Miscellaneous Image Processing Functions 11

Earlier chapters in this book have discussed several classes of XIL image-processing functions. Chapter 7, "Presentation Functions," talked about the XIL functions you're most likely to use in preparing an image for display. Chapter 9, "Arithmetic, Relational, and Logical Functions," considered the XIL library's arithmetic and logical functions. And Chapter 10, "Geometric Functions," discussed the library's geometric functions. This chapter covers the library's remaining image-processing functions. These functions enable you to:

- Find the minimum and maximum values in an image
- Produce a histogram for an image
- Threshold an image
- Fill an area in an image
- Filter an image
- Detect an image's edges
- Dilate or erode an image
- Pass an image through a lookup table
- Perform a linear combination of the values in the different bands of an image
- Blend images
- Paint on an image
- Set and get the values of pixels in an image
- Copy a pattern to an image

These operations are discussed in the sections below.

11

Finding the Minimum and Maximum Values in an Image

The function `xil_extrema()` finds the minimum and maximum values in a single-band image or the minimum and maximum values in each band of a multiband image. The prototype for this function is shown below:

```
void xil_extrema(XilImage src, float *max, float *min);
```

The parameter `src` is a handle to a source image. The parameter `max` is an array of floating-point numbers in which the function will return the maximum value found in each band of the source image. The parameter `min` is an array of `floats` in which the function will return the minimum value found in each band of the image. The number of elements in the arrays `max` and `min` must match the number of bands in the image.

Consider the following example:

```
XilImage src;
float max[3], min[3];

xil_extrema(src, max, min);
```

After this call, `max[0]` will contain the maximum value in band 0 of `src`, `max[1]` will contain the maximum value in band 1, and `max[2]` will contain the maximum value in band 2. Similarly, `min` will contain the minimum values in the three bands of the image.

Note – `xil_extrema()` returns a minimum or maximum value, depending on which one you specify. Don't confuse this with finding the larger or lesser of pixels between two source images and storing the results in a destination image. To perform that relational function, use `xil_max()` or `xil_min()`, discussed on page 143.

Producing a Histogram for an Image

The XIL library defines a data type called `XilHistogram`. A data structure of this type is designed to hold gray-level (or color-level) information about an image. This section discusses how to produce a histogram for an image and how to read intensity-level information from an XIL histogram structure. The primary tasks you will need to perform are listed below:

1. Create a histogram data structure.
2. Write gray- or color-level information about an image to the data structure.
3. Read the histogram data stored in the structure.
4. Destroy the histogram.

These tasks are discussed in detail in the sections that follow.

Creating a Histogram

An XIL histogram structure contains a set of *bins* for each band of the image whose histogram you want to take. These bins are used to hold information about gray or color levels. For instance, if you want to take the histogram of an 8-bit grayscale image, you might create a histogram structure that contains 256 bins. When you actually take the histogram, the number of 0's in the image will be stored in bin 0, the number of 1's in the image will be stored in bin 1, and so on.

Your histogram need not contain a bin for each possible value in the image. You can specify the lowest and highest values that will result in a bin count being incremented. In addition, you can specify that the histogram structure contain a number of bins less than the number of levels you will be checking for. In this case, each bin holds a count for a range of values. For example, if you create a histogram for an 8-bit grayscale image and specify four bins, occurrences of the values 0 to 63 will be stored in bin 0, occurrences of 64 to 127 will be stored in bin 1, and so on.

11

You create a histogram data structure by calling the function `xil_histogram_create()`, whose function prototype is shown below.

```
XilHistogram xil_histogram_create(XilSystemState state,
    unsigned int nbands, unsigned int *nbins, float *low_value,
    float *high_value);
```

Table 11-1 discusses the purpose of each of these parameters.

Table 11-1 Parameters to `xil_histogram_create()`

Parameter	What It Represents
`state`	The system-state data structure returned when you initialized the library.
`nbands`	The number of bands in the histogram. This number must match the number of bands in the image whose histogram you want to take.
`nbins`	An array of `unsigned int`s, each element of which specifies the number of bins to be used for one band of the image. The number of elements in the array must match the number of bands in the image.
`low_value`	An array of `float`s, each element of which specifies the lowest gray or color level that will be checked for in one band of the image. The number of elements in the array must match the number of bands in the image.
`high_value`	An array of `float`s, each element of which specifies the highest gray or color level that will be checked for in one band of the image. The number of elements in the array must match the number of bands in the image.

Writing Level Information to the Histogram Structure

Once you have created an appropriate histogram structure, you generate the histogram for an image using the function `xil_histogram()`. The prototype for this function is shown below.

```
void xil_histogram(XilImage src, XilHistogram histogram,
    unsigned int skip_x, unsigned int skip_y);
```

The parameter `src` is a handle to the image whose histogram you want to take. The parameter `histogram` is a handle to the histogram structure you created earlier. The parameters `skip_x` and `skip_y` tell `xil_histogram()` whether it should count every pixel in the source image, or whether it can skip some pixels, either horizontally or vertically.

If `skip_x` is set to 1, `xil_histogram()` counts every pixel on a scanline; if it is set to 2, the function counts every other pixel; and so on. The value of `skip_y` has an analogous effect on whether `xil_histogram()` counts every pixel in the vertical direction.

Reading Data from a Histogram

Once you've generated histogram data for an image, you'll want to read the counts stored in the bins of the histogram structure. You read this data using the function `xil_histogram_get_values()`, whose prototype is shown below.

```
void xil_histogram_get_values(XilHistogram histogram,
    unsigned int *data);
```

The parameter `histogram` is a handle to your histogram structure, and `data` is a pointer to an array of `unsigned int`s in which `xil_histogram_get_values()` can place the histogram data.

The code fragment below shows how you might retrieve the histogram data for a 3-band `XIL_BYTE` image. This code assumes that 32 bins were used for each band of the image.

```
#define BINS 32

XilHistogram histogram;
unsigned int *data;

data = (unsigned int *)malloc(BINS * BINS * BINS *
    sizeof(unsigned int));
xil_histogram_get_values(histogram, data);
```

The XIL library also contains functions that enable you to retrieve other information from histogram structures and to name histograms. These are listed in Table 11-2.

Table 11-2 Additional Histogram Functions

Function Name	What The Function Does
`xil_histogram_get_nbands`	Returns the number of bands in the histogram
`xil_histogram_get_nbins`	Fills a user-supplied array with values representing the number of histogram bins for each histogram band
`xil_histogram_get_limits`	Fills one user-supplied array with floating-point numbers that represent the low values for the histogram's bands and a second array with numbers representing the high values
`xil_histogram_get_info`	Returns information about the number of bands, the number of bins, and the low and high values in the histogram
`xil_histogram_set_name`	Sets the name of a histogram
`xil_histogram_get_name`	Returns a copy of a histogram's name
`xil_histogram_get_by_name`	Returns a handle to a histogram that has the name you specify

Destroying a Histogram

After reading the histogram data for your image, you should destroy the histogram structure if you will not be using it again. Destroying the histogram frees the memory that was allocated to store it.

You destroy a histogram structure by calling the function `xil_histogram_destroy()`, whose prototype is shown below.

```
void xil_histogram_destroy(XilHistogram histogram);
```

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

Thresholding an Image

The XIL library's thresholding function, `xil_threshold()`, provides a simple mechanism for defining the boundaries of objects that appear on a contrasting background. The function's prototype is shown below.

```
void xil_threshold(XilImage src, XilImage dst, float *lowvalue,
    float *highvalue, float *mapvalue);
```

The parameters `src` and `dst` are handles to the source and destination images for the operation. The parameter `lowvalue` is a pointer to an array of floating-point numbers that define the lower bound for the threshold operation for each band of `src`. That is, the operation will affect only values greater than or equal to `lowvalue[0]` in band 0, only values greater than or equal to `lowvalue[1]` in band 1, and so on. The parameter `highvalue` is a pointer to an array of `float`s that define the upper bound for the threshold operation for each band of `src`. The final parameter, `mapvalue`, is also a pointer to an array of `float`s and is used as follows. For band *n* of an image, all values in the range `lowvalue[`*n*`]` to `highvalue[`*n*`]` inclusive are set to `mapvalue[`*n*`]`.

