ImageVision Library Programming Guide

Document Number 007-1387-030

CONTRIBUTORS

Written by Jackie Neider and Eleanor Bassler
Illustrated by Seth Katz, Nancy Cam, Bill Pickering, and Eleanor Bassler
Edited by Nan Schweiger
Production by Laura Cooper
Engineering contributions by Chris Walker, Nancy Cam, Venkatesh Narayanan, Dan Baca, Jon Brandt, Don Hatch, and Casey Leedom
Photography by Jackie Neider, Jim Winget, Nancy Cam, and Judith Quenvold

ImageVision Library Programming Guide
Document Number 007-1387-030

Contents

Examples

Figures

Tables

About This Guide

The ImageVision Library™ (IL) is an object-oriented, extensible toolkit designed for developers of image processing applications. Typical image processing programs access existing image data, manipulate it, and display and save the processed results. The IL provides a robust framework within which developers can easily create such programs to run on all Silicon Graphics® workstations.

The IL consists of a library written in the C++ programming language; interfaces for the C and Fortran languages are also available. The object-oriented nature of C++ provides a simplified programming model based on abstractions of what images are and how they're manipulated. This model relieves developers of many tedious programming details and allows them to conceptually design creative programming solutions. Also, because the IL is written in C++, developers can easily extend it, for example, to incorporate their own image processing algorithms or to include support for their own image file formats. Several examples of images produced using the IL appear in Chapter 4, "Operating on an Image," of this programming guide.

What This Guide Contains

This guide presents a task-oriented perspective of the IL. The topics in this guide are arranged to coincide with the order in which you need to refer to them while writing an image processing program. To illustrate the use of the IL, code examples are sprinkled liberally throughout the guide. Additional sample source code is provided online; see "Online Source Code" on page 352. Brief descriptions of the chapters in this guide follow:

- Chapter 1, "Writing an ImageVision Library Program," shows what a typical image processing application that uses the IL looks like. It presents an IL program that performs the tasks common to many image processing applications. It also summarizes the differences among the C++, C, and Fortran interfaces to the IL.
- Chapter 2, "The ImageVision Library Foundation," explains the general architecture and design philosophy of the IL. Most of this chapter is devoted to discussion of the principal image class (ilImage), from which virtually all IL classes derive, and the class that implements a key part of the IL's execution model (ilCacheImg).
- Chapter 3, "Accessing External Image Data," describes how to read and write image data from and to either a file on disk or memory.
- Chapter 4, "Operating on an Image," discusses the more than 70 image processing algorithms provided with the IL. It explains how to use them and what effect they have on image data.
- Chapter 5, "Displaying an Image," describes how to display and manage a set of images on the screen in an interactive program. You can allow a user of your program to move images, perform wipes, roam around an image, and create split views of multiple images.
- Chapter 6, "Extending the IL," explains how to extend the capabilities of the IL to implement your own derived classes. You might extend the IL to include support for your own file format or to incorporate your own image processing algorithm.
- Chapter 7, "Optimizing Your Application," provides information on optimizing your IL programs by reducing memory usage, taking advantage of hardware acceleration, and making use of the IL's multi-threading facility.
- Chapter 8, "The Programming Environment," provides information on the programming environment available on Silicon Graphics

workstations. It mentions special tools that may help you in writing, compiling, and debugging your IL program.

In addition to these chapters, this guide includes several appendices as handy summaries of useful information:

- Appendix A, “Introduction to C++,” contains a brief introduction to the principles of C++ programming.
- Appendix B, “Summary of All Classes,” provides a brief summary of all the classes that make up the IL.
- Appendix C, “Data Types, Data Orderings, and Color Models,” provides a summary of the default data type, data ordering, and color model attributes of IL images.
- Appendix D, “Results of Operators,” contains illustrations showing the results of using the IL’s operators to process data.
- Appendix E, “Auxiliary Classes, Functions, and Definitions,” describes IL classes not fully discussed elsewhere in this guide. It also lists all the error codes and enumerated types used by the IL.

Other documentation on the IL is contained in the *ImageVision Library Reference Pages*. These reference pages provide concise yet thorough descriptions of each C++ class included in the IL. They’re only available online in versions for C++, C, and Fortran programmers. See “Reading the Reference Pages” on page 348 for more information on the exact content of the reference pages.

Suggestions for Further Reading

Because the IL is written in C++, it’s easiest to describe its design philosophy and how to program with it by talking about the C++ classes that compose the IL. While it’s not necessary that you know how to program in C++, you will gain more from this guide if you understand the concepts of object-oriented programming. Where possible, however, this guide avoids focusing on topics directly related to the C++ implementation of the IL. In addition, a brief introduction to C++ is included in Appendix A. Programming examples in Chapter 1, “Writing an ImageVision Library

Program," are given in C++, C, and Fortran. Some books on C++ you might find helpful include:

- Ellis, Margaret, and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. AT&T Bell Laboratories, 1990. The official C++ language reference manual.
- *The C++ Programmer's Guide*. A short manual that provides information about implementing C++ programs on Silicon Graphics workstations.
- Lippman, Stanley. *C++ Primer*. AT&T Bell Laboratories, 1991. An introductory-level, tutorial-style presentation of C++.

This guide assumes that you're familiar with the principles of image processing. A good, general discussion of image processing can be found in any of several textbooks, such as:

- Jain, Anil K. *Fundamentals of Digital Image Processing*. Prentice-Hall, Inc., 1989. A thorough presentation of the major concepts of image processing, written for graduate students.
- Pratt, William K. *Digital Image Processing*. John Wiley & Sons, 1991.
- Gonzalez, Rafael C., and Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992.

To learn more about the RealityEngine™ architecture, read:

- Akeley, Kurt, and Tom Jermoluk. RealityEngine Graphics™. In *Proceedings of SIGGRAPH '93* (August 1993), pp. 109-116.

Most sample programs in this guide include calls to the IRIS Graphics Library™ (GL), and the IL itself uses the GL to perform rendering in the frame buffer. These calls are not explained in much detail since the GL is documented separately in these Silicon Graphics books:

- Graphics Library Programming Guide
- Graphics Library Reference Pages
- Graphics Library Programming Tools and Techniques

The IL provides support for manipulating files stored in the format defined by Tag Image File Format (TIFF), Revision 6.0, distributed by Aldus Corp. You might want to obtain the official specification of this format *directly from Aldus (*411 First Avenue South; Suite 200; Seattle, WA 98104; (206) 628-6593).

- TIFF 6.0 Specification

The IL provides support for multi-threading on single- and multi-processor machines. If you want to know more about writing multi-threaded applications, refer to this document:

- Parallel Programming on Silicon Graphics Computer Systems

The IL uses dynamic linking. To learn more about using dynamic linking with your applications, read:

- the dlopen, dlsym, and dlerror reference pages
- IRIX™ Programming Guide

Adding a User Interface to Your ImageVision Library Program

The IL doesn't impose any particular user interface (UI), so you can use any UI toolkit—such as IRIS IM™, Silicon Graphic's port of the industry-standard OSF Motif™—to allow the user to control your program. To support such interactive control, the IL provides many functions for altering parameters dynamically. The IL also keeps track of when parameters have changed so that image data can be updated automatically. These user-interface manuals are available from Silicon Graphics:

- OSF/Motif Programmer's Guide
- OSF/Motif Programmer's Reference
- OSF/Motif Style Guide
- IRIS IM Programming Notes

Silicon Graphics recommends that you write mixed-model programs rather than pure GL programs. A mixed-model program is essentially an X program that uses the GL to handle graphics; the GL is completely removed from all areas governed by the X server. If you're creating a mixed-model X Window System™ and IL program, you might also want to refer to these volumes in the O’Reilly X Window System Series, published by O’Reilly & Associates, Inc., Sebastopol, California:

- Volume One: *XLIB Programming Manual*, by Adrian Nye
- Volume Four: *X Toolkit Intrinsics Programming Manual*, by Adrian Nye and Tim O’Reilly

Volumes One and Four are available from Silicon Graphics as part of the IRIS Development Option (IDO).

Style Conventions

These style conventions are used in this guide:

- **Bold**—Functions, data members, and data types
- *Italics*—Variables, filenames, spatial dimensions, and command
- Regular—Class names and enumerated types

Code examples are set off from the text in a fixed-space font.

Chapter 1

Writing an ImageVision Library Program

This chapter describes the tasks typically performed by an application using the ImageVision Library. It illustrates the implementation of these tasks with sample programs in C++, C, and Fortran.

Writing an ImageVision Library Program

The C++ classes that make up the ImageVision Library are designed to be used together in an image processing program. Programs that use these classes usually assume the program structure defined by the IL. This chapter shows what a typical image processing application that uses the IL looks like. Chapter 2, “The ImageVision Library Foundation,” discusses the basic concepts encapsulated in the IL’s principal classes.

This chapter contains the following major sections:

- “A Sample Program in C++” on page 4 presents a sample program written in C++ that uses the IL. The section shows the program in two versions, one that uses X window management (mixed-model) and one that uses GL window management (pure GL).
- “The C and Fortran Interfaces” on page 15 explains the differences between the C++ and C and Fortran interfaces to the IL.
- “A Sample Program in C” on page 19 presents the sample program written in C. This section contains two versions of the program, one that uses X window management and one that uses GL window management.
- “A Sample Program in Fortran” on page 24 presents the sample program written in Fortran.

A Sample Program in C++

The sample C++ program presented in this section reads image data from a file, processes it, displays it, and saves the processed data in a new file. Each task the program performs is described in more detail in subsequent chapters. This chapter gives you a brief introduction to the capabilities of the IL and provides you with a code example that can serve as a template for programs you write.

Image processing applications typically perform at least some of the following tasks:

- Read image data—Read formatted image data from a file on disk, for example, and decompress it if necessary.
- Process the data—Manipulate the data, for example to enhance the original image or to produce a statistical analysis of the data.
- Display the image on the screen—Allow a user to interactively view selected portions of simultaneously displayed images.
- Save the processed data in a file—Format and possibly compress the data.

The C++ program outlined and listed below demonstrates how the IL accomplishes these tasks. (Versions of this program in the C and Fortran languages appear later in this chapter.) In this particular example, the user invokes the program from the command line and specifies a file of image data to be processed. The program first sharpens the image data (like improving the focus through a camera lens) and then rotates it 90 degrees. Next it displays the processed image on the screen and writes it to a file.

The sample program performs these tasks:

1. Opens the input file of image data.
2. Constructs a sharpening operator that uses the file of image data as input.
3. Constructs a rotate operator that uses the output of the sharpening operator as input.
4. Displays the sharpened and rotated image data on the screen.

5. Copies the sharpened and rotated image to a file on disk.
6. Continues to display the processed image until the user quits by pressing the **<Esc>** key or by using the window menu.

The code for this program (in both mixed-model and pure GL) is available online so that you can easily compile and run it. Look in:

/usr/people/4Dgifts/examples/ImageVision/ilguide

Other sample code is also available online; see "Online Source Code" on page 352.

Mixed-Model Version of the Sample Program

The code in Example 1-1 shows the mixed-model version of the sample program. Silicon Graphics recommends that you write mixed-model programs rather than pure GL programs.

Example 1-1 Sample Program (in C++) Using X Window Management

```
/* sampleProgX: X window (mixed-model) version
 * Opens a file image and then sharpens and rotates it. Sets
 * up the window configuration, opens an X window, and
 * displays the processed image.
*/
#include <stdlib>
#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/keysym.h>
#include <il/ilGLXConfig.h>
#include <il/ilGenericImgFile.h>
#include <il/ilSharpenImg.h>
#include <il/ilRotZoomImg.h>
#include <il/ilDisplay.h>

void main (int argc, char **argv)
{
  if (argc < 2) {
    printf("usage: sampleProgX <filename>\n");
    exit(0);

  }
```

Step 1: Open the file of image data.

```
ilFileImg *inImg = ilOpenImgFile (argv[1], "r");
```

Step 2: Create IL objects for sharpening and rotating.

```
ilSharpenImg sharperImg(inImg, 0.5);

// If page width equals image width Then adjust cache size
// to hold pages needed for ilRotZoomImg
int px, py, pz, pc;
inImg->getPageSize(px, py, pz, pc);
if (px == inImg->getXsize()) ;
  sharperImg.setCacheWindow(inImg->getXsize(),
                inImg->getYsize());
ilRotZoomImg rotatedImg(&sharperImg, 90.0);
```

Step 3: Configure and open a window for display.

```
int rgbMode = getgdesc(GD_BITS_NORM_SNG_RED) > 0;
int doubleBuffer = getgdesc(GD_BITS_NORM_DBL_RED) >= 8;

ilGLXConfig glx;
glx.addEntry(GLX_NORMAL, GLX_RGB, rgbMode);
glx.addEntry(GLX_NORMAL, GLX_DOUBLE, doubleBuffer);
glx.addEntry(0, 0, 0); // terminator

ilSize size;
rotatedImg.getSize(size);
Display* dpy = XOpenDisplay(NULL);
Window win = glx.createWindow(dpy,
                RootWindow(dpy, DefaultScreen(dpy)),
                0, 0, size.x, size.y, 0);
XStoreName(dpy, win, "Processed Image"); //set window title
XSelectInput(dpy, win, ExposureMask | KeyPressMask);
XMapWindow(dpy, win);

// If SGI format Then adjust cache size
if (inImg->getImageFormat() == ilSGI_IMG)
  inImg->setCacheWindow(size.x, size.y);
```

Step 4: Display the processed data.

```
ilDisplay disp(dpy, win);
disp.addView(&rotatedImg, ilLast);
```

Step 5: Write the processed data to a file.

```
  ilFileImg *tmpFile = ilCreateImgFile("outFile.tif", size,
                  inImg->getDataType(),
                  inImg->getOrder(), ilTIFF_IMG);
  tmpFile->copyTile(0,0,size.x,size.y,&rotatedImg,
                  0,0,0,TRUE);
  delete tmpFile;     //flush
```

Step 6: Display until the user quits.

```
  XEvent event;
  int ever = 1;
  for (;ever;) {
    XNextEvent(dpy, &event);
    switch (event.type) {

      case DestroyNotify:
        ever = 0;
        break;

      case KeyPress:
        switch(XLookupKeysym(&event.xkey, 0)) {
        case XK_Escape: // Escape key
          ever = 0;
          break;
       }
        break;

      case Expose:
        disp.redraw();
        break;
    }

  }
  XCloseDisplay(dpy);
  exit(0);
}
```

Pure GL Version of the Sample Program

The code in Example 1-2 shows a pure GL implementation of the sample program.

Example 1-2 Sample Program (in C++) Using GL Window Management

```
/* sampleProgGL: Pure GL version
 * Opens a file image, then sharpens and rotates it. Sets up
 * the window configuration, opens a GL window, and displays
 * the processed image.
 */

#include <stdio.h>
#include <stdlib.h>
#include <il/ilGenericImgFile.h>
#include <il/ilSharpenImg.h>
#include <il/ilRotZoomImg.h>
#include <il/ilDisplay.h>
#include <gl/gl.h>
#include <gl/device.h>

void main (int argc, char **argv)
{
 if (argc < 2) {
    printf("usage: sampleProgGL <filename>\n");
    exit(0);
 }
```

Step 1: Open the file of image data.

```
  ilFileImg *inImg = ilOpenImgFile (argv[1], "r");
```

Step 2: Create IL objects for sharpening and rotating.

```
  ilSharpenImg sharperImg(inImg, 0.5);

// If page width equals image width Then adjust cache size
 // to hold pages needed for ilRotZoomImg
 int px, py, pz, pc;
 inImg->getPageSize(px, py, pz, pc);
 if (px == inImg->getXsize()) ;
   sharperImg.setCacheWindow(inImg->getXsize(),
               inImg->getYsize());
   ilRotZoomImg rotatedImg(&sharperImg, 90.0);
```

Step 3: Configure and open a window for display.

```
  ilSize size;
  rotatedImg.getSize(size);
  prefsize(size.x, size.y);
  long wid = winopen("Processed Image");
  if (getgdesc(GD_BITS_NORM_SNG_RED) !=0) RGBmode();
  gconfig();
```

Step 4: Display the processed data.

```
  ilDisplay disp(wid);
  disp.addView(&rotatedImg);
  disp.redraw();
```

Step 5: Write the processed data to a file.

```
  ilFileImg *tmpFile = ilCreateImgFile("outFile.tif", size,
         inImg->getDataType(), inImg->getOrder(),
         ilTIFF_IMG);
  tmpFile->copyTile(0,0,size.x,size.y,&rotatedImg,
         0,0,0,TRUE);
  delete tmpFile;       //flush
```

Step 6: Display until the user quits.

```
  qdevice(REDRAW);
  qdevice(ESCKEY);
  qenter(REDRAW, short(wid));
  short val;

  int active = 1;
  while (active) {
    switch (qread (&val)) {
    case ESCKEY:
      active = 0;
      break;
    case REDRAW:
      disp.redraw();
      break;
    }
  }
  exit(0);
}
```

More about the Sample Program

Both the mixed-model and pure GL versions of the sample program use the IL in a recommended way, but many good programming habits were *not* followed in the interest of keeping the program short. More specifically, this program does *not* do any of the following things, and you shouldn't regard it as recommended practice in at least these areas:

- check return arguments and write error messages as appropriate
- strip arguments off the command line in an elegant way and check them for appropriate values (or provide a graphical user interface)
- provide feedback to the user—for example, to indicate that a file of processed image data has been created

The remaining paragraphs in this section walk through the sample program, explaining how it uses the IL. This discussion is intended to give you a taste of the kinds of things the IL can do and what you as a programmer need to do to accomplish them. Each of the topics touched on is discussed extensively elsewhere in this guide.

Header Files

The first few lines of code include the necessary header files from the IL. These header files also include other IL header files, as well as header files from the Graphics Library and the standard C library. If you use this program as a template and modify it to suit your needs, be sure you include the header files necessary for your program. Since the IL provides many more capabilities than you'll need for any particular program, you don't need to include all of its header files. To minimize compile time and the size of your executable, you should include only those header files actually required by your program.

In this example,

- the header *il/ilGLXConfig.h* is included to configure an X window for GL rendering.
- the header *il/ilGenericImgFile.h* is included to implement the ilFileImg class.
- the header *il/ilSharpenImg.h* is included to implement the ilSharpenImg class.

- the header *il/ilRotZoomImg.h* is included to implement the ilRotZoomImg class.
- the header *il/ilDisplay.h* is included to manage views in an X or GL window.

In general, when writing an IL program in C++, you'll need to include an IL header file for each IL class you use. More information about programming and compiling IL programs is included in "Compiling and Linking an IL Program" on page 344.

Step 1: Open the File of Image Data

In step 1 of the **main()** function, an image data file specified by the user is opened by invoking the **ilOpenImgFile()** function. This function takes two arguments: the pathname of the file and a mode argument, which indicates whether the file is read-only or read-write. In this example, the filename is taken as an argument from the command line, and the file is opened for reading. An example image from a file is shown in Figure 1-1.

Before any image data can be read, **ilOpenImgFile()** must determine the file format used to store the data by returning a pointer to one of the supported ilFileImg types. The IL supports the following six file formats:

- an extended version of the Tag Image File Format (TIFF), Revision 6.0
- Silicon Graphics' format for storing image data, typically in files suffixed with *.bw* (black-and-white) or *.rgb* (red, green, blue—or RGB)
- Photo CD image pack file format to support images produced and stored in a Kodak Photo CD™
- a simple tiled file format called FIT, developed primarily as an example of extending the IL to include other file formats
- GIF (Graphics Interchange Format), Compuserve's image file format

Also, as suggested by the existence of FIT, you can easily add support to the IL for other file formats. Chapter 3, "Accessing External Image Data," presents detailed information about reading and writing image data in specific file formats. "Implementing Your Own File Format" on page 258 tells you how to extend the IL to support your own file format.

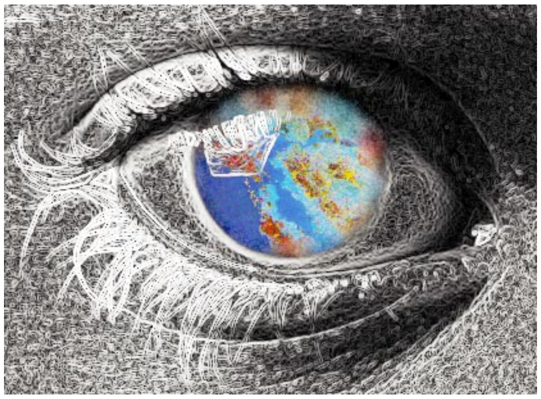


Figure 1-1 An Image before Processing

Step 2: Create IL Objects for Sharpening and Rotating

Now that the source of the image data to be processed is ready, the IL classes used for processing the data are created in step 2. For this sample program, data is first sharpened and then rotated by using the ilSharpenImg and ilRotZoomImg classes. These two classes are among the many operators provided by the IL that perform image processing algorithms; however, you can invoke any number of operators on a set of data. See Chapter 4, “Operating on an Image,” for more information about how the IL allows you to operate on image data. You can also easily add your own algorithms; “Implementing an Image Processing Operator” on page 273 tells you how to extend the IL to include a new image processing operator.

- Create Sharpening Object—As shown in the program example, the parameter 0.5 is passed along with a pointer to the input image data file. This parameter, which is a single-precision floating point number, can range in value from 0 to 1; it determines how much the data is sharpened. The specific algorithm that ilSharpenImg uses to sharpen image data is described in detail in its *class reference page* (read “Reading the Reference Pages” on page 348 for an explanation of the difference between normal reference pages and class reference pages). If this were an interactive program, you could allow the user to change the sharpness factor dynamically, perhaps with a slider; you would then reset the ilSharpenImg parameter to respond to the user’s input.

For better performance, the sample program checks to make sure the cached area in ilSharpenImg is set to the size of the image. This ensures that the program can store the entire sharpened image in cache.

- Create Rotate Object—You can use the ilRotZoomImg class to rotate and / or zoom (magnify or minify) an image. In this example, the sharpened image data is rotated 90 degrees, in a counterclockwise direction, as specified by the parameter passed to ilRotZoomImg. The ilRotZoomImg class is discussed in detail in its reference page.
- On-demand Processing—As an IL program executes, image data is processed only on demand—for example, when it's needed for displaying or writing to a file. This execution model eliminates unnecessary processing and minimizes transfers of data in and out of memory. In this sample program, data isn't actually processed until step 4. The execution model is discussed in detail in "The IL Execution Model" on page 68.

Step 3: Configure and Open a Window for Display

In the mixed-model version of the program (*sampleProgX*), calls are made to the X Window library to initialize and open a window on the screen. The **getgdesc()** function is used to avoid switching to RGB or double buffer mode on machines that don't support these modes. An IL object, ilGLXConfig, is used to make the job of configuring and opening an X Window easier. For more information on the ilGLXConfig object and creating mixed-model programs, see Chapter 5, "Displaying an Image." For information on X library calls, see the manuals listed in the Introduction.

In the pure GL version of the program (*sampleProgGL*), calls are made to the GL to initialize the GL and to open a window on the screen. The **getgdesc()** function is used to avoid switching to RGB mode on machines that don't support this mode. In this example, the window is the size of the processed image (measured in pixels) and is titled "Processed Image." For information on GL calls, see the manuals listed in the Introduction.

Step 4: Display the Processed Data

In step 4, an ilDisplay object is created to display the processed image data. This must be done after the window is opened. In a more interactive image processing program, you would use an ilDisplay object to manage the dynamic display of multiple images. Also, you could rewrite the program to display the sharpened image before it's rotated. This program, however, simply displays the final image. Displaying processed images is covered in detail in Chapter 5, "Displaying an Image." The result of running the sample program with the image from Figure 1-1 is shown in Figure 1-2.

In the IL's execution model, data is processed in conveniently sized chunks, called *pages*. As you execute this sample program, you can watch as successive pages of image data are displayed after they've been processed.

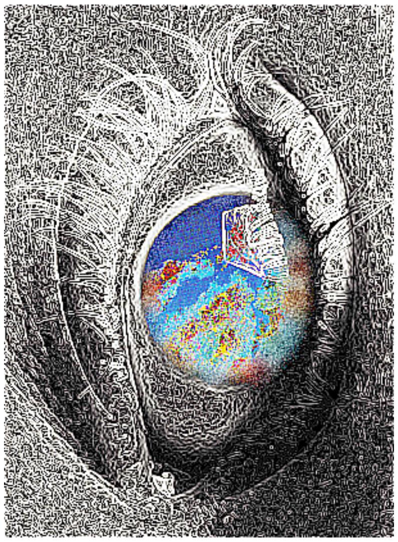


Figure 1-2 The Image after Processing

Step 5: Write the Processed Data to a File

Many image processing applications need to write processed image data to a file. In step 5, **ilCreateImgFile()** creates a file for writing data using the TIFF file format. This function needs to know the name of the file, the size of the image in pixels, the data type of the pixels (for example, **float** or **int**), and how those pixels are ordered. See "Creating a TIFF, SGI, or FIT File" on page 94 for more information about **ilCreateImgFile()**.

The **ilCreateImgFile()** function only creates a file. The **copyTile()** function actually writes the processed (sharpened and rotated) image data directly into the file.

Step 6: Display Until the User Quits

Finally, in step 6, the program sets up a loop using GL calls (in *sampleProgGL*) or X library calls (in *sampleProgX*) to display the image data until the user quits the program by pressing the **`<Esc>`** key (or by using the window menu). Also, the program redraws the image as necessary, for example, if the user moves, resizes, or pops the window on the screen.

The C and Fortran Interfaces

Since the IL was written in C++, it implements the C and Fortran interfaces as wrappers to C++ member functions. These wrappers have names that are similar to those of the C++ member functions. Thus, the concepts explained in this guide apply to C, Fortran, and C++ programmers even though most of the code examples are shown in C++. If you're programming in C or Fortran, feel free to refer to this section as necessary for information about how to translate C++ functions into C and Fortran function calls.

Creating and Deleting C++-style Objects

A C++ class object must be defined as something the C language recognizes to make it usable in a C program. For example, the header file *il/ilCdefs.h* defines all the IL classes as being of data type **struct**. To "create" such a **struct** in your program, call the appropriate function, which is of the form ***ClassNameCreate*()**. The call to create an ilDisplay **struct**, for example, is **ilDisplayCreate()**.

In C, use these statements:

```
ilDisplay* disp;
disp = ilDisplayCreate(dpy, win, ilGLRender, ilDefault);
```

In Fortran, use these statements:

```
integer*4 disp
disp = ilDisplayCreate(dpy, win, ilGLRender, ilDefault)
```

In C++, use these statements:

```
ilDisplay disp(dpy, win);                //automatic storage
ilDisplay* disp=new ilDisplay(dpy, win); //dynamic allocation
```

You can see in this example some other differences among the C, Fortran, and C++ calls. In C++, you can have variables created automatically for you, or you can allocate them dynamically yourself. The C variable *disp* must be declared as a pointer to type **ilDisplay**. The Fortran variable must be of type **integer*4**, which is indirectly a pointer. In all instances where C and C++ expect a pointer, Fortran will expect an **integer*4**. A **void*** becomes an array in Fortran. (The difference in the number of arguments given to these functions is explained in “Calling Functions” on page 17.)

Since you can create an ilDisplay for use with either an X or GL window, two functions are provided to create an ilDisplay **struct**. Both functions have the same name in C++, but must have distinct names in C and Fortran. The second function is called **ilDisplayGL()** in these two languages, since it creates an ilDisplay given only a GL window identifier.

In C, use these statements:

```
ilDisplay* disp;
disp = ilDisplayGL(wid);
```

In Fortran, use these statements:

```
integer*4 disp
disp = ilDisplayGL(wid)
```

In C++, use these statements:

```
ilDisplay disp(wid);                  //automatic storage
ilDisplay* disp = new ilDisplay(wid); //dynamic allocation
```

Since *disp* appears as just a **struct** to C, you will need to call a destructor directly when you need to delete it. The destructor naming scheme is similar to the creator scheme. In order to delete the display you created with the calls above, use **ilDisplayDelete()**.

In C, use this statement:

```
ilDisplayDelete(disp);
```

In Fortran, use this statement:

```
call ilDisplayDelete(disp)
```

In C++, use this statement:

```
delete ilDisplay; // not needed unless created with new
```

Calling Functions

Once you've accomplished the C or Fortran equivalent of creating an object, you can manipulate it with the C or Fortran versions of the functions associated with that object. The C and Fortran function names generally include the C++ class name, and the functions themselves take a pointer to the "object" as an additional argument.

In C, use this statement:

```
status = ilDisplayAddView(disp, rotatedImg, 0, ilCenter);
```

In Fortran, use this statement:

```
status = ilDisplayAddView(disp, rotatedImg, 0, ilCenter)
```

In C++, use this statement:

```
status = disp.addView(rotatedImg);
```

As you can see, the C++ function **addView()**, which is a member function of the ilDisplay class, becomes **ilDisplayAddView()**. Most functions will follow this form and prefix the name of the base class. C++ functions from the ilImage base class (or from ilImage's parent class, ilLink) will just add "il," not "ilImage." ilCacheImg's **flush()** function does this as well; it becomes **ilFlush()**, not **ilCacheImgFlush()**.

The C++ versions of the IL functions fill in default values for some arguments. If you omit those arguments, C++ simply calls the function with the defaults. C and Fortran, however, won't fill in defaults for you. You must supply values for each argument. The C++ sample program takes advantage of this feature when creating a new **ilSharpenImg** object.

In C, use this statement:

```
sharperImg = ilSharpenImgCreate(theImg, 0.5, 1.5, ilPadSrc);
```

In Fortran, use this statement:

```
sharperImg = ilSharpenImgCreate(theImg, 0.5, 1.5, ilPadSrc)
```

In C++, use this statement:

```
ilSharpenImg sharperImg(theImg);
```

0.5, 1.5, and ilPadSrc are the default values for the sharpness factor, radius, and edge mode arguments, respectively. In C and Fortran, you must pass them explicitly.

Including Header Files

To use the IL in your C programs, you need to include only *il/ilCdefs.h*. In your Fortran programs, include *il/ilFdefs.h*. These two header files each include information about all of the IL classes and functions.

A Sample Program in C

This section contains C code for the sample program, both mixed-model and pure GL versions.

Mixed-Model Program Written in C

The code in Example 1-3 shows the mixed-model version of the sample written in C.

