` in that you use it to scale down an image and it guarantees that all the pixels in the source image will make a contribution to the value of one pixel in the scaled-down image. However, `xil_subsample_binary_to_gray()` is designed specifically for scaling down `XIL_BIT` images.

This specialized function is necessary because if you scale down a 1-bit image using `xil_subsample_adaptive()` or `xil_scale()`, no matter how many source pixels you consider in determining the value of a destination-image pixel, the destination pixel can only have one of two values: 0 or 1. Obviously, a great deal of information about the image can be lost in this way.

The function `xil_subsample_binary_to_gray()` helps solve this problem by allowing up to 256 gray levels in the destination image, which must have the data type `XIL_BYTE`. For defining gray levels, one colormap index is used for each possible gray level that could result from the *x* and *y* scaling factors, and the indexes are consecutive values, with 0 representing all 0's in the source image. You must modify your colormap to define a gray level for each resulting index.

For example, Figure 10-5 shows a source image being scaled down by a factor of .5 in both the *x* and *y* dimensions. This means that a block of four pixels in the source image will determine the value of each pixel in the destination.

Since there are five possible combinations of values that can appear in a 2-by-2 block of pixels in the source, there will be five possible gray levels in the destination. The equations below indicate how the subsampling is performed.

- Zero 1's in the source (all 0's) = 0 in the destination
- One 1 in the source = 1 in the destination
- Two 1's in the source = 2 in the destination
- Three 1's in the source = 3 in the destination
- Four 1's in the source = 4 in the destination

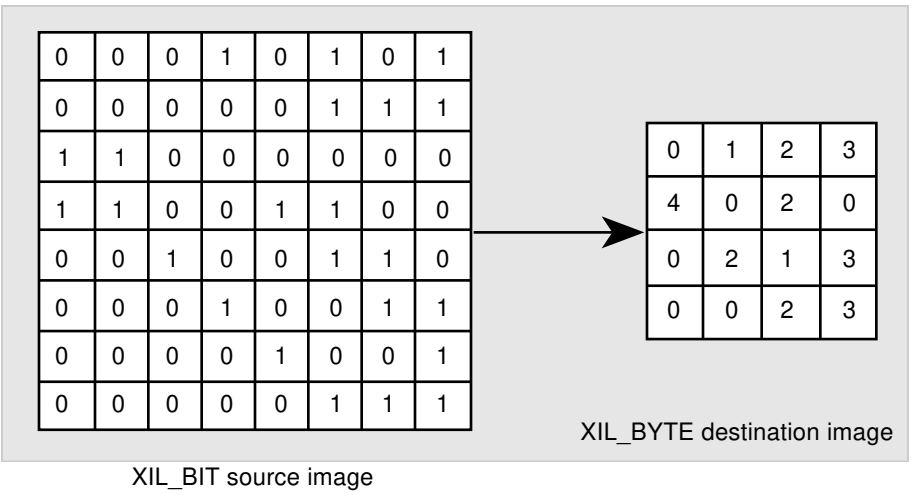


*Figure 10-5* Subsampling Bit Images

If the scaling factors require a fractional block of pixels in the source to determine the destination pixel values, the block size is rounded up. For example, if a 2.2-by-2.2 block of pixels would be required to determine pixel values in the destination, a 3-by-3 block is used, resulting in 10 possible gray levels and therefore 10 colormap indexes whose values are 0 through 9.

The result of this type of operation is much clearer than the image obtained by scaling down a 1-bit image using `xil_subsample_adaptive()` or `xil_scale()`.

**Note –** The function `xil_subsample_binary_to_gray()` has been optimized for the case where the *x* and *y* scale factors are .33 and the width of the destination image is a multiple of 8.

## *Rotating Images*

The XIL library's rotation function is called `xil_rotate()`. This function's prototype is shown below.

```
void xil_rotate(XilImage src, XilImage dst, char *interpolation,
    float angle);
```

The parameter `src` is a handle to the image to be rotated, and `dst` is a handle to the destination image. The parameter `interpolation` is a string that specifies the type of interpolation to be used for the operation: nearest neighbor, bilinear, bicubic, or general. For information about these interpolation methods and how to request one of them, see "Interpolation Options" on page 150. The last parameter, `angle`, indicates the desired angle of rotation in radians. (One radian is equal to 57.2958 degrees.) A positive angle results in a counterclockwise rotation, and a negative angle results in a clockwise rotation.