A standard way of arriving at the optimal values for the elements of the arrays `lowvalue` and `highvalue` is to create a histogram for the image. For information about XIL histograms, see the section "Producing a Histogram for an Image" on page 179.

Filling an Area in an Image

The XIL library contains two fill functions: `xil_fill()` and `xil_soft_fill()`. The first function performs boundary fills, and the second performs soft fills.

`xil_fill()`

The function `xil_fill()` fills a 4-connected region of pixels with a specified color. (One pixel is 4-connected to another if it is located directly above, below, to the left of, or to the right of that pixel.) The region begins at a starting pixel (the *seed*) and grows until it encounters a boundary color or the edge of the image.

Figure 11-1 shows a fill operation in which the boundary color is black, the fill color is dark gray, and the starting point for the fill is 3,2.

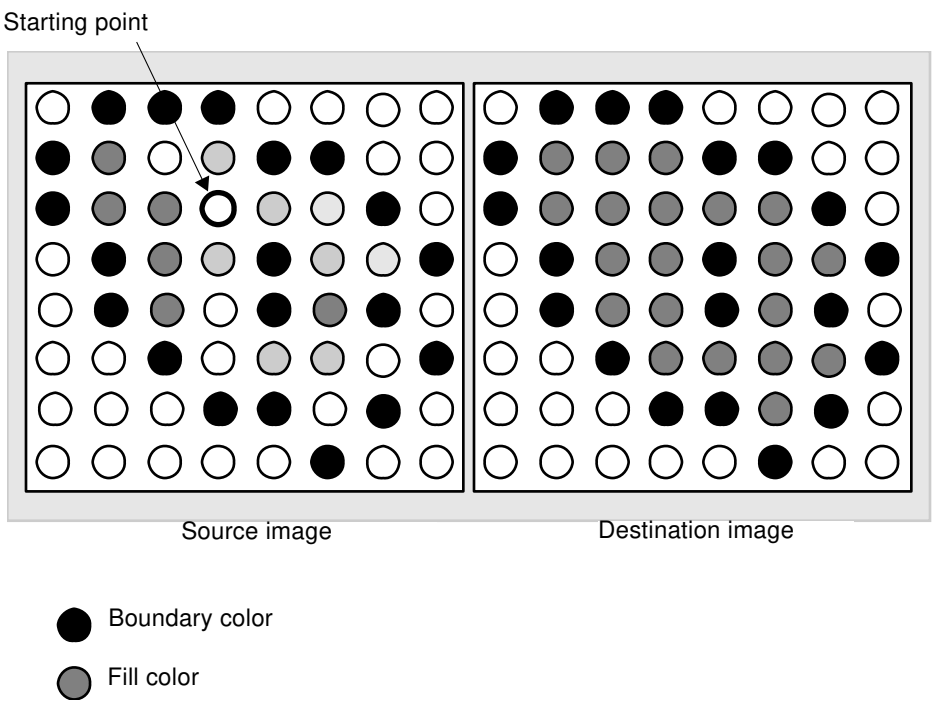

Figure 11-1 Boundary Fill

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

```
void xil_fill(XilImage src, XilImage dst, float xseed,
    float yseed, float *boundary, float *fill_color);
```

The parameters `src` and `dst` are handles to the source and destination images for the operation. These images must have the same number of bands and the same data type. The parameters `xseed` and `yseed` represent the *x* and *y* coordinates of the starting point. The parameters `boundary` and `fill_color` are arrays of floating-point numbers. Both arrays must contain a number of

elements equal to the number of bands in the source and destination images. The elements of `boundary` represent the boundary color, and those in `fill_color` represent the fill color.

xil_soft_fill()

The function `xil_soft_fill()` is designed to solve the problem of filling a region that does not have a distinct boundary, usually because the outline of the region has been antialiased. It also fills regions that are shaded or contain specular reflections without losing the special effects.

Here's how `xil_soft_fill()` works. Assume for the purpose of illustration that you want to fill a red object that appears on a blue background and that the object has been antialiased so that there's a transition at the edge of the object from red to magenta to blue. You want to fill the object with green. To get started, you specify:

- The color of the *foreground*, the region to be filled: red
- The color of the background: blue. (There may be more than one background color.)
- A fill color: green
- A starting point (seed) known to be inside the region to be filled

The soft-fill function then determines which pixels are inside the region and sets those pixels appropriately. To be considered part of the region to be filled, a pixel must be 4-connected to a pixel inside the region and must contain some fraction of the foreground color. That is, the value of each pixel in the region can be expressed as:

Pixel value = (fraction * foreground-color) + ((1 - fraction) * background-color)

When setting the values of pixels in the destination image, `xil_soft_fill()` uses the equation:

Pixel value = (fraction * fill-color) + ((1 - fraction) * background-color)

As a consequence, where there was a transition from red to magenta to blue in the source, there will be a transition from green to cyan to blue in the destination.

The prototype for `xil_soft_fill()` is shown below.

```
void xil_soft_fill(XilImage src, XilImage dst, float xseed,
    float yseed, float *fgcolor, unsigned int num_bgcolor,
    float *bgcolor, float *fill_color);
```

Table 11-3 below explains the purpose of each of these parameters.

Table 11-3 Parameters to `xil_soft_fill()`

Parameter	What It Is Used For
`src`	A handle to the source image for the operation.
`dst`	A handle to the destination image. This image must have the same number of bands and the same data type as the source image.
`xseed`	The *x* coordinate of the starting point for the operation.
`yseed`	The *y* coordinate of the starting point for the operation.
`fgcolor`	An array of `floats` containing a number of elements equal to the number of bands in the source and destination images. These elements specify the current color of the region to be filled.
`num_bgcolor`	The number of background colors in the image.
`bgcolor`	An array of `floats` containing a number of elements equal to the number of background colors times the number of bands in the source and destination images. If your RGB source image contained two background colors—black and red—your declaration of `bgcolor` might look like this: `float bgcolor[6] = {0.0,0.0,0.0,0.0,0.0,255.0};`
`fill_color`	An array of `floats` containing a number of elements equal to the number of bands in the source and destination images. These elements specify the color to be used in filling the region of interest.

Filtering an Image

The XIL function for filtering is called `xil_convolve()`. Depending on the convolution filter, or *kernel*, you specify as a parameter to this function, the routine can perform a variety of tasks, including:

- Sharpening images
- Blurring images
- Highlighting the edges in images

See Color Plate 2 for a few examples of how different kernels can affect a source image.

Note – To detect edges, you can also use `xil_edge_detection()`, which uses the Sobel algorithm. For information about `xil_edge_detection()`, see "Detecting Edges in an Image" on page 192.

The kernel you supply as a parameter is a two-dimensional array of weighted values that the `xil_convolve()` function uses as follows. To calculate the value of each pixel in the destination image, the function takes the following steps:

1. It lays the kernel over the corresponding pixel in the source image.

   To be precise, the function lays the key value of the filter—usually the value at the center of the filter—over this source-image pixel. This means that the other values in the filter lie on top of neighbors of this pixel. Figure 11-2 shows a 3-by-3 kernel being laid on top of the source-image pixel at 1,1 and its neighbors.