Example 1-3 Sample Program (in C) Using X Window Management

```
/* sampleCProgX
*/
#include <il/ilCdefs.h>
#include <X11/keysym.h>
#include <stdlib.h>
#include <stdio.h>

void main (int argc, char **argv)
{
  ilFileImg           *inImg, *tmpFile;
  ilSharpenImg        *sharperImg;
  ilRotZoomImg        *rotatedImg;
  ilDisplay           *disp;
  ilSize              size;
  ilGLXConfig         glx;
  Display             *dpy;
  Window              win;
  XEvent              event;
  long                dev, wid;
  short               val, ever;
  int                 rgbMode, doubleBuffer;
  int                 px, py, pz, pc;

  if (argc < 2) {
    printf("usage: sampleCProgX <filename>\n");
    exit(0);
  }
```

Step 1: Open the file of image data.

```
  inImg = ilOpenImgFile(argv[1], "r");
```

Step 2: Create IL objects for sharpening and rotating.

```
sharperImg = ilSharpenImgCreate((ilImage*)inImg,
                0.5, 1.5, ilPadSrc);
/* If page width equals image width Then adjust cache */
/* size to hold pages needed for ilRotZoomImg */
ilGetPageSize((ilImage *)inImg, &px, &py, &pz, &pc);
if (px == ilGetXsize((ilImage *)inImg))
  ilSetCacheWindow((ilImage *)sharperImg,
            ilGetXsize((ilImage *)inImg),
            ilGetYsize((ilImage *)inImg), 0, 0, TRUE);

rotatedImg = ilRotZoomImgCreate((ilImage*)sharperImg,
            90.0,1,1,0,ilBiLinear);
```

Step 3: Open a window for displaying in RGB mode.

```
rgbMode = getgdesc(GD_BITS_NORM_SNG_RED) > 0;
doubleBuffer = getgdesc(GD_BITS_NORM_DBL_RED) >= 8;

glx = ilGLXConfigCreate(5);
ilGLXConfigAddEntry(glx, GLX_NORMAL, GLX_RGB, rgbMode);
ilGLXConfigAddEntry(glx,GLX_NORMAL,GLX_DOUBLE,
                doubleBuffer);
ilGLXConfigAddEntry(glx, 0, 0, 0); /* terminator */

dpy = XOpenDisplay(NULL);
ilGetSize((ilImage*)rotatedImg, &size);
win = ilGLXConfigCreateWindow(glx, dpy,
             RootWindow(dpy, DefaultScreen(dpy)),
             0, 0, size.x, size.y, 0);
XStoreName(dpy, win, “Processed Image”);//set window title
XSelectInput(dpy, win, ExposureMask | KeyPressMask);
XMapWindow(dpy, win);

/* If SGI format Then adjust cache size */
if (ilFileImgGetImageFormat(inImg) == ilSGI_IMG)
ilSetCacheWindow((ilImage *)inImg, size.x, size.y, 0, 0,
              TRUE);
```

Step 4: Display the processed data.

```
disp = ilDisplayCreate(dpy, win, ilGLRender, ilDefault);
ilDisplayAddViewTop(disp, (ilImage*)rotatedImg, ilCenter);
ilDisplayRedraw(disp, ilDefault);
```

Step 5: Write the processed data to a file.

```
  tmpFile = ilCreateImgFile(“outFile.tif”, &size,
              ilGetDataType((ilImage*)inImg),
              ilGetOrder((ilImage*)inImg),
              ilTIFF_IMG, NULL);
  ilCopyTile((ilImage*)tmpFile, 0,0,size.x,size.y,
              (ilImage*)rotatedImg,0,0,0,1);
  ilImageDelete((ilImage*)tmpFile);
```

Step 6: Display until the user quits.

```
  ever = 1;
  for (;ever;) {
    XNextEvent(dpy, &event);
    switch (event.type) {

      case DestroyNotify:
        ever = 0;
        break;

      case KeyPress:
        switch(XLookupKeysym(&event.xkey, 0)) {
        case XK_Escape:      /* Escape key */
          ever = 0;
          break;
       }
        break;

      case Expose:
        reshapeviewport();
        ilDisplayRedraw(disp, ilDefault);
        break;
    }
  }
  XCloseDisplay(dpy);
  exit(0);
}
```

You can see several examples of function name changes in this sample program. For example, the C++ call **inImg->getOrder()** becomes **ilGetOrder(inImg)** in C, **inImg->getSize()** becomes **ilGetSize(inImg)**, and so on. Note also where ilFileImg, ilSharpenImg, and ilRotZoomImg pointers are cast to ilImage pointers in the argument lists for several functions.

Pure GL Sample Program Written in C

The code in Example 1-4 shows the version of the sample written in C using GL window management.

Example 1-4 Sample Program (in C) Using GL Window Management

```
/* sampleCProgGL */
#include <il/ilCdefs.h>
#include <gl/device.h>
#include <stdlib.h>
#include <stdio.h>

void main (int argc, char **argv)
{
  ilFileImg              *inImg, *tmpFile;
  ilSharpenImg           *sharperImg;
  ilRotZoomImg           *rotatedImg;
  ilDisplay              *disp;
  ilSize                 size;
  long                   dev, wid;
  short                  val, active;
  int                    px, py, pz, pc;

  if (argc < 2) {
    printf("usage: sampleCProgGL <filename>\n");
    exit(0);
  }
```

Step 1: Open the file of image data.

```
  inImg = ilOpenImgFile(argv[1], "r");
```

Step 2: Create IL objects for sharpening and rotating.

```
  sharperImg = ilSharpenImgCreate((ilImage*)inImg,
               0.5, 1.5, ilPadSrc);
  /* If page width equals image width Then adjust cache */
  /* size to hold pages needed for ilRotZoomImg */
  ilGetPageSize((ilImage *)inImg, &px, &py, &pz, &pc);
  if (px == ilGetXsize((ilImage *)inImg))
    ilSetCacheWindow((ilImage *)sharperImg,
               ilGetXsize((ilImage *)inImg),
               ilGetYsize((ilImage *)inImg), 0, 0, TRUE);

  rotatedImg = ilRotZoomImgCreate((ilImage*)sharperImg,
               90.0,1,1,0,ilBiLinear);
```

Step 3: Open a window for displaying in RGB mode.

```
    ilGetSize((ilImage*)rotatedImg, &size);
    prefsize(size.x, size.y);
    wid = winopen(argv[1]);
    if (getgdesc(GD_BITS_NORM_SNG_RED) != 0) RGBmode();
    gconfig();
```

Step 4: Display the processed data.

```
    disp = ilDisplayGL(wid);
    ilDisplayAddViewTop(disp, (ilImage*)rotatedImg, ilCenter);
    ilDisplayRedraw(disp, ilDefault);
```

Step 5: Write the processed data to a file.

```
    tmpFile = ilCreateImgFile("outFile.tif", &size,
                ilGetDataType((ilImage*)inImg),
                ilGetOrder((ilImage*)inImg),
                ilTIFF_IMG, NULL);
    ilCopyTile((ilImage*)tmpFile, 0,0,size.x,size.y,
                (ilImage*)rotatedImg,0,0,0,1);
    ilImageDelete((ilImage*)tmpFile);
```

Step 6: Display until the user quits.

```
    qdevice(REDRAW);
    qdevice(ESCKEY);

    active = 1;
    while (active) {
      switch (dev = qread (&val)) {

      case ESCKEY:
        active = 0;
        break;

      case REDRAW:
        reshapeviewport();
        ilDisplayRedraw(disp, ilDefault);
        break;

      }
    }
    exit(0);
  }
```

A Sample Program in Fortran

Here's the pure GL sample program as you would write it in Fortran.

Example 1-5 Sample Program (in Fortran) using GL Window Management

```
c program sampleFProgGL

c Include Fortran header file for Image Library
#include <il/ilFdefs.h>
#include <gl/fgl.h>

       integer*4 theImg, sharperImg, rotatedImg, disp
       integer*4 status, viewptr, wid, ordtype, datatype
       integer*4 sizex, sizey, j, i, chanlist(3)
       character*1 tmp(80)
       character*80 fname
       equivalence (fname, tmp(1)
       integer*4 dev
       integer*2 val
       integer*4 outimagesize(4), tmpfile

       j = iargc()
       if (j .ne. 1) then
          write(6,*)'usage: sampleFProgGL <filename>'
          stop
       endif
       call getarg(1, fname)
c strip blanks off the filename and stuff with nulls
       i = index(fname,' ')
       do 100 j = i, 80
          tmp(j) = '\0'
100    continue
```

Step 1: Open the file of image data.

```
       theImg = ilOpenImgFile(fname, 'r')
```

Step 2: Create IL objects for sharpening and rotating.

```
       sharperImg =
    1       ilSharpenImgCreate(theImg,0.5,1.5,ilPadSrc)
       rotatedImg = ilRotZoomImgCreate(sharperImg,90.0,
    1       1.0,1.0,0,ilBiLinear)
```

Step 3: Open a window for displaying in RGB mode.

```
      sizex = ilGetXsize(rotatedImg)
      sizey = ilGetYsize(rotatedImg)
      call prefsi(sizex, sizey)
      wid = winope('sampleFProgGL', 13)
      if (getgde(GD_BITS_NORM_SNG_RED) .ne. 0) call RGBmod()
      call gconfi()
```

Step 4: Display the processed data.

```
      disp = ilDisplayGL(wid, ilDefault)
      viewptr = ilDisplayAddView(disp,rotatedImg,0,ilCenter)
      status = ilDisplayRedraw(disp, ilDefault)
```

Step 5: Write the processed data to a file.

```
      ordtype = ilGetOrder(rotatedImg)
      datatype = ilGetDataType(rotatedImg)
      outimagesize(1) = sizex
      outimagesize(2) = sizey
      outimagesize(3) = ilGetZsize(rotatedImg)
      outimagesize(4) = ilGetCsize(rotatedImg)
      tmpFile = ilCreateImgFile("outFile.tif",
     1     outimagesize(1), datatype, ordtype, ilTIFF_IMG,
     2     %VAL(0), %VAL(11), %VAL(4))
      status = ilCopyTile(tmpFile,0,0,sizex,sizey,
     1    rotatedImg,0,0,%VAL(0),1)
      call ilImageDelete(tmpFile)
```

Step 6: Display until the user quits.

```
      call qdevic(REDRAW)
      call qdevic(ESCKEY)

200   continue
      dev = qread(val)
      if (dev .eq. REDRAW) then
        call reshap()
        status = ilDisplayRedraw(disp,ilDefault)
      end if
      if (dev .eq. ESCKEY) goto 99
      if (dev .ne. 0) go to 200
 99   stop
      end
```

The Fortran IL calls are nearly identical to the C calls, with the exception that each pointer is replaced in Fortran by an **integer*4**. As in the C calls, you must include all arguments to each function. (For more information on calling C functions from Fortran, see the chapter titled "Fortran Programming Interfaces" in the *FORTRAN 77 Programmer's Guide*.)

Chapter 2

The ImageVision Library Foundation

This chapter describes the architecture and design philosophy of the ImageVision Library. This description includes the IL class hierarchy, image attributes, the caching and accessing of image data, and the chaining of image operators for optimum performance.

Chapter 2

The ImageVision Library Foundation

This chapter explains the general architecture and design philosophy of the ImageVision Library. All subsequent chapters assume knowledge of the basic concepts presented here. This chapter contains the following major sections:

- "The IL Class Hierarchy" on page 30 gives a brief overview of the main classes that compose the IL.
- "Foundation Classes" on page 31 introduces the IL foundation classes, particularly ilLink and ilImage, from which most IL classes derive.
- "Image Attributes" on page 34 discusses in detail the attributes used to describe an image and the functions available for retrieving and setting these attributes.
- "The Cache" on page 46 describes the role of cache in holding raw and processed image data.
- "Accessing Image Data" on page 54 discusses the general capabilities for reading and writing image data that are common to all image classes.
- "The IL Execution Model" on page 68 discusses the IL's demand-driven model that optimizes memory usage and performance as image data is processed.
- "Working with Image Chains" on page 75 shows how you can manipulate image chains in a dynamic environment.
- "Object Properties" on page 80 describes how you can assign property values to objects and retrieve these values.

The IL Class Hierarchy

The architecture and functionality of the IL is contained in a hierarchy of C++ classes. Most of this chapter is devoted to a discussion of the principal image class (ilImage), from which most IL classes derive, and the ilMemCacheImg class, which implements a key part of the IL's execution model. However, a brief look first at the IL base classes provides a perspective for better understanding the role of the ilImage and ilMemCacheImg classes.

The base classes can be divided into four functional groupings:

ilLink	The ilLink class defines the chaining of image operators and the images associated with these operators. "The ilLink Class" on page 32 contains more information about the ilLink class.
Multi-threading	The IL contains several base classes that implement the multi-threading feature in the IL. "Multi-threading" on page 71 describes how multi-threading works in the IL. "Controlling Multi-threading" on page 336 tells you how you can query or control the classes used in multi-threading.
ilDisplay	The ilDisplay class allows you to create and manage one or more processed images in a graphics window. Read Chapter 5, "Displaying an Image," to learn more about this class.
Miscellaneous	Some base classes, like ilLut, ilPixel, and ilSize, provide a variety of auxiliary functions to support the function of the IL. For example, ilPage defines a page of image data and ilLut defines a color palette lookup table (LUT) used to interpret the data in some images. "Auxiliary Classes" on page 396 contains more detail about many of these miscellaneous base classes.

The entire IL inheritance hierarchy is shown on the quick reference card included with this guide. All the IL classes are briefly summarized in "Summary of All Classes" on page 359.

Foundation Classes

Figure 2-1 shows the portion of the IL class hierarchy that derives from ilLink. The classes in the shaded boxes are the IL foundation classes. These classes provide much of the functionality and flexibility of the ImageVision Library.

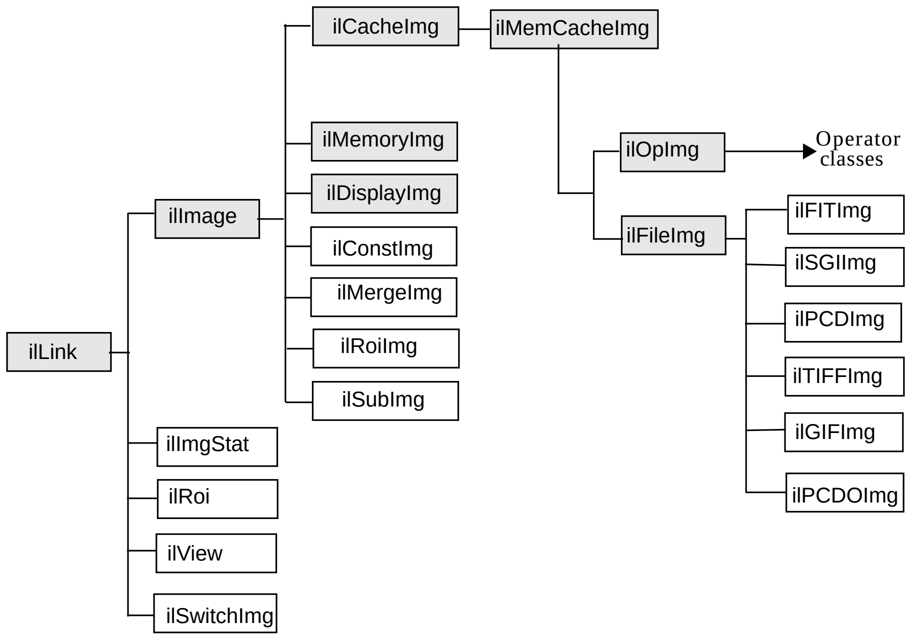


Figure 2-1 The ilLink Class Inheritance

The foundation classes shown in Figure 2-1 are abstract classes and can’t be used directly. However, understanding the capabilities these classes provide is key to understanding how the IL works and how to use it. Also, if you extend the IL to meet your specific image processing needs, you will derive your own classes from these abstract classes.

The ilLink Class

The IL allows you to access, manipulate, store, and display images. You can perform a series of operations on one or more images by creating a chain of operators and passing the image or images down this chain of operators. An operator is a class derived from ilOpImg (the base class for all IL operators) that applies its image processing algorithm to an image. The image output from each operator becomes the input to the next operator in the chain.

An element in a chain of operators can be:

- a raw or processed image
- an object containing statistical information about an image, for example, a histogram
- a region of interest within an image
- a portion of the image to be displayed

The result of a chain of operations is either a display of the processed image or a file on disk containing the processed image. Figure 2-2 illustrates this concept by showing an image chain whose elements are raw and processed images.

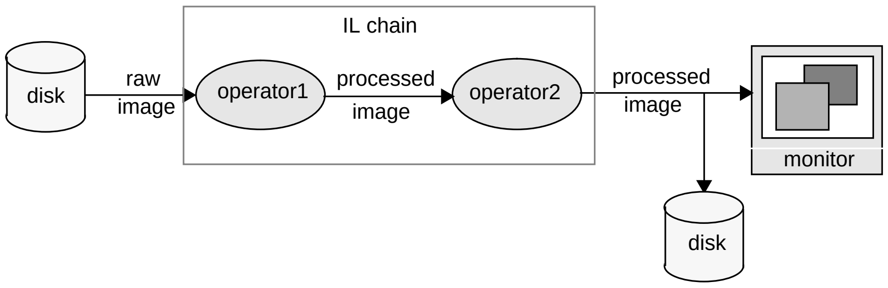


Figure 2-2 An IL Chain

The ilLink class implements the chaining model by defining the mechanism for linking the image objects together. This model defines the concept of parent (input) and child (output) images. The ilLink class also provides functions that allow you to manipulate image attributes by providing functions that keep track of whether an attribute is allowed to change or has been altered. "The IL Execution Model" on page 68 contains more information about chaining in the IL.

The ilImage Class

The ilImage class is the root of the majority of the IL's image class hierarchy and provides the IL's abstract concept of what images are and how they're manipulated. The IL defines an image as a four-dimensional array of pixels, *x*, *y*, *z*, and *c*. An image has certain attributes, such as what size (in pixels) the image is, what data type the pixel elements are (for example, **float** or **int**), and what color model should be used to interpret the data (for example, RGB or CMYK).

The ilImage class provides two main categories of functions to support this abstraction of an image:

- image attribute functions, for querying an image about its attributes and setting these attributes (Programmers can explicitly set some attributes, even though many attributes are determined at the time the image is instantiated.)
- data access functions, for reading, writing, and copying image data

All classes that derive from ilImage (see Figure 2-1) inherit these general capabilities for querying and setting attributes and accessing data. Thus, the IL allows you to manipulate all images in the same way, regardless of the actual source or destination of the data. The same mechanism is used for data that's associated with any type of image; for example:

- an image stored in memory (ilMemoryImg)
- an image that's displayed on the screen and that resides in the framebuffer (ilDisplayImg)
- an image operator, which applies an image processing algorithm to its data (ilOpImg)
- an image that resides in a file on disk and is buffered in memory (ilFileImg)

Classes derived from ilImage implement their own versions of the data access functions as necessary to add specificity. For example, ilMemCacheImg defines versions of the data access functions that read data from or write it to a partial copy of the image buffered in main memory. Similarly, ilTIFFImg adds capabilities specifically for reading and writing TIFF file headers and data. The ilSharpenImg class incorporates a sharpening algorithm into its access functions.

Image Attributes

In the IL, an image has several descriptive attributes. These include:

- image size
- data type of image pixels
- data ordering of channels in an image
- color model
- color palette
- image type
- coordinate space
- fill value
- minimum and maximum pixel values
- data compression
- page border
- image format

Many of these attributes are assigned default values when an image is created. Some of them are changed subsequently, usually as a result of applying—or preparing to apply—an image operator. Some can be changed explicitly by the programmer. Each class that derives from ilImage chooses which attributes it allows to be explicitly modified. (For more information about how this mechanism works, see “Propagating Image Attributes” on page 78 and “Managing Image Attributes” on page 249.)

This section describes the image attributes and the functions available for retrieving and setting them. These functions are defined by the ilImage and ilLink classes and therefore can be used on any type of image.

For your convenience, Table 2-1 provides a summary of the image attribute functions. All of these functions are described later in this section except for image format (described in “Querying a File Image” on page 98) and page border (described in “Page Borders” on page 75).

Table 2-1 Image Attribute Summary

Image Attribute	Retrieving Attributes	Changing Attributes

Size	getSize()
	getXsize()
	getYsize()
	getZsize()
	getCsize()
		setSize()
		setCsize)
		Apply an operator that affects size.

Data type	getDataType()
	isSigned()
		setDataType()

Data ordering	getOrder()	setOrder()
Color model	getColorModel()	setColorModel()
Color palette	getColorMap()	setColorMap()
Image type	getImageType()	Set upon file creation and cannot be changed
Coordinate Space	getCoordSpace()	setCoordSpace()
Fill value	getFill()	setFill()

Min and max pixel values	getminPixel()
	getMaxPixel()
	getMaxValue()
		setMinPixel()
		setMaxPixel()
		setMaxValue()

Data compression	getCompression()	setCompression()
Page border	getPageBorder()	setPageBorder()
Image format	getImageFormat()	Use the *imgCopy* utility to convert from one IL-supported format to another

In addition to the functions shown above, which allow you to set image attributes individually, you might decide to use the IL’s ilConfig class, which allows you to specify several image attributes at once. An ilConfig contains several elements that describe pixel data: the data type, pixel ordering, number of data channels, ordering of data channels, channel offset, coordinate space, and zoom factors. This class is defined in the header file *il/ilConfig.h* and described in more detail in "ilConfig" on page 397 as well as in its reference page.

Error Codes

As you read the following sections, you'll note that many of the functions described return a value of data type ilStatus. This enumerated type, which is defined in the header file *il/ilError.h*, contains the error codes used by the IL to indicate that an unexpected result occurred. If no unexpected result occurred, an image's status is ilOKAY. The error codes and their meanings are listed in "Error Codes" on page 407.

At any point, you can query an ilImage about its current status by using **getStatus()**, a function defined in ilLink that takes no arguments and returns a value of type ilStatus. You can also set an image's status to ilOKAY by using **clearStatus()** (a function defined in and inherited from ilLink).

Size

One key attribute of an image is its size, which is determined initially when an image is created. In the sample program in Chapter 1, "Writing an ImageVision Library Program," the size of the image data is determined when the **ilOpenImgFile()** function is called. The IL defines an ilSize data structure, which consists of four integers that correspond to the image's size in the *x*, *y*, and *z* dimensions and the number of data channels, *c*, per pixel.

The *x* and *y* dimensions specify the width and height of the image as measured in pixels. The *z* dimension, or "depth," may refer to the number of *xy* planes of image data. The *xy* planes are usually related in some way; for example, they might be a time-series of a single animation scene or a set of CAT scan images. (CAT stands for computerized axial tomography, a medical imaging technique used to create three-dimensional images.) Different image representations require different numbers of data channels to describe each pixel of data. An RGB (red, green, blue) image, for example, requires three channels, one for each of the three colors.

The ilImage class defines functions for retrieving the entire ilSize structure for an image at once and functions for returning each of the elements separately:

```
ilSize imgSize;
int imgXSize, imgYSize, imgZSize, imgChans;

myImg.getSize(imgSize);
imgXSize = myImg.getXsize();
imgYSize = myImg.getYsize();
imgZSize = myImg.getZsize();
imgChans = myImg.getCsize();
```

You can change an image's size by applying an image operator that affects its size or by setting its size explicitly (if you're allowed to set it). For example, in most cases, the ilRotZoomImg operator (used in the sample program in Chapter 1, "Writing an ImageVision Library Program") produces a processed image with a size that differs from that of the original image, as shown in Figure 2-3:

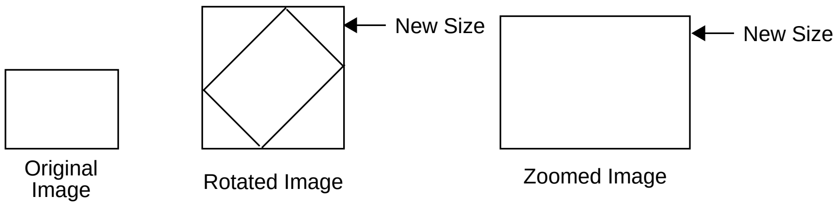


Figure 2-3 Sizes of Original and Processed Images

You can set an image's size explicitly by using **setSize()**, which takes a reference to the desired ilSize structure as an argument. A separate function, **setCsize()**, allows you to restrict the number of channels associated with an image.

Data Type

An image's pixel components must all be of the same data type. The IL defines an enumerated set of data types (ilType) and a function, **getDataType()**, to return the data type of an image's pixels:

```
ilType imgType;
imgType = myImg.getDataType();
```

The ilType returned can be one of the following: ilBit, ilChar, ilUChar (an unsigned char), ilShort, ilUShort, ilLong, ilULong, ilFloat, or ilDouble. (These types are defined in the *il/ilTypes.h* header file and listed in "Describing Image Attributes" on page 409.)

Use **isSigned()** to query an ilImage about whether its data type is signed:

```
int sign = myImg.isSigned();
```

As shown, this function takes no arguments and returns TRUE (nonzero) if the image's data type is signed and FALSE (zero) otherwise.

Operators accept input images of any data type, even if the actual image processing computation is performed using a different type. In these cases, the data is converted as needed to perform the computation. If you know what data type you'll need at the end of the computation, you can use the **setDataType()** function to force the data type.

Data Ordering

The channels composing an image's pixel data can be ordered in any of several ways. The IL defines a corresponding enumerated type, ilOrder, and a query function **getOrder()**, as shown below. (See the *il/ilTypes.h* header file and "Describing Image Attributes" on page 409.)

```
ilOrder imgOrder;
imgOrder = myImg.getOrder();
```

The ilOrder value returned can be either ilInterleaved, ilSequential, or ilSeparate. The meanings of these three orders are shown in Figure 2-4 and explained below:

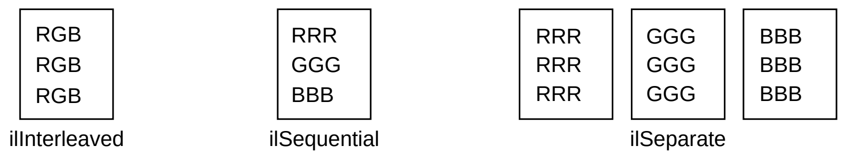


Figure 2-4 Pixel Data Ordering for an RGB Image

Interleaved	In interleaved ordering, all pixel components are clustered together. For an interleaved RGB image, data is stored as: RGBRGBRGB....
Sequential	With sequential ordering, each component is stored as a separate line. In the example, three lines of data (one each for red, green, and blue data) are needed to describe one line of pixels.
Separate	An image using separate ordering stores each component in a separate page. (See "The Cache" on page 46 for more information about pages.)

Thus, the order defines which dimensions vary most rapidly relative to the others in a chunk of data. For example, in the interleaved case, the channel dimension varies most rapidly, and the *z* dimension varies least rapidly. Here's how the dimensions vary for each of the orders, listed from most to least rapidly: ilInterleaved *(c,x,y,z)*, ilSequential (*x,c,y,z*), ilSeparate (*x,y,z,c*).

In the rare cases where you need to set an image's order, use the **setOrder()** function. Some classes derived from ilImage, such as ilFileImg, won't let you change an image's order.

Color Model

An image's color model determines the meaning of the data channels from which a pixel is constructed. The IL defines an ilColorModel enumerated type (in the header file *il/ilTypes.h*) that can refer to the following color models:

ilRGB	red, green, blue
ilRGBA	red, green, blue, alpha
ilRGBPalette	color index mapped to an RGB lookup table
ilHSV	hue, saturation, value
ilCMY	cyan, magenta, yellow
ilCMYK	cyan, magenta, yellow, black
ilMinWhite	grayscale, with the minimum value interpreted as white
ilMinBlack	grayscale, with the minimum value interpreted as black
ilBGR	variation of RGB, for images generated by Silicon Graphics
ilABGR	variation of RGBA, for images generated by Silicon Graphics
ilMultiSpectral	generally more than three channels; requires a special interpretation
ilYCC	a luminance / chrominance data metric based on video primaries

The **getColorModel()** function allows you to query an image about its color model. If necessary, you can change the data interpretation by using the **setColorModel()** function.

Color Palette

Some images include a color palette that's used to interpret their data. A color palette is also referred to as a lookup table or LUT. The most common use of such a table is to store color map values. The ilLut class, defined in the header file *il/ilLut.h* and described in “ilLut” on page 399, is provided for such purposes. To set an image’s LUT, use **setColorMap()**:

```
ilStatus setColorMap(const ilLut& lut);
```

The table pointed to by *lut* is established as the image’s look-up table. This function copies the specified ilLut but not its data. The **getColorMap()** function returns by reference an image’s LUT:

```
void getColorMap(ilLut& lut);
```

Two other functions—**ilSGICmapLUT()** and **ilSGIFileLut()**—create look-up tables for use in managing color map data. They’re described in “ilLut” on page 399 and in their own reference pages.

Image Type

The way in which an image stores its data defines its image type. The IL defines the ilImageType data type in the header file *il/ilTypes.h*. The IL supports nine image types:

ilMEM_IMG	an image residing in memory
ilFILE_IMG	an image residing in a file on disk
ilGLDISP_IMG	a GL image residing in the framebuffer
ilXDISP_IMG	an X image residing in the framebuffer
ilOP_IMG	an operator image
ilX_IMG	an X image
ilSYNTH_IMG	a synthetic image created interactively
ilTEX_IMG	an image residing in texture memory
ilAUX_IMG	an image residing in off-screen framebuffer memory

The last three are used internally and do not correspond to any particular image class in the IL hierarchy.

The **getImageType()** function allows you to query an image about its image type. The proper type is set upon creation and may not be changed by the programmer.

Coordinate Space

Different file formats arrange their data in different ways. By default, a TIFF file image considers its origin to be the upper left corner; if you scan through the data, you should read from left to right, working your way down the image. An SGI RGB image considers its origin to be the lower left corner; to read through its data, again read from left to right, but work your way up the image.

The IL defines an ilCoordSpace data type to represent the possible orientations of image data. To query an image about the orientation of its coordinate space, use **getCoordSpace()**, which returns one of the eight values listed below. (You can set an image's coordinate space with the **setCoordSpace()** function.) These four coordinate spaces have the traditional orientation of the *x* and *y* dimensions (the *x* dimension goes across, and the *y* dimension runs up and down):

ilUpperLeftOrigin	the same as the TIFF example
ilLowerLeftOrigin	the same as the Silicon Graphics RGB example
ilUpperRightOrigin	the origin is in the upper right corner, and you read data from right to left, working your way down the image
ilLowerRightOrigin	the origin is in the lower right corner, and you read data from right to left, working your way up the image

The following four coordinate spaces have the *x* and *y* dimensions transposed so that the *x* dimension runs up and down, and the *y* dimension goes across.

ilLeftUpperOrigin	The origin is in the upper left corner, and you read from top to bottom, working your way across the image to the right.
ilLeftLowerOrigin	The origin is in the lower left corner, and you read from the bottom to the top, working your way across the image to the right.

ilRightUpperOrigin	The origin is in the upper right corner, and you read data from top to bottom, working your way across the image to the left.
ilRightLowerOrigin	The origin is in the lower right corner, and you read data from bottom to top, working your way across the image to the left.

Figure 2-5 illustrates the difference between ilUpperLeftOrigin and ilLeftUpperOrigin orientation of image data.

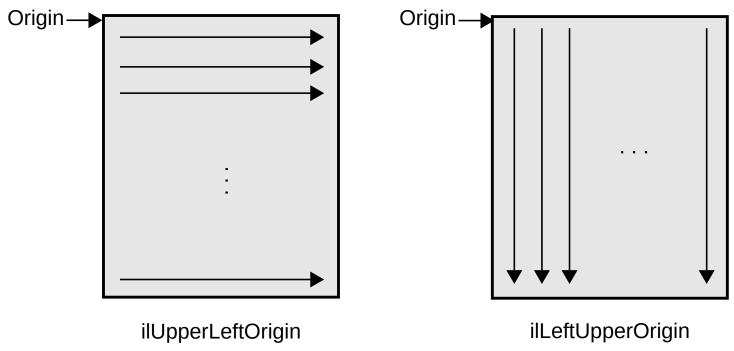


Figure 2-5 Image Coordinate Spaces

Fill Value

When a function tries to access pixels that are beyond an image's edge, those pixels are set to the image's fill value. By default, an image's fill value is 0, but you can set a different fill value with the **setFill()** function:

```
static float fillData[3] = {127.0, 127.0, 127.0};
myImg.setFill( ilPixel(ilFloat, 3, fillData) );
```

As shown, **setFill()** takes a reference to an ilPixel as an argument. An ilPixel defines the pixel's data type (in this case, **ilFloat**), the number of data channels (3), and the pixel data itself (*fillData[]*). (In this example, the ilPixel value is passed in-line so that the compiler automatically constructs and deletes the object.) The image makes its own copy of the pixel data.