One thing to keep in mind when you're rotating an image is that `xil_rotate()` rotates an image around its origin. (For information about image origins, see "Origin" on page 46.) For example, consider the situation where you have not set the origin of the source or destination image. Figure 10-6 shows what will happen when you rotate the source image by 90 degrees counterclockwise.

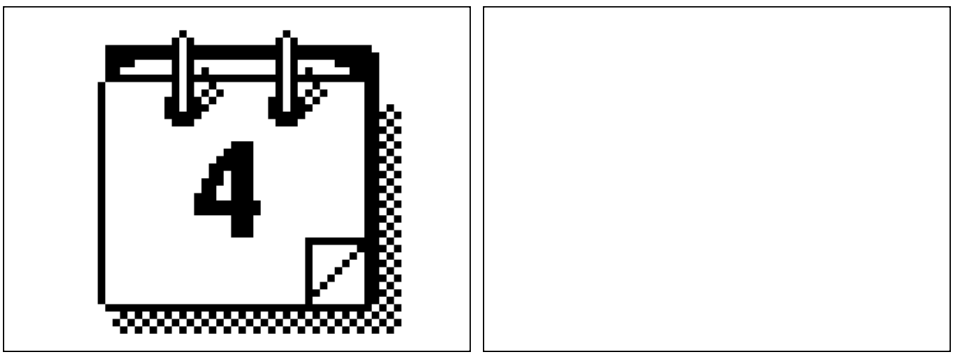


*Figure 10-6* Rotating an Image Around Its Default Origin

Because you're rotating the image about its default origin, which is in the upper-left corner, the image is basically rotated out of sight. Only the pixels in the leftmost column of the source image are written to the destination. They become part of the topmost scanline in that image.

To rotate the source image in the usual sense, you must first set the origins of the source and destination images to their centers.

```
unsigned int width, height;

width = xil_get_width(src);
height = xil_get_height(src);
xil_set_origin(src, width/2.0, height/2.0);
xil_set_origin(dst, width/2.0, height/2.0);
xil_rotate(src, dst, "nearest", 1.5708);
```

This code produces the results shown in Figure 10-7.

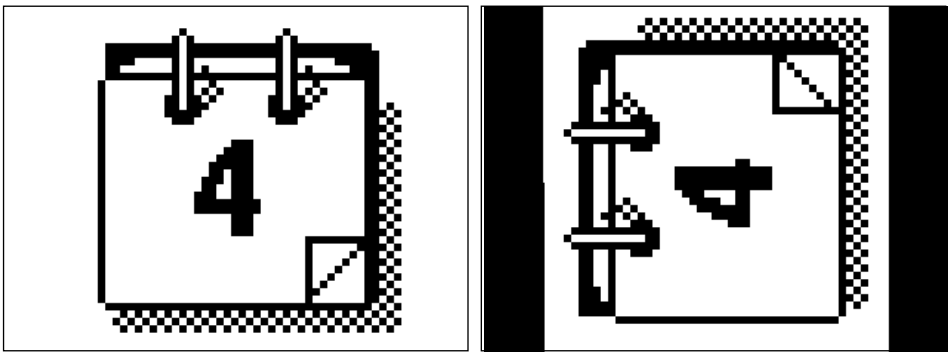


*Figure 10-7* Rotating an Image Around Its Center

## *Performing General Affine Transforms*

An *affine transform* is a transformation of an image in which straight lines remain straight and parallel lines remain parallel, but the distance between lines and the angles between lines may change. Thus, translation, scaling, and rotation are all affine transforms.

The XIL functions that perform *one* of these transforms are discussed in "Translating Images," "Scaling and Subsampling Images," and "Rotating Images." However, the XIL library also contains a function that can perform *any* of these transforms, or a combination of them, such as a rotate and a scale. This function is called `xil_affine()`, and its prototype is shown below.

```
void xil_affine(XilImage src, XilImage dst, char *interpolation,
    float *matrix);
```

The parameter `src` is a handle to the image to be transformed, and `dst` is a handle to the destination image. The parameter `interpolation` is a string that specifies the type of interpolation to be used for the operation. For information about the available methods of interpolation and how to request one of them, see "Interpolation Options" on page 150. The last parameter, `matrix`, must be a six-element array of floating-point numbers.