11

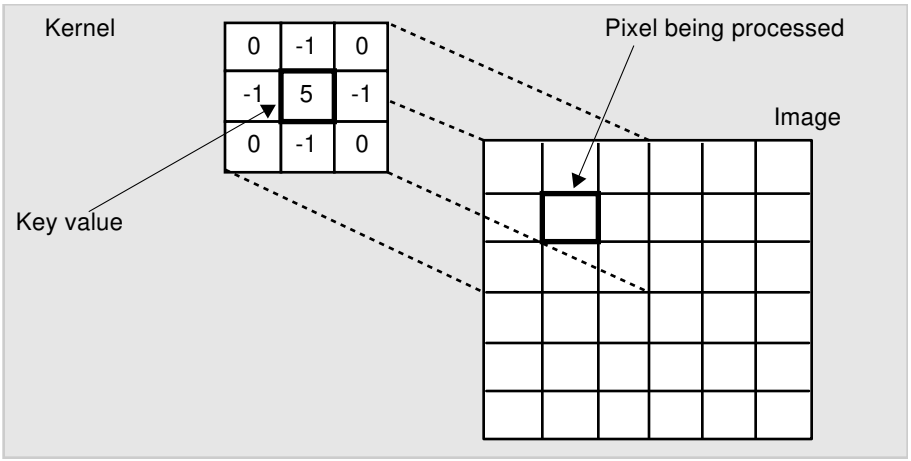

Figure 11-2 Convolution Operation

2. The convolution function multiplies each pixel in the neighborhood defined by the filter by the appropriate value in the filter. Then, it sums the products of these multiplications. This sum becomes the value of the pixel in the destination.

Three basic steps are involved in performing this type of convolution operation using the XIL library. These are:

- Creating the convolution kernel
- Filtering your image
- Destroying the kernel

These tasks are discussed in detail in the next several sections.

Creating a Convolution Kernel

Convolution kernels are XIL data structures of type `XilKernel` and are created with the function `xil_kernel_create()`. The prototype for this function is shown below.

```
XilKernel xil_kernel_create(XilSystemState state,
    unsigned int width, unsigned int height, unsigned int keyx,
    unsigned int keyy, float *data);
```

The parameter `state` is a system-state data structure. You received a handle to this structure when you initialized the library. The parameters `width` and `height` are the width and height of the kernel in pixels. These values are usually odd numbers so that the kernel will have a center value, but need not be. Common sizes for kernels are 3-by-3 and 5-by-5. The parameters `keyx` and `keyy` define the coordinates of the key value in the kernel. These coordinates are specified with respect to the upper-left value in the kernel (0,0). As mentioned earlier, the key value is usually in the center of the kernel. The last parameter, `data`, is a pointer to the floating-point values that will be written to the kernel.

The code needed to create the kernel shown in Figure 11-2 would look something like this.

```
XilKernel high_pass_filter;
XilSystemState state;
unsigned int width, height, keyx, keyy;
float data[] = {0.0, -1.0, 0.0,
                -1.0, 5.0, -1.0,
                0.0, -1.0, 0.0};

width = height = 3;
keyx = keyy = 1;
high_pass_filter = xil_kernel_create(state, width, height, keyx,
    keyy, data);
```

The kernel created here is a high-pass filter, used for sharpening an image. Two other common high-pass filters are shown in Figure 11-3.

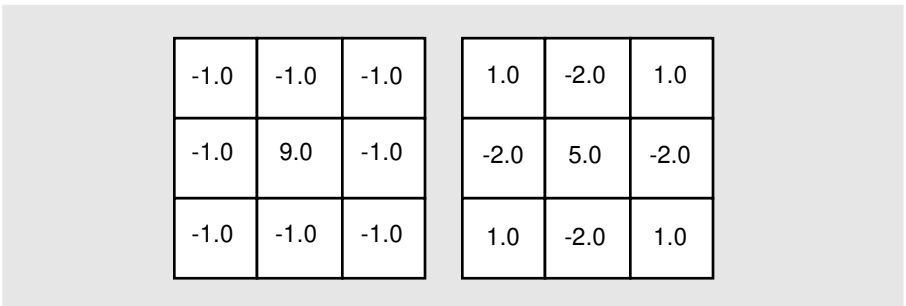

Figure 11-3 High-Pass Filters

Figure 11-4 shows several low-pass filters. These are useful for smoothing or blurring images.

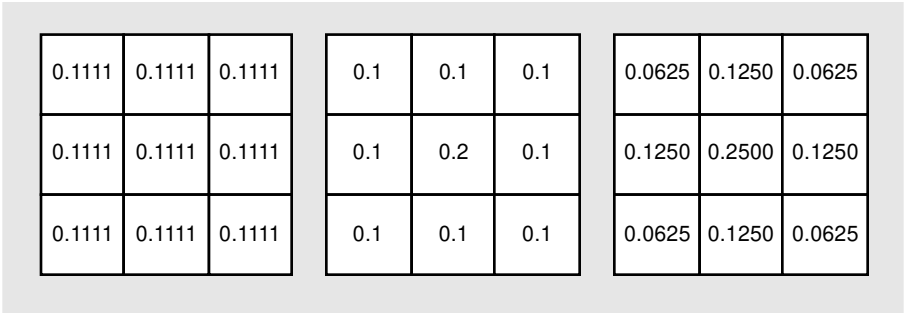

Figure 11-4 Low-Pass Filters

Filtering an Image

Once you have created the convolution kernel you want to use for your filtering operation, you perform the actual filtering by calling `xil_convolve()`. The prototype for this function is shown below.

```
void xil_convolve(XilImage src, XilImage dst, XilKernel kernel,
    XilEdgeCondition edge_condition);
```

The parameters `src` and `dst` are handles to a source and destination image. These two images must have the same number of bands and contain the same type of data. The parameter `kernel` is a handle to the convolution kernel you created earlier. The final parameter, `edge_condition`, is an enumeration

constant that indicates how `xil_convolve()` should handle pixels at, or near, the edge of the source image. Table 11-4 lists the three constants you can use here and explains what effect they have.

Table 11-4 Handling Edges in a Convolution Operation

Value for **`edge_condition`**	Effect on Convolution Operation
`XIL_EDGE_NO_WRITE`	The values of pixels at the edge of the source image are copied to the corresponding pixels in the destination. Thus, the edge of the source image is not filtered.
`XIL_EDGE_ZERO_FILL`	Pixels at the edge of the destination image are set to zero.
`XIL_EDGE_EXTEND`	The convolution operator temporarily extends the width and height of the source image by replicating the pixels at the edge of the image. This strategy enables the operator to filter the pixels that were originally at the edge of the image.

Destroying a Convolution Kernel

After filtering your image, you should destroy the convolution kernel you used for the filtering if you will not be using it again. Destroying this kernel frees the memory that was allocated to store it.

You destroy a convolution kernel by calling the function `xil_kernel_destroy()`, whose prototype is shown below.

```
void xil_kernel_destroy(XilKernel kernel);
```

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

11

Additional Kernel-Related Functions

The preceding sections have discussed the most frequently used XIL functions that affect kernels. However, the library also contains the kernel-related functions shown in Table 11-5. These functions enable you to make a copy of a kernel, read the values of kernel attributes, and assign a name to a kernel, among other things.

Table 11-5 Utility Functions for Convolution Kernels

Function Name	What the Function Does
`xil_kernel_create_copy`	Creates a copy of a kernel
`xil_kernel_get_width`	Gets the width of a kernel in pixels
`xil_kernel_get_height`	Gets the height of a kernel in pixels
`xil_kernel_get_key_x`	Gets the *x* coordinate of the kernel's key value
`xil_kernel_get_key_y`	Gets the *y* coordinate of the kernel's key value
`xil_kernel_set_name`	Assigns a name to a kernel
`xil_kernel_get_name`	Returns a copy of a kernel's name
`xil_kernel_get_by_name`	Gets a handle to a kernel by specifying the name of the kernel

Detecting Edges in an Image

The `xil_edge_detection()` function detects edges in a source image and writes the result to a destination image. The prototype for this function is shown below.

```
void xil_edge_detection(XilImage src, XilImage dst,
    XilEdgeDetection edge_detection_method);
```

The parameters `src` and `dst` are handles to a source and destination image. The two images must have the same number of bands and contain the same type of data. The parameter `edge_detection_method` is an enumeration constant describing the type of edge detection to perform.