Use **getFill()** to query an image about its fill value:

```
ilPixel theFillValue;
myImg.getFill(theFillValue);
```

Minimum and Maximum Pixel Values

By default, no restrictions are placed on the range of a pixel’s allowable values. However, when an image is displayed—for example, using the ABGR color model—its pixel values may need to be converted to the range that’s meaningful for the framebuffers, which is 0 to 255. If you explicitly set an image’s minimum and maximum allowable pixel values, they’re used to color-scale the data as it’s displayed.

You might want to set the allowable pixel values for a processed image so that the resulting data has certain characteristics, especially if you’ll be displaying the data. For example, suppose you’re using an edge detection filter that theoretically produces data ranging in value from -1000 to +1000. However, you know that the images you’ll be filtering will actually yield filtered data ranging from -100 to +100. If you set the allowable values to match this range and then display the filtered data, the display will be more useful, since the data will be scaled and stretched out over the framebuffer’s meaningful range.

Setting Maximum and Minimum Pixel Values

Minimum and maximum values are image attributes that are stored with an image. You can set the minimum and maximum allowable values for an image’s pixel data by using the **setMinPixel()** and **setMaxPixel()** functions. Both these functions take an ilPixel reference as an argument:

```
ilStatus setMinPixel(const ilPixel& pix);
ilStatus setMaxPixel(const ilPixel& pix);
```

Use **getMinPixel()** and **getMaxPixel()** to query an image about its minimum and maximum allowable pixel values:

```
void getMinPixel(ilPixel& pix);
void getMaxPixel(ilPixel& pix);
```

These functions return the minimum or maximum pixel value by reference.

Setting Maximum and Minimum Pixel Values for a Channel

You can also set the minimum and maximum values for an individual channel of an image:

```
ilStatus setMinValue(double val, int c=0);
ilStatus setMaxValue(double val, int c=0);
```

These functions set channel *c*'s minimum or maximum value to *val*.

To query an image about its channel value limits, use **getMinValue()** and **getMaxValue()**:

```
double getMinValue(int c=-1);
double getMaxValue(int c=-1);
```

These functions return the minimum or maximum allowable value for the specified channel (the default, -1, returns the minimum or maximum of all channels).

Setting Maximum and Minimum Scaling Values For Color Conversion

Minimum and maximum scaling values are used by the IL during color conversion. By default, the scale minimum and maximum are the same as the image minimum and maximum values. The IL provides functions you can use to retrieve current maximum and minimum scaling values and set new ones.

The **initScaleMinMax()** function initializes the scale minimum and maximum to the image minimum and maximum values. If scale minimum and maximum have already been set, they are unchanged, unless *force* is TRUE.

```
void initScaleMinMax(int force=0);
```

The function **setScaleMinMax()** sets the minimum and maximum scaling values to *min* and *max*. The **setScaleType()** function sets the scale minimum and maximum to the minimum and maximum values of the data type passed in *type*.

```
ilStatus setScaleMinMax(double min, double max);
ilStatus setScaleType(ilType type=ilType(0));
```

The **getScaleMax()** and **getScaleMin()** functions return the maximum and minimum value used for scaling during color conversion.

```
double getScaleMax();
void getScaleMin();
```

Data Compression

Often, images stored in a file on disk are compressed to minimize their size. Such images need to be decompressed before they can be read. There are many different compression algorithms, and each specific file format (for example, TIFF) determines which algorithms it supports. See “Setting a File’s Compression” on page 96 for more information about which compression algorithms the IL supports. From a programmer’s point of view, as data is read or written in an IL program, its compression or decompression is handled transparently.

The Cache

The IL uses the term *cache* to mean a portion of memory that holds raw and processed image data accessed by a process. This is not the same as the hardware cache accessed by the CPU. The IL cache holds image data in rectangular pieces called pages. The cache does not necessarily hold all the pages for each image being processed, but only those pages that have been referenced and have not been bumped out of cache to make room for more recently referenced pages. Thus, only part of an image may reside in cache.

Figure 2-6 shows a cache that contains three images being used by an IL application. The three rectangles on the left show a logical map of the pages for each image. The shaded boxes indicate the pages of each image resident in cache. For example, the raw image contains four pages, only two of which are in cache. The rectangle on the right shows cache as it might contain the pages from the three images.

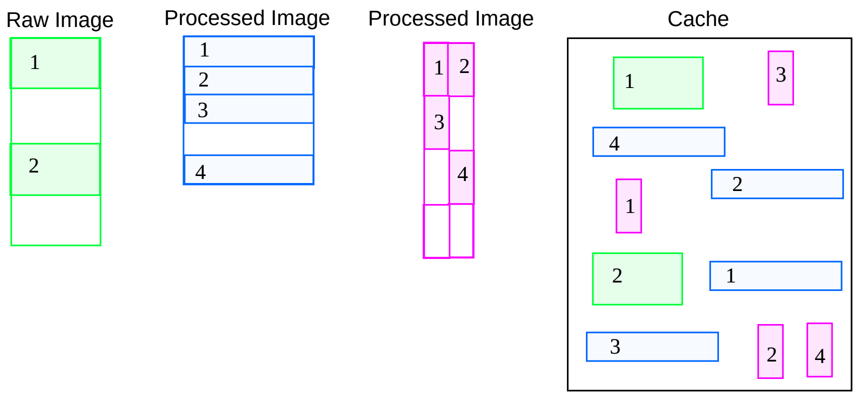


Figure 2-6 Cache Containing Portions of Three Images

The IL keeps track of the pages in cache, brings in a page when the program requests data on a page not in cache, and chooses a page to be overwritten when it needs to read a new page for which there is no space.

Every IL class that derives from ilMemCacheImg uses cache and the caching mechanism defined by ilMemCacheImg. Both ilOpImg and ilFileImg inherit directly from ilMemCacheImg and use caching in these ways:

- IL operators (those classes that derive from ilOpImg) use the cache to hold the results of applying their image processing algorithms to an image.
- Classes that derive from ilFileImg place raw, uncompressed data in the cache.

While the cache holds image data in pages, an IL program can access image data in rectangular blocks of any size, without regard to page boundaries. These rectangular blocks are referred to as tiles. As shown in Figure 2-7, tiles can cross page boundaries or can be smaller than a page.

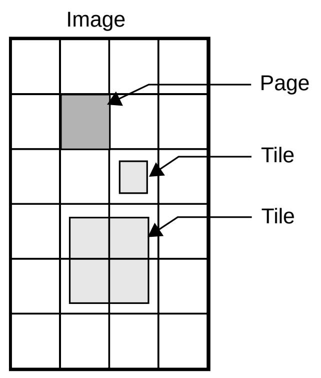


Figure 2-7 Pages and Tiles of Image Data

When a program requests a data tile, the IL checks the cache. If the data corresponding to the tile isn't among the pages already in the cache, the IL brings additional pages into the cache as necessary. If the cache is already full, it must discard some of the resident pages in order to read the new pages. The page replacement algorithm is based on a combination of these factors:

- The number of times each page has been referenced; the more times a page is referenced, the more likely it will remain in the cache.
- The priority of the references; higher priority requests tend to have their pages retained longer.
- The relative time since each page was last referenced; the pages that have been in the cache the longest without being referenced are discarded first.

The overall effect of the page replacement algorithm is that data toward the end of a chain tends to get preferentially cached. Other data that is frequently referenced (for instance, the input to an operator whose parameters are being repeatedly adjusted) also tends to remain in the cache. To prevent data from being recomputed for successive tile requests, the cache must be large enough so that pages just discarded aren't reread. (See the following two sections for more information on setting the size of the cache and adjusting priorities.)

Since operators place processed image data in the cache, data is operated on as it's brought into the cache. To maximize efficiency under this execution model, only the pages needed to satisfy any given tile request are brought into the cache. For example, if a **getTile()** request specifies only a single channel of an image that's stored in a separate format, only the pages containing that channel are accessed. Thus, processing multispectral data (or any data stored in a separate format) is made as efficient as possible.

Managing Cache

By default, the cache size is set to 30% of the total user memory on the host system. The IL provides two functions to override the default size of the cache, **ilSetMaxCacheSize()** and **ilSetMaxCacheFraction()**, which are defined as shown below:

```
void ilSetMaxCacheSize(int maxBytes);
void ilSetMaxCacheFraction(float fraction);
```

The first function sets the cache size to the number of bytes indicated. The second function computes the size of the cache as the indicated fraction of the total user memory on the host computer.

You can change these limits without modifying an IL- based program by using the environment variables IL_CACHE_SIZE or IL_CACHE_FRACTION to set either the size in bytes or the fraction of user memory, respectively. The IL_CACHE_SIZE value overrides the value specified by IL_CACHE_FRACTION. Any value established with these environment variables is overridden by calls to **ilSetMaxCacheSize()** or **ilSetMaxCacheFraction()**.

The current value of these cache size limits can be obtained with either **ilGetMaxCacheSize()** or **ilGetMaxCacheFraction()**. The current actual size of the object's cache can be retrieved with **ilGetCurCacheSize()**. These functions are defined as shown below:

```
int ilGetCurCacheSize();
int ilGetMaxCacheSize();
float ilGetMaxCacheFraction();
```

The IL maintains global cache in a special memory pool that allows the cache memory to be compacted to eliminate memory fragmentation problems. When fragmentation exceeds a defined threshold, the pool is automatically compacted. You can use the **ilSetCompactFraction()** function to set the fragmentation threshold to the maximum fraction of the pool that is allowed to be wasted space before compaction occurs. The current value of this threshold can be obtained with **ilGetCompactFraction()**. The default compact fraction value is .2 or 20%.

```
ilSetCompactFraction(float maxWastedFraction);
float ilGetCompactFraction();
```

You can force compaction of the pool at any time by calling **ilCompactCache()**. If the pool is more fragmented than the fraction passed to this routine, it is compacted. You can pass zero to cause the pool to be unconditionally compacted.

```
ilCompactCache(float maxWastedFraction=0);
```

You can use the **getCacheSize()** function of ilMemCacheImg to query the cache size for a individual object:

```
int getCacheSize();
```

You can use the **flush()** member function of ilMemCacheImg to flush the cache for a individual object:

```
ilStatus flush(int discard=FALSE));
```

You can free the memory in the global cache to get it down to a desired maximum size with **ilFlushCache()**. This call also compacts the cache memory.

```
int ilFlushCache(int maxsize);
```

Priority

When an image operator requests a tile and any of the pages needed by the tile are not in cache, the missing pages must be brought into cache. If the cache is full, some of the resident pages must be discarded and replaced with new pages. The IL then has to decide which pages to discard.

The IL assigns priorities to pages in cache and uses these priorities to make decisions about which pages to discard. The priority associated with pages in cache ranges from zero (lowest) to seven (highest); the higher the priority, the greater the likelihood the page will remain resident.

The IL maintains a linked list of the pages in cache for each of priority levels 0 through 7. Figure 2-8 illustrates this concept. This simplified diagram shows a cache with three pages at priority level seven, two pages at priority level three, and three pages at level zero.

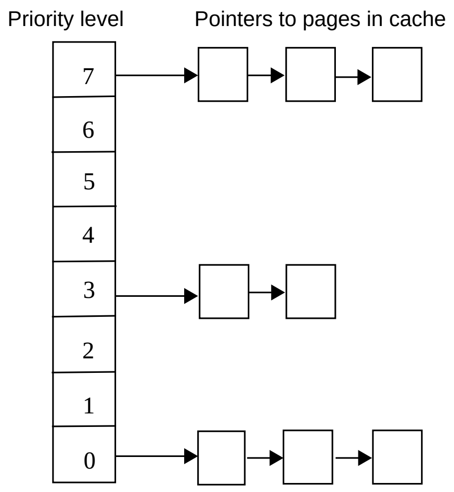


Figure 2-8 Priority Lists in Cache

The initial (default) priority level of a page is zero. The following events can cause a change to the priority level:

- The priority of a page is increased by one each time the page is accessed (for example by a **copyTile()** or **getPage()**). This is essentially a reference count; the more times a page is referenced, the higher its priority.
- The priority of the page is increased by one when you use the **lockPage()** method to lock a page.
- If you use the **setPriority()** method to set the priority of the image containing the page, the IL increases the priority of the page by one plus the value specified in **setPriority()** each time the page is referenced. The **setPriority()** definition is shown below:

  ```
  void setPriority(int priority);
  ```
- The maximum number of pages at each priority level is one eighth of the total number of pages in cache. If the number of pages at any one

priority level exceeds this limit, the priority level of the last page at that level is reduced by one. In other words, the page is moved to the head of the list at the next lower priority level.

You can use the *ilmonitor* utility to monitor the activity of pages in cache. See the *ilmonitor* reference page or “Image Tools” on page 351 for more information about ilmonitor.

Page Size

The page size for each operator is defined by its input images; for an ilFileImg, the page dimensions match those used to store the image on disk. Some images also let you set the size of the pages in the cache and the data type and ordering of the cached data. The data type and ordering affect how data is cached, so if you change these attributes, you might also want to change the size of the cache. To set the data type or the ordering of data in the cache, use the appropriate functions defined by ilImage, **setDataType()** and **setOrder()**. These functions are described in “Image Attributes” on page 34. “Managing Cache” on page 49 describes how to set the size of the cache.

Not all images allow you to set the page size; in fact, generally only operators do. If you change the page size of an image, you should follow the suggestions in “Cache Priority” on page 311.

To set the size of the pages used in the cache for a particular image, use **setPageSize()**, which is defined by ilImage as follows:

```
ilStatus setPageSize(int nx, int ny, int nz=1, int nc=0);
```

The arguments specify the *x* and *y* dimensions of the page in pixels. By default, the *z* dimension, or depth of the image, is 1, and the channel dimension matches that of the image. This function calculates the number of bytes needed to store a page with the specified dimensions.

You can use any of a number of functions to query an image about its page size, depending on whether you want the answer in page dimensions, bytes, or pixels:

```
int getPageSize();
int getPageSize(int& nx, int& ny, int& nz, int& nc);

int getPageSizePix();
int getPageSizeVal();
```

getPageSize() returns the page size in bytes; this function is overloaded as shown to take no arguments, or to take arguments that indicate locations into which the dimensions of the page in pixels are returned.

getPageSizePix()
returns the total number of pixels represented by a page; this value is found by multiplying the *x*, *y*, and *z* dimensions.

getPageSizeVal()
returns the total number of data elements represented by a page; this function multiplies the page's channel dimension by the value returned from **getPageSizePix()**.

Paging Support

The ilImage class provides functions to support paging in a multi-threaded environment. These functions allow you to lock pages to ensure that those pages stay in memory until you unlock them. The five virtual functions that control paging are:

```
virtual int hasPages();
virtual ilPage* lockPage(int x, int y, int z, int c,
          ilStatus& status, int mode=ilLMread);
virtual void unlockPage(ilPage* page);
virtual ilStatus lockPageSet(ilLockRequest* set,
          int mode=ilLMread, int count=1);
virtual void unlockPageSet(ilLockRequest* set, int count=1);
```

hasPages() returns TRUE for ilMemCacheImg and all of its descendants and FALSE for all other classes in the IL. This is useful for determining whether the ilImage in question supports paging.

lockPage()	is used to *lock down* the page located at *x*, *y*, *z*, and *c* in the cache; it returns a pointer to that page, which later must be passed to **unlockPage()** to free up that page again.
unlockPage()	frees the page specified by the pointer in the argument list.
lockPageSet()	processes a set of ilLockRequest structures and returns pointers to the requested pages in the structures.
unlockPageSet()	releases the set of pages obtained by the **lockPageSet()** function.

These methods provide a mechanism to bypass the overhead of **getTile()** and **setTile()**, but they require that you be aware of all of the attributes of the page: size, data type, and order.

Accessing Image Data

All classes derived from ilImage read, write, and copy image data using the same set of data access functions defined by the ilImage base class. Each derived class implements the functions as necessary to suit its particular requirements. A key feature of these functions is that they allow you to access any arbitrary rectangle, or tile, of image data, regardless of how that data is stored. This flexibility allows the IL's demand-driven execution model to be implemented. As part of this model, calls to some of these functions are generated automatically. However, you can also call these functions explicitly as needed. The execution model is discussed in detail in "The IL Execution Model" on page 68. The ilImage class defines both three-dimensional and, for convenience, two-dimensional data access functions, as shown in Table 2-2.

Table 2-2 Data Access Functions

Three-dimensional	**Two-dimensional**	**Description**

getTile3D()
setTile3D()
copyTile3D() or <<[a]
copyTileCfg()
	getTile()
	setTile()
	copyTile() or <<[a]
		reads, writes, and copies a tile of data

getSubTile3D()
setSubTile3D()
	getSubTile()
	setSubTile()
		reads and writes a subtile of data

getPixel3D()
setPixel3D()
	getPixel()
	setPixel()
		reads and writes a pixel

fillTile3D()	fillTile()	fills a tile with a constant value
seekTile3D()	seekTile()	finds and updates a tile in memory

a. << is the left-shift or output operator; it's redefined in the C++ version of the IL.

The two-dimensional data access functions work through their three-dimensional counterparts. Since the two-dimensional versions are slightly easier to comprehend, they're discussed first, in the next section. Two other sets of functions are also described below; one set provides support while accessing data, and the other helps manage coordinate space translations.

Two-dimensional Functions

The two-dimensional functions you're likely to use most frequently are **getTile()**, **setTile()**, and **copyTile()**. As their names suggest, these functions read (get), write (set), and copy a tile of data. They assume that the data buffer being read into or written from is the exact size necessary to hold the tile being read or written; if the buffer is larger use **getSubTile()** or **setSubTile()**. Another pair of functions, **getPixel()** and **setPixel()**, allow you to read and write pixels rather than tiles. The **fillTile()** function allows you to fill a two-dimensional tile of data with a specified constant value. The

seekTile() function updates a tile of data so that a subsequent access will find that tile in memory.

getTile() and setTile()

The calling sequences for **getTile()** and **setTile()**, which take the same arguments, are shown below:

```
ilStatus getTile(int x, int y, int nx, int ny, void*
               data,const ilConfig* config=NULL);
```

```
ilStatus setTile(int x, int y, int nx, int ny, void* data,
               const ilConfig* config=NULL);
```

As you might expect, **getTile()** retrieves a tile of data from a source image and places it in the location pointed to by *data*. This source image is the one whose **getTile()** function is called. The tile that's retrieved is specified by its origin in the source image (*x*,*y*) and its size (*nx* and *ny*), which is measured in pixels. (Since the tile's origin is specified in the image's coordinate space, the (*x*,*y*) point is specified relative to the image's origin.) The optional *config* argument allows you to change the configuration of the data (including the coordinate space) as it's read and placed in the buffer. If this argument isn't supplied, the configuration of the source image is used. One element of an ilConfig is an ordered list of the image's channels; see the section on **copyTile()** for an example of using this channel list to reorder channels as data is retrieved.

The **setTile()** function writes a tile of data from the location pointed to by *data* to the destination image. In this case, the destination image's **setTile()** is called. The location of the tile being written is specified by its origin in the destination image (*x*,*y*) and its size (*nx* and *ny*). The optional *config* argument for **setTile()** describes the configuration of the data being written; if necessary, the data is automatically reconfigured to match the configuration of the destination image. If this argument isn't supplied, it's assumed that the data being written already has the same configuration as the destination image.

copyTile()

The **copyTile()** function is an efficient way to copy a tile of data from one ilImage to another:

```
ilStatus copyTile(int x, int y, int nx, int ny,
             ilImage* other, int ox, int oy,
             int* chanList=NULL, int from=1);
```

By default, the tile is copied to the calling image from the image pointed to by *other*. The *x* and *y* arguments specify the origin of the tile in the destination image, and *nx* and *ny* specify the size of the tile. The tile that's to be copied is located at (*ox*,*oy*) in the *other* image. (If the tile is at the same location in both the source and destination images, then *x*=*ox* and *y*=*oy*.) If the source and destination images have different coordinate spaces, the data is transposed automatically as necessary. The last argument, *from*, allows you to reverse the direction of the copy; if it's 0, the tile is copied to *other* from the calling image.

The default direction (from *other* to the calling image) is the most efficient direction when you're copying to a file image (that is, one that inherits from ilFileImg). See "Cache Priority" on page 311 for an explanation.

No configuration argument is needed for **copyTile()** because the destination image's configuration is always used; data is automatically converted as necessary to match the destination image's data type, order, and coordinate space. However, you can choose a subset of the source image's channels and / or reorder them using the optional *chanList* argument. This argument is an **int** array that specifies a channel mapping between the *other* image and the calling image; its interpretation is the same, regardless of the direction of the copy. The number of entries in the array should always match the number of channels in the calling image; a negative one (-1) in the array means that no data will be written for that channel.

As an example, suppose you have an RGB image (with red, green, and blue channels) that you want to display as an ABGR image (with alpha, blue, green, and red channels). (You can do this most simply with the color conversion operators that derive from ilColorImg, but this example is presented for discussion purposes.) The code for accomplishing this with **getTile()** and **setTile()** is:

```
/* allocate the data buffer */
int xsize = 20;
int ysize = 10;
char data[xsize*ysize*3];

/* specify the channel list and configuration */
static int chans[] = {3, 2, 1};
ilConfig config(ilUChar, ilInterleaved, 3, chans);

/* read the data from one image and write it to the other */
RGBImg.getTile(0, 0, xsize, ysize, data);
ABGRImg.setTile(0, 0, xsize, ysize, data, &config);
```

First, a buffer, *data*, is allocated to hold the 20-pixel by 10-pixel three-channel tile as it's copied. Next, the configuration that the data should be mapped into is specified. The channel list *chans* maps the channels of the RGB *data* to the channels of the ABGR image, as explained below. (Keep in mind that channels are numbered beginning with 0 and that there is no channel offset.)

- Channel 0 of the RGB *data* (the red channel) is mapped to channel 3 of the ABGR image (also the red channel).
- Channel 1 of *data* (green) is mapped to channel 2 of the ABGR image (also green).
- Channel 2 of *data* (blue) is mapped to channel 1 of the ABGR image (blue).
- Nothing is available to map to channel 0 of the ABGR image (the alpha channel).

Finally, the data is read into the buffer from *RGBImg*, and then it's written to *ABGRImg* from the buffer.

Here's what the code looks like if **copyTile()** is used:

```
int xsize = 20;
int ysize = 10;
static int chans[] = {-1, 2, 1, 0};

ABGRImg.copyTile(0, 0, xsize, ysize, RGBImg, 0, 0, chans);
```

In this case, no intermediate data buffer needs to be allocated; the tile is copied directly from RGBImg to ABGRImg. The channel list specifies how the channels of RGBImg are mapped to those of ABGRImg, as shown in Table 2-3.

Table 2-3 Channel Mapping

Channel List	RGBImg Channel	ABGRImg Channel
-1	none	0 (alpha)
2	2 (blue)	1 (blue)
1	1 (green)	2 (green)
0	0 (red)	3 (red)

The interpretation of the channel list is the same if the direction of the copy is reversed. If the same channel list were used with a call to **copyTile()** that specified 0 as the direction argument, data would be copied from ABGRImg to RGBImg as follows:

- Channel 0 of ABGRImg isn't copied at all.
- Channel 1 of ABGRImg is copied to channel 2 of RGBImg.
- Channel 2 of ABGRImg is copied to channel 1 of RGBImg.
- Channel 3 of ABGRImg is copied to channel 0 of RGBImg.

If you need to offset channels you need to use **copyTileCfg()** instead of **copyTile()**. This function is discussed in "Three-dimensional Functions" on page 63. To force a two-dimensional interpretation of **copyTileCfg()**, specify zero values for the *z*, *nz*, and *oz* parameters.

The Left-Shift or Output Operator, <<

The C++ language allows you to overload the definition of operators as long as the new definition takes at least one class argument. The IL overloads the operator **<<** so that it requires a reference to an ilImage as an argument and so that it becomes a shorthand for **copyTile()**. Here's how you invoke this operator (assume the two ilImages *srcImage* and *destImage* are already created):

```
destImage<<srcImage;
```

This operator copies *srcImage*'s data to *destImage*, aligning the data with *destImage*'s origin. If the two images are different sizes, as much data as possible is copied. This operator works for two- and three-dimensional images.

getSubTile() and setSubTile()

One limitation of **getTile()** and **setTile()** is that the data buffer must be the exact size needed to hold the data being read or written. If the buffer you're reading data into or writing it from is larger than the tiles being read or written, use **getSubTile()** or **setSubTile()** to specify a subtile of the larger buffer. (Be sure the buffer is at least as large as the tile being read or written and that the tile is completely contained in the buffer.) The calling sequences for these functions are shown below:

```
ilStatus getSubTile(int x, int y, int nx, int ny, void* data,
               int dx, int dy, int dnx, int dny,
               const ilConfig* config=NULL);

ilStatus setSubTile(int x, int y, int nx, int ny, void* data,
               int dx, int dy, int dnx, int dny,
               const ilConfig* config=NULL);
```

The *x, y, nx, ny, data*, and *config* parameters have the same meanings as they have in **getTile()** and **setTile()**. The remaining parameters specify the origin of the *data* buffer (*dx*,*dy*) relative to the image and the size of the buffer (*dnx* and *dny*), as shown in Figure 2-9. (This figure assumes that the image's coordinate space defines the origin as the lower left corner.) With either function, if the *data* buffer is the same size as the source tile, then *x*=*dx*, *y*=*dy*, *nx*=*dnx*, and *ny*=*dny*.

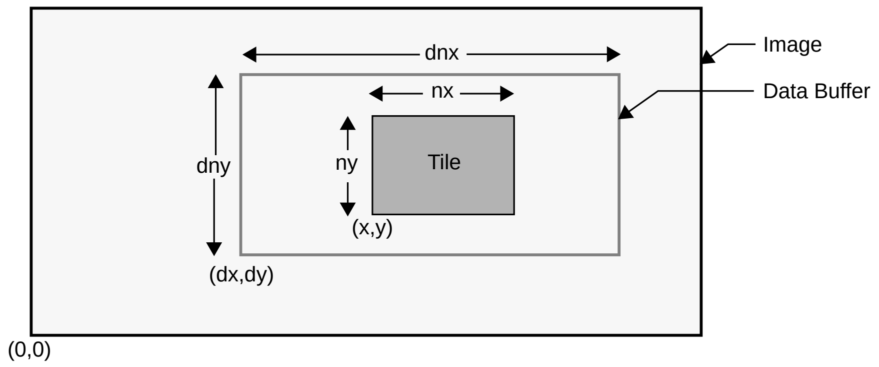


Figure 2-9 Parameters for **getSubTile()** and **setSubTile()**

getPixel() and setPixel()

If you'd rather read or write pixels than tiles, use **getPixel()** or **setPixel()**:

```
ilStatus getPixel(int x, int y, ilPixel& pix);
ilStatus setPixel(int x, int y, ilPixel& pix);
```

These functions read or write the pixel at location (*x*,*y*) in the calling image. When a pixel of data is read, it's placed in the location referenced by *pix*. The *pix* argument for **setPixel()** references the data that's written into the calling image at (*x*,*y*).

fillTile()

As a special case of writing a tile of data, you can set an arbitrary rectangular area of an image to a constant value with **fillTile()**:

```
ilStatus fillTile(int x, int y, int nx, int ny,
              void* data, const ilConfig* config=NULL,
              const ilTile* fillMask=NULL);
```

The rectangular area to be filled is specified by its origin (*x*,*y*) and size (*nx* and *ny*), measured in pixels. The *data* argument specifies the value used to fill the tile; it's typically an ilPixel object (for C++ programmers). For example, to fill a tile with white, use an ilPixel with these values: 255, 255, 255. The optional *config* argument describes the configuration of *data*; if it's omitted, *data* is assumed to have the same configuration as the image being filled.

The last argument, *fillMask*, allows you to define a mask that prevents a portion of the tile from being filled. (See “Auxiliary Classes” on page 396 for a detailed description of the ilTile class.) If it’s not NULL, only the portion outside of the *fillMask* is filled.

seekTile()

The **seekTile()** function, still under development, will allow you to prefetch data to ensure smooth performance. You might use this function if the user of your application roams across an image in a particular direction, and you want to maximize performance by making sure that the tiles the user is approaching will be in memory before they need to be displayed. To ensure that an upcoming memory access will find a particular tile, call **seekTile()**:

```
ilStatus seekTile(int x, int y, int nx, int ny,
               const ilConfig* config=NULL,
               ilSemaphore* sem=NULL,
               ilImage* target=NULL);
```

The *x, y, nx, ny,* and *config* parameters have the same meaning as they have in **getTile()** and **setTile()**. There is no *data* parameter, since the tile specified isn’t being written to or read from a buffer; it’s merely being updated in the image’s cache. The *sem* and *target* parameters are not currently implemented.

Prefetching

The IL contains a prefetch feature that improves the speed and smoothness of some display operations. The prefetcher attempts to predict new pages that are going to be required as a user pans smoothly over a displayed IL image. It accomplishes this prediction by recording the coordinates of a series of **copyTile()** operations to an ilDisplayImg and then making a linear prediction of future **copyTile()** requests. Based on this prediction, input pages are scheduled to be interleaved with the normal **copyTile()** operations. For instance, suppose you are roaming over an image having page size 64 x 64 pixels. The size of each frame is 512 x 512 pixels and you offset your position by one pixel in the *x* direction and one pixel in the *y* direction in each frame. In this case, every 64 frames, 17 new pages need to be loaded into the cache. If this is done all at once on the 64th frame, there is a visible delay on this frame. The prefetcher detects that the image is being linearly translated in the displayed window and distributes the 17 lockPages evenly over the 64 frame interval so that one extra lockPage is done about every 4 frames. On the 64th frame all the necessary new pages are already resident in the cache and there is no hesitation.

The prefetch feature is enabled by default. You can use the **ilEnablePreFetch()** function to disable prefetching and the **ilPreFetchIsEnabled()** feature to check the status of prefetching.

```
void ilEnablePreFetch(int enable);
```

```
int ilPreFetchIsEnabled();
```

You can also use the environment variable IL_ENABLE_PREFETCH to enable or disable prefetching.

```
setenv IL_ENABLE_PREFETCH 0
```

The prefetcher works best with small, mostly square page sizes. If you use unusual page sizes, its behavior can be erratic and you may wish to disable it. The behavior of the prefetcher can be observed using the ilmonitor tool. See “Image Tools” on page 351 for more information about ilmonitor.

Three-dimensional Functions

The three-dimensional data access functions are the same as their two-dimensional counterparts, except that they take extra arguments as necessary to handle an image’s *z* dimension. For example, **getTile3D()**, **setTile3D()**, and **copyTile3D()** take arguments to specify the origin and size in the *z* dimension:

```
ilStatus getTile3D(int x, int y, int z, int nx, int ny,
        int nz, void* data, const ilConfig* config=NULL);
```

```
ilStatus setTile3D(int x, int y, int z, int nx, int ny,
        int nz, void* data, const ilConfig* config=NULL);
```

```
ilStatus copyTile3D(int x, int y, int z,
        int nx, int ny, int nz,
        ilImage* other, int ox, int oy, int oz,
        int* chanList=NULL, int from=0);
```

The overloaded left-shift or output operator **<<** works for three-dimensional images as well as two-dimensional ones, as described above.