If we call the six elements in the array a, b, c, d, e, and f, the equations used in performing the transform look like this:

x‘ = (a * x) + (c * y) + e
y’ = (b * x) + (d * y) + f

Or written as a matrix operation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

To illustrate how these equations relate to particular transforms, look at how you would perform the translation, scale, and rotation considered earlier in the chapter using `xil_affine()`.

The following code fragment illustrates an image being translated 50 pixels to the right and 50 pixels down:

```
XilImage src, dst;
float matrix[6] = {1.0, 0.0, 0.0, 1.0, 50.0, 50.0};

xil_affine(src, dst, “nearest”, matrix);
```

The fifth value in the matrix is used to control the horizontal translation of the image, and the sixth is used to control the amount of vertical translation.

The following code fragment shows an image being zoomed by a factor of 2 in both the *x* and *y* directions:

```
XilImage src, dst;
float matrix[6] = {2.0, 0.0, 0.0, 2.0, 0.0, 0.0};

xil_affine(src, dst, “nearest”, matrix);
```

And this code fragment shows an image being rotated 90 degrees counterclockwise.

```
XilImage src, dst;
float matrix[6];

matrix[0] = (float)cos(1.5708);
matrix[1] = (float)-sin(1.5708);
matrix[2] = (float)sin(1.5708);
matrix[3] = (float)cos(1.5708);
matrix[4] = 0.0;
matrix[5] = 0.0;
xil_affine(src, dst, "nearest", matrix);
```

You can also use `xil_affine()` to perform many other affine transforms. For example, the code below shears a source image along the *x* axis and then translates the image to the left so that it is centered in the destination.

```
XilImage src, dst;
float matrix[6] = {1.0, 0.0, -0.4, 1.0, 50.0, 0.0};

xil_affine(src, dst, "nearest", matrix);
```

**Note –** In this case, the *y* coordinate of a pixel in the source image affects the *x* coordinate of a pixel in the destination. Figure 10-8 shows an image before and after this shearing operation.

# 10

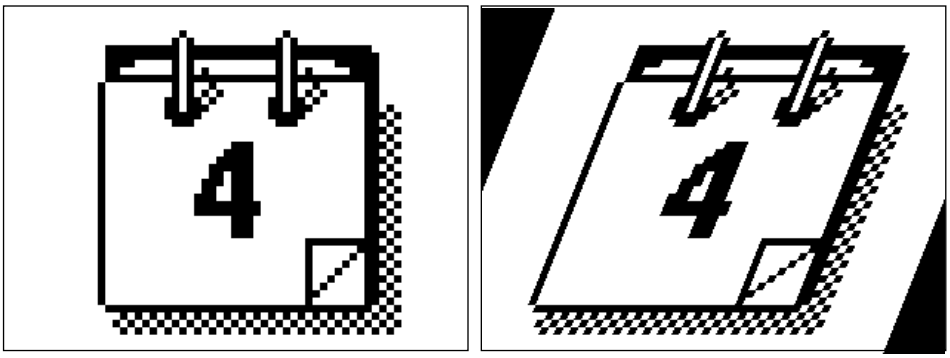


*Figure 10-8* Shearing an Image Along Its *x* Axis

## *Warping Images*

Each of the geometric functions discussed so far in this chapter performs a specific type of affine transformation: `xil_translate()` moves an image up, down, left, or right; `xil_scale()` changes an image's width or height; and so on. The XIL library also contains a set of general functions that let you customize a transformation to meet an individual set of needs by creating *warp* tables that perform pixelwise displacement horizontally, vertically, or in both directions. The warp tables are XIL images whose pixel values define the backward mapping from a pixel in the destination to a pixel in the source.

Warp tables are typically used to stretch an image according to predefined rules and are most useful for performing nonlinear transformations. For example, they might be used to correct distortions that were imposed on an image by the equipment used to capture it. Or they might be used for cartographic projection of an image.