Currently, XIL_EDGE_DETECT_SOBEL is the only edge detection method supported; to detect vertical and horizontal edges in an image, it uses the filtering kernels shown in Figure 11-5.

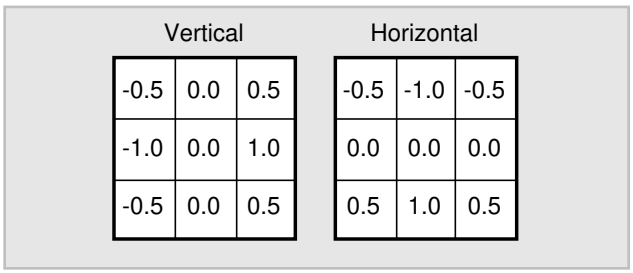

Figure 11-5 Filters Used by the XIL_EDGE_DETECT_SOBEL Algorithm

The XIL_EDGE_DETECT_SOBEL method detects edges by finding the gradient of an image as follows:

- It performs two convolution operations on the source image; one operation detects vertical edges using the vertical filter shown in Figure 11-5, the other detects horizontal edges using the horizontal filter. This yields two intermediate images: *a* and *b*.
- It squares all the pixel values in image *a* and then in image *b*, yielding the intermediate images a^2 and b^2.
- It forms the final destination image by taking the square root of $a^2 + b^2$.

Note – The convolution operations duplicate the source-image edges during convolution, similar to using the XIL_EDGE_EXTEND edge condition on xil_convolve(). For information on xil_convolve(), see "Filtering an Image" on page 187.

Note – For a discussion of the Sobel edge condition algorithm, consult Gonzalez and Wintz, *Digital Image Processing* (see Appendix E, "Bibliography").

11

Dilating or Eroding an Image

The XIL library includes two functions—`xil_dilate()` and `xil_erode()`—that enable you to dilate and erode regions within images. One of the main uses of these functions is to perform *trapping* on color images that will be printed. This trapping involves creating some overlap where regions of different colors meet. Once a trap is in place, slight registration problems during the printing process should not result in gaps appearing between the edge of an object and its background.

Dilation and erosion are similar to convolution in that both calculate destination-image pixels by looking at a neighborhood of source-image pixels and the values in a matrix (called a *structuring element*). Like a convolution kernel, this structuring element contains a key value. As each pixel in the destination is calculated, this key value is laid over the corresponding pixel in the source. See Figure 11-6.

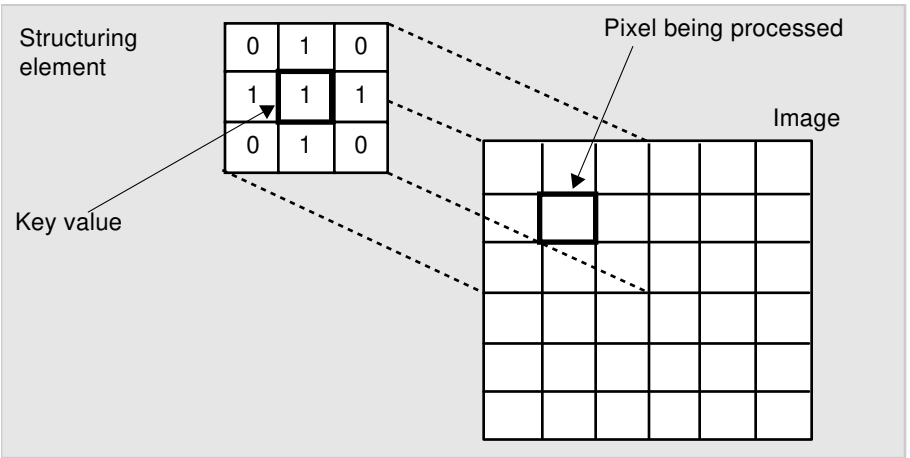

Figure 11-6 Dilating and Eroding Images

Note that a structuring element contains only 0's and 1's.

For each placement of the structuring element, the dilation function calculates a destination pixel value as follows. It looks at the values of the source-image pixels that correspond to a 1 in the structuring element and assigns the maximum of these values to the appropriate destination pixel. In the case of an `XIL_BIT` image, this means that if any of the source-image pixels associated

with a 1 in the structuring element is set, the destination pixel will be set. Figure 11-7 shows a binary image before and after dilation of the image's white region. The dilation was performed using the structuring element shown in Figure 11-6.

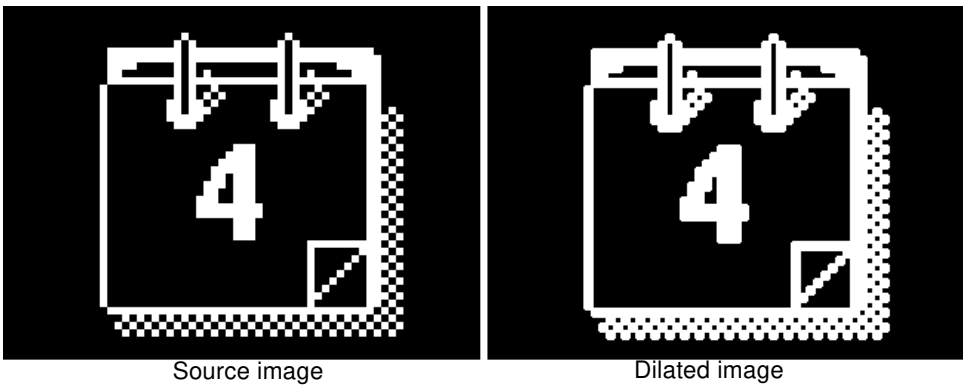

Figure 11-7 Dilating an Image

The erosion function does the opposite of this. It looks at the values of the source-image pixels that correspond to a 1 in the structuring element and assigns the minimum of these values to the appropriate destination pixel. Figure 11-8 shows a binary image before and after erosion of the image's white region.

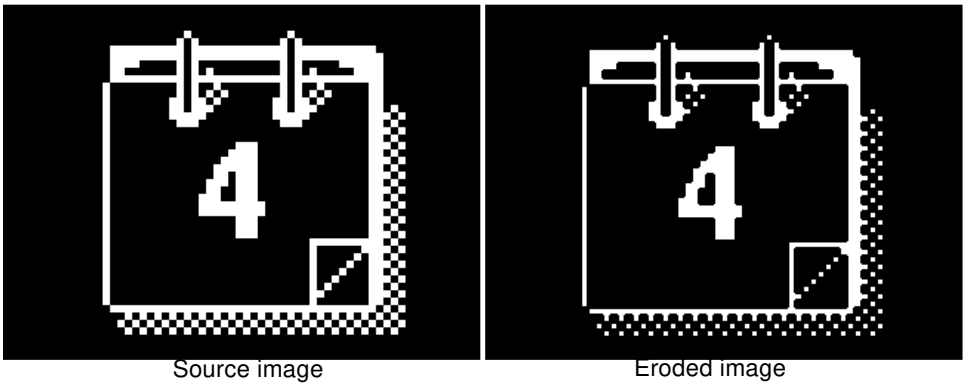

Figure 11-8 Eroding an Image

Performing either a dilation or an erosion in an XIL program is a three-step process:

1. Create a structuring element.
2. Perform the dilation or erosion.
3. Destroy the structuring element.

These tasks are discussed in detail in the sections below.

Creating a Structuring Element

Structuring elements, which are XIL data structures of type `XilSel`, are created with the function `xil_sel_create()`. The prototype for this function is shown below.

```
XilSel xil_sel_create(XilSystemState state,
    unsigned int width, unsigned int height, unsigned int keyx,
    unsigned int keyy, unsigned int *data);
```

The parameter `state` is a system-state data structure. You received a handle to this structure when you initialized the library. The parameters `width` and `height` are the width and height of the structuring element in pixels. These values are usually odd numbers so that the structuring element will have a

center value, but need not always be odd. Common sizes for structuring elements are 3-by-3 and 5-by-5. The parameters `keyx` and `keyy` define the coordinates of the key value in the kernel. These coordinates are specified with respect to the upper-left value in the structuring element (0,0). The key value is usually in the center of the structuring element. The last parameter, `data`, is a pointer to the Boolean values that will be written to the kernel.

The code needed to create the structuring element shown in Figure 11-6 would look something like this.

```
XilSel cross_sel;
XilSystemState state;
unsigned int width, height, keyx, keyy;
unsigned int data[] = {0, 1, 0,
                       1, 1, 1,
                       0, 1, 0};

width = height = 3;
keyx = keyy = 1;
cross_sel = xil_kernel_create(state, width, height, keyx, keyy,
    data);
```

You can also assign names to structuring elements that you've created and then get handles to those structuring elements using the function `xil_sel_get_by_name()`. To assign a name to a custom structuring element, you use the function `xil_sel_set_name()`, whose prototype is shown below.

```
void xil_sel_set_name(XilSel sel, char *sel_name);
```

You can determine whether a structuring element has a name and, if it does, what that name is using the function `xil_sel_get_name()`.

```
char *xil_sel_get_name(XilSel sel);
```

Dilating or Eroding an Image

Once you have created a structuring element, you perform a dilation by calling `xil_dilate()`, whose prototype is shown below.

```
void xil_dilate(XilImage src, XilImage dst, XilSel sel);
```

The parameters `src` and `dst` are handles to the source and destination images for the operation. These images must have the same number of bands and the same data type. The parameter `sel` is a handle to the structuring element you created earlier.

To erode a region in an image, you call `xil_erode()`, which takes the same parameters as `xil_dilate()`.

```
void xil_erode(XilImage src, XilImage dst, XilSel sel);
```

Destroying a Structuring Element

After performing your dilation or erosion, you should destroy your structuring element if you will not be using it again. Destroying the structuring element frees the memory that was allocated to store that structure.

You destroy a structuring element by calling `xil_sel_destroy()`.

```
void xil_sel_destroy(XilSel sel);
```

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

Additional Structuring-Element Functions

The preceding sections have discussed the most frequently used XIL functions that affect structuring elements. However, the library also contains the structuring-element-related functions shown in Table 11-6. These functions enable you to make a copy of a structuring element and read the values of structuring-element attributes.

Table 11-6 Utility Functions for Structuring Elements

Function Name	What the Function Does
`xil_sel_create_copy`	Returns a copy of a structuring element
`xil_sel_get_width`	Gets the width of a structuring element
`xil_sel_get_height`	Gets the height of a structuring element
`xil_sel_get_key_x`	Gets the *x* coordinate of the key value of a structuring element
`xil_sel_get_key_y`	Gets the *y* coordinate of the key value of a structuring element

Passing an Image Through a Lookup Table

XIL lookup tables provide a very general mechanism for modifying images. A lookup table enables you to 1) convert a single-band image of any data type to a single-band or multiband image of any data type, or 2) convert a multiband image of any data type to a multiband image of any data type. Each lookup table allows you to specify precisely the correlation between the values of pixels in the source image and values in the destination image.

The general procedure for performing a lookup operation involves three steps:

1. Create a lookup table.
2. Pass a source image through the lookup table.
3. Destroy the table.

These tasks are discussed in detail in the sections that follow.

11

Creating a Lookup Table

The steps for creating a lookup table vary slightly, depending on whether the input image is a single-band or multiband image. For a single-band input image, you create a single lookup table; for a multiband input image, you create a single lookup table for each band in the input image, then combine those tables into a *combined* lookup table. Regardless of whether you create a single lookup table or a combined lookup table, you pass the image through the table the same way.

Lookup tables for single-band and multiband input images are discussed separately in the following sections.

Creating Lookup Tables for Single-Band Input Images

To create a lookup table for a single-band input image, you make a single call to the function `xil_lookup_create()`, whose prototype is shown below.

```
XilLookup xil_lookup_create(XilSystemState state,
    XilDataType input_data_type, XilDataType output_data_type,
    unsigned int output_nbands, unsigned int num_entries,
    short first_entry_offset, void *data);
```

This function returns a handle to a data structure of type `XilLookup`, which is the lookup table. Table 11-7 lists the parameters to `xil_lookup_create()` and explains what each parameter represents.

Table 11-7 Parameters to `xil_lookup_create()`

Parameter	What It Represents
`state`	A handle to the system-state data structure that was created when you initialized the XIL library.
`input_data_type`	The data type of the pixel values in the source image. The three possible data types are `XIL_BIT`, `XIL_BYTE`, and `XIL_SHORT`.
`output_data_type`	The data type of the pixel values in the destination image.
`output_nbands`	The number of bands in the destination image.

Table 11-7 Parameters to `xil_lookup_create()`

Parameter	What It Represents
`num_entries`	The number of entries in the lookup table. Each entry specifies the correspondence between a source-image pixel value and the value, or values, that will define the corresponding pixel in the destination image.
`first_entry_offset`	The source-image pixel value for the first entry in the table. The source-image pixel value for the last entry in the table is `first_entry_offset` + `num_entries` - 1.
`data`	A pointer to the data to be stored in the table.

As an example of how this function might be used, say that you have an `XIL_BIT` image that you want to convert to a three-band `XIL_BYTE` image so that you can display it on a 24-bit display. Further, you want black pixels in the source to appear dark gray in the destination image, and white pixels to appear light blue. The code to create a lookup table for this operation is shown below.

```
XilLookup lookup_table;
XilSystemState state;
Xil_unsigned8 data[] = {79, 79, 47, 216, 216, 191};

lookup_table = xil_lookup_create(state, XIL_BIT, XIL_BYTE, 3, 2,
    0, data);
```

The lookup table this code would produce is shown in Figure 11-9.

0	79	79	47
1	216	216	191

Figure 11-9 Single Lookup Table

The numbers to the left of the bold line are XIL_BIT values to be looked up in the source image. The numbers to the right of the line are XIL_BYTE values to be written to the destination image. Because the destination has three bands, the table contains three values for each pixel.

A lookup table can also be used to convert an XIL_BIT image to an 8-bit XIL_BYTE image; however, it may be easier to use xil_cast() to cast the XIL_BIT data type to XIL_BYTE. Generally, the xil_cast() function casts the values 0 and 1 in the XIL_BIT image to indices 0 and 1 in the XIL_BYTE image. If you need different indices, convert the image by passing it through a lookup table.

After creating the lookup table, you must pass the image through it, as discussed in "Passing an Image Through the Table" on page 204.

Creating Lookup Tables for Multiband Input Images

To create a lookup table for a multiband input image, you call xil_lookup_create() once for each band in the input image, then combine these single lookup tables into a *combined* lookup table. To create the combined lookup table, you call the function xil_lookup_create_combined(), whose prototype is shown below:

```
XilLookup xil_lookup_create_combined(XilSystemState state,
    XilLookup lookup_list[], unsigned int num_lookups);
```

This function returns a handle to a data structure of type XilLookup, which is the combined lookup table. The parameter state is a handle to the system-state data structure that was created when you initialized the XIL library, lookup_list[] is an array of type XilLookup that stores the single lookup tables created for each of the input image's bands, and num_lookups indicates how many lookup tables are stored in the lookup_list[] array.

To create the combined lookup table:

1. **Create an array variable of data type** XilLookup**.**
   The variable should have as many array elements as there are bands in the input image so you can store each band's values in a separate element. Store the values for band 0 in element 0, those for band 1 in element 1, and so on.

2. **For each band in the input image, create a single lookup table by calling** `xil_lookup_create()`**, as discussed in "Creating Lookup Tables for Single-Band Input Images" on page 200.**
   Each single lookup table defines values for only one band in the destination image; thus, you must pass a `1` for the `xil_lookup_create()` function's `output_nbands` argument. The single lookup tables must all have the same data type, but each can use a different offset.

3. **Combine the single tables by calling the** `xil_lookup_create_combined()` **function, passing it the array variable you created in Step 1.**

4. **Pass the input image through the combined lookup table, as discussed in the next section.**

The following code fragment shows how you might alter a 3-band XIL_BYTE input image whose green band is accented but whose red and blue bands are subdued:

```
XilImage image;
XilLookup lookup_tables[3]; /* var to store 3 lookup components
*/
XilLookup combined_lookup_table;
Xil_unsigned8 red[256];     /* red component of lookup */
Xil_unsigned8 green[256];   /* green component of lookup */
Xil_unsigned8 blue[256];    /* blue component of lookup */
int i;

for(i=0; i<256; i++) {
   green[i] = (i + 20) < 255 ? i + 20 : 255;
    blue[i] = red[i] = (i - 10) < 0 ? 0 : i - 10;
}
/* create single lookup tables for each input band */
lookup_tables[0] = xil_lookup_create(state, XIL_BYTE, XIL_BYTE,
                                     1, 256, 0, red);
lookup_tables[1] = xil_lookup_create(state, XIL_BYTE, XIL_BYTE,
                                     1, 256, 0, green);
lookup_tables[2] = xil_lookup_create(state, XIL_BYTE, XIL_BYTE,
                                     1, 256, 0, blue);
/*combine the tables and pass input image through combined
table*/
combined_lookup_table = xil_lookup_create_combined(state,
                                                   lookup_tables, 3);
xil_lookup(image, image, combined_lookup_table);
```

Passing an Image Through the Table

Once you've created a lookup table, you call xil_lookup() to perform the actual lookup operation. The prototype for this function is shown below.

```
void xil_lookup(XilImage src, XilImage dst, XilLookup lookup);
```

The parameter src is a handle to your source image. The data type of this image must match the data type of the values on the input side of your lookup table. The parameter dst is a handle to your destination image. The data type of this image must match the data type of the values on the output side of the

table. For a single-band input image, the number of bands in the destination image must match the number of values per pixel on the output side of the table; for a multiband input image, the number of bands in the destination image must match the number of lookup tables that compose the combined lookup table. The final parameter, `lookup`, is the handle to the lookup table you received when you called `xil_lookup_create()` for a single-band input image, or `xil_lookup_create_combined()` for a multiband input image.

Destroying a Lookup Table

After performing your lookup operation, you should destroy your lookup table if you won't be using it in a subsequent operation. Destroying the table frees the memory that was allocated to store it. For multiband input images, be sure to destroy the tables created for each input band, as well as the combined table.

You destroy a lookup table by calling the function `xil_lookup_destroy()`, whose prototype is shown below.

```
void xil_lookup_destroy(XilLookup lookup);
```

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

11

Additional Lookup-Table Functions

The preceding sections have discussed the most frequently used XIL functions that affect lookup tables. However, the library also contains the lookup-table-related functions shown in Table 11-8. These functions enable you to make a copy of a lookup table, read the values of lookup-table attributes, assign a name to a lookup table, and so on.

Note – Some of the functions cannot be used on a combined lookup table. However, you can extract a band from a combined lookup table and perform those functions on the extracted copy. You can then use the altered table for creating another combined lookup table.

Table 11-8 Additional Functions for Lookup Tables

Function Name	What the Function Does
`xil_lookup_create_copy`	Creates and returns a copy of an existing lookup table (LUT).
`xil_lookup_get_input_datatype`	Gets the data type of an LUT's input.
`xil_lookup_get_output_datatype`	Gets the data type of an LUT's output.
`xil_lookup_get_input_nbands`	Gets the number of bands in the LUT's input.
`xil_lookup_get_output_nbands`	Gets the number of bands defined for the LUT's output.
`xil_lookup_get_num_entries`	Gets the number of entries in the LUT. Cannot be used on a combined LUT.
`xil_lookup_get_offset`	Gets the input value for the first entry in the LUT. Cannot be used on a combined LUT.
`xil_lookup_set_offset`	Sets the input value for the first entry in the LUT. Cannot be used on a combined LUT.
`xil_lookup_get_band_lookup`	Gets a particular lookup table out of a combined LUT.
`xil_lookup_get_values`	Retrieves the values in an LUT. Cannot be used on a combined LUT.

Table 11-8 Additional Functions for Lookup Tables

Function Name	What the Function Does
`xil_lookup_set_values`	Sets the values in an LUT. Cannot be used on a combined LUT
`xil_lookup_get_version`	Gets a unique identifier associated with an LUT.
`xil_lookup_convert`	Creates an LUT that includes input data from one lookup table and output data from another. Cannot be used on a combined LUT.
`xil_squeeze_range`	Creates an LUT that will map a single-band image to a single-band image with contiguous values. Cannot be used on a combined LUT.
`xil_lookup_get_name`	Reads the name of an LUT.
`xil_lookup_set_name`	Assigns a name to an LUT.
`xil_lookup_get_by_name`	Gets a handle to an LUT by specifying the name of the table.

Linear Combination

The function `xil_band_combine()` calculates each value in a destination image by performing a linear combination (matrix multiply) of the values of all the bands of a pixel in the source image. The nonimage values used in this calculation are stored in a matrix. For example, assume that `xil_band_combine()` is operating on the three-band single-pixel image shown in Figure 11-10 using the matrix shown in the figure and is writing its output to a one-band single-pixel image.

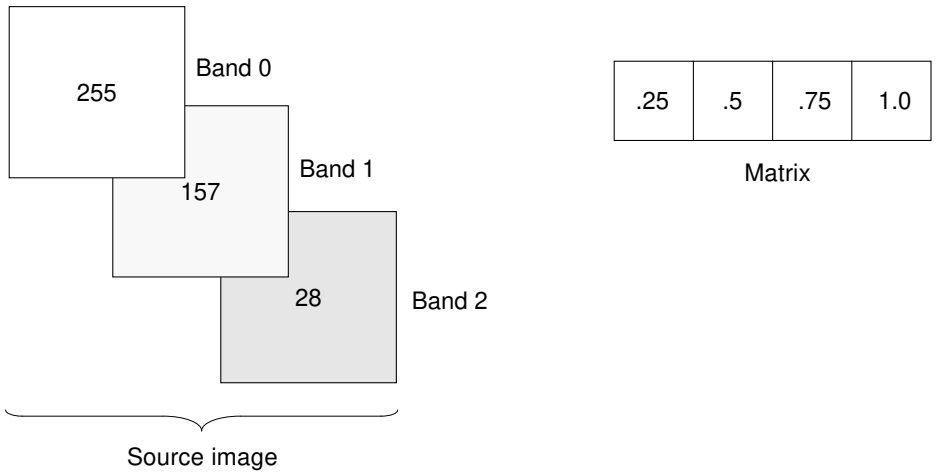

Figure 11-10 Interband Linear Combination

The equation used to calculate the value of the destination-image pixel would be

dst = (255 * .25) + (157 * .5) + (28 * .75) + 1.0

As you can see, the number of columns in the matrix is equal to the number of bands in the source image plus one. This number of columns provides a multiplier for each band in the source image plus a constant that is to be added to the sum of the products obtained using the multipliers and the values in the source image. The number of rows in the matrix must equal the number of bands in the destination image. If the destination image had three bands, the values in the second row of the matrix would be used in calculating the values in the second band of the destination, and the values in the third row would be used in calculating the values in the third band.

Thus, the matrix shown in Figure 11-11 would operate on a three-band source image and would produce a three-band image identical to the source.

1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0

Figure 11-11 Linear Combination Matrix

Performing a Linear Combination

Performing a linear combination is a two-step process. You first create an XIL kernel (a object of type `XilKernel`) in which to store a matrix. Then, you call `xil_band_combine()` to perform the operation.

To create the kernel that holds the matrix, you call the function `xil_kernel_create()`. For instance, to create a kernel that describes the matrix shown in Figure 11-11, you could use the code shown below.

```
XilSystemState state;
XilKernel matrix;
unsigned int width = 4, height = 3, keyx = 0, keyy = 0;
float data[] = {1.0, 0.0, 0.0, 0.0,
                0.0, 1.0, 0.0, 0.0,
                0.0, 0.0, 1.0, 0.0};

matrix = xil_kernel_create(state, width, height, keyx, keyy,
    data);
```

The values of `keyx` and `keyy`, which define the key value of the kernel, are irrelevant if the kernel will be used in a linear-combination operation because `xil_band_combine()` ignores these values. For more information about key values, see the section "Filtering an Image" on page 187.

Note – When you've finished using this kernel, you should destroy it using the function `xil_kernel_destroy()`.

Once you've created this kernel, you can call `xil_band_combine()`, whose function prototype is shown below.

```
void xil_band_combine(XilImage src, XilImage dst, XilKernel
    matrix);
```

The parameter `src` is a handle to the source image, `dst` is a handle to the destination image, and `matrix` is a handle to the XIL kernel created earlier.

How to Use Linear Combinations

The function `xil_band_combine()` is a very general-purpose function, but let's look at a couple of ways in which it could be used.

One possibility is to use the function to convert an image from one color space to another. For example, you could use the matrix shown in Figure 11-12 to convert a three-band `XIL_BYTE` image from the RGB to the CMY color space.

0.0	0.0	-1.0	255.0
0.0	-1.0	0.0	255.0
-1.0	0.0	0.0	255.0

Figure 11-12 RGB-to-CMY Conversion Using `xil_band_combine()`

Similarly, you could use the matrix shown in Figure 11-13 to convert a three-band `XIL_BYTE` RGB image to a one-band Y image.

0.114	0.587	0.299	0.0

Figure 11-13 RGB-to-Y Conversion Using `xil_band_combine()`

Note, however, that in most cases using `xil_band_combine()` is not the best way of performing color-space conversions using the XIL library. See the section "Color-Space Conversion" on page 122 for information about the preferred method of performing these conversions.

Another possibility would be to use the matrix shown in Figure 11-14 to convert a three-band source image to a single-band destination of the same data type.

0.333	0.333	0.333	0.0

Figure 11-14 Calculating the Normalized Sum of an Image

Each pixel in the destination would equal the normalized sum of the three values of the corresponding pixel in the source.

Blending Images

The XIL library's blending function, `xil_blend()`, blends two images using an alpha mask. The function's prototype is shown below.

```
void xil_blend(XilImage src1, XilImage src2, XilImage dst,
    XilImage alpha);
```

The parameters `src1` and `src2` are handles to the two images to be blended, and `dst` is a handle to the destination image. The last parameter, `alpha`, is a handle to an XIL image that serves as an alpha mask for the blending operation. This alpha image must be a single-band image.

The equation below shows how `xil_blend()` uses the values in the mask as it calculates the values of pixels in the destination image:

dst = (((1 - normalized-alpha) * src1) + (normalized-alpha * src2))

Note – A normalized value is one that has been scaled into the range 0 to 1, where 0 corresponds to the minimum value for the image's data type and 1 corresponds to the maximum value. For example, in an `XIL_BYTE` image, a value of 127 would be normalized to approximately .5 since it falls about half way between 0 and 255.

Figure 11-15 illustrates the blending of two `XIL_BYTE` images, labeled `src1` and `src2`. The pixels in the dark areas of these images have values of 100, and the white pixels have values of 255. The alpha mask, `alpha`, has all its pixels set to 127, so normalized `alpha` is approximately .5. Thus, the equation used to calculate destination-image values in this example is

dst = ((.5 * src1) + (.5 * src2))

There will only be three distinct values in the destination:

- Where `src1` and `src2` were both white, the destination value is calculated as follows:

  (.5 * 255) + (.5 * 255) = 255

  These pixels appear white in the destination.

- Where `src1` and `src2` were both dark (pixel value of 100), the destination value is calculated as follows:

  (.5 * 100) + (.5 * 100) = 100

  These pixels appear dark gray in the destination.

- Where `src1` was white and `src2` was dark, or vice versa, the destination value is calculated as follows:

  (.5 * 255) + (.5 * 100) = 178

  These pixels appear light gray in the destination.

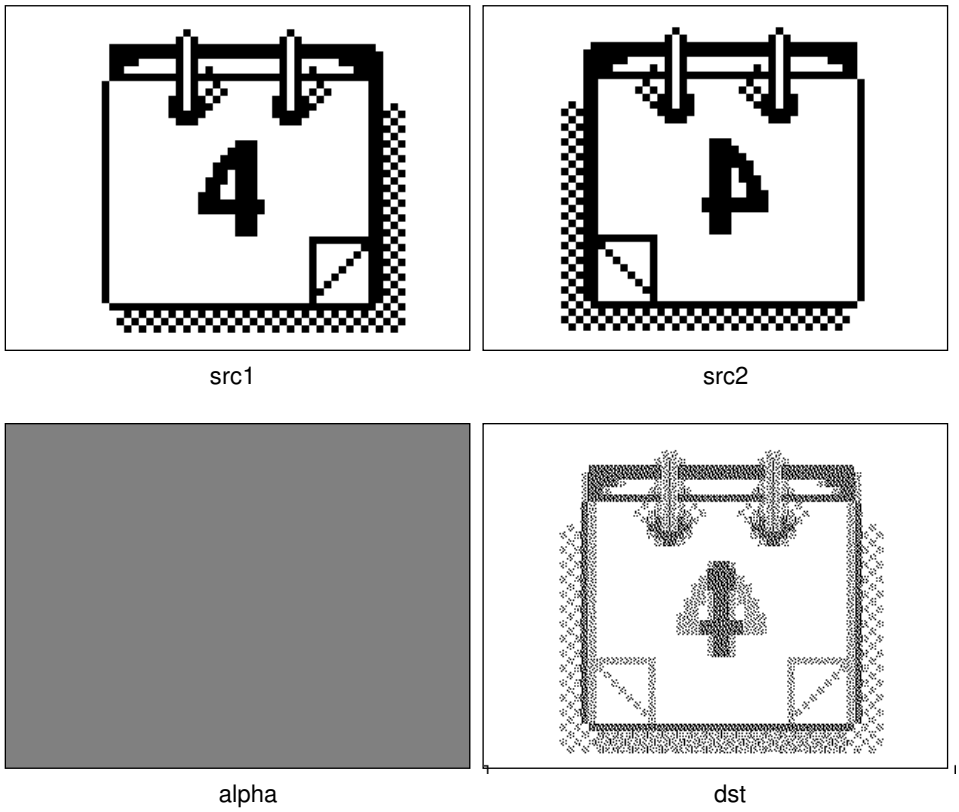

Figure 11-15 Blending Images

To see an example of the blending of two color images, see Color Plate 3.

Painting on an Image

The library's paint operation requires you to provide a list of the pixels in the source image you want to paint by specifying their coordinates. You also specify a kernel that the operation will use as a brush. This kernel is like the kernel you use with `xil_convolve()`, except that all the values in the kernel must fall in the range 0.0 to 1.0.

Note – For information on creating an XIL kernel, see the section "Creating a Convolution Kernel" on page 188.

Finally, you specify a color that you want to use for your painting.

In one sense, the paint function uses the kernel you supply much as the convolution operation does. For each pixel that you want to paint, the paint function lays the key value of the kernel over the pixel. See Figure 11-16.

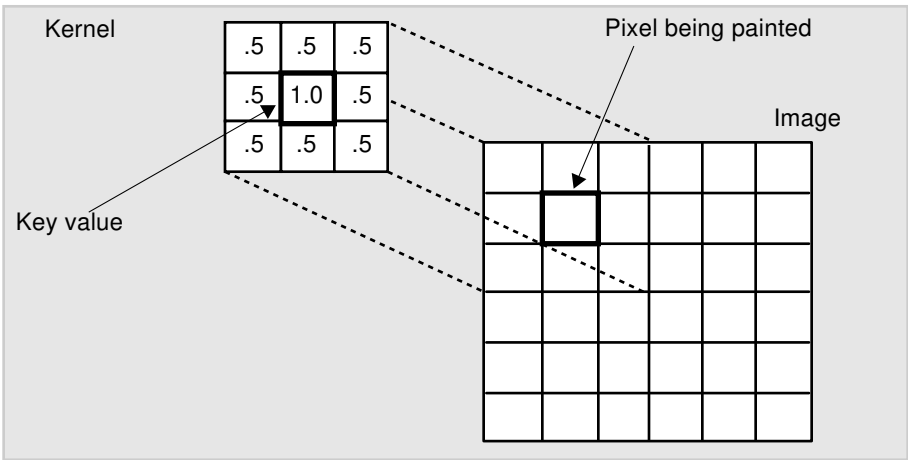

Figure 11-16 Painting on an Image

In another sense, however, the paint function uses the kernel differently. In a paint operation, a neighborhood of pixels in the source does not contribute to the value of a singe pixel in the destination. Instead, for each placement of the kernel, each source-image pixel that lies under the kernel affects the value of the corresponding pixel in the destination. This part of the painting operation is similar to a blending operation. The pixels under the kernel are blended with the paint color, and the kernel serves as the equivalent of an alpha mask. Destination-image pixel values are determined by the following formula:

```
dst = (brush-value * color) + ((1.0 - brush-value) * src)
```

You perform a paint operation in an XIL program by calling `xil_paint()`, whose prototype is shown below.

```
void xil_paint(XilImage src, XilImage dst, float *color,
    XilKernel brush, unsigned int count, float *coord_list);
```

Table 11-9 explains the purpose of each of the parameters shown above.

Table 11-9 Parameters to `xil_paint()`

Parameter	What It Represents
`src`	A handle to the source image for the operation.
`dst`	A handle to the destination image. This image must have the same number of bands and the same data type as the source image.
`color`	An array of `floats` containing a number of elements equal to the number of bands in the source and destination images. The elements in this array define the color to be used for painting.
`brush`	A handle to the kernel that will be used as the brush.
`count`	The number of pixels to be painted.
`coord_list`	An array of `floats` containing `2 * count` elements. These elements represent the *x* and *y* coordinates of the pixels to be painted.

Setting and Getting the Values of Pixels in an Image

The XIL library contains three functions that set or retrieve the values of pixels in an image. Two of these functions—`xil_set_pixel()` and `xil_get_pixel()`—set or get the value of individual pixels. The third, `xil_set_value()`, sets all the pixels in an image.

xil_set_pixel() and xil_get_pixel()

The prototypes for these two functions are shown below.

```
void xil_set_pixel(XilImage image, unsigned int x,
    unsigned int y, float *values);

void xil_get_pixel(XilImage image, unsigned int x,
    unsigned int y, float *values);
```

In both prototypes, the parameter `image` is a handle to the image you're working with. The parameters `x` and `y` are the *x* and *y* coordinates of the pixel you want to read or write. These coordinates should describe the location of the pixel of interest with respect to the image's origin, which by default is in the upper-left corner of the image (0,0). The final parameter, `values`, is an array of floating-point numbers. The number of elements in this array must match the number of bands in the image.

When you read a pixel, the value of band 0 of that pixel is written to `values[0]`, the value of band 1 is written to `values[1]`, and so on. Similarly, when you write a pixel, `values[0]` is written to band 0 of the pixel, and so forth. Before the writing actually takes place, the floating-point numbers in `values` are converted to integers.

Note – If you attempt to write to a pixel a value that is out of range for the source image's data type, the value will be clamped to the low or high limit for the data type. For example, an `XIL_BYTE` image can only accommodate values in the range 0 to 255. If you try to write a 400 to such an image, the function will actually write a 255.

---

xil_set_value()

The function `xil_set_value()` is useful for clearing an image by setting all the pixels in the image to the same color. The prototype for this function is shown below.

```
void xil_set_value(XilImage dst, float *values);
```

The parameter `dst` is a handle to the image you are working with, and `values` is an array of floating-point numbers. The number of elements in this array must match the number of bands in the image `dst`. The first element in the array is rounded to an integer and is then written to all of band 0 of your image; the second element is rounded to an integer and written to band 1 of the image; and so on.

Note – If you attempt to write to a band a value that is out of range for the image's data type, the value will be clamped to the low or high limit for the data type. For example, an `XIL_BYTE` image can only accommodate values in the range 0 to 255. If you try to write a 400 to a band, the function will actually write a 255.

Copying a Pattern to an Image

Besides `xil_copy()`, which copies a source image to a destination image, the XIL library includes a function, `xil_copy_pattern()`, that writes as many copies of the source image as possible to the destination image. The function prototype for `xil_copy_pattern()` is shown below.

```
void xil_copy_pattern(XilImage src, XilImage dst);
```

The parameter `src` is a handle to the source image, and `dst` is a handle to the destination.

If your destination image is the same size as, or smaller than, your source image, using `xil_copy()` and `xil_copy_pattern()` will produce the same result. However, if your destination image is larger than your source, `xil_copy()` will write one copy of the source image to the destination, while `xil_copy_pattern()` will write as many copies as possible. Some of these copies will be partial copies if the dimensions of the destination are not multiples of the dimensions of the source. Figure 11-17 below illustrates the difference between using `xil_copy()` and `xil_copy_pattern()` to copy a source image to a destination that is twice as high and wide as the source.

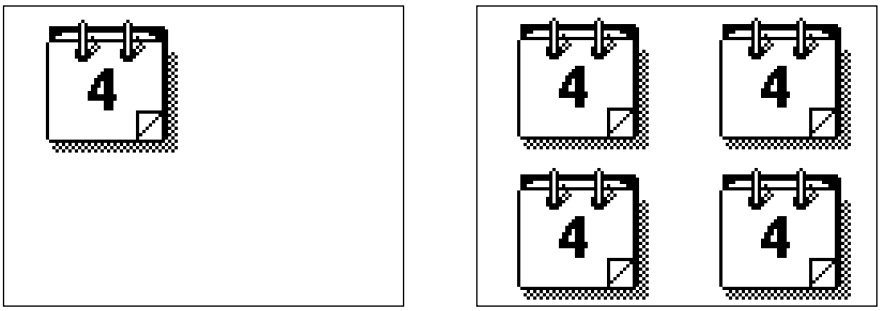

Figure 11-17 Replicating a Source Image