The **copyTileCfg()** function works similarly to the **copyTile3D()** function, except that it allows the channels of the copied data to be offset as well as reordered at the same time it is being copied:

```
virtual ilStatus copyTileCfg(int x, int y, int z,
        int nx, int ny, int nz,
        ilImage* other, int ox, int oy, int oz,
        const ilConfig* config=NULL, int from=1);
```

Note that this function takes an ilConfig argument rather than an **int***. Only fields in the ilConfig that refer to the number of channels, channel list, and channel offset are used during the copy; the other fields are ignored.

The **getSubTile3D()** and **setSubTile3D()** functions require several additional arguments to specify the origin and size of the *z* dimension in both the source and the destination:

```
virtual ilStatus getSubTile3D(int x, int y, int z,
        int nx, int ny, int nz,
        void* data, int dx, int dy, int dz,
        int dnx, int dny, int dnz,
        const ilConfig* config=NULL) = 0;
virtual ilStatus setSubTile3D(int x, int y, int z,
        int nx, int ny, int nz,
        void* data, int dx, int dy, int dz,
        int dnx, int dny, int dnz,
        const ilConfig* config=NULL) = 0;
```

The pixel functions take an additional *z* component to the origin specification:

```
ilStatus getPixel3D(int x, int y, int z, ilPixel& pix);
ilStatus setPixel3D(int x, int y, int z, ilPixel& pix);
```

The **fillTile3D()** and **seekTile3D()** functions take arguments that are similar to the two-dimensional versions:

```
virtual ilStatus fillTile3D(int x, int y, int z,
        int nx, int ny, int nz,
        void* data, const ilConfig* config=NULL,
        const ilTile* fillMask=NULL);
```

```
virtual ilStatus seekTile3D(int x, int y, int z,
        int nx, int ny, int nz,
        const ilConfig* config=NULL,
        ilSemaphore* sem=NULL);
```

Data Access Support Functions

This section discusses a few functions designed to perform tasks related to accessing data. These functions help you step through a buffer of image data, **getStrides()** and **getStrides3D()**, or clip a tile to the dimensions of the image, **clipTile()**.

getStrides() and getStrides3D()

In some situations, you might want to step through a buffer of image data pixel by pixel, rather than simply reading or writing a single tile of data. Or you might want to move some specific number of pixels in a particular direction. To do this, you need to know where one pixel's data ends and the next one's begins. This information, called the *stride*, depends on the image's data type, pixel ordering, and the size of the data buffer. The two functions, **getStrides()** and **getStrides3D()**, return data strides by reference:

```
void getStrides(int nx, int& xs, int& ys,
      int& cs, int& nc, ilOrder ord=numilOrders);
```

```
void getStrides3D(int nx, int ny, int& xs, int& ys, int& zs,
      int& cs, int& nc, ilOrder ord=numilOrders);
```

You specify the size of the data buffer (*nx* and *ny* in the three-dimensional case) and the pixel ordering, *ord*. The default value numilOrders means that the calling image's ordering should be used. The remaining values are returned by reference:

- *xs*, the x stride, steps to the next pixel in the same row.
- *ys*, the y stride, steps to the next pixel in the same column.
- *zs*, the z stride, steps to the next pixel along the z axis at the same xy location.
- *cs*, the channel stride, steps to the next channel of the same pixel.
- *nc* is the number of channels in the data.

clipTile()

Another useful function, **clipTile()**, clips a specified tile to an image's boundaries:

```
ilStatus clipTile(int& x, int& y, int& z,
        int& nx, int& ny, int& nz,
        int includeBorder=FALSE);
```

The arguments specify by reference the origin (*x*, *y*, *z*) and size (*nx*, *ny*, *nz*) of the tile. The *includeBorder* argument specifies whether the page borders of the image should be used to determine clipping. If *includeBorder* is TRUE, the clipped tile will include a border at the edge of the image whose size is determined by the IL (or by **setPageBorder()** if you choose to use this function). If *includeBorder* is FALSE (which it is by default), the tile is clipped to the actual image edge, not including any borders. If any part of the tile lies outside the image's boundaries, the corresponding argument is adjusted as necessary to clip the tile. You can then use the parameters in a call to **getTile()** or **setTile()**, for example. If the tile is clipped, **clipTile()** returns ilDATACLIPPED; otherwise, it returns ilOKAY.

Coordinate Space Support

Several functions are defined to help you translate image data from one coordinate space to another:

```
ilCoordSpace mapFlipTrans(ilCoordSpace fromSpace,
        ilFlip& flip, int& transXY,
        ilCoordSpace workSpace=ilCoordSpace(0));

void mapTile(ilCoordSpace fromSpace, ilTile& tile,
        ilFlip& flip, int& transXY,
        ilCoordSpace workSpace=ilCoordSpace(0));

void mapTile(ilCoordSpace fromSpace, ilTile& tile,
        ilCoordSpace workSpace=ilCoordSpace(0));

void mapXY(ilCoordSpace fromSpace, int& x, int& y,
        ilCoordSpace workSpace=ilCoordSpace(0));
void mapXY(ilCoordSpace fromSpace, float& x, float& y,
        ilCoordSpace workSpace=ilCoordSpace(0));

void mapXYSign(ilCoordSpace fromSpace, float& x, float& y,
        ilCoordSpace workSpace=ilCoordSpace(0));
ilCoordSpace mapSpace(int flipX, int flipY,
        int transXY=FALSE);

void getSize(ilSize &sz, ilCoordSpace workSpace);

int isMirrorSpace(ilCoordSpace otherSpace,
        ilCoordSpace workSpace=ilCoordSpace(0));
```

The **mapFlipTrans()** function determines the flips and / or transpositions necessary to map coordinates from the *fromSpace* coordinate space to *workSpace* (and returns them by reference). The **mapTile()** and **mapXY()** functions map the specified *tile* or (*x*,*y*) point from *fromSpace* to *workSpace*. The **mapXYSign()** function reverses the sign of the (*x*,*y*) values if *workSpace* is flipped with respect to *fromSpace*; it also swaps the values (that is, exchanges *x* for *y* and vice versa) if the coordinate spaces are transposed. The **mapSpace()** function returns the coordinate space that results from performing the specified flips and / or transpositions. The other two functions return information related to an image's coordinate space. The **getSize()** function maps the image's size to the *workSpace* coordinate space and returns it by reference. The **isMirrorSpace()** function returns whether *otherSpace* is a mirror image of *workSpace*.

For more information about these functions, see the ilImage reference page.

Geometric Mapping Support

These four functions are defined in ilImage to support functionality in image processing operators that perform geometric transformations:

```
void mapToSource(ilXYfloat& src, const ilXYfloat& self);
void mapFromSource(ilXYfloat& self, const ilXYfloat& src);
virtual void evalXY(ilXYfloat& xy, const ilXYfloat& uv);
virtual void evalUV(ilXYfloat& uv, const ilXYfloat& xy);
```

mapToSource() transforms the coordinates in *self* into the ultimate source image's coordinate space and places them in *src*. **mapFromSource()** transforms the coordinates in *src* into the calling image's coordinate space and places them in *self*. **evalXY()** maps from the calling image's coordinate space to the immediate input image's coordinate space, and **evalUV()** maps from the immediate input image's coordinate space to the calling image's coordinate space.

The IL Execution Model

This section describes the IL execution model and explains in general how it works in an IL program. Features of the IL execution model are:

- on-demand processing of image data using chains of IL operators
- multi-threading to allow some portions of an IL program to execute in parallel
- the use of hardware acceleration hardware whenever possible to improve the performance of operators in an IL chain

The IL incorporates these features into your program automatically. You don't have to do anything to make them happen. You will, however, need to understand them to tune your program for optimum performance.

On-demand Processing

In the IL's execution model, image data is processed only on demand. This technique minimizes both the need to store intermediate results and the frequency of disk input and output operations so that overall program performance is optimized. IL programs that apply multiple successive image processing operators or that deal with large images especially benefit from this execution model. (An operator is a class derived from ilOpImg that applies its image processing algorithm to the data encapsulated in an ilImage object. See Chapter 4, "Operating on an Image," for more information.)

An IL program implements the demand-driven execution model in two stages:

1. It creates a chain of image processing operators by creating the desired operator classes.
2. It pulls data through the chain as it is needed. The impetus for pulling data through the processor chain is the need for the image data at the end of the chain, either for display or storage back to disk. The data is pulled by processing one to several pages at a time.

In the IL program listed in "A Sample Program in C++" on page 4, a relatively simple image chain is constructed. Figure 2-10 shows this chain, with arrows indicating the path that image data follows as it's read from disk, processed (sharpened and rotated), and then both displayed and written back to disk.

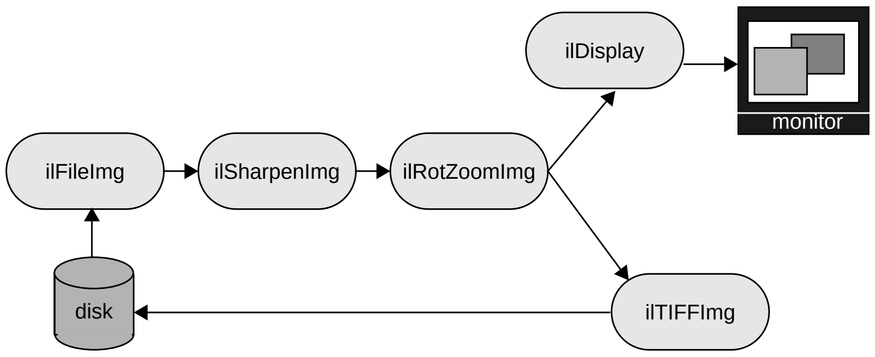


Figure 2-10 Image Chain for the Sample Program

An image processing library that uses a conventional execution model shuffles data in and out of memory at each stage of the chain. Such a program:

1. Reads the initial image data from disk into a buffer
2. Sharpens it
3. Writes the sharpened data into a different buffer
4. Rotates the sharpened data
5. Writes this final, processed data into yet another buffer
6. Writes the final data into the framebuffer and back to disk

If the image is too large to be cached in memory, a conventional library will write at least some of the processed data to disk for each intermediate stage. This data then needs to be read back in from disk for the next stage.

In contrast, the IL pulls one or several pages of image data at a time all the way through the chain; after a page is completely processed—in this example, read from disk, sharpened, rotated, displayed, and written to a file on disk—the next page is pulled through the chain. When multi-threading is enabled, several pages can be in process through the chain at any one time. This execution model eliminates the need to save intermediate processing results for all images, regardless of their size. Thus, all those intermediate buffers don't need to be allocated. The IL's model also minimizes startup time for IL programs, particularly those that allow the user to roam around a large image. The data for the entire image isn't processed before startup; it's processed only as needed, which in this case is as the user roams.

The backward red and blue arrows in Figure 2-11 show how data is pulled through the image chain in the sample program.

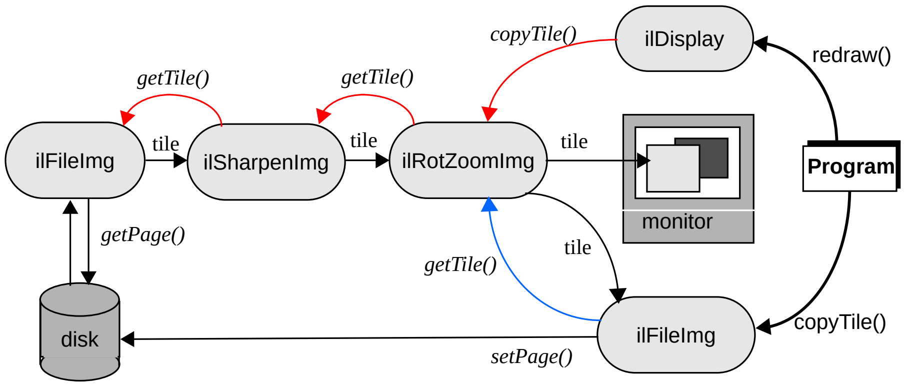


Figure 2-11 Image Chain Showing Demand-driven Execution Model

In this example, the **redraw()** and **copyTile()** function calls issued by the program instigate the processing of image data. They cause successive tiles of image data to be pulled through the chain and sent to the display or back to the disk. As each tile is written, another tile is requested from the previous stage of the chain with a **getTile()**, **copyTile()**, or **getPage()** call. If the tile requested doesn't already reside in the cache, the page containing that tile is pulled through the chain—read from disk, sharpened, and rotated. The ilDisplay class manages the transfer of data from the end of the chain to the framebuffer.

In the sample program in Chapter 1, "Writing an ImageVision Library Program," the instigating functions—ilDisplay's **redraw()** and ilTIFFImg's **copyTile()**—are actually called in the program. The other function calls are generated automatically as the program executes. Thus, only data that's actually needed is pulled through the chain.

This particular sample program displays and writes to disk the entire processed image, a tile at a time. Other image processing programs might not even process an entire image. For example, suppose that instead of simply displaying the entire final image, the program allowed the user to roam around the image, viewing only a fraction of it at a time. This kind of user interface is typically provided with programs that deal with huge images. Since IL programs process data only as it's needed, only those portions of the image that the user demands to see are processed. It's quite possible—often the case, in fact—that the user will never view some portions of a large image, and those portions won't be read from disk or processed. Thus, the IL helps minimize your program's overall processing requirements.

Multi-threading

The multi-threading part of the IL's execution model optimizes overall program performance by allowing portions of an IL program to execute in parallel. For example, when a tile covering several pages is copied from one operator to the next in a chain and the tiles are not resident in cache, they must be fetched from disk. The IL implements the parallel fetching of pages by queueing a request for each page and creating a process thread to service each request.

Figure 2-12 shows how long it takes to read in and perform computations on four pages in a non-multi-threaded application, a multi-threaded application running on a single-processor machine, and a multi-threaded application running on a multiple-processor machine. As you can see, the multi-threaded applications complete this transaction more quickly than the non-multi-threaded application.

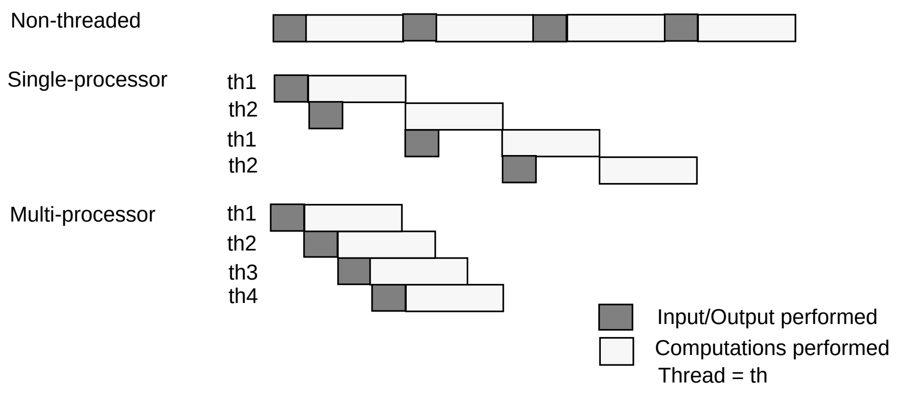


Figure 2-12 Performance Comparison of Non-threaded, Single-processor, and Multi-processor Applications

The IL supports parallel execution on single- and multiple-CPU machines by creating process threads that execute portions of an IL program simultaneously. This multi-threading facility is implemented transparently and automatically: there are no special function calls to make or header files to include. When you derive new classes from the existing classes in the library, however, you must ensure that the code you produce is *reentrant*, or able to be called from several process threads running concurrently. You will find information in Chapter 6, “Extending the IL,” about how to do this.

When debugging your application or linking with other libraries that perform multi-threading, you may want to turn off the IL’s multi-threading facility. The preferred way to do this is to set the environment variables IL_COMPUTE_THREADS and IL_SPARE_THREADS to zero by using the convenience function **ilMpSetMaxProcs()**, as shown below. This is a global function that doesn’t belong to any class.

```
ilMPSetMaxProcs(0,0);
```

More specific information on how to adjust the IL’s multi-threading facility is located in “Controlling Multi-threading” on page 336.

How Multi-threading Works

When the IL processes a **getTile()** or **copyTile()** call, it determines the pages needed for the requested tile and dispatches a request for each page. It then maintains these requests in a queue and creates process threads to service the queue. Figure 2-13 illustrates the concept of multi-threading as well as the on-demand processing described in the preceding section.

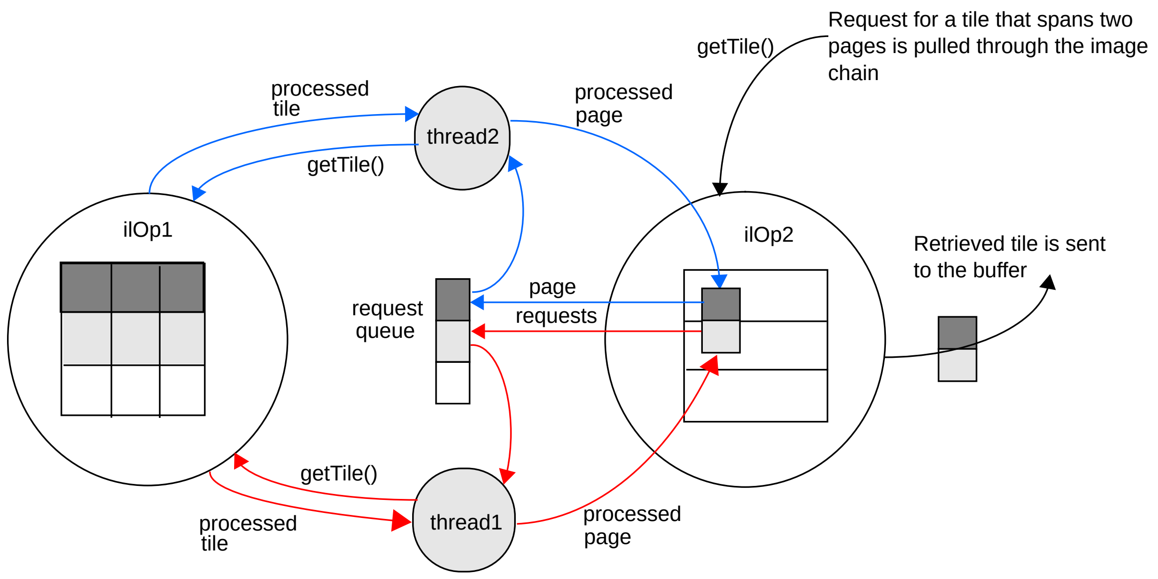


Figure 2-13 Operators, Requests for Pages, and Threads

Using Graphics Hardware for Acceleration

The hardware acceleration facility built into the IL allows your application to automatically take advantage of specialized graphics hardware whenever possible in order to make certain operations more efficient. It does this by performing one or more operations at the end of a chain in the graphics hardware instead of the CPU. On some architectures, it does this by reserving part of the framebuffer as an auxiliary buffer for the IL. Computations are then performed on the data stored in the framebuffer and displayed more quickly than if the data were being operated on in the CPU and brought in from main memory. If the IL needs a tile that is not in the reserved part of the framebuffer, the tile is brought into the framebuffer from main memory. This model is implemented transparently and automatically; there are no header files to include or function calls to make.

Disabling Hardware Acceleration

Sometimes you'll need to disable the hardware acceleration facility. For instance:

- when you are debugging your program. You cannot debug with this facility enabled if the operator you need to test is a CPU operation that is accelerated in the hardware.
- when you need more accurate results. Computing some operations in the CPU (for example, those that require a resampling method) gives more accurate results at the expense of speed.
- if your IL application includes certain GL calls, such as **scrmask()**, **pixmode()**, **logicop()**, **blendfunction()**, **ortho()**, **viewport()**, or texture loading calls. These GL routines depend on state variables that the IL can change at any time. (You can safely use hardware acceleration in conjunction with these GL calls, but only if you restore the GL state after making IL calls.)

You can enable and disable the hardware acceleration facility:

- globally for all features of the IL
- for a specific objects of an operator class
- for all objects of a specified class

"Controlling Hardware Acceleration" on page 315 gives detailed information about how to enable and disable hardware acceleration.

Using a Dedicated GL Thread for Hardware-Accelerated Rendering

The IL allows you the option of creating a dedicated thread to perform all hardware-accelerated rendering operations. You can enable the use of this thread with the **ilHwThreadEnable()** function. The default is that no dedicated thread is used.

Using a dedicated thread can improve performance in intensive rendering situations but it also requires extra effort to avoid collision with user-initiated rendering operations. The simplest way to ensure that the rendering thread doesn't collide with user rendering is to bracket all user rendering with calls to **ilHwSuspend()** and **ilHwThreadResume()**. See "Using a Dedicated GL Rendering Thread" on page 332 for more information about using a dedicated rendering thread.

Page Borders

To be accelerated in the graphics hardware, some image processing operations (specifically, those that perform image warps) require the data in the pages of the cache to overlap somewhat. A set of *page borders* determines how much the pages in the cache can overlap for these operations. The page borders are set automatically for you by the IL and should rarely be changed. You can use the **setPageBorder()** and **getPageBorder()** functions to query and set page borders.

Working with Image Chains

Your IL programs always contain image chains—the string of operators that define the kinds of operations you want performed on your images and the order in which these operations are to be performed. You can manipulate these chains after they are created.

Dynamically Reconfiguring a Chain

Some IL programs need to construct new image chains dynamically, as the program executes. For example, imagine a program with a graphical user interface that allows its user to specify input images and select operations to be performed on them. Once processing has been performed, the user can choose to operate further or to start again with new images and operators. Such a program is most easily implemented by taking advantage of the IL's facility for reconfiguring an image chain.

Each image in a chain maintains two lists, one of the images directly preceding it in the chain (its inputs or parents) and one of the images succeeding it in the chain (its children). In the chain shown in Figure 2-14, for example, the ilRotZoomImg object has one parent, the ilSharpenImg object, and one child, the ilTIFFImg object.

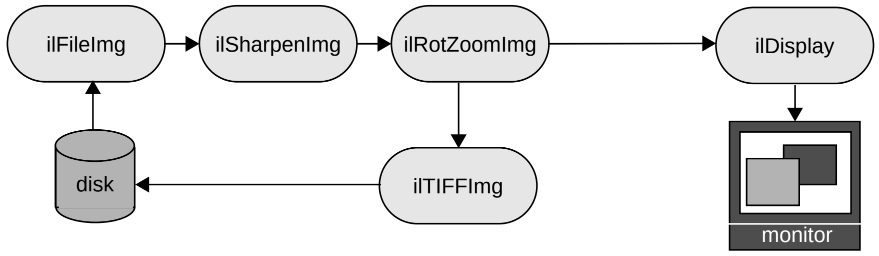


Figure 2-14 An Image Chain

The lists start numbering at 0; that is, the first item on the list is at index 0.

Note: An ilDisplayImg object is a special kind of image, and it isn't considered a child. (An ilView created by ilDisplay is the child of ilRotZoomImg).

Replacing a Chained Operator

Let's say you want to modify the sample program so that it can dynamically add a threshold operator in place of the ilSharpenImg operator. The ilThreshImg operator examines each pixel in an image and sets the pixel to a new value, depending on whether its value is higher or lower than a specified threshold value. If a pixel is higher than or equal to the threshold, it's set to the image's maximum pixel value; if the pixel is lower, it's set to the minimum value.

Here's what the code might look like to replace the ilSharpenImg operator with an ilThreshImg operator (this code can be inserted just before step 3 in the sample program in Chapter 1, "Writing an ImageVision Library Program"):

```
// set the threshold value to 127.5
float threshValue = 127.5;
ilPixel threshPixel(ilFloat, 1, &threshValue);

// create the ilThreshImg operator
ilThreshImg myThresher(inImg, threshPixel);

// replace ilSharpenImg with ilThreshImg
rotatedImg.setInput(&myThresher);
```

This example is simplified, but it demonstrates the use of **setInput()** to reconfigure a chain. A more realistic program will let the user specify the threshold value to be used and also might let the user specify any of a number of different operators to be replaced or added to the chain.

In this code fragment, the threshold value is explicitly set to 127.5 (which is halfway between the 0 and 255 endpoints for a standard RGB image), and an ilPixel object is created with this value. Next, the ilThreshImg operator is created and given the input image *inImg* (which is the ilFileImg created in the sample program to read an image file from disk) and the ilPixel.

The **setInput()** function removes the ilSharpenImg operator from the chain by replacing it with the new ilThreshImg operator. This function, which is declared in ilImage, takes a pointer to the new, already created input ilImage as its first argument. In this example, the ilThreshImg operator is now the input image for the ilRotZoomImg object, *rotatedImg*. The old input, which in the sample program was an ilSharpenImg, isn't deleted by the IL, so you might want to delete it if it isn't needed anymore. The attributes of the new input image are propagated down the operator chain as described in "Propagating Image Attributes" on page 78.

A second, optional argument for **setInput()** is of type **int**, and it specifies the position on the indexed list of inputs at which the input is to be added. By default, this argument is 0, indicating the first position on the list. Before the **setInput()** call, the ilSharpenImg operator occupies position 0 on ilRotZoomImg's list of inputs. Afterward, the ilThreshImg operator is at position 0, having replaced the ilSharpenImg operator.

Querying Chained Images

Although you probably won't frequently need to query a chained image about the operators it's chained to, the ilImage base class defines functions for you to do so. The function **getNumInputs()** returns an **int**, indicating the number of inputs or links backward; **getNumChildren()** (inherited from ilLink) returns the number of children or links forward.

You can also obtain a pointer to the preceding or succeeding linked images:

```
ilImage* myInput;
ilImage* myChild;

myInput = theImg.getInput(0);
myChild = theImg.getChild(0);
```

As its name implies, **getInput()** returns a pointer to the ilImage preceding it in the chain; **getChild()** (inherited from ilLink) returns a pointer to the ilImage succeeding it. Since there can be multiple inputs and children, both of these functions allow you to specify the indexed position of the image you wish to retrieve. By default, this argument is 0, indicating the first position on the indexed list.

Adding and Removing Inputs

Some operator images can have a variable number of inputs. For such operators, you may need to dynamically change the number of inputs as a chain is reconfigured. The two functions that are provided for this purpose are shown below:

```
ilStatus addInput(ilImage* img);
ilStatus removeInput(int index = 0);
```

The **addInput()** function adds the ilImage supplied as an argument to the end of its current list of inputs. As its name suggests, **removeInput()** removes the ilImage located at the specified index from its list of inputs. The ilImage removed from the chain isn’t deleted, so you might want to delete it if it won’t be used anymore.

The **setNumInputs()** function sets the maximum number of inputs to the **int** passed in as its argument. Since this function is declared protected, you can use it only when you’re deriving a class from ilImage.

Propagating Image Attributes

One important property of image chains is that they propagate attribute values to succeeding stages of the chain. In other words, each stage of the chain receives some or all of the attributes of the preceding stage. The attributes that are propagated—image size, data type, order, coordinate space, color model, lookup table, page size, minimum and maximum pixel values, and the fill value—are defined in ilImage and discussed in “Image Attributes” on page 34.

Changing Image Attributes

Image attribute values can change, either from being set explicitly or as a result of performing an operation. You can override a propagated value by explicitly setting it (if the operator allows you to do so), in which case the IL discards any data residing in the cache so that only data with the correct attributes is processed.

Operators can restrict the values for certain attributes. A supported value won’t be overridden by an unsupported propagated one. In addition, chains can be constructed so that one link has more than one preceding link (for example, ilBlendImg blends two images). In these cases, the most appropriate value is propagated; usually, this is the largest (for the size attribute, for example) or the most general value.

Typically, if you’ve explicitly set an attribute value using one of the appropriate functions defined in ilImage, for example, **setDataType()** or **setPageSize()**, you don’t want it to be overridden automatically by a propagated one. The IL assumes this to be the case, so it keeps track of any attributes that you’ve set. These attributes won’t be allowed to change through propagation down the chain unless you indicate that they should be. To allow an attribute to change even though you’ve set it, call **clearSet()** (inherited from ilLink):

```
myImg.clearSet(ilIPdataType);
```

The argument to **clearSet()** can be any logical combination of the enumerated type ilImgParam, which is defined in the header file *il/ilImage.h* and discussed in more detail in Chapter 6, “Extending the IL.” For more information about how the propagation mechanism is implemented, see “Deriving from ilImage” on page 244.

Automatic Color Conversion of Inputs

If the input(s) to an operator does not match its color model (either as inherited from multiple inputs or as set by the user), then an ilColorImg is automatically inserted between the operator and its input(s). The ilColorImg converts any mismatched input to match the operator’s color model.

In some cases, this automatic conversion is not desired, especially for operators such as ilColorImg and ilFalseColorImg that perform color conversions as part of their operations. These operators can prevent the insertion of an ilColorImg by setting the member variable *allowDiffCM* to TRUE, either in their constructor or when they initialize their state. When *allowDiffCM* is TRUE, the operator must be prepared to handle inputs of any color model for proper operation to be guaranteed. The default value is FALSE.

Object Properties

The IL allows you to assign property values and associated property names to objects derived from ilLink and then to query these values. This feature allows you to tag an object with arbitrary attributes. A property value can be an integer, a floating point number, or a pointer. The property name is a character string.

The IL provides three scope levels for property values:

- ilInstanceScope – defines the scope as a specified object
- ilClassScope – defines the scope as an object class
- ilGlobalScope – defines a global scope

The IL provides several redundant functions to set and query property values. In each of these functions, a scope argument specifies the search range for property lookup. This argument can be any logically OR'ed combination of ilInstanceScope, ilClassScope, and ilGlobalScope. If ilInstanceScope is specified, the object's property set is searched. If ilClassScope is specified, the object's class property set is searched. Finally, if ilGlobalScope is specified, the global property set is searched. If more than one of the search scopes is specified, each of the specified scopes is searched in this order: the object instance scope, then the object class scope, then the global scope. The default value for scope is ilInstanceScope.

The functions provided for the property value feature refer to a property associated with a character string name or, alternatively, with an ilName pointer that is used as a search key. It is more efficient to lookup a property by ilName pointer than by a string because hashing is avoided. See the ilGlobalName reference page to find out how to obtain an ilName pointer from a string.

The **getIntProp()** functions return the integer property value associated with either the string *s* or an ilName pointer. These functions return 0 if no such property has been defined.

```
int getIntProp(Char *s, ilScope scope_ilInstanceScope);
int getIntProp(ilName* n, ilScope scope_ilInstanceScope);
```

The **getFloatProp()** functions return the float property value associated with the string *s* or an ilName pointer. These functions return 0 if no such property has been defined.

```
float getFloatProp(char* s, ilScope scope=ilInstanceScope);
float getFloatProp(ilName* n, ilScope scope=ilInstanceScope);
```

The **getPtrProp()** functions return the pointer property value associated with the string *s* or the ilName. These functions return NULL if no such property has been defined.

```
void* getPtrProp(char* s, ilScope scope=ilInstanceScope);
void* getPtrProp(ilName* n, ilScope scope=ilInstanceScope);
```

The **getProp()** functions return the property associated with the string *s* or an ilName pointer. These functions return NULL if no such property has been defined.

```
ilProperty* getProp(char* s, ilScope scope=ilInstanceScope);
ilProperty* getProp(ilName* n,
              ilScope scope=ilInstanceScope);
```

You can use one of the following **setProp()** functions to assign a property value to be associated with the string *s* or an ilName pointer. These functions return ilOKAY if *scope* is one of the following: ilInstanceScope, ilClassScope, or ilGlobalScope. Otherwise, it returns ilUNSUPPORTED. The object is not marked altered as a result of **setProp()**.

```
ilStatus setProp(char* s, int i,
             ilScope scope=ilInstanceScope);
ilStatus setProp(ilName* n, int i,
             ilScope scope=ilInstanceScope);
ilStatus setProp(char* s, float f,
             ilScope scope=ilInstanceScope);
```

```
ilStatus setProp(ilName* n, float f,
             ilScope scope=ilInstanceScope);
ilStatus setProp(char* s, void* p,
             ilScope scope=ilInstanceScope;
ilStatus setProp(ilName* n, void* p,
             ilScope scope=ilInstanceScope);
ilStatus setProp(char* s, const ilPropValue& val,
             ilScope scope=ilInstanceScope);
ilStatus setProp(ilName* n, const ilPropValue& val,
             ilScope scope=ilInstanceScope);
```

The **removeProp()** functions remove the property associated with the string *s* or the ilName pointer *n* from the specified property set.The object is not marked altered as a result of **removeProp()**.

```
ilStatus removeProp(char* s, ilScope scope=ilInstanceScope);
ilStatus removeProp(ilName* n,
             ilScope scope=ilInstanceScope);
```

The **getClassPropSet()** function returns a pointer to the property set associated with the object's class

```
ilPropSet* getClassPropSet();
```

The **getPropSet()** function returns a pointer to the object's property set.

```
ilPropSet* getPropSet();
```

Chapter 3

Accessing External Image Data

This chapter describes the six file formats supported by the ImageVision Library and the ways in which you can create and access image data in these formats.

Accessing External Image Data

This chapter describes how to use the IL to read and write image data from and to either a file on disk or memory. This chapter contains the following major sections:

- "Supported IL Image File Formats" on page 87 describes the six supported IL image file formats.
- "Using the IL Image File Formats" on page 91 tells you how to access data in the six file formats.
- "Importing and Exporting Image Data" on page 101 discusses how to import and export image data between the IL and other libraries or devices.

The IL classes discussed in this chapter are shown shaded in Figure 3-1.

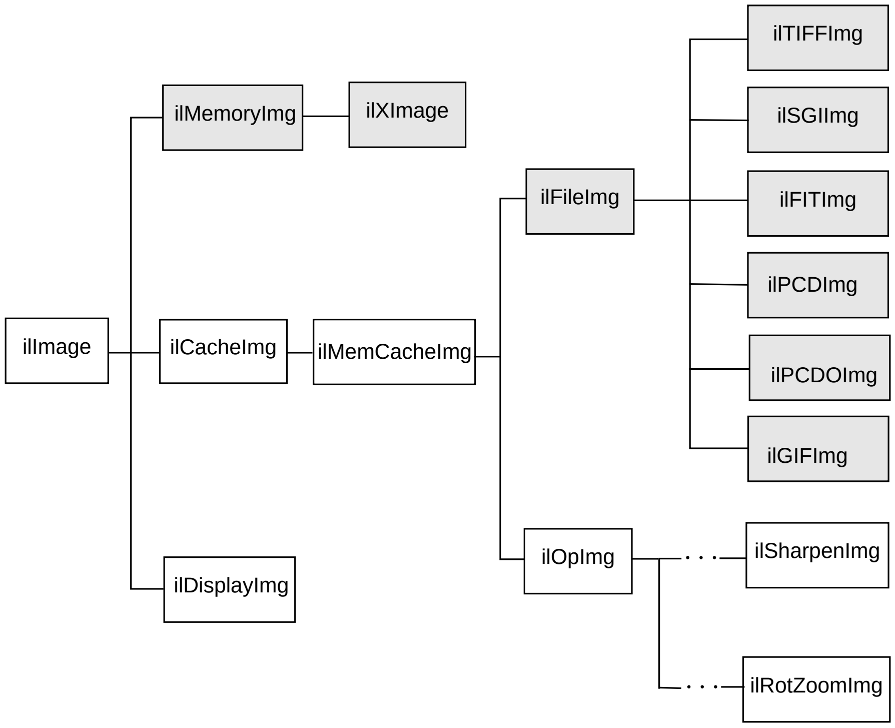


Figure 3-1 File and Memory Classes

The six classes ilTIFFImg, ilSGIImg, ilFITImg, ilPCDImg, ilPCDOImg, and ilGIFImg encapsulate the IL's support for these six file formats: TIFF, SGI, FIT, Photo CD Image Pack, Photo CD Overview Pack, and GIF, respectively.

You use the ilTIFFImg, ilSGIImg, and ilFITImg classes directly to read or write image data in these formats, as described in "Using the IL Image File Formats" on page 91. You can use the ilPCDImg class only to read data in the ilPCDImg format. The ilPCDOImg and ilGIFImg classes are also currently read-only.

The ilFileImg class, from which these six classes derive, provides the basic support needed to access data in any file format. To integrate support for your particular file format, you must derive a class from ilFileImg and implement the necessary functions. (See “Implementing Your Own File Format” on page 258 to learn more about deriving from ilFileImg.) The ilMemoryImg class allows you to import and export raw image data between the IL and other libraries or devices.

Supported IL Image File Formats

The following sections describe the six file formats supported by the ImageVision Library.

TIFF

The TIFF file format is an extended version of the Tag Image File Format, Revision 6.0. The purpose of TIFF is to describe and store raster image data. TIFF can describe bilevel, grayscale, palette-color, and full-color image data in several color spaces. TIFF includes a number of compression schemes that allow you to choose the best space or time trade-off for your applications. The IL uses these extensions to TIFF 6.0: Tilewidth, Tilelength, and SampleFormat. These tags provide necessary support for the image data types and tiles as defined by the IL.

The Introduction of this Programming Guide tells you how to obtain more information about the TIFF 6.0 specification. Refer to *il/ilTIFF.h* to see a list of the available tags and to the TIFF specification, generated by Aldus Corporation, to learn more about TIFF tags.

GIF

The GIF file format is used to read image files stored in the CompuServe Graphics Image File (GIF) format. To obtain more information about the GIF specification, contact CompuServe, Incorporated; Columbus, Ohio.

Kodak Photo CD Image Pack

The PCD file format supports image files produced by the Kodak Photo CD system. Photo CD establishes a system for storing high-resolution digital photographic images on compact discs. The Kodak Photo CD™ system:

- scans photographic film
- processes the resultant images (color correction, color encoding, hierarchical decomposition, and compression)
- records these images as a series of digitally coded images on a Kodak Photo CD disc

In addition to digital images, Kodak Photo CD can also produce digital psaudio data and playback control data. However, the IL only handles the image data files from a Photo CD disc.

The IL allows you to read Kodak Photo CD discs and process the images retrieved from the discs. Figure 3-2 shows the sequence of operations that occur as photographic film becomes an image in an IL file.

.

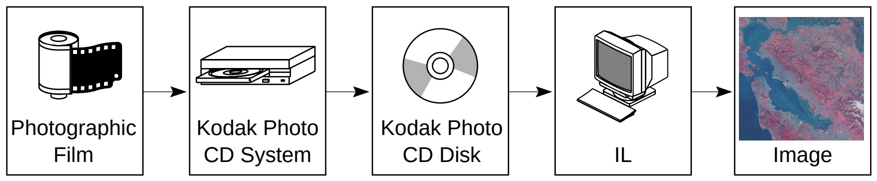


Figure 3-2 Processing Kodak Photo CD Images

Photo CD Images

A photographic image on a Kodak Photo CD disc is stored as a hierarchy of images, each of which represents the original image in a different resolution. This image hierarchy is stored in a structure called an image pack. You can get a maximum of six different resolutions of an image from an image pack. These resolutions are:

Base / 64	(96x64)
Base / 16	(192x128)
Base / 4	(384x256)
Base	(768x512)
4Base	(1536x1024)
16Base	(3072x2048)

An image pack file always contains the first four resolutions listed above. The last two resolutions, 4Base and 16Base, can be omitted when the Photo CD disc is created. Resolutions Base / 64 through Base are stored directly and can be accessed quickly. Resolutions 4Base and 16Base, if they are available in the image file, are stored in a compressed form.

You can use the inherited ilFileImg member function **getNumImgs()** to determine the number of images in your ilPCDImg file. You can use the **setCurrentImg()** and **getCurrentImg()** functions to select and query the current resolution. If you use **setCurrentImg()** to select an image resolution that does not exist in the image pack, the function returns the ilStatus value ilOUTOFBOUND but does not set the image’s status.

The IL determines page sizes for the varying Photo CD image resolutions in the following way:

- For the Base / 4, Base, 4Base, and 16Base resolutions, a page contains 16 scan lines.
- For the Base / 64 and Base / 16 resolutions, a page contains the whole image.

Photo CD Color Model

The color model of a Kodak Photo CD image is YCC. Photo YCC is a luminance / chrominance data metric that is based on video primaries and is designed to allow simple video display without compromising the colors available in photographic media. You can convert from the YCC color model to another color model using the IL. Currently, you cannot do the reverse, convert from another color model to YCC.

Kodak Photo CD Overview Pack

Every Kodak Photo CD contains a file in the Kodak Photo CD overview pack format. This format contains a low resolution representation of each image on the Photo CD. The ilPCDOImg class allows you to retrieve each of the overview images at either Base / 16 or Base / 64 resolution (the default is Base / 16).

You can use the inherited ilFileImg member function **getNumImgs()** to determine the number of images in your ilPCDOImg file. You can use the **setCurrentImg()** and **getCurrentImg()** functions to select and query the current resolution. If you use **setCurrentImg()** to select an image resolution that does not exist in the overview pack, the function returns the ilStatus value ilOUTOFBOUND but does not set the image's status.

SGI

SGI is the first format defined by Silicon Graphics for storing image data. SGI files are typically stored in files suffixed by *.bw*, *.rgb*, *.rgba*, *.sgi*, or *.screen*. SGI files support full color, color palette, and monochrome images of either one or two bytes per color component. Image data can be stored in either raw form or RLE (run-length encoding) form. You can create SGI files with compression but you cannot later rewrite a portion of a compressed SGI file.

Note: If an SGI formatted image is RGB Palette, its corresponding color map must be stored in a separate (also SGI formatted) file with the name *img.map*, where *img* is the name of the SGI image.

Page width for SGI files is the width of the image. Page height is a value in the range 16 through 32 that evenly divides the overall height of the image.

The SGI format makes the image order interleaved. SGI supports only unsigned data and a lower left coordinate space.

FIT

The FIT file format is a simple tiled format developed along with the IL. You might use FIT as a starting point for defining your own file format.

FIT supports the full flexibility of the IL model: all data types, orders, and page sizes. It uses a default page size of 128 x 128. FIT allows you to reserve space to hold user extensions to the file format. FIT is the only format that supports paging in the channel dimension, which is useful for multispectral imagery.

Code that implements the FIT file format is located in the */usr/people/4Dgifts/examples/ImageVision/ilsrc* file.

Using the IL Image File Formats

The IL allows you to read and write image data in any of the three file formats ilTIFFImg, ilSGIImg, and ilFITImg, and to read image data from the ilPCDImg, ilPCDOImg, and ilGIFImg file formats. You can also integrate support for your own file format, as described in “Implementing Your Own File Format” on page 258.

The easiest way to open existing files or create new ones in any of these formats is to use the convenience functions **ilOpenImgFile()** and **ilCreateImgFile()**, as discussed in the next section. These two functions are declared in the header file *il/ilGenericImgFile.h* and are described in detail in their own reference pages. You can use the constructors for the ilTIFFImg, ilSGIImg, ilPCDImg, ilPCDOImg, ilGIFImg, and ilFITImg classes directly, but the convenience functions are simpler and allow you to write format-independent code. If you choose to use the constructors to open or create files, you must include the appropriate header files in your program.

You must link to the appropriate libraries in */usr/lib/ImageVision/filefmt*. Before running your program, enter the command:

```
setenv LD.LIBRARY_PATH.filefmt
```

Opening an Existing File

The following example opens an existing file for reading. The name of the file is specified in the first argument to **ilOpenImgFile()**.

```
ilFileImg* myFile;
myFile = ilOpenImgFile("anExistingFileName", "r");
```

The second argument to **ilOpenImgFile()**, the file access mode, can be either:

- “r” to indicate that the file is to be opened only for reading
- “r+” if the file will be read from and written to. Remember that you can’t write to ilPCDImg, ilPCDOImg, and ilGIFImg files, you can only read from them.

The **ilOpenImgFile()** function opens the named file and returns a pointer to one of the six ilFileImg types—ilTIFFImg, ilSGIImg, ilPCDImg, ilPCDOImg, ilGIFImg, or ilFITImg. If the named file doesn’t exist, or if it’s in another, unsupported format, a NULL pointer is returned. If you specify an invalid file access mode, for example if you try to write to an ilPCDImg file, the function returns a NULL pointer. **ilOpenImg()** also returns NULL if you attempt to open an existing file without read permission.

When you open an ilPCDImg file, you can modify the filename argument to the **ilOpenImgFile()** function to select the initial resolution. To do this, append a colon to the filename argument, followed by the index of the desired resolution. The index values are 0 through 5 for resolutions Base / 64 through 16Base; the default index is 3 (Base resolution). The following example opens an existing ilPCDImg file for reading. The index, specified as part of the file name, selects an initial resolution of Base / 4.

```
ilFileImg* photoFile;
photoFile = ilOpenImgFile (“myPhotoFile:2”, “r”);
```

You can also append an index to the filename to select the initial index for any multi-image file in the formats (TIFF, GIF, PDC, PDCO) that support multiple images in a file.

You can use **ilOpenImgFile()** to open a file in a format you’ve designed, as described in “Implementing Your Own File Format” on page 258.

After you open a file of image data, you can read the data. Example 3-1 illustrates this.

Example 3-1 Opening an Image File and Reading Data

```
// open the file
ilFileImg* someFile = ilOpenImgFile("someFileName", "r");

// check for errors
if (someFile == NULL) {
  printf("file %s could not be opened", fname);
  exit (1);
}

// obtain image attributes
ilType theDataType = someFile -> getDataType();
int theXdimension = someFile -> getXsize();
int numChannels = someFile -> getNumChans();

// allocate buffer
char* buf = new char[ ilDataSize(theDataType,
            theXdimension*numChannels) ];

// read data into buffer
someFile -> getTile(0,0,theXdimension,1,buf);
```

In this example:

1. A file is opened for reading—and a corresponding ilFileImg is created—with **ilOpenImgFile()**. If the file can't be opened, the program exits.
2. The ilFileImg is queried about some of its attributes to determine what size buffer to allocate for holding one row of the image's data.
3. The buffer is allocated. The **ilDataSize()** function returns the number of bytes needed for the data type indicated by its first argument, multiplied by the optional second argument. This function is declared in the header file *il/ilDataSize.h* and described in "Computing the Size of Data Types" on page 401.
4. The **getTile()** function reads the first row of the image's data into the buffer.

Creating a TIFF, SGI, or FIT File

To create a new file for writing image data, you need to specify the characteristics of the data, such as its data type, and indicate what file format will be used. The calling sequence for **ilCreateImgFile()** is shown below, and the image data attributes that you need to specify are discussed in the next paragraph. (All of these attributes are discussed in detail in “Image Attributes” on page 34, along with the constants that specify particular values for these attributes.)

```
ilFileImg* ilCreateImgFile(const char* name,
   const ilSize& size, ilType type, ilOrder order,
   char* format=NULL, const ilSize* pageSize=NULL);
```

This function creates an image file with the requested attributes and returns a corresponding pointer to one of the ilFileImg types. The first two arguments specify the name of the file to be created (which can be a pathname) and the size of the image to be written. The next three arguments indicate the data type of the data to be written (such as ilFloat), the pixel ordering (for example, ilInterleaved), and the format (for example, “TIFF”). If no format is supplied, the format is determined by the file name extension (for example “.tif” denotes TIFF format). If the format cannot be deduced from the argument, the default format TIFF is used.

The *pageSize* argument defines the *x*, *y*, *z*, and channel dimensions of the pages that the image is broken into as it’s stored on disk. If no page size is supplied, the default page size for that particular format and image size is computed, as described in Table 3-1. (The *x* and *y* dimensions are specified in pixels.)

Table 3-1 Default Page Dimensions

File Format	x-Dimension	y-Dimension	z-Dimension	Channel Dimension
TIFF	x-dimension of image[a]	64K / (x-dim*c-dim)[ab]	1	number of channels in the image[c]
SGI	width of image	32[d]	1	number of channels in the image (1, 3, or 4)
FIT	128	128	1	number of channels in the image[c]

a. The *x* -dimension must be either the width of the image or a multiple of 8.

b. The *y* -dimension must be a multiple of 8; it’s set so that the total page size is approximately 64 KB.

c. If the image’s ordering is ilSeparate, the channel dimension of its page size is 1.

d. The image is broken up as evenly as possible into pages that are less than or equal to 32 pixels in the *y* dimension.

The attributes specified when **ilCreateImgFile()** is called must match those supported by the file format being used. For example, TIFF files support any data type except ilDouble, SGI files support only ilUChar and ilUShort, and FIT files can handle any data type. See the reference pages for the various ilFileImg types for more information about what they support.

Once you create a file, you can write data to it. The example shown below assumes that *theImg* of size *size* has been previously created; its data is written to the file *outFile.tif* using **copyTile()**.

```
ilFileImg* tmpFile = ilCreateImgFile("outFile.tif", size,
          theImg.getDataType(), theImg.getOrder());

tmpFile->copyTile(0, 0, size.x, size.y, theImg, 0,0,0,1);
delete tmpFile;
```

You can change some attributes of a file after it is created, or even after an existing file is opened. Each of the different file formats uses the **setAllowed()** function inherited from ilLink to permit a few attributes to be modified. If you modify any of these attributes, you must do so before you write data to the file. Table 3-2 lists which attributes can be set for each of the three file formats.

Table 3-2 Modifiable File Attributes

File Format	File Type	Modifiable Attributes
TIFF	Existing	color model, color palette
	New	color model, color palette, compression, coordinate space
SGI	Existing	color model, color palette
	New	color model, color palette
FIT	Existing	z size
	New	color model, coordinate space, z size

The attributes shown in Table 3-2 are discussed in detail in "Image Attributes" on page 34.

Setting a File's Compression

Often, images stored in a file on disk are compressed to minimize their size. Such images need to be decompressed before you can read them. There are many different compression algorithms, and each specific file format (for example, TIFF) determines which algorithms it supports. From a programmer's point of view, as data is read or written in an IL program, its compression or decompression is handled transparently.

The compression attribute indicates which compression algorithm, if any, is used to compress the data before it's stored on disk. You should not compress files that will be interactively modified, rather than just written once and then read. Modifying portions of a compressed, existing file is dangerous because the amount of data written must be the same as what was originally in the file. In general, the size of a file image, once created, is fixed.

Currently, TIFF and SGI are the only IL file formats that support the creation of compressed files. Of course, you can implement your own file format as described in “Implementing Your Own File Format” on page 258 and have it support compression.

To set a TIFF file’s compression algorithm after you’ve created the file (and before you’ve written any data to it), use the **setCompression()** function declared by ilImage:

```
myTIFFFile->setCompression(ilLZW);
```

The argument passed to **setCompression()** is of type ilCompress and indicates which supported compression algorithm to use. Table 3-3 lists the ilCompress constants defined in the header file *il/ilTypes.h* and their corresponding compression algorithms.

Table 3-3 Compression Algorithms Supported for ilTIFFImg Files

ilCompress Constant	Compression Algorithm
ilNoCompression	none
ilCCITTFAX3	CCITT Group 3 fax encoding
ilCCITTFAX4	CCITT Group 4 fax encoding
ilLZW	Lempel-Ziv and Welch algorithm
ilPACKBITS	Apple® Computer, Inc., Macintosh® RLE (run-length encoding)

To query an existing TIFF file about which compression algorithm it uses, call **getCompression()**:

```
ilCompress whichCompression;
whichCompression = myTIFFFile->getCompression();
```

This function returns a value of type ilCompress corresponding to one of the supported algorithms.

Querying a File Image

Once you've created an ilFileImg, you can query its attributes with any of these functions:

```
char* getFileName();
int getFileDesc();
int getFileMode();
char* getImageFormat();
int getNumImgs();
```

getFileName()	returns the name of the file
getFileDesc()	returns the file descriptor
getFileMode()	returns either O_RDWR or O_RDONLY, depending on whether the file was opened for reading and writing or just reading
getImageFormat()	returns the file format—TIFF, SGI, PhotoCD Image Pack, PhotoCD Overview Pack, GIF, or FIT
getNumImgs()	returns the number of images stored in the file

Managing TIFF Tags and Directories

The ilTIFFImg implementation matches the TIFF 6.0 specification. For more information on how to obtain the TIFF 6.0 specification, see the Introduction of this Programming Guide. Also, the header file *il/ilTIFF.h* lists the TIFF tags supported by the IL. In addition, two new tags, TIFFTAG_IMAGEDEPTH and TIFFTAG_TILEDEPTH, have been added in the IL implementation to define the image depth, or what the IL refers to as the *z* dimension of an image's size.

You can use **getTIFFTag()**, which is declared in the header file *il/ilTIFFImg.h*, to retrieve any of the TIFF fields specified by TIFF 6.0:

```
char* tagValue;
myTIFFImg->getTIFFTag(TIFFTAG_IMAGEDESCRIPTION, &tagValue);
```

This function returns the requested tag value into the location referenced by *tagValue*.

You can set most TIFF tags using **setTIFFTag()**, as shown below.

```
myTIFFImg->setTIFFTag(TIFFTAG_IMAGEDESCRIPTION, tagData);
```

This function sets the TIFF tag using information pointed to by *tagData*.

These are the TIFF tags you're not allowed to set:

```
TIFFTAG_IMAGEWIDTH
TIFFTAG_IMAGELENGTH
TIFFTAG_BITSPERSAMPLE
TIFFTAG_SAMPLESPERPIXEL
TIFFTAG_ROWSPERSTRIP
TIFFTAG_TILEWIDTH
TIFFTAG_TILELENGTH
TIFFTAG_TILEDEPTH
TIFFTAG_DATATYPE
TIFFTAG_IMAGEDEPTH
TIFFTAG_PLANARCONFIG
TIFFTAG_SAMPLEFORMAT
TIFFTAG_COLORMAP
```

In order to call either **getTIFFTag()** or **setTIFFTag()**, you need a pointer to an ilTIFFImg. However, **ilOpenImgFile()** and **ilCreateImgFile()** return a pointer to an ilFileImg. To solve this problem, you can query your ilFileImg about its format and then cast the pointer appropriately, as shown below:

```
ilTIFFImg* myTIFFImg = NULL;

if (strcmp(myFileImg->getFileFormat(), “TIFF”) == 0)
  myTIFFImg = (ilTIFFImg*)myFileImg;

if (myTIFFImg != NULL) {
  // retrieve or set TIFF tags
}
```

Within fax-encoded TIFF files, each page of the fax is typically considered a separate image. The separate images are stored individually in the same TIFF file, each with its own header. The header at the top of the file points to the file’s first image. You can specify a particular image with its index (page number minus one). The example below reads the fourth page of the file:

```
ilStatus stat;
stat = myTIFFImg.setCurrentImg(3); // fourth page
if (stat == ilOKAY) {
  // continue work on this page
} else {
  // NULL data
}
```

You can also obtain the index for the current ilTIFFImg with **getCurrentImg()**.

You can use the index feature of **ilOpenImgFile()** to select the desired page of the fax-encoded file when you open the file:

```
ilFileImg* faxFile
faxFile = ilOpenImgFile(“faxFile:3”, “r”);
```

Extending the FIT Format

The ilFITImg class allows you to support extensions to the format. To do this, you must first reserve space in the file header and then use this space to write the data corresponding to the extension.

The function **reserveExtension()**, declared in the header file *il/ilFITImg.h*, takes a single **int** argument that indicates the number of bytes of space you wish to reserve. Once this space is reserved, you can write the header data using **writeExtension()**. This function takes two arguments; the first is a pointer to the data to be written, and the second indicates the number of bytes to be written:

```
ilStatus readExtension(void* data, int length);
ilStatus writeExtension(void* data, int length);
```

To read the additional header data, call **readExtension()**. This function also takes two arguments; the first points to a location into which the data should be read, and the second indicates the number of bytes to be read. The interpretation of the extension data is up to you.

Importing and Exporting Image Data

The IL provides a convenient mechanism for importing or exporting raw image data between the IL and other libraries or devices. This mechanism is encapsulated in the ilMemoryImg class, which interprets a contiguous array of data residing in memory as an ilImage object. Since ilMemoryImg inherits from ilImage, you can use any of the data access, query, and other functions defined in ilImage. In addition, ilMemoryImg defines a function that returns a pointer to its array of data so that you can read the data (for exporting) or write new data (for importing). The class ilXImage, derived from ilMemoryImg, allows you to convert an XImage (an X Window data structure that defines X's representation of an image) to an ilImage and vice versa.

Images in Memory

The ilMemoryImg class provides four constructors. You can use these constructors to:

- allocate an array to hold data that will be written
- use an already existing array
- create an ilMemoryImg object from an ilImage
- create an empty ilMemoryImg that will be populated later

The first constructor:

```
ilMemoryImg(const ilSize& size, ilType datatype,
               ilOrder order);
```

allocates an array large enough to hold *size.x*size.y*size.z*size.c* pixels of the indicated data type. This array is deallocated when the ilMemoryImg object is destroyed.

The second constructor allows you to import data. It takes as an argument an already existing array of data:

```
ilMemoryImg(void* data, const ilSize& size, ilType datatype,
                ilOrder order);
```

This constructor creates an ilMemoryImg object and initializes its data array with the data pointed to by *data*. The array needs to be large enough to hold *size.x*size.y*size.z*size.c* pixels of the indicated data type. Since this array wasn't allocated by ilMemoryImg, it won't be deallocated automatically when the ilMemoryImg object is destroyed.

Both of these constructors set the ilMemoryImg's attributes—size, data type, and order—to the values passed in the constructor so that you can use the query functions defined in ilImage, such as **getDataType()**. The minimum and maximum allowable pixel values are set by default to the minimum and maximum values allowed for the image's data type. In addition, the coordinate space attribute is set to ilLowerLeftOrigin. The color model is set depending on the number of channels in the image. If the channel dimension is 1, the color model is ilMinBlack; if it's 3, the color model is ilRGB. If the channel dimension is 4, the color model is ilABGR. Otherwise, the color model is ilMultiSpectral. You're allowed to change the color model and coordinate space of an ilMemoryImg after it's created.

The third constructor:

```
ilMemoryImg(ilImage* img, int autoSyncEnable = TRUE);
```

takes an ilImage as an argument. The ilMemoryImg object created has the same attributes as the ilImage. In addition, if autoSyncEnable is TRUE (the default), the attributes and data of the ilMemoryImg are automatically synchronized to match those of the ilImage during ilMemoryImg's reset operation. To turn off this feature:

- use **setAutoSync()** and pass in FALSE as the argument
- pass in FALSE for *autoSyncEnable*

You can call the **sync()** function on this type of ilMemoryImg to synchronize the attributes and data at any point, regardless of whether automatic synchronization is in effect.

The fourth constructor:

```
ilMemoryImg();
```

returns an ilMemoryImg object with no data or attributes. You can use this constructor when you need to create an ilMemoryImg sooner than you can supply its data. Use **setDataPtr()** to specify the data.

To change the image data residing in an ilMemoryImg object, call **setDataPtr()** and pass a pointer to the new data. You should also call **markDirty()** to indicate that the data in the ilMemoryImg object has been altered.

```
void setDataPtr(void* data);
void markDirty();
```

To gain direct access to the image data residing in a ilMemoryImg object, call **getDataPtr()**. This function returns a void pointer to the data, as shown below.

```
void* getDataPtr();
```

Because an ilMemoryImg resides in memory, you can use it to hold temporary copies of images that you need to access quickly.

Note: Since the entire image resides in memory at once, the IL's on-demand execution model doesn't take effect when an ilMemoryImg is used.

X Window Images

An XImage, (the X library defines **struct** XImage), can be converted to an ilImage using the ilXImage class, which is derived from ilMemoryImg. Since an XImage cannot be chained as an ilImage is, you need to convert any XImage to an ilImage before you can use the IL to perform image processing on it. Note that an ilXImage holds its entire contents in memory at once and thus doesn't give the benefits of the IL's demand-driven paging model. You can remedy this by using the ilNopImg operator, which does nothing but allow you to tile and cache an otherwise non-cached IL object. See "Using a Null Operator" on page 184 for more information about the ilNopImg class.

You can construct an ilXImage in four different ways:

```
ilXImage(ilImage* ilImg, Display* xDisply = NULL);
ilXImage(XImage* xImg, Display* xDisply = NULL);
ilXImage(const ilSize& size, ilType type,
          Display* xDisply = NULL);
ilXImage(void* data, const ilSize& size, ilType type,
          Display* xDisply = NULL);
```

The first and second constructors create an ilXImage from an ilImage and an XImage, respectively. The third constructor creates an ilXImage of the specified size and data type. The fourth constructor creates an ilXImage from *data* of the specified size and data type. The Display pointer (an X Window **struct**) associated with the image is optionally specified in each constructor with *xDisply* (which is NULL by default).

If you are programming using the X library and need X Window information from an ilXImage, you may wish to use these functions:

```
XImage* getXImage();
Visual* getXVisual();
XVisualInfo* getXVisualInfo();
```

You can get an XImage pointer from the ilXImage with **getXImage()**. Use **getXVisual()** to get the Visual pointer associated with the ilXImage and **getXVisualInfo()** to return an XVisualInfo pointer. Additionally, for C++ programmers, the function call operator, (), is overloaded to perform the same function as **getXImage()**. (Visual and XVisualInfo are **struct**s defined in the X Window library.)

Use these functions to set X Window information in the ilXImage:

```
void setXDisplay(Display* xDisply);
void setXVisual(Visual* xVisual);
```

Several protected functions are provided for convenience to extract IL and X information from the ilXImage. These functions can be useful for X programming; for example, functions are provided to determine the color map associated with the image and its depth. See the ilXImage reference page for details.

Chapter 4

Operating on an Image

This chapter describes how to use the image processing operators defined by the ImageVision Library. It also tells you how to restrict an image processing operation to just a portion of an image.

Chapter 4

Operating on an Image

Much of the ImageVision Library implementation consists of image processing algorithms, or *operators*. An operator applies its algorithm to the image data encapsulated in an ilImage object. To maximize the efficiency of the computation required to perform such an operation, the IL uses the demand-driven execution model discussed in Chapter 2, “The ImageVision Library Foundation.”

This chapter explains how to use each of the operators defined by the IL. “Implementing an Image Processing Operator” on page 273 explains how you can implement your own image processing algorithm as an IL operator. This chapter contains the following major sections:

- “Image Processing Operators Provided with the IL” on page 110 describes the set of approximately 70 image processing operators implemented in the IL.
- “Defining a Region of Interest” on page 185 explains how to mask out portions of an image and restrict processing to a desired area.

The IL classes covered in this chapter are mainly those that derive from ilOpImg. The relevant portion of the IL inheritance hierarchy is shown shaded in Figure 4-1.

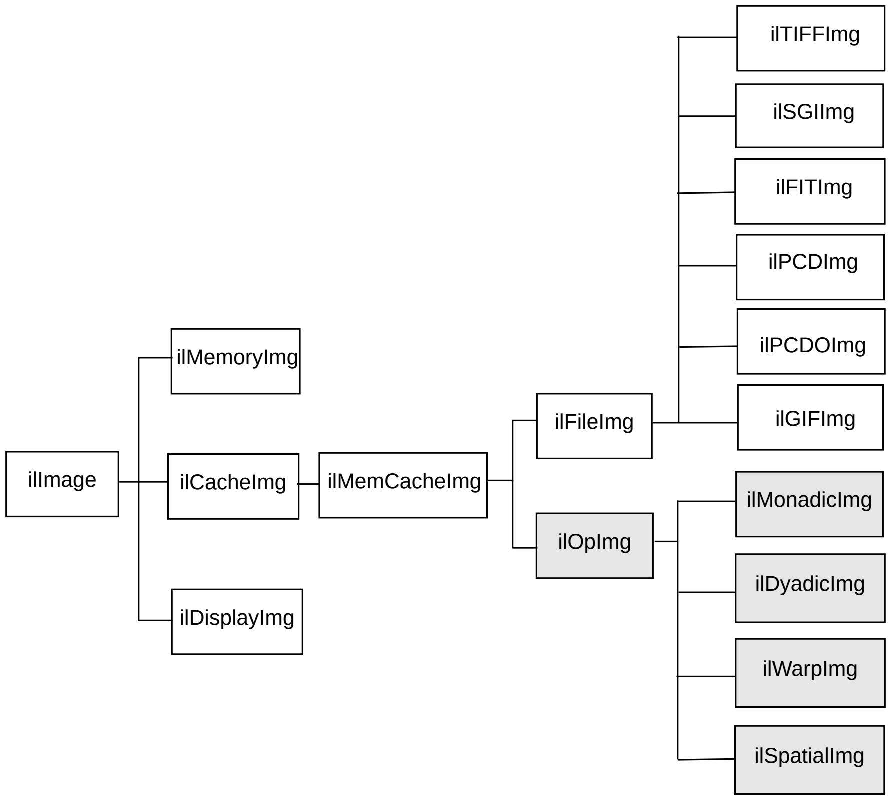


Figure 4-1 ilOpImg and the IL Inheritance Hierarchy

Some operators derive directly from ilOpImg, which is itself an abstract class. In addition, several abstract or generalized classes inherit from ilOpImg, and these classes have operator subclasses. Each of the sections that follows shows how the relevant classes fit into the IL inheritance hierarchy.

The ilOpImg class defines the basic support for all operator classes. It provides functions for setting attributes, accessing data, and propagating attributes down an operator chain. All of these functions are declared protected, so while they're available for use in a subclass's implementation, they're not available (or needed) if you're simply using an operator. In fact, ilOpImg defines only three sets of public functions:

```
double getBias();
ilStatus setBias(double biasValue=0);

ilStatus setClamp(ilType type=numilTypes);
ilStatus setClamp(double min, double max);

void hwAccelerate(int enable);
int isAccelerated();
```

Some operators take a bias argument in their constructors and use it in their image processing algorithms. This bias value is discussed in the sections describing the relevant operators in the remainder of this chapter.

The **setClamp()** functions allow you to set values that pixels is clamped to if underflow or overflow occurs. Not all operators allow the clamp values to be modified, so you need to check that the returned status isn't ilUNSUPPORTED if you're assuming you've changed the values. The first version of **setClamp()** sets the clamp values to be the minimum and maximum values allowed for the data type *type*; the default value of numilTypes means to use the operator's data type. The second version allows you to specify actual clamp values. You won't generally need to use either of these functions since most operators handle overflow and underflow conditions appropriately.

In addition, all operators that alter the data range of their inputs compute the worst case minimum and maximum pixel values to ensure that the processed data can be displayed. For example, if you multiply two images and then display the result, you can easily end up with pixel data that's all black. To solve this problem, ilMultiplyImg automatically computes the worst case minimum and maximum values. When the data is displayed using ilDisplay, the data is automatically scaled between these values (or those allowed by the display) so that a meaningful display is produced.

The **hwAccelerate()** function allows you to turn hardware acceleration on (*enable* = TRUE) or off (*enable* = FALSE) for a particular operator. Hardware acceleration is enabled by default. See “Using Graphics Hardware for Acceleration” on page 73 for a discussion of hardware acceleration. “Using Hardware Acceleration” on page 315 lists operators and the hardware on which they're accelerated.

Image Processing Operators Provided with the IL

This section discusses all the operators provided with the IL. They're grouped functionally as listed below:

- "Color Conversion and Transformation" on page 111 describes operators that convert an image from one color model to another.
- "Arithmetic and Logical Transformations" on page 117 describes operators that perform pixelwise arithmetic or logical computations.
- "Geometric Transformations" on page 125 describes operators that warp, rotate, and zoom (magnify or minify) an image.
- "Spatial Domain Transformations" on page 133 describes operators that transform an image in the spatial domain—for example, by sharpening, blurring, convolving, or rank filtering it in the spatial domain.
- "Edge Detection" on page 144 describes gradient operators such as compass, Laplace, Roberts, and Sobel.
- "Frequency Domain Transformations" on page 148 describes operators that incorporate forward or inverse Fourier transforms and frequency-domain filters.
- "Generation of Statistical Data" on page 162 describes the operator that computes the histogram, mean, and standard deviation of an image.
- "Radiometric Transformations" on page 166 describes operators that perform radiometric transformations such as histogram normalization and thresholding.
- "Combining Images" on page 177 describes operators that blend, merge, or combine two images.
- "Constant-valued Images" on page 184 describes an image class that returns a constant value for all data accesses.
- "Using a Null Operator" on page 184 describes an operator that performs a "null" operation.

Color Conversion and Transformation

The IL provides several operators that perform color conversions and color transformations of IL images. These operators can be summarized as follows:

- The ilColorImg operator converts an existing image from any IL-supported color model to a requested color model. (See “Color Model” on page 40 for a description of the color models supported by the IL.)
- Several operators, derived from ilColorImg, convert an existing image to one of the more commonly used color models: CMYK, grayscale, HSV, and RGB.
- The ilFalseColorImg operator converts an image from one multispectral color model to another.
- The ilSaturateImg operator provides a mechanism to transform the color saturation of an image.

These color conversion and transformation operators are described in the following paragraphs. Their positions in the IL inheritance hierarchy are shown in Figure 4-2.

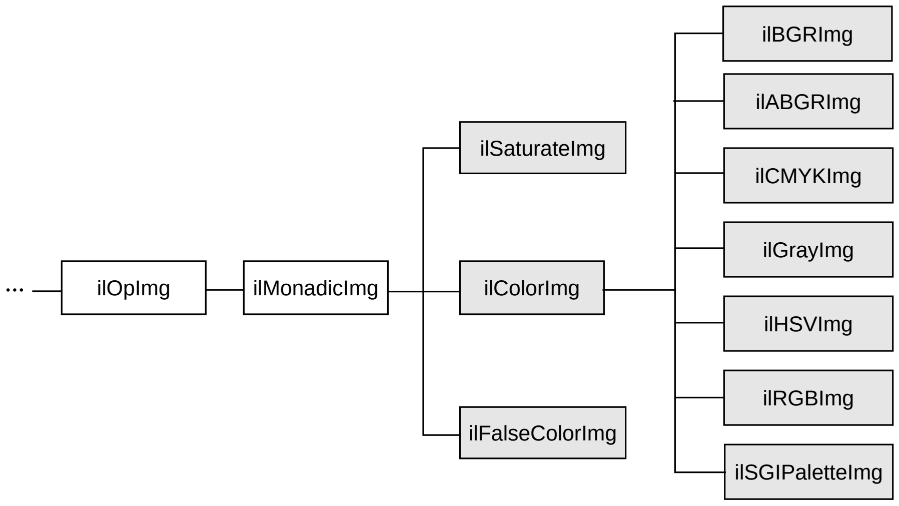


Figure 4-2 Color Conversion Operators Inheritance Hierarchy

Color Conversion

The base class for the color conversion operators, ilColorImg, defines the generic support for performing color conversions on image data. It converts data from any supported color model to any other supported color model, except multispectral.

```
ilColorImg(ilImage* img, ilColorModel cm)
```

For example, the following code converts an ilRGB image (*theimg*) to one whose color model is ilYCC.

```
ilColorImg(ilImage* theimg, ilColorModel ilYCC);
```

The ilColorImg class is not normally used directly to do color-model conversion. Instead, use derived classes. Each of the six classes derived from ilColorImg performs a specific conversion. The algorithms used to perform the various conversions are detailed in the respective reference pages. The six derived classes are summarized below:

- ilABGRImg converts data to the ABGR color model used by Silicon Graphics' framebuffer.
- ilCMYKImg converts data to the CMYK color model. This color model is used primarily as an output format for color printers.
- ilGrayImg converts an image to minBlack.
- ilHSVImg converts to the HSVcolor model.
- ilRGBImg converts an image to the ilRGB color model.
- ilSGIPaletteImg converts data to the ilRGBPalette color model. This color model is suitable for data that is to be displayed in a color-mapped window.

Using any of these derived classes is simple since the only public member function most of them define is a constructor. To convert an ilImage, call the constructor for the desired color model and supply as an argument a pointer to the ilImage to be converted. For the following example, assume that *theImg* has already been created and that it uses any one of the supported IL color models:

```
ilCMYKImg* cnvrtdImg;
cnvrtdImg = new ilCMYKImg(theImg);
```

In this example, the constructor for the ilCMYKImg class returns a pointer to an ilCMYKImg, which produces image data converted to the CMYK color model. Similarly, the constructors for any of the derived classes—ilABGRImg, ilCMYKImg, ilGrayImg, ilHSVImg, ilRGBImg, or ilSGIPaletteImg—return a pointer to an object of that class, which produces converted image data. That's really all there is to it.

If you want to convert to the color models for which there is no derived class (ilRGBA, ilCMY, ilBRG or ilYCC), use the ilColorImg operator.

If an operator image has two or more inputs with different color models, the color model of the resulting image depends on the color models of the input images. The IL converts the color models of the input images to a common color model before performing the operation. The resulting image has this color model. You can use the diagram in Figure 4-3 to determine how the IL determines the common color model. Just find the nodes for the input images and follow the paths from these nodes to a common node. This nodes determines the color model of the resulting image. For example, if the color models of two inputs to an operator are ilHSV and ilYCC, the color model of the resulting image is ilRGB.

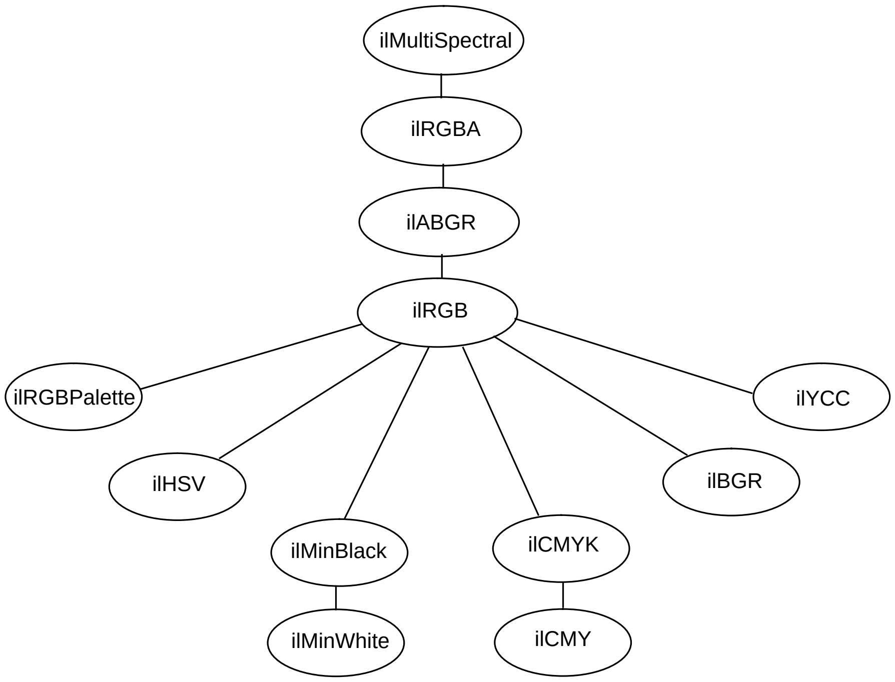


Figure 4-3 Determining the Color Model of Multi-Input Operators

ilFalseColorImg

The ilFalseColorImg operator performs false coloring of multispectral images. It accomplishes this by computing the weighted sum of the input channels for each channel of the resulting false-color image. The constructor for ilFalseColorImg takes a pointer to the input image and arguments that define the conversion algorithm:

```
ilFalseColorImg(ilImage *img, ilFloatMatrix& Tm);
ilFalseColorImg(ilImage *img, ilFloatMatrix& Tm,
             ilFloatMatrix& Bm);
```

The conversion is defined by the transformation matrix, *Tm*. (The ilFloatMatrix type is described in “Auxiliary Classes” on page 396.) This matrix has dimensions *N* x *M*, where *N* is the number of channels the output image should have and *M* is the number of channels in the source image. Each row of this matrix defines a set of weights used to produce one channel of the output. Each weight is multiplied by the pixel values in the corresponding input channel, and the weighted sum forms the output channel. The conversion may also include a bias vector, *Bm*. This vector contains a constant value for each input channel that is added to each input value before it is weighted. Thus, the transformation equation for each channel of the output image is:

$$Output_{Nx1} = T_{NxM}(Input_{Mx1} + B_{Mx1})$$

You can set the transformation matrix and bias vector with **setTransform()**:

```
ilStatus setTransform(ilFloatMatrix& Tm)
ilStatus setTransform(ilFloatMatrix& Tm, ilFloatMatrix& Bm)
```

An image transformed by ilFalseColorImg appears in Figure 4-4.

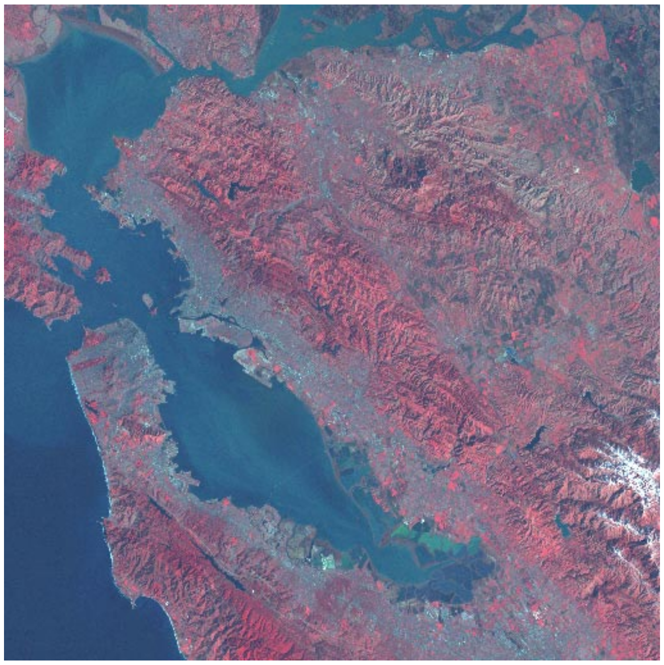


Figure 4-4 A Falsely Colored Image

ilSaturateImg

This operator performs a color saturation of its input. If the input color model is not RGB, the input is first converted to RGB. The constructor for ilSaturateImg takes a pointer to the input image and an initial saturation value:

```
ilSaturateImg(ilImage* img=NULL, float sat=1);
```

The transformation is defined as:

$lum = .3red_{in} + .59green_{in} + .11blue_{in}$

$red_{out} = lum + (red_{in} - lum)sat$

$green_{out} = lum + (green_{in} - lum)sat$

$\text{blue}_{out} = \text{lum} + (\text{blue}_{in} - \text{lum})\text{sat}$

You can set the saturation value interactively with **setSaturation()**:

```
void setSaturation(float saturation);
```

The current value of the saturation factor can be queried with **getSaturation()**:

```
float getSaturation();
```

A value of zero completely desaturates the image (equivalent to ilGrayImg), a value of one leaves the image unchanged, and values greater than one increase the color saturation of the image. Output values are clamped to the minimum and maximum values of the operator image, which by default are simply inherited from the input.

Arithmetic and Logical Transformations

There are numerous IL operators that perform pixelwise arithmetic transformations of image data. Some of these require two input images—for example, to add them together—while others perform computations on a single image's data, such as determining the absolute value. In the inheritance hierarchy shown in Figure 4-5, operators that inherit from ilDyadicImg take two images as inputs, and those that derive from ilMonadicImg take only one.

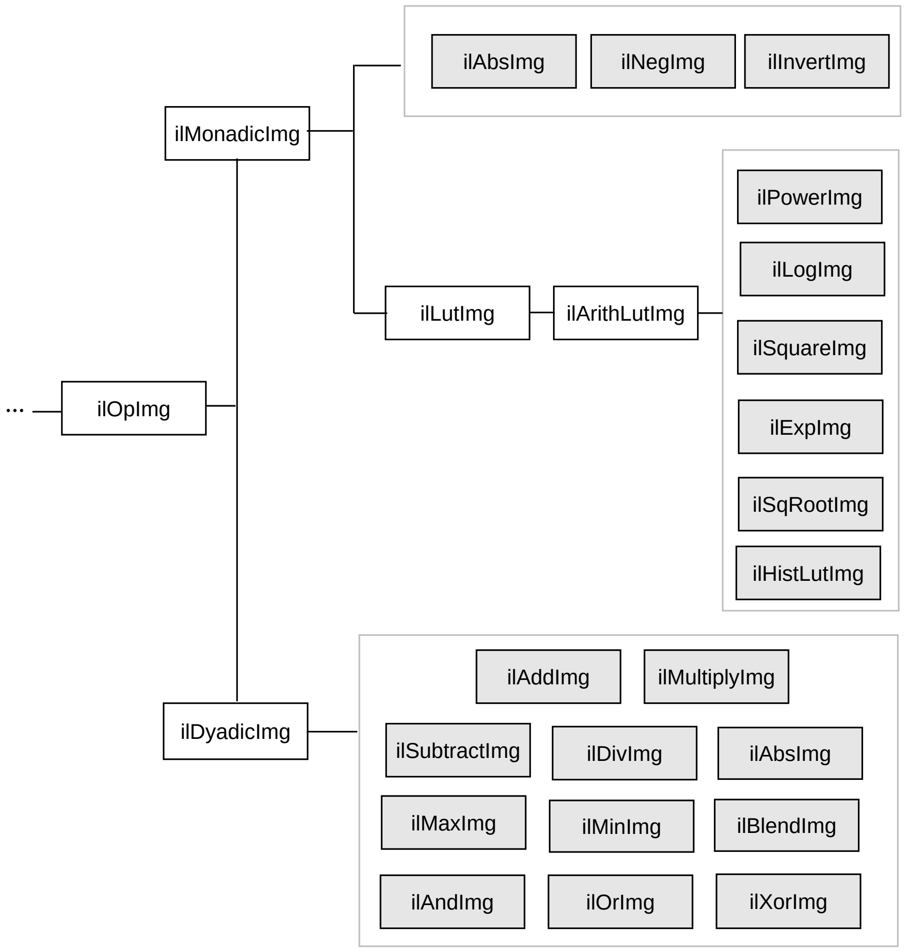


Figure 4-5 Arithmetic and Logical Operators Inheritance Hierarchy

When using one of the dual-input operators, you might want to use an ilConstImg as one of the inputs. An ilConstImg returns the same value for all of its pixels, so you can use it to multiply each of an image's pixels by a constant value, for example. For more information on how to create an ilConstImg, see "Constant-valued Images" on page 184.

Single-input Operators

The single-input arithmetic operators are listed in Table 4-1, along with the operation they perform on each pixel of image data and the pixel data types each operation can produce. The last five operators in Table 4-1 (ilSquareImg, ilSqRootImg, ilExpImg, ilPowerImg, and ilLogImg) descend directly from ilArithLutImg. The ilArithLutImg abstract class optimizes the performance of operators that derive from it by pulling precomputed square, square root, exponent, power, and log values from a lookup table. This is much more efficient than computing values on a per-pixel basis.

The ilArithLut class in turn inherits from ilLutImg, Thus, the last five operators in Table 4-1 inherit the ability to be accelerated further in the CPU or in specialized graphics hardware. See “Radiometric Transformations” on page 166 and “Using Hardware Acceleration” on page 315 for details about ilArithLutImg and hardware acceleration, respectively.

Table 4-1 Single-input Arithmetic Operators and Their Valid Output Data Types

Operator	**Operation Performed**	**Valid Data Types**
ilAbsImg	absolute value	ilUChar, ilUShort, ilULong, ilFloat, ilDouble
ilNegImg	two’s complement	any signed data type[a]
ilInvertImg	one’s complement	ilBit, ilChar, ilUChar, ilShort, ilUShort, ilLong, ilULong
ilSquareImg	$(\text{pixelvalue})^2$	any type except ilBit
ilSqRootImg	$\sqrt{\text{pixelvalue}}$	any type except ilBit
ilExpImg[b]	$\text{base}^{(\text{pixelvalue})}$	any type except ilBit
ilPowerImg[b]	$(\text{pixelvalue})^{\text{power}}$	any type except ilBit
ilLogImg[b]	$\log_{\text{base}}(\text{pixelvalue})$	any type except ilBit

a. ilChar ilShort, ilLong, ilFloat, and ilDouble are the signed data types.

b. These operators allow you to apply *scale* and *bias* values to the *pixelvalue*, so that it becomes *scale*pixelvalue+bias*

An example of processing by an arithmetic operator is given in Figure 4-6, which shows an original image constructed from simulation data processed with ilNegImg. See “Arithmetic and Logical Transformations” on page 383 for examples of the output of other arithmetic and logical operators.

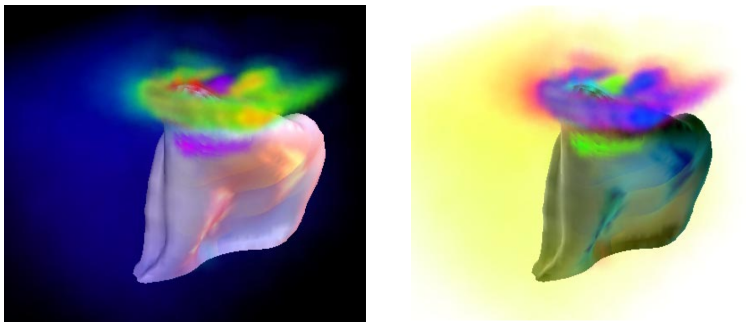


Figure 4-6 A Positive and Negative Image Pair

The only public member function defined in ilAbsImg, ilNegImg, ilInvertImg, ilSquareImg, and ilSqRootImg is a constructor that takes a single argument, the input image. Thus, to include any of these operators in a chain, you simply call its constructor and pass as the argument a pointer to the input ilImage. In this example, assume that *inputImg* is a pointer to an already existing ilImage:

```
ilAbsImg* someAbsImg = new ilAbsImg(inputImg);
```

The constructors for the ilAbsImg, ilNegImg, ilInvertImg, ilSquareImg, and ilSqRootImg classes all return a pointer to the operator image.

The constructors for the remaining three classes—ilExpImg, ilPowerImg, and ilLogImg—take three additional arguments, all of type **double**. The second argument for each of these constructors specifies base or power, the third specifies scale, and the fourth bias.

```
ilExpImg(ilImage* inImg = NULL, double expBase=0,
              double scl=1., double bs=0.);
ilPowerImg(ilImage* inImg = NULL, double pow = 2,
              double scl=1., double bs=0.);
ilLogImg(ilImage* inImg = NULL, double logBase=0,
              double scl=1., double bs=0.);
```

The ilExpImg, ilPowerImg, and ilLogImg classes define a function for setting the value of the second parameter after the operator is created, so that you can dynamically alter the computation:

```
void setBase(double expBase=0);          // for ilExpImg
void setPower(double power=2);           // for ilPowerImg
void setBase(double logBase=0);          // for ilLogImg
```

The ilExpImg, ilPowerImg, and ilLogImg classes define functions for setting and retrieving the scale parameter after the operator is created.

```
void setScale(double scl);
double getScale();
```

Dual-input Operators

As their names suggest, the dual-input operators ilAddImg, ilSubtractImg, ilMultiplyImg, and ilDivImg perform standard arithmetic computations—addition, subtraction, multiplication, and division of two images. The constructors for each of these classes take as arguments pointers to the two input images, which can be different sizes but must have the same number of channels. If they're different sizes, by default the output image is the larger of the two sizes; the smaller input image is padded with its fill value, and then the operator performs its computation on corresponding pixels in the two images. You can explicitly set the desired output size with **setSize()**.

You may also offset one image with respect to the other:

```
void setOffset(int x, int y, int z = 0, int input = 0);
void getOffset(int &x, int &y, int &z, int input = 0);
```

setOffset() offsets the first image with respect to the second by *x*, *y*, and *z* if *input* is 0. If *input* is 1, the second image is offset with respect to the first. **getOffset()** queries the dual-input operator for its offsets. If *input* is 0, the offset of the first image relative to the second is given; if *input* is 1, the offset of the second image relative to the first is given.

Here are the constructors for the dual-input operators:

```
ilAddImg(ilImage* in1 = NULL, ilImage* in2 = NULL,
         double bias=0);
ilSubtractImg(ilImage* in1 = NULL, ilImage* in2 = NULL,
         double bias=0);
ilMultiplyImg(ilImage* in1 = NULL, ilImage* in2 = NULL);
ilDivImg(ilImage* in1 = NULL, ilImage* in2 = NULL, ckDiv=1);
```

ilAddImg adds the *bias* value to the sum found by adding the corresponding pixels of *in1* and those of *in2*. The ilSubtractImg operator subtracts the corresponding pixels of *in2* from every pixel of *in1* and then adds the *bias* value. ilMultiplyImg multiplies the pixels in the two input images, and ilDivImg divides the pixels of *in1* by the corresponding pixels of *in2*. All these operators can produce an image containing any data type except ilBit. An example using ilAddImg appears in Figure 4-7. The two original images appear as well; one is the flipped version of the other.

The *ckDiv* argument for ilDivImg's constructor specifies whether the operator should check for division by zero. By default, it does check and responds as described below:

- If the divisor is zero and the dividend is positive, the quotient is set to the maximum value possible for the final image's data type.
- If the divisor is zero and the dividend is negative, the quotient is set to the minimum value possible for the final image's data type.
- Zero divided by zero produces a zero.

You can use **setCheck()** to change whether this check is made.

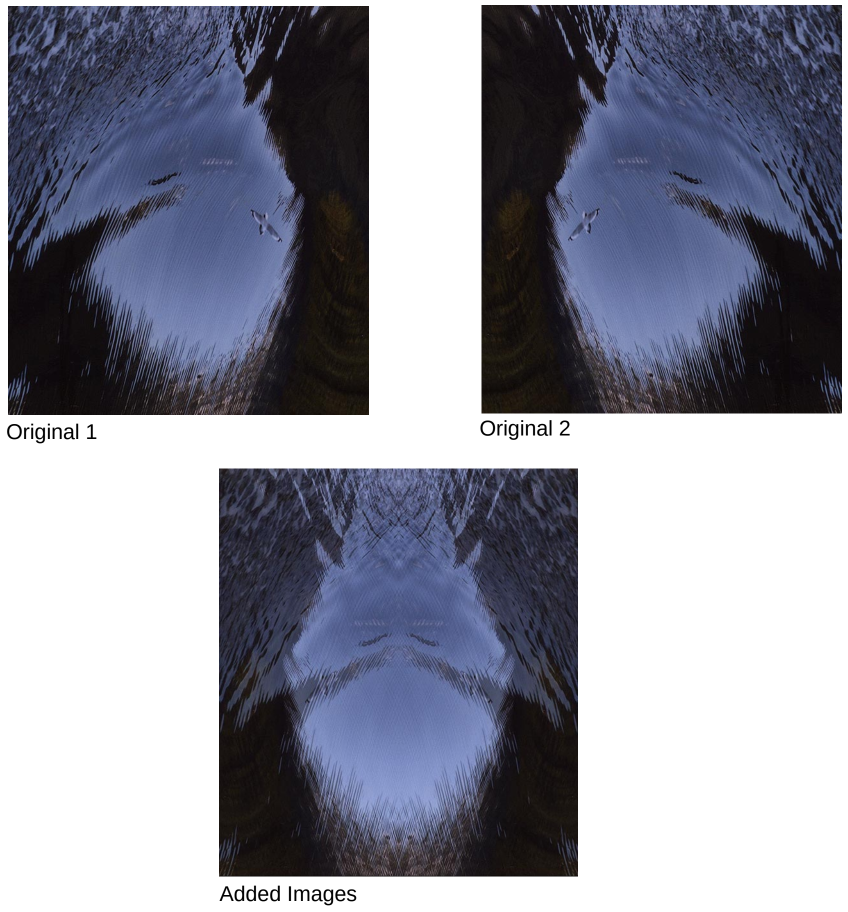


Figure 4-7 Adding Two Images

The two classes ilMaxImg and ilMinImg compare each corresponding pixel in the two input images and select the greater or the lesser value, respectively. Their constructors take pointers to the two input images as arguments. These input ilImages must have the same number of channels; the output image can contain any data type except ilBit. (There are also simple in-line functions defined in the header file *il/ilMinMax.h* that compare two values and return the greater or the lesser one. See “Minimum and Maximum Comparisons” on page 402 for more information about these functions.) An example of using ilMinImg appears in Figure 4-8. Two original images are shown, followed by the image that results if you apply ilMinImg to these images.

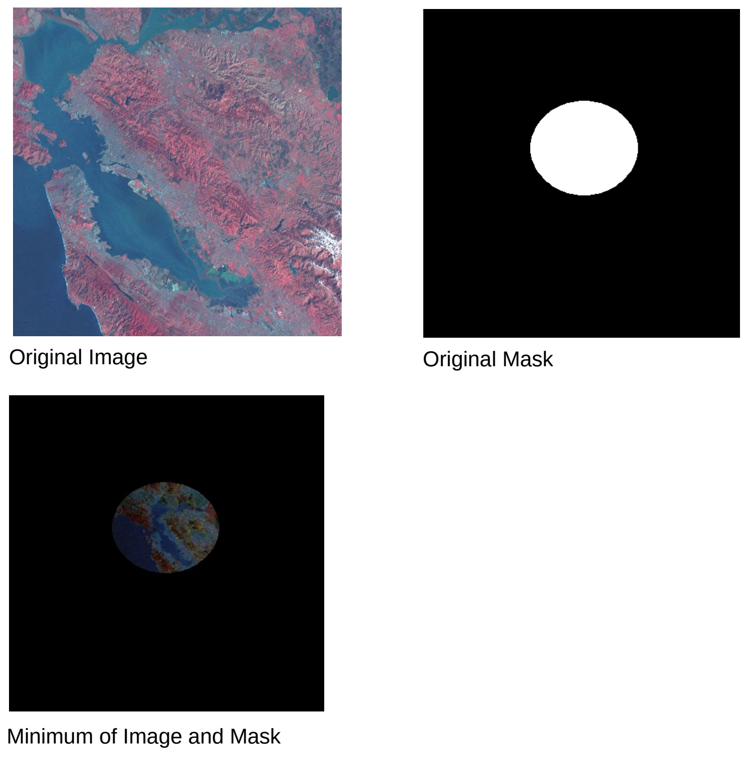


Figure 4-8 Minimum of Two Images

Similarly, the logical-operator classes—ilAndImg, ilOrImg, and ilXorImg—perform their computations (logical AND, OR, and exclusive-OR) by combining each corresponding pixel in the two input images. The constructors for these classes take pointers to the two input images as arguments. The input ilImages must have the same number of channels; the output image can contain any of the following data types: ilChar, ilUChar, ilShort, ilUShort, ilLong, or ilULong. Figure 4-9 shows an example of using ilAndImg and ilOrImg on the original images from Figure 4-7.

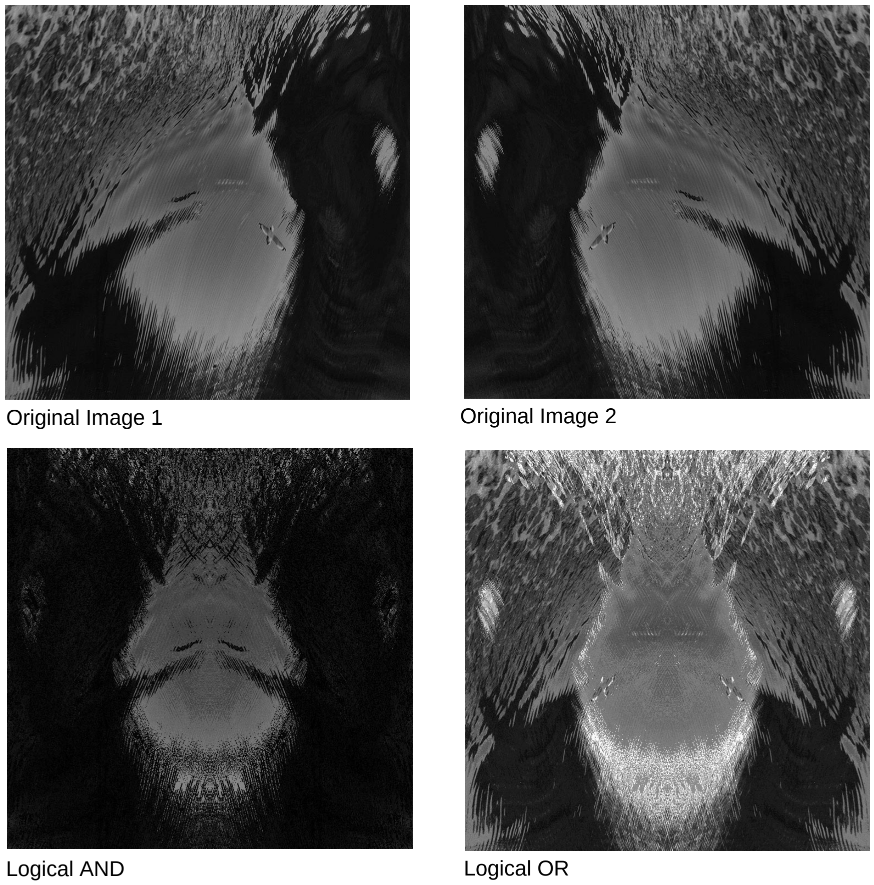


Figure 4-9 Logical AND and OR of Two Images

Geometric Transformations

The heart of a geometric transformation, or warp, is the algorithm that maps output image coordinates to input coordinates. (See Figure 4-10.) The general support for such transformations is encapsulated in the abstract class, ilWarpImg. Classes that derive from ilWarpImg—ilPolyWarpImg, ilTieWarpImg, and ilRotZoomImg—implement specific warping algorithms; these algorithms are most efficient for images that are relatively square.

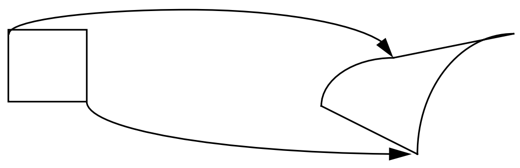


Figure 4-10 A Warped Image

The warping classes are shown in Figure 4-11 and discussed in the following paragraphs.

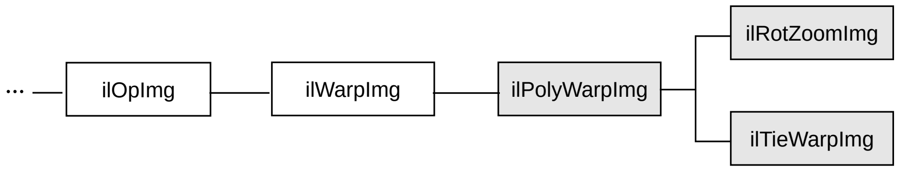


Figure 4-11 Geometric Operator Inheritance Hierarchy

Warping an Image

The ilWarpImg class, from which ilPolyWarpImg, ilTieWarpImg, and ilRotZoomImg derive, performs up to a two-dimensional, seventh-order warp. The output image space is mapped to the input image space with a transformation defined by two sets of polynomials (which can be up to seventh order), one for the *x*-dimension and one for the *y*-dimension. Since the coefficients for the polynomials aren't always integers, the addresses computed for the output space sometimes contain fractional components. Therefore, a resampling method must be applied to convert these fractional addresses into meaningful pixel locations.

To use ilWarpImg, you must choose a resampling algorithm and specify the coefficients of the warping polynomials. The constructor takes as its arguments a pointer to the input image and a constant that corresponds to a resampling method:

```
ilWarpImg(ilImage* img, ilResampType rs);
```

The ilResampType enumerated type is defined in the header file *il/ilTypes.h* and shown in “Controlling Operators” on page 411. It has these six members:

- ilNearNb (nearest neighbor)
- ilBiLinear
- ilBiCubic
- ilMinify
- ilAutoResamp
- ilUserDef (for a resampling algorithm you implement)

If you choose a bicubic resampling method, you can use **setBicubicFamily()** to fine-tune its algorithm.

If you choose the ilMinify resampling method, you can use **setMinifyKernel()** to specify your own kernel instead of the default box (all 1s) kernel. In the default case, the kernel size is dynamically adjusted so that the entire input is sampled (that is, all the input image pixels are used to compute the output). If you use the default kernel, you can speed up the operation by using **setMaxSamples()** to set the number of input image pixels to be averaged to produce a single output pixel. For example, if you set the maximum number of samples to 10 and you are minifying by a factor of 8, thus necessitating the use of an 8 x 8 kernel, only 10 input pixels (instead of 64) uniformly interspersed throughout the 8 x 8 area are averaged to produce one output pixel.

ilAutoResamp is not another resampling algorithm, but a mechanism through which derived classes can dynamically set the resampling method to one of the six listed above. ilAutoResamp sets a flag, autoResamp, that derived classes inherit and can use to set the resampling method, using the inherited member variable resampType, as shown below:

```
If (autoResamp) {
  if(condition1)resampType = ilBiLinear;
  else if (condition2)resampType = ilMinify;
  else if (condition3)resampType = ilBiCubic;
  else resampType = ilNearNb;
}
```

This code typically appears in the **resetOp()** method of a derived class. IlRotZoomImg has the resampling method set to ilAutoResamp by default. When there is no hardware acceleration, it uses the mechanism described above to set the resampling method to ilMinify for pure minification (x and y zoom factors less than 1.0 and no rotation) and to ilNearNb for all other cases. If there is hardware acceleration, it sets the resampling method to ilBiLinear for pure minification and to ilNearNb for all other cases.

To define your own resampling method, use **setResampFunc()** and pass in a pointer to your algorithm. The reference page for ilWarpImg explains what the supported algorithms are, which one you might want to use, and how to define your own algorithm.

You can dynamically change and retrieve the resampling method with **setResampType()** and **getResampType()**, which are inherited from ilWarpImg:

```
void setResampType(ilResampType rs);
ilResampType getResampType();
```

Additionally, ilWarpImg lets you determine the amount of error allowed in a warp performed in graphics hardware with **setAddressError()**. Its one parameter, *maxPixelsOff*, determines by how many pixels the warped data may be incorrect. The previously set parameter can be retrieved with **getAddressError()**:

```
void setAddressError(float maxPixelsOff);
float getAddressError();
```

ilPolyWarpImg adds to the capabilities of ilWarpImg in that it allows you to define the coefficients of the warping polynomial. Once you've created an ilPolyWarpImg object, you need to set the coefficients with the **setCoeff()** function:

```
void setCoeff(const ilCoeff_2d& xcoeff, const ilCoeff_2d& ycoeff);
```

You can query the ilPolyWarpImg object for its coefficients with **getCoeff()** and for the order of its polynomial with **getPolyOrder()**:

```
void getCoeff(ilCoeff_2d& xcoeff, ilCoeff_2d& ycoeff);
int getPolyOrder();
```

The ilCoeff_2d structure contains floating point numbers for the coefficients; it’s defined in the header file *il/ilPolyDef.h* as shown below:

```
struct ilCoeff_2d {
  int order;// equation’s total degree or order
  struct ilCoeff7_2d c;
};

struct ilCoeff7_2d {
  float con,          // constant
  y, x,               // first-order terms
  y2, xy, x2,         // second-order terms
  y3, xy2, x2y, x3,   // third-order terms ...
  y4, xy3, x2y2, x3y, x4,
  y5, xy4, x2y3, x3y2, x4y, x5,
  y6, xy5, x2y4, x3y3, x4y2, x5y, x6,
  y7, xy6, x2y5, x3y4, x4y3, x5y2, x6y, x7;
};
```

Using ilTieWarpImg is similar to using ilPolyWarpImg. This class also performs a two-dimensional warp, but it doesn’t allow you to specify the coefficients of the warping polynomial directly. Instead, you specify pairs of tie points in the input and the output images that should match after the image is warped as shown in Figure 4-12. The coefficients of the polynomial, which you can choose to be first- to seventh-order, are then computed from these tie points. The minimum number of pairs of points necessary to determine the coefficients of a polynomial of order *ord* is given by the formula:

$$pairs = \frac{(ord+1)\ (ord+2)}{2}$$

Thus, you need to specify at least three pairs of points for a first-order polynomial, six pairs for a second-order, and so on.

The constructor for ilTieWarpImg takes the same arguments as that for ilPolyWarpImg. After creating an ilTieWarpImg operator, you must specify the tie points from which the warping polynomial is computed. For this, use **setTiePoints()**:

```
void setTiePoints(const ilXYSfloat* uv, const ilXYSfloat* xy,
          int n);
```

This function takes pointers to arrays of *n* tie points in the input image (*xy*) and the output image (*uv*) and computes the polynomial's coefficients. (The data type ilXYSfloat is defined in the header file *il/ilCoord.h* as an (*x*,*y*) coordinate pair of data type **float**.) The function **isWellDefined()** can be used to check if the polynomial coefficients can be computed from the specified tie points. If the polynomial is successfully computed, 1 is returned; if not, 0 is returned. Before you call **setTiePoints()**, you might want to set the order of the polynomial that will be computed by calling **setPolyOrder()** and passing in 1, 2, 3, 4, 5, 6, or 7 as the desired order. If you don't explicitly set the order, a first-order polynomial will be used. The function **getPolyOrder()** returns the order of the warping polynomial.

ilPolyWarpImg defines functions (which ilTieWarpImg and ilRotZoomImg inherit) that, given a point in the input (or output) image, compute the corresponding point in the output (or input) image, using the mapping specified by the polynomial:

```
void evalUV(ilXYSfloat& uv, const ilXYSfloat& xy);
```

```
void evalXY(ilXYSfloat& xy, const ilXYSfloat& uv);
```

The function **evalUV()** takes the input image point *xy* and returns by reference the corresponding point *uv* in the output image. Similarly, **evalXY()** computes the input image point *xy* from the output image point *uv*.

Figure 4-12 shows the result of applying ilTieWarpImg to an image. "Geometric Transformations" on page 386 illustrates the effect of other geometric transformation on an image.

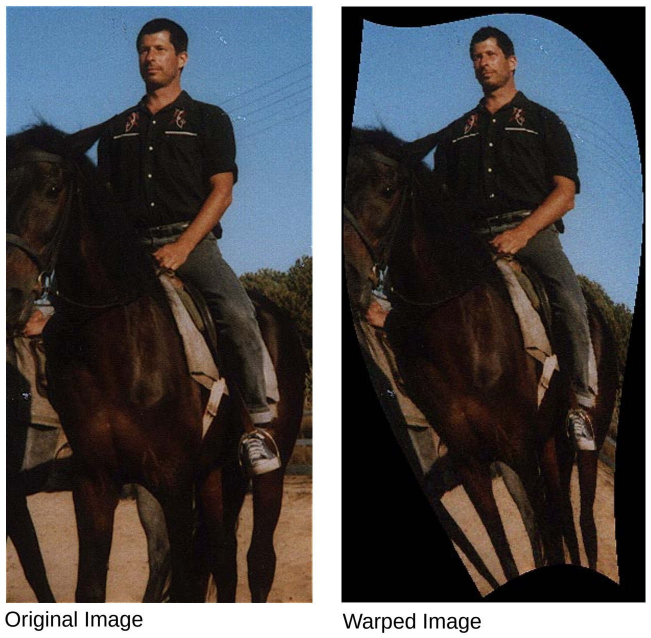


Figure 4-12 Warping an Image

Rotating, Zooming, and Flipping an Image

Unlike the various warping classes, the ilRotZoomImg operator is limited to performing two-dimensional affine transformations on an image. This single operator can rotate, zoom (magnify or minify), and mirror (or flip) image data:

```
ilRotZoomImg(ilImage* img=NULL, float rotAngle=0,
          float xzoom=1., float yzoom=1.,
          ilFlip flip=ilNoFlip,
          ilResampleType rs=ilAutoResamp);
```

The input image, *img*, is rotated by *rotAngle* degrees in a counterclockwise direction, and magnified or minified in the appropriate dimension by the *xzoom* and *yzoom* factors. The *flip* argument should be ilXFlip or ilYFlip to specify mirroring across the image's *x* or *y* axis; use ilNoFlip if you don't want the image mirrored (this is the default). The default resampling method is ilAutoResamp. This method, when there is no hardware acceleration, chooses ilMinify resampling for pure minification (x and y zoom factors < 1.0 and rotation angle = 0.0) and ilNearNb otherwise. If there is hardware acceleration, then ilBiLinear is chosen for pure minification and ilNearNb otherwise. This operator is especially efficient when the rotation is a multiple of 90 degrees and when the resampling method is ilNearNb.

Functions are provided for you to dynamically change all the parameters:

```
void setAngle(float rotAngle);
void setZoom(float x, float y);
void setCenter(float x, float y);
void setFlip(ilFlip flp);
```

An analogous set of functions is provided to retrieve the parameters:

```
float getAngle();
void getZoom(float& x, float& y);
int getCenter(float& x, float& y);
ilFlip getFlip();
```

You can also select a portion of the image to be operated on by using **setSize()** (inherited from ilImage) and **setCenter()**. (Alternatively, you can ask for only the desired portion using **getTile()** or **copyTile()** with the appropriate arguments, or you can define a region of interest.) The **setSize()** and **setCenter()** functions limit the transformation to the area specified with **setSize()**, centered on the point given in **setCenter()**. The center point is specified in the input image's coordinate space. These functions also translate the image's coordinate space so that the image's origin becomes the corner of the region specified by **setCenter()** and **setSize()**. You can clear the center point set with **setCenter()** by calling **clearCenter()**.

You can zoom the input image to a particular size by calling **sizeToFit()**:

```
void sizeToFit(float x, float y, int keepAspect=FALSE);
```

You specify the desired image width and height with *x* and *y*. If you want the image to keep its aspect ratio, set *keepAspect* to TRUE. The default behavior allows the image's aspect ratio to change.

Spatial Domain Transformations

Spatial operators transform image data by computing a weighted sum of the pixels in the neighborhood surrounding the target pixel. The size of the neighborhood and the weights used for neighboring pixel values are defined by the *kernel*. Some spatial operators predefine their kernels, while others allow the user to specify them. In addition, a method for handling pixels at the edge of the image must be specified, since a pixel's neighborhood is undefined beyond the edge of a page. The spatial operators provided with the IL are shown in Figure 4-13.

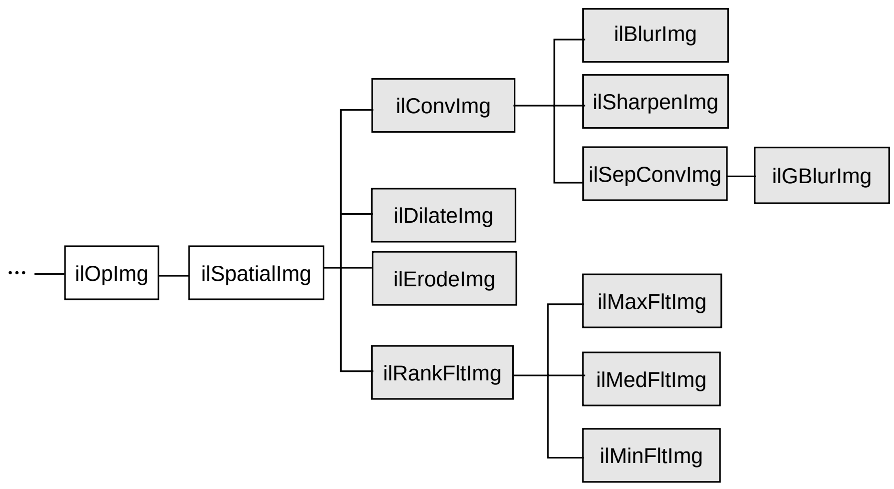


Figure 4-13 Spatial Domain Operator Inheritance Hierarchy

The ilSpatialImg class, which is an abstract class, defines the basic support for spatial operators that derive from it. The public functions it defines are those that allow you to set and retrieve the kernel and the edge-handling method (remember that some operators predefine their kernel and thus don't allow you to set it):

```
void setKernel(ilKernel* kern=NULL);
void setKernelSize(int x, int y, int z=1);
void getKernelSize(int& x, int& y, int& z);
void getKernelSize(int& x, int&y);
```

```
void setEdgeMode(ilEdgeMode eMode = ilPadSrc);
ilEdgeMode getEdgeMode();
```

The ilKernel class defines a kernel as consisting of five elements:

- the size of the kernel in the *x*, *y*, and *z* dimensions
- the size of the data type used to specify kernel weights
- a pointer to the data specifying the weights

The *x*, *y*, and *z* dimensions should be odd numbers so that a neighborhood can be exactly centered on a single, target pixel. If they're even numbers, the data may be shifted. See the reference page for ilKernel, *il/ilKernel.h*, and "Auxiliary Classes" on page 396 for more information about this class.

The *origin* of an ilKernel normally falls at its center pixel. The origin can be specified with ilKernel's **setOrigin()** function to correspond to any of the pixels in the kernel; the arguments *x*, *y*, and *z* indicate the origin's offset from the upper-left-front corner of the kernel. **getOrigin()** returns the offset by reference.

```
void setOrigin(int x, int y, int z=0);
void getOrigin(int &x, int &y, int &z);
```

ilSpatialImg's **setEdgeMode()** function specifies how the neighborhood is defined for pixels at the edge of the image. Explanations of the supported edge modes, which are defined in *il/ilTypes.h*, follow:

ilReflect	Sufficient data near the edge of the image is reflected so that a full-sized output image can be processed without producing artifacts at the image edge. This mode gives the best results for most operators.
ilWrap	Sufficient data is taken from the opposite edge of the source image so that a full-sized output image can be processed.
ilPadSrc	The edge of the input image is padded with the input image's fill value so that a full-sized output image can be processed (see Figure 4-14). See "Fill Value" on page 43 for more information on an image's fill value.

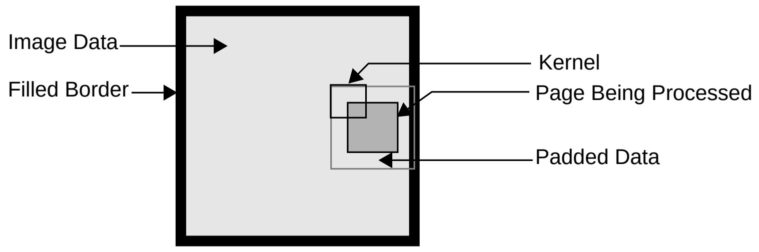


Figure 4-14 The ilPadSrc Edge Mode

ilNoPad	No padding is done, and the output image shrinks by the size of the kernel minus one in each dimension.
ilPadDst	Similar to ilNoPad, except that the output, or destination, image's border is sufficiently padded with its fill value so that the final image is the same size as the source image.

Convolving an Image

The ilConvImg operator performs general image convolution. This class isn't an abstract class, so you can use it directly to convolve image data. The constructor for ilConvImg, which is its only public member function, is shown below:

```
ilConvImg(ilImage* inputImage=NULL,
          ilKernel* inputKernel=NULL, double biasValue=0.0,
          ilEdgeMode edgeMode=ilPadSrc);
```

This function takes a pointer to the source or input image, a pointer to the kernel, and an enumerated type that matches one of the supported edge modes. The other argument, *biasValue*, is added to the weighted sum (image data multiplied by kernel weight) for each neighborhood. You can set the bias value with the **setBias()** function.

You can also perform certain convolutions more efficiently with a separable kernel (one that is specified by row and column vectors). The ilSepConvImg, descended from ilSpatialImg, provides this feature. Its constructor accepts the input image, the row and column kernels, the sizes of the kernels, an optional bias value, and an optional edge mode:

```
ilSepConvImg(ilImage *inputImg = NULL,
       float *xkernel=NULL, float *ykernel=NULL, int xsize=5,
       int ysize=5, double biasVal=0.0,
       ilEdgeMode eMode = ilPadSrc);
```

As shown, the default bias is 0.0, and the default edge mode is ilPadSrc. The default kernel size for each kernel is 5. This operator is especially efficient for kernel sizes 3 x 3, 5 x 5, and 7 x 7.

ilSepConvImg also defines a set of functions to set and get the kernel vectors:

```
void setXkernel(float *xval);
void setYkernel(float *yval);
float* getXkernel();
float* getYkernel();
```

setXkernel() allows you to change the row kernel; **getXkernel()** returns its value. **setYkernel()** allows you to change the column kernel; **getYkernel()** returns its value. If you replace either kernel with one that has a different size, don’t forget to use **setKernelSize()** (inherited from ilSpatialImg) to update the sizes.

Blurring or Sharpening an Image

The two blurring operators, ilBlurImg and ilGBlurImg, both blur an image by performing a convolution, but they use different kernels and algorithms for the convolution. ilBlurImg convolves the image with a blurring kernel using the general convolution algorithm defined by ilConvImg. ilGBlurImg (descended from ilSepConvImg) convolves an image with a separable two-dimensional Gaussian kernel. Because ilGBlurImg uses a separable kernel, it’s generally more efficient than ilBlurImg. Although different methods are used, often the blurred results don’t look significantly different. The reference pages for these classes provide more detailed information on the kernels and convolution algorithms used. Figure 4-15 shows an original image that’s used as an example in the following pages.

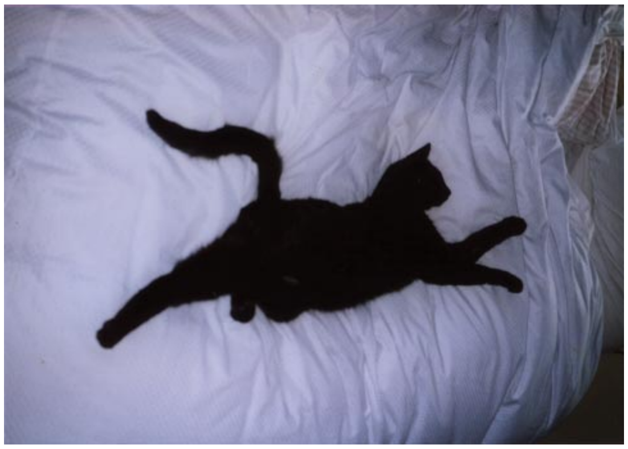


Figure 4-15 An Original Image

The ilBlurImg and ilGBlurImg classes have slightly different interfaces:

```
ilBlurImg(ilImage* inputImage, float blurFactor=1.0,
   float radius=2.0, ilEdgeMode edgeMode=ilPadSrc);
```

```
ilGBlurImg(ilImage* inputImage, float blurFactor=1.0,
   int xsize=5, int ysize=5, double biasVal=0.0,
   ilEdgeMode edgeMode=ilPadSrc);
```

Both constructors take as arguments a pointer to the source image, a blur factor ranging from 0.0 (no blur) to 1.0 (maximum blur), and an enumerated type specifying the edge mode. By default, the blur factor is set to 1.0, and the edge mode is ilPadSrc. The *radius* argument for ilBlurImg (with a default value of 2.0) and the *xsize* and *ysize* arguments for ilGBlurImg (with default values of 5) control the size of the kernel used for blurring. (The ilBlurImg kernel size is equal to 1+*radius**2.) ilGBlurImg's *biasValue* argument, which by default is zero, is added to the final weighted sum.

Both classes allow you to dynamically modify the amount of blur by passing a **float** value to the **setBlur()** function. You can also change the size of the kernel with **setBlurRadius()** (for ilBlurImg) or **setBlurKernelSize()** (for ilGBlurImg). An image blurred with ilBlurImg is shown in Figure 4-16.

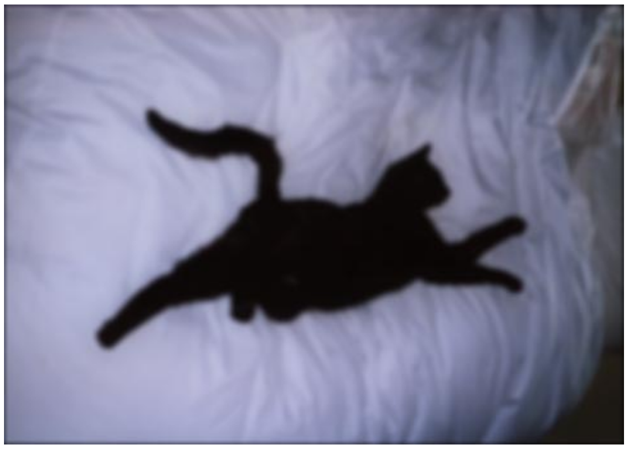


Figure 4-16 An Image Blurred with ilBlurImg

The ilSharpenImg class is similar to ilBlurImg, except that instead of using a kernel that blurs, it uses a kernel that sharpens the image data. Its constructor takes a similar set of arguments:

```
ilSharpenImg(ilImage* img, float sharpness=0.5,
          float radius=1.5, ilEdgeMode edgeMode=ilPadSrc);
```

The sharpness factor indicates the degree of sharpening that should occur. This factor can have a value between 0.0 and 1.0, with a default value of 0.5. A sharpened image appears in Figure 4-17.

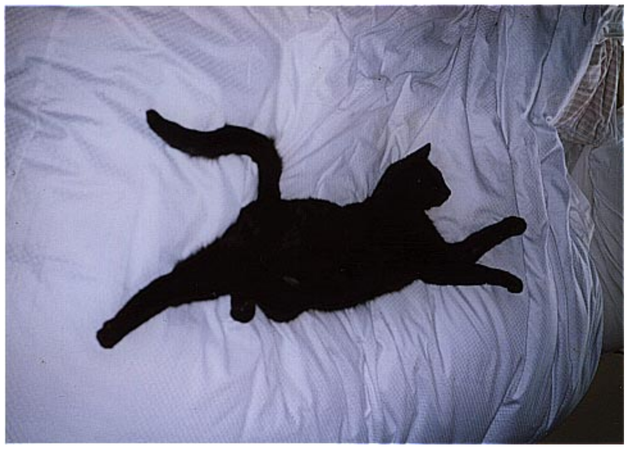


Figure 4-17 An Image Sharpened with ilSharpenImg

As with ilBlurImg, you can dynamically change the sharpness factor (with **setSharpness()**) and the size of the radius (with **setSharpenRadius()**). **getSharpness()** and **getSharpenRadius()** are the query methods that return the values of the sharpness factor and radius. Making the size of the radius too large or repeatedly cycling an image through the sharpening operation can result in a grainy, high-contrast image. Figure 4-18 shows an example of this.

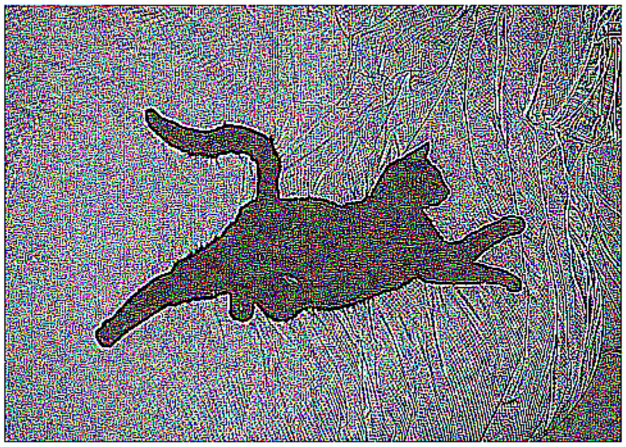


Figure 4-18 An Over-sharpened Image

To see additional illustrations of the ilBlurImg and ilGBlurImg transformations, refer to “Spatial Domain Transformations” on page 387.

Rank Filtering an Image

The ilRankFltImg class performs two-dimensional rank filtering, which is typically—though not exclusively—done on black-and-white images. It involves sorting all the pixel values (for each channel) for a neighborhood of pixels. Then, the target pixel is assigned the values corresponding to a specified rank. For example, suppose you have chosen a 3 x 3 neighborhood and a desired rank of 0 (the minimum). In this case, each pixel is assigned the lowest value found among itself and its eight surrounding pixels.

The classes that derive from ilRankFltImg—ilMinFltImg, ilMaxFltImg, and ilMedFltImg—assume that the desired rank is the minimum possible rank, the maximum possible rank, and the median, respectively. Median filtering is useful for removing binary, or impulse, noise in image data. Minimum and maximum rank filtering produce morphological erosion and dilation. An example of an image processed with ilMedFltImg appears in Figure 4-19.

The only public member function defined by these three classes is a constructor, and each of these constructors takes the same set of arguments. ilMinFltImg’s constructor is shown below:

```
ilMinFltImg(ilImage* inputImage = NULL,
          ilEdgeMode edgeMode = ilPadSrc,
          ilKernel* inputKernel = NULL);
```

As shown, you need to specify the input image, how pixels at the edge of the image are to be handled, and the kernel. The kernel is treated as a mask. Only nonzero elements are included in the neighborhood; the rest are ignored, as are the kernel weights.

The constructor for the ilRankFltImg superclass takes the same set of arguments and an additional one for specifying the desired rank for the target pixel:

```
ilRankFltImg(ilImage* inputImage = NULL, int filterRank = -1,
          ilEdgeMode edgeMode = ilPadSrc,
          ilKernel* inputKernel = NULL);
```

The default rank of minus 1 indicates that median rank should be used. You can dynamically change the desired rank with the **setRank()** function; you can also determine what the maximum possible rank is with **getMaxRank()**.

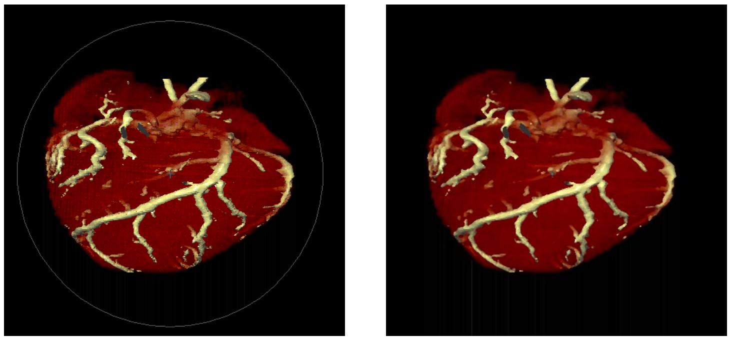


Figure 4-19 Median Rank Filtering on an Image

To see additional illustrations of the rank filtering transformations, refer to “Spatial Domain Transformations” on page 387.

Morphological Operators

Morphological operators include shape-dependent, nonlinear image transformations such as erosion and dilation. The operators implemented in the IL, ilDilateImg and ilErodeImg, can be used on 1-D, 2-D or 3-D data sets. More powerful morphological operations such as “opening” and “closing” can be performed by chaining together dilation and erosion operations. Opening can be accomplished by an erosion followed by a dilation; closing can be done with a dilation followed by an erosion.

These operations are defined on binary or grayscale images. Note that you can operate on color images if you remember that “binary” and “grayscale” indicate how the pixel values or intensities in each channel of the image are interpreted. A binary image contains no more than two levels or intensity values: zero and not zero. An 8-bit image with 256 pixel intensities can be treated as a binary image by collapsing the intensities into two groups; for example, a zero pixel intensity could be represented with a zero, and all intensities between 1 and 255 could be represented with a nonzero value. A grayscale image, of course, includes more than two intensity values. Thus, an 8-bit image can be treated as an input image with 256 pixel intensities. Typically the image has a single channel. (For multichanneled input, the operations are performed on each channel independently.)

Both ilErodeImg and ilDilateImg are derived from ilSpatialImg and thus involve moving a kernel across an image, but the operation performed isn't a computed sum. Instead, in morphological operations, the kernel is called a ‘‘structuring element’’ (SE) and is represented by an ilKernel. The SE, like the input image, can be interpreted as binary or grayscale. When applied to an image, a morphological operator returns a quantitative measure of the image's geometrical structure in terms of the SE.

The interpretation of the numbers that make up an SE depends on the type of morphological operation being performed. Negative SE elements are always treated as logical "don't cares"; when the operation is in progress, image pixels under negative SE elements are ignored. Thus, the support of the SE is limited to those elements that are nonnegative. This permits the creation of odd-shaped SEs. The image pixel under the origin is the one potentially modified. (You can change the origin of the SE by using ilKernel's **setOrigin()** method. The default is in the center of the SE.)

The result of erosion or dilation on a binary image (regardless of whether the SE is binary or grayscale) is to turn every pixel either "on" or "off." A pixel in the output image can then be assigned one of two intensities, corresponding to whether it's on or off. These two intensities are typically the maximum and minimum values of the operator image, which can be set using **setMaxValue()** and **setMinValue()** (inherited from ilImage). If they're not explicitly set, the maximum and minimum values are inherited from the input image. For the example of an 8-bit image, the minimum value might be 0 and the maximum 255. A pixel that's 0 in the input image might have a value of 255 in the output image, and a nonzero input pixel might be 0 in the output.

The interpretation of the image or the SE as binary or grayscale can be controlled through the enumerated type ilMorphType, as described below.

- If the input image and the SE are binary (ilMorphType = ilBinBin), then the SE is used to perform a hit-or-miss transformation. That is, if a zero image pixel falls under a zero SE element, or if a nonzero image pixel falls under a nonzero SE element, the image pixel beneath the SE origin is turned on (assigned the maximum value) for dilation and turned off (assigned the minimum value) for erosion. Typically, for binary images, an SE is composed of negative and positive 1s.

- If the input image is binary and the SE type is grayscale (ilMorphType = ilBinGray), then the nonnegative SE elements determine the support area. In other words, image pixels under negative SE elements are ignored, but if a positive image pixel falls under a non-negative SE element, then the target pixel (under the SE origin) is turned on for dilation or off for erosion.
- If the input image is grayscale and the SE type is binary (ilMorphType = ilGrayBin), then the maximum or minimum (depending on whether dilation or erosion is being performed, respectively) of image pixels falling under positive SE elements is computed.
- If the input image and the SE are grayscale and a “set” operation is desired (ilMorphType = ilGrayGraySet), then the maximum or minimum (depending on whether dilation or erosion is being performed) of image pixels falling under nonnegative SE elements is computed.
- If a “function” operation is desired (ilMorphType = ilGrayGrayFct), then the computation is the same as for ilGrayGraySet, except that the SE elements are added to the image pixels before computing the minimum or maximum.

The constructors for erosion and dilation are shown below:

```
ilDilateImg(ilImage* inputImage = NULL,
             ilMorphType mtype = ilBinGray,
             ilKernel* se = NULL, double biasVal = 0.,
             ilEdgeMode eMode = ilPadSrc);
```

```
ilErodeImg(ilImage* inputImage = NULL,
             ilMorphType mtype = ilBinGray,
             ilKernel* se = NULL, double biasVal = 0.,
             ilEdgeMode eMode = ilPadSrc);
```

Each operator accepts a pointer to an input image (*inputImage*), a specification of the type of morphological operation (*mtype*), a *structuring element* (the ilKernel pointer *se*), a bias (*biasVal*), and an edge mode (*eMode*).

The morphological transform types, which are members of the enumerated type ilMorphType (defined in *il/ilTypes.h*), are summarized below. These types define whether data in the image and the structuring element (SE) is treated as binary (that is, having a zero or a nonzero value) or as grayscale (that is, with an appropriate range for its data type).

BinBin	Dilation or erosion on a binary image with a binary SE.
BinGray	Dilation or erosion of a binary image with a grayscale SE. The operation is performed over the support of nonnegative SE elements.
GrayBin	Dilation or erosion of a grayscale image with a binary SE. The operation is performed over the positive support of the SE.
GrayGraySet	Dilation or erosion of a grayscale image with a grayscale SE. The operation is performed over the nonnegative support of the SE.
GrayGrayFct	Dilation or erosion of a grayscale image with a grayscale SE. The dilation or erosion is performed as a function operation over the nonnegative support of the SE; that is, the SE elements are added to the image pixels before the dilation or erosion is performed.

Both ilDilateImg and ilErodeImg define these two functions:

```
void setMorphType(ilMorphType type);
ilMorphType getMorphType();
```

setMorphType() allows you to set the type of morphological operation and **getMorphType()** returns the type of operation.

Edge Detection

The operators described in this section are gradient operators that produce edge-enhanced images by performing orthogonal convolutions with particular kernels. This section focuses on how to use these operators rather than on the specific algorithm implemented by each of these operators. For more information about the algorithms, see the reference pages for the specific class.

The classes described in this section inherit directly or indirectly from ilSpatialImg, as shown in Figure 4-20.

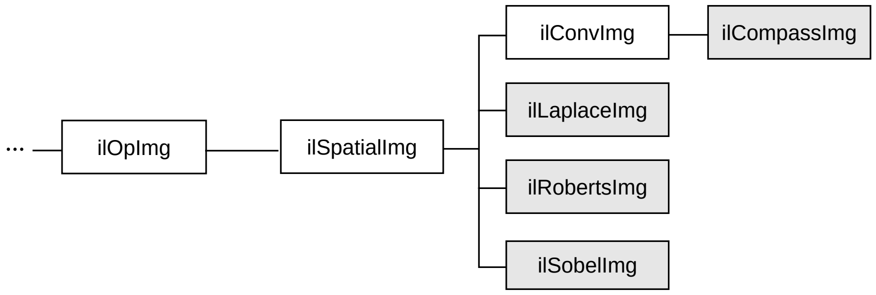


Figure 4-20 Edge Detection Operator Inheritance Hierarchy

The constructors for the ilRobertsImg and ilSobelImg operators take the same arguments:

```
ilRobertsImg(ilImage* inImg = NULL, double biasVal = 0.0,
          ilEdgeMode eMode = ilPadSrc);
ilSobelImg(ilImage* inImg = NULL, double biasVal = 0.0,
          ilEdgeMode eMode = ilPadSrc);
```

The image to be transformed is specified by *inImg*. The other two arguments, which have default values, indicate a bias value to be added as each pixelwise convolution is performed and how pixels at the edge of a page are to be handled. These arguments have the same meaning as the ones supplied in the ilConvImg constructor, which is described in the preceding section. As explained in more detail in the reference pages, these operators perform two orthogonal two-dimensional convolutions, which are then combined with predefined kernels. The resulting images are edge-enhanced images. An example image produced by ilRobertsImg is shown in Figure 4-21.

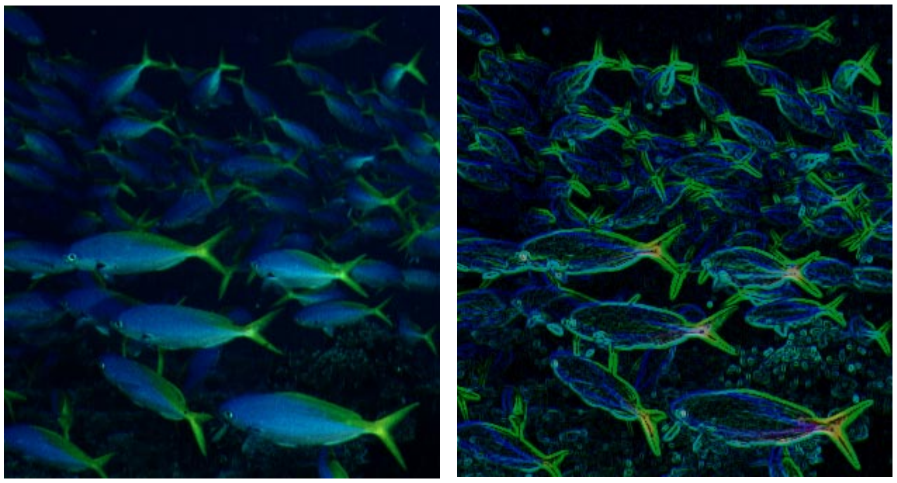


Figure 4-21 Edge Image Produced by ilRobertsImg

The constructor for the ilLaplaceImg operator uses the same arguments as the constructors shown above, plus an additional argument that allows you to select one of two predefined kernels:

```
ilLaplaceImg(ilImage* inImg = NULL, double biasVal = 0.0,
          ilEdgeMode eMode = ilPadSrc, int kerno = 1);
```

The *kerno* argument can be either 1 or 2; the corresponding kernels are listed in the reference page for ilLaplaceImg. You can use **setKernel()** to specify either kernel after you've created an ilLaplaceImg object.

A compass operator measures gradients in a specified direction. The ilCompassImg operator allows you to specify the desired direction as an angle between 0 and 360 degrees or as one of eight compass points. You can also specify the size of the kernel to be used. Once all this information is supplied, a square kernel is generated, which is then convolved with the image data. Here's the class constructor:

```
ilCompassImg(ilImage* inImg= NULL,
  float angleDir = ilCompassN, double biasVal = 0.0,
  int kernSize = 3,ilEdgeMode edgeMode = ilPadSrc);
```

An example image produced by using ilCompassImg appears in Figure 4-22.

The *angleDir* argument can be a number or one of the following values (see Table 4-2), which correspond to the compass points.

Table 4-2 Compass Directions for the ilCompassImg Operator

Value	Angle (in degrees
ilCompassN	0
ilCompassNE	45
ilCompassE	90
ilCompassSE	135
ilCompassS	180
ilCompassSW	225
ilCompassW	270
ilCompassNW	315

North, or 0 degrees, is the top of an image (as it's displayed using ilDisplay); angles are measured from north in a clockwise direction. The bias value and edge mode arguments for the constructor have the same meaning as those for ilLaplaceImg. Since the kernel is always square, only one dimension of its size needs to be specified. You can set and retrieve the bias value with **setBias()** and **getBias()**, which are defined by ilOpImg.

Once you've created an ilCompassImg operator, you can dynamically change the direction of the gradient with either **setAngle()** or **setXYWt()**:

```
void setAngle(float angleDir = ilCompassN);
void setXYWt(float Xwt = 0.0, float Ywt = 1.0);
```

The **setXYWt()** function specifies weights in the *x* and *y* dimensions, which are then used to generate the kernel. The ilCompassImg reference page describes in more detail how the kernel is generated from the angle or weights.

You can query an ilCompassImg about its angle or weights with these functions:

```
float getAngle();
void getXYWt(float& Xwt, float& Ywt);
```

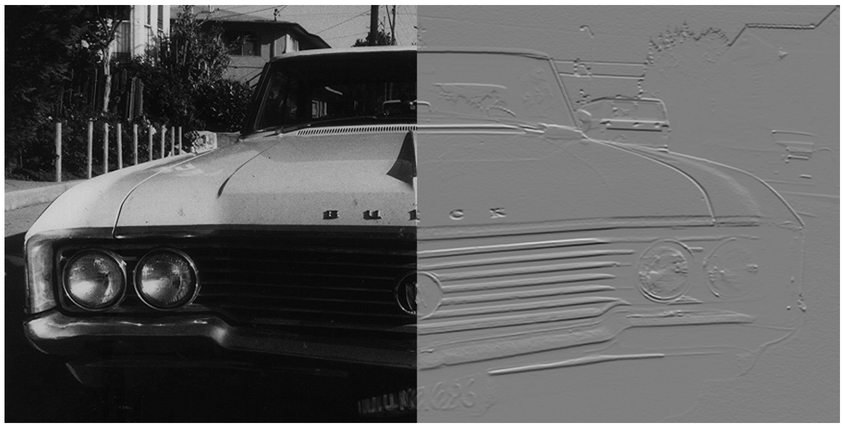


Figure 4-22 A Compass Filtered Image

Frequency Domain Transformations

It's often convenient to manipulate data in the frequency domain, particularly when restoring, enhancing, or removing noise from images. The ilRFFTfImg operator described in this section performs a forward fast Fourier transform (FFT) on an image (containing "real-valued" data, not complex). Once you've converted an image into the frequency domain, you can use any of the numerous Fourier operators to manipulate the image data. Then, when you're finished, you can use ilRFFTiImg, which performs an inverse FFT, to convert back to the spatial domain. Figure 4-23 shows the frequency domain operators and how they fit into the IL inheritance hierarchy.

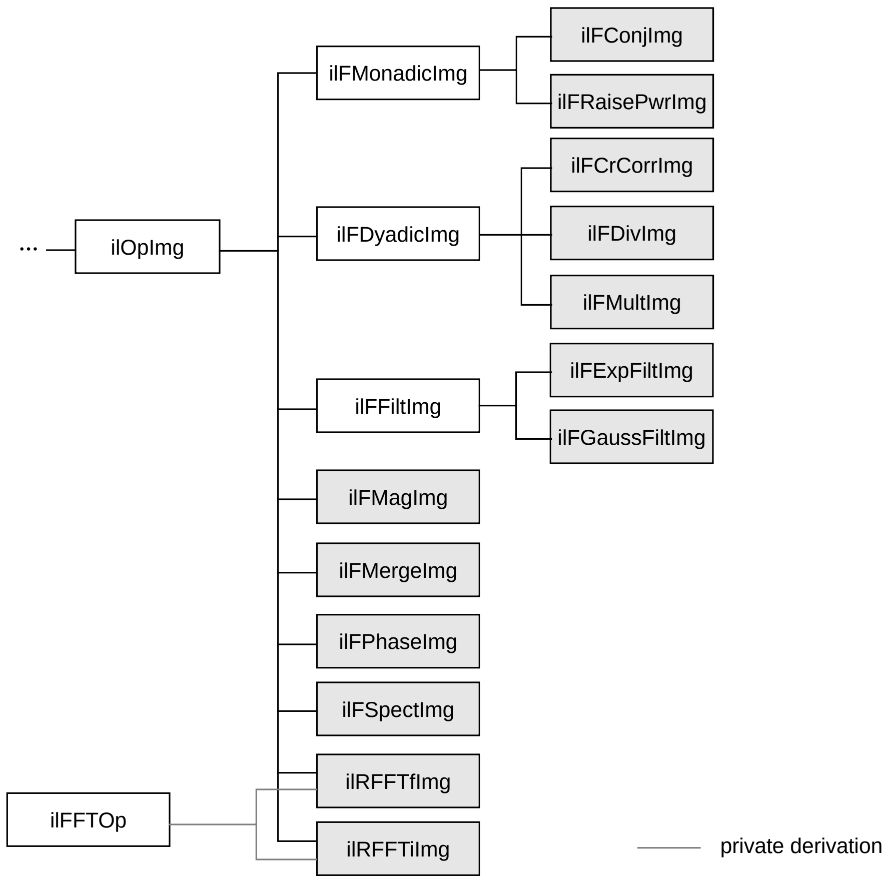


Figure 4-23 Frequency Domain Operator Inheritance Hierarchy

Forward and Inverse Fourier Transforms

As shown in Figure 4-23, both ilRFFTfImg and ilRFFTiImg inherit publicly from ilOpImg and privately from ilFFTOp. You should think of these two classes as operators that simply use the forward and inverse transform functions defined by ilFFTOp. This class, which doesn't derive from any superclass, defines functions that do the actual computation necessary to perform forward and inverse FFTs. The ilFFTOp class also defines a function that determines the average power spectrum of an image. Both ilRFFTfImg and ilRFFTiImg try to set the page size large enough to hold an entire channel of the image. If your image is too large, you can use ilFFTOp's functions directly on each channel in turn. Remember that even though ilFFTOp applies its algorithms to ilImages, it doesn't derive from ilOpImg and thus can't be linked into an operator chain as ilRFFTfImg and ilRFFTiImg can. Multiprocessing on an ilFFTOp object can be turned on or off and queried using the **enableMP()** and **isMPenabled()** functions.

The FFTs are performed using the Prime Factor algorithm, using floating point arithmetic. (For more information on the specifics of this algorithm, see the ilFFTOp reference page and the article "Symmetric FFTs," by Paul N. Swarztrauber, *Mathematics of Computation*, Vol. 47, Number 175, July 1986, pp. 323-346.) The only restriction this algorithm places on the input image is that it have a real (non-complex) data type other than ilBit. However, the algorithm is most efficient if the image already contains floating point data (so it doesn't have to be converted for processing and then converted back again), has an ilSeparate order, and has dimensions that are products of small primes. Dimensions that are a power of two yield the most efficient computation. The reference pages for each of the Fourier operators described in this section contain more information about the methods used to perform the computations as well as hints about how to achieve the greatest possible efficiency.

The constructor for the ilRFFTfImg operator and the member function **ilRfft()** perform a forward FFT.

```
ilRFFTfImg(ilImage* src=NULL, short option=3);
ilRfftf(ilImage src, int srcCh, ilImage* dst, int dstCh,
              short option=3);
```

Using the ilRFFTfImg operator to perform a forward FFT is relatively easy. The first argument is a pointer to the source image that's to be transformed. The second argument, called *option*, allows you to choose whether a one- or two-dimensional transform is performed; if it's:

- 1, a one-dimensional FFT is performed on the rows of data
- 2, a one-dimensional FFT is performed on the columns of data
- 3, a two-dimensional FFT is performed (the default)

You can dynamically change this parameter with the **setOption()** function.

If you're using ilFFTOp to perform a forward FFT, you have to first create an ilFFTOp object and then use its **ilRfftf()** member function on a particular channel of the source image. In the following example, assume that *srcImg* already exists and is a pointer to an ilImage.

```
ilFFTOp myFFTOp;
ilImage* destImg;
myFFTOp.ilRfftf(srcImg, 0, destImg, 0, 3);
```

The first four arguments to ilFFTOp's **ilRfftf()** function specify which channel of the source image is to be transformed and into which channel of the destination image the result should be put. In this example, channel 0 of *srcImg* is transformed and placed into channel 0 of *destImg*; the size of both of these images must be the same. The last argument for this function specifies which of the three options described above is desired. (It has the same meaning as the second argument to the ilRFFTfImg constructor.)

Since the source image must contain real data (not complex numbers), the output is conjugate-symmetric. In other words, only two of the four quadrants are unique, and only these are computed for the output. The output is complex, however, so both the real and imaginary results must be reported; because of this, the destination image has the same *x* and *y* dimensions as the source image. Table 4-3 shows the format of the output from either the ilRFFTfImg operator or ilFFTOp's **ilRfftf()** function. (The origin is in the upper left corner.)

Table 4-3 Output of a Forward Fourier Transform (if *nx* and *ny* are even)

	0	**1**	**2**	**3**	**4**	**...**	***nx*-3**	***nx*-2**	***nx*-1**
0	real	real	imag	real	imag	...	real	imag	real
1	real	real	imag	real	imag	...	real	imag	real
2	imag	real	imag	real	imag	...	real	imag	imag
3	real	real	imag	real	imag	...	real	imag	real
4	imag	real	imag	real	imag	...	real	imag	imag
...	...	...	...	...	...	...	...	...	...
ny-3	real	real	imag	real	imag	...	real	imag	real
ny-2	imag	real	imag	real	imag	...	real	imag	imag
ny-1	real	real	imag	real	imag	...	real	imag	real

Columns 1 through *nx*-2 contain the real and imaginary components of a complex transform; for example, column 1 contains the real component and column 2 the corresponding imaginary component of the first complex FFT output. The column 0 represents the 0-frequency (or DC) component, and column *nx*-1 represents the highest (Nyquist) frequency along the *x*-direction. These two columns resemble the output of a real-valued FFT. In the example shown, both *nx* and *ny* are assumed to be even. If *nx* were odd, the Nyquist column would be missing. If *ny* were odd, the last row shown would be missing. Table 4-4 shows the output format if both *nx* and *ny* are odd.

Table 4-4 Output of a Forward Fourier Transform (if *nx* and *ny* are odd)

	0	**1**	**2**	**3**	**4**	**...**	***nx*-2**	***nx*-1**
0	real	real	imag	real	imag	...	real	imag
1	real	real	imag	real	imag	...	real	imag
2	imag	real	imag	real	imag	...	real	imag
3	real	real	imag	real	imag	...	real	imag
4	imag	real	imag	real	imag	...	real	imag
...	...	...	...	...	...	...	...	...
ny-2	real	real	imag	real	imag	...	real	imag
ny-1	imag	real	imag	real	imag	...	real	imag

This format is what's expected as input by all the Fourier operators described in this section. In particular, the constructor for the ilRFFTiImg operator and the **ilRffti()** member function of the ilFFTOp class both expect this format in their source image. They perform an inverse FFT, which is to say they convert the input Fourier data back to the spatial domain:

```
ilRFFTiImg(ilImage* src = NULL, short option = 3);
```

```
ilStatus ilRffti(ilImage* src, int srcCh,
          ilImage* dst, int dstCh, short option = 3);
```

The ilRFFTiImg constructor takes a pointer to the source image and the same *option* argument described above. (The ilRFFTiImg operator also defines the same **setOption()** function described above.) For the **ilRffti()** function, the source and destination images (*src* and *dst*) must be the same size; the *srcCh* and *dstCh* arguments specify the channel to be transformed and the destination channel number. Both the constructor and the function produce output data that's real. The output of the forward transform is multiplied by 1.0 / (*nx***ny*) so that the forward transform followed by the inverse returns the original image unscaled.

The ilFFTOp class also defines functions that allow you to control the size of the buffer it uses to hold data that's being transformed. Ideally, this buffer is large enough to hold an entire channel of the source image. By default, it holds 4 MB, which is just enough for one channel of a 1-MB image. You can set the size of the buffer with **setBufSize()**, which takes an **int** argument that specifies the desired size of the buffer in bytes. Remember that computations are performed using floating point data, so the number of pixels that the buffer can hold is its size in bytes divided by four (size of float). You can retrieve the current size of the buffer with **getBufSize()**.

Another useful function defined by ilFFTOp, **ilFFTAvg()**, computes the average power spectrum of an image. This function uses a relatively complicated algorithm that's described in some detail in the ilFFTOp class reference page.

Separating the Magnitude and Phase Components

The operators described in this section allow you to separate the magnitude and phase components of a complex Fourier image so that you can process or filter them independently, and then combine them into a complete image when you're finished. Such an operator chain would look like Figure 4-24.

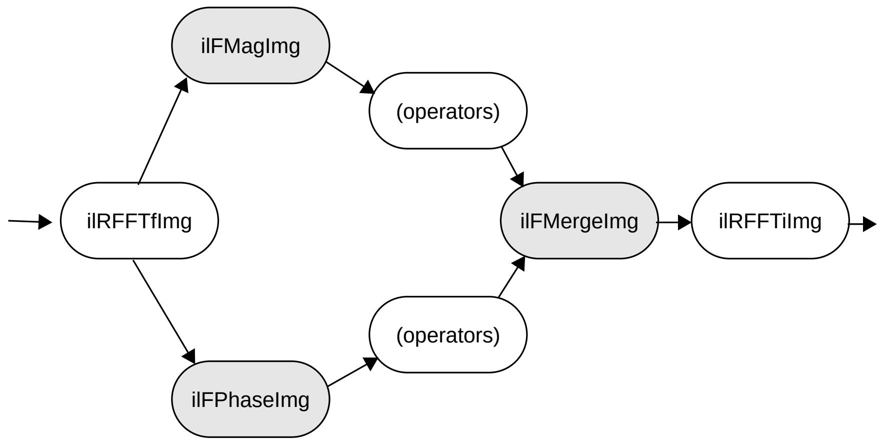


Figure 4-24 Magnitude and Phase Fourier Operators

As you might expect from their names, the ilFMagImg operator computes the magnitude of an input complex Fourier image, and ilFPhaseImg determines the phase component. The constructors for both of these operators expect the format produced by ilRFFTfImg (which is described above):

```
ilFMagImg(ilImage* src = NULL);
ilFPhaseImg(ilImage* src = NULL);
```

The *x* -dimension of the output image for both these operators is half of the input image's size, plus one; the *y* dimension is unchanged. The *x* dimension shrinks because the input image uses two columns for each Fourier element, one for the real component and one for the imaginary, whereas the magnitude and phase aren't complex. (For a complex number represented by $a + ib$, the

magnitude is $\sqrt{a^2 + b^2}$ and the phase is $\operatorname{atan}(b/a)$.)

An operator that's similar to ilFMagImg, ilFSpectImg, computes the spectrum of a Fourier image. The computation is the same as that performed by ilFMagImg, but all quadrants are represented in the output image, not just the two that are unique. As a result, the size of the output image is the same as that of the input image, and the origin of the output image is at its center rather than its upper left corner. You might use an ilFSpectImg object for displaying, although you probably want to scale the spectral values using ilHistScaleImg. (This operator is described in "Radiometric Transformations" on page 166.) An ilFMagImg object is more efficient for processing since redundant calculations aren't performed.

The constructor for ilFSpectImg simply takes a pointer to the source image:

```
ilFSpectImg(ilImage* src = NULL);
```

The ilFMergeImg operator merges an ilFMagImg and an ilFPhaseImg to produce the original whole Fourier image. The merged image is converted from polar to rectangular form so that it's in the format expected by ilRFFTiImg. The constructor for ilFMergeImg takes pointers to the two images and an **int** that specifies the desired *x* dimension of the final image:

```
ilFMergeImg(ilImage* mag, ilImage* phase, int xsize);
```

The *xsize* argument is required because the *x* dimension of a merged image can't be uniquely determined from the *x* dimension of *mag* or *phase*. For example, if *mag* and *phase* have *x* dimensions of 129, the merged image could have an *x* dimension of either 256 or 257. You can explicitly set the *x* dimension with **setXsize()**.

Filtering

Two filter operators are provided for use on Fourier images: ilFExpFiltImg and ilFGaussFiltImg. These operators derive from ilFFiltImg, an abstract class that implements the basic support for frequency domain filtering. (You can derive your own filter as described in “Deriving from ilFFiltImg” on page 299.) Both ilFExpFiltImg and ilFGaussFiltImg expect input in the format produced by ilRFFTfImg; typically, you’ll apply the ilRFFTiImg operator to the filtered image in order to view the results in the spatial domain.

The constructors for these operators are shown below:

```
ilFExpFiltImg(ilImage* src, float alpha, float beta,
          float gamma, float eccent, float theta);
```

```
ilFGaussFiltImg(ilImage* src, float hfgain, float dcgain,
          float minhalf, float majhalf, float theta);
```

For more information about what these arguments mean, see the filter equations below and the reference pages for these two operators.

This is the filtering equation used by ilFExpFiltImg:

$$H(u, v) = \alpha + \beta e^{\{\gamma[(a_{11}u + a_{12}v)^2 + (a_{21}u + a_{22}v)^2]\}}$$

This is the filtering equation used by ilFGaussFiltImg:

$$H(u, v) = hf + (dc - hf)\, e^{-\{(a_{11}u + a_{12}v)^2 + (a_{21}u + a_{22}v)^2\}}$$

where for both equations:

H() = transfer function of the filter

u,*v* = two-dimensional frequency coordinates

$$a_{11} = \frac{\sigma_S \cos\theta'}{xSize},\ a_{12} = \frac{\sigma_S \sin\theta'}{ySize},\ a_{21} = -\frac{\sigma_L \sin\theta'}{xSize},\ a_{22} = \frac{\sigma_L \cos\theta'}{ySize}$$

$\theta' = \frac{\pi\theta}{180}$, where θ = angle in degrees of the filter’s orientation

xSize = *x* dimension of the source image *ySize* = *y* dimension of the source image

and where for ilFExpFiltImg:

α = high-frequency asymptote

β = decay coefficient

γ = exponential decay coefficient

$\sigma_S = 1.0$ and $\sigma_L = \frac{1}{\sqrt{1-\varepsilon^2}}$ where ε = eccentricity of equal contours of the filter

and where for ilFGaussFiltImg:

hf = gain of filter at the Nyquist (highest) frequency

dc = gain of filter at zero frequency

$\sigma_S = \sqrt{\frac{0.693147}{minHalf^2}}$ and $\sigma_L = \sqrt{\frac{0.693147}{majHalf^2}}$

minHalf = frequency of half-power point along the minor elliptical axis

majHalf = frequency of half-power point along the major elliptical axis

Table 4-5 shows two examples of specific values that might be passed in for ilFGaussFiltImg.

Table 4-5 Sample Parameter Values for ilFGaussFiltImg

Parameter	**High-pass**	**Low-pass**
dc	0.004	1.0
hf	3.0	0.002
minHalf	0.01	0.05
majHalf	0.01	0.05
θ (theta)	0.0	0.0

The high-pass values create a two-dimensional circular high-pass filter with a cutoff value of 0.01 on both axes; its DC gain is 0.004, and its gain at the highest frequency is 3.0. A high-pass filter diminishes the constant or slowly changing portions of an image and thereby accentuates the edge portions (creating a high-contrast, edge image). The low-pass values create a two-dimensional circular low-pass filter with a cutoff value of 0.05 on both axes; its DC gain is 1.0, and its gain at the highest frequency is 0.002. A low-pass filter diminishes the dramatically changing values at edges in an image and thereby accentuates the constant or slowly varying portions (creating a blurry image). See Figure 4-25 and Figure 4-26.

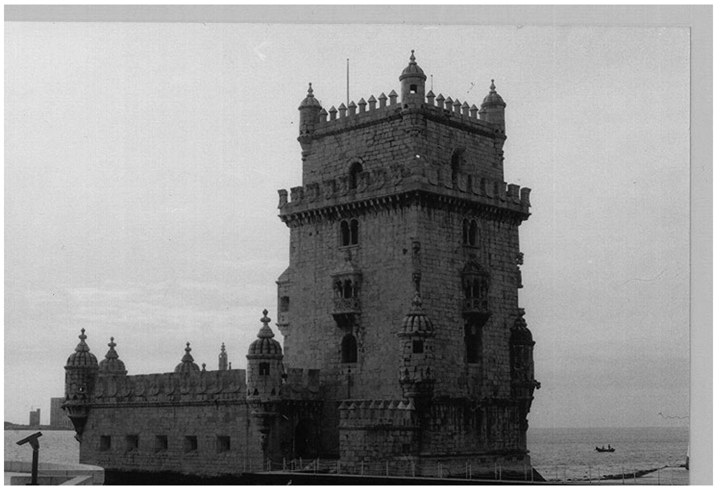


Figure 4-25 Original Image