To displace pixels in a single dimension, use either the `xil_tablewarp_horizontal()` or `xil_tablewarp_vertical()` function.

```
void xil_tablewarp_horizontal(XilImage src, XilImage dst,
    char* interpolation, XilImage warp_table);
```

```
void xil_tablewarp_vertical(XilImage src, XilImage dst,
    char* interpolation, XilImage warp_table);
```

To displace pixels in two dimensions, use the `xil_tablewarp()` function.

```
void xil_tablewarp(XilImage src, XilImage dst,
    char* interpolation, XilImage warp_table);
```

`xil_tablewarp_horizontal()` specifies a horizontal displacement, `xil_tablewarp_vertical()` specifies a vertical displacement, and `xil_tablewarp()` specifies a displacement in both directions. For these functions, the parameters `src` and `dst` are handles to the source and destination images. The parameter `interpolation` is a string that specifies an interpolation type for the operation: nearest neighbor, bilinear, bicubic, or general (see "Interpolation Options" on page 150). The parameter `warp_table` is a handle to a defined warp table. For a one-dimensional displacement, the warp table is a 1-banded image that specifies the displacement in one direction. For a two-dimensional displacement, the warp table is a 2-banded image; the first band specifies the horizontal displacement and the second band specifies the vertical displacement.

A warp table is applied to the destination image's origin; the source image's origin is then added to the backward mapping position specified by the warp table. By default, an image's origin is 0,0; to change the origin, use the `xil_set_origin()` function.

A warp table must have the data type `XIL_SHORT`, though it can be used to warp images of any data type; the `XIL_SHORT` value is interpreted as fixed point data with a 12-bit value and 4 bits of precision. Both the source and destination images must have the same type and number of bands. Pixels that map to positions outside the source image are considered zeros. The table warp operations cannot be performed in place.

Warp tables can be used to perform any type of geometric transformation. For example, as discussed in "Translating Images" on page 160, you can move a source image up, down, left, or right with `xil_translate()`. You can also

use a warp table to move the image. The following code fragment shows how you could define a two-dimensional warp table to move an image to the left 100 pixels and down 500 pixels.

```
XilSystemState state;
XilImage src, dst, warp_table;
float values[2];
int width, height;

warp_table = xil_create(state, width, height, 2, XIL_SHORT);
/* multiply offsets by 16 because of 12 bit values with
 * 4 bit precision */
values[0] = 100.0 * 16;
values[1] = 500.0 * 16;
xil_set_value(warp_table, values);

xil_tablewarp(src, dst, "bilinear", warp_table);
```

In this example, variable `warp_table` stores the warp table for the operation; it is assumed that the source image's width and height have already been stored in the `width` and `height` variables. The `warp_table` image has two bands; the first band is needed to store the horizontal displacement defined by `values[0]`, and the second is needed to store the vertical displacement defined by `values[1]`. The call to `xil_set_values()` sets all pixels in the table-warp image to the values defined by the `values` array; thus, the displacement is identical for all pixels. The call to `xil_tablewarp()` then applies the warp table to the source image, using a bilinear interpolation.

Of course, you wouldn't use a warp table to perform a simple translation; warp tables are better used to define algorithms that implement nonlinear displacements. The algorithm would typically define a pixelwise displacement for source-image pixels, in which case you would set the displacements on the warp table by calling `xil_set_pixel()` rather than `xil_set_value()`. For a discussion of warping transformations, see Gonzalez and Wintz, *Digital Image Processing* (see Appendix E, "Bibliography").

## *Transposing Images*

The XIL library's transposition function, `xil_transpose()`, enables you to:

- Flip an image across an imaginary horizontal line that runs through the center of the image
- Flip an image across an imaginary vertical line that runs through the center of the image
- Flip an image across its main diagonal
- Flip an image across its antidiagonal
- Rotate an image counterclockwise about its center by 90, 180, or 270 degrees

The function's prototype is shown below:

```
void xil_transpose(XilImage src, XilImage dst,
    XilFlipType fliptype);
```

The parameters `src` and `dst` are handles to the source and destination images. The parameter `fliptype` is an enumeration constant of type `XilFlipType`. The constants in the enumeration and their meanings are shown in Table 10-2.

*Table 10-2* Constants in the Enumeration `XilFlipType`

| Constant | Meaning |
|---|---|
| `XIL_FLIP_X_AXIS` | Flips the image across a horizontal line running through its center |
| `XIL_FLIP_Y_AXIS` | Flips the image across a vertical line running through its center |
| `XIL_FLIP_MAIN_DIAGONAL` | Flips the image across its main diagonal |
| `XIL_FLIP_ANTIDIAGONAL` | Flips the image across its antidiagonal |
| `XIL_FLIP_90` | Rotates the image 90 degrees counterclockwise about its center |
| `XIL_FLIP_180` | Rotates the image 180 degrees counterclockwise about its center |
| `XIL_FLIP_270` | Rotates the image 270 degrees counterclockwise about its center |

**Note –** You don't specify an interpolation type when you call `xil_transpose()`. This is not necessary because `xil_transpose()` differs slightly from the other geometric operators in that it maps pixels in the source

image directly to pixels in the destination. Another important thing to note about `xil_transpose()` is that it ignores image origins when it flips or rotates images. The function always flips an image across a line that passes through the center of the image and always rotates an image around its center.

Figure 10-9 illustrates several of the operations that `xil_transpose()` can perform.

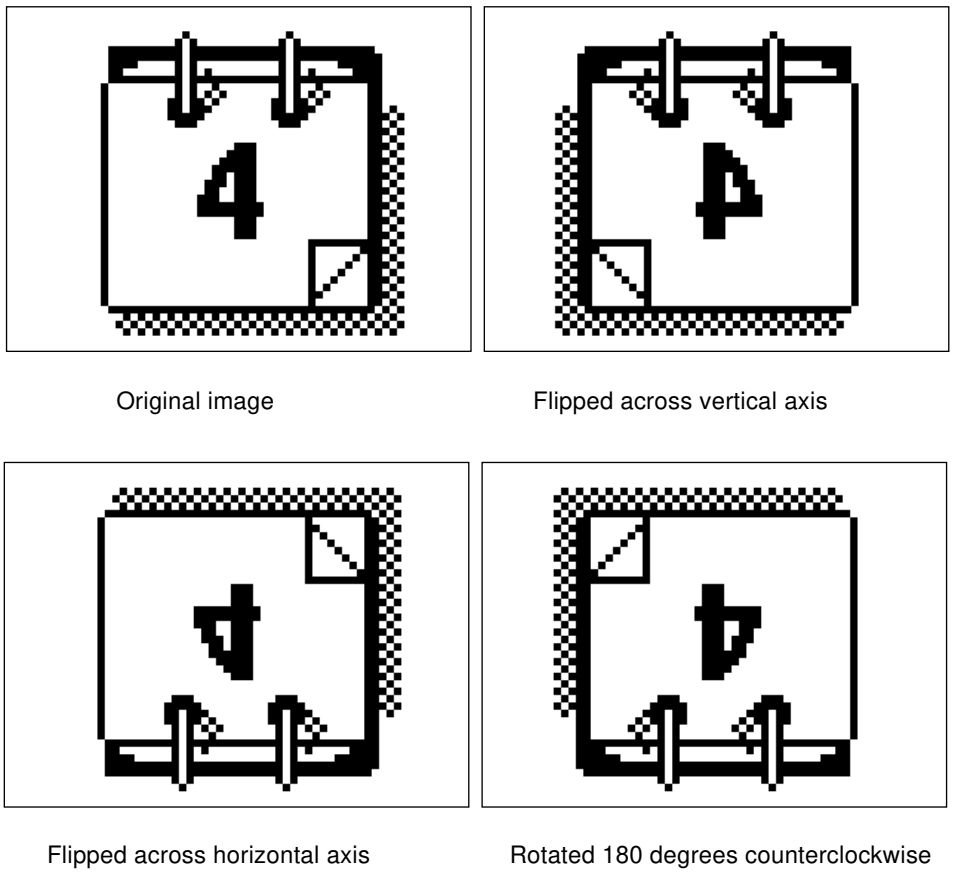


*Figure 10-9* Flipping and Rotating Images Using `xil_transpose()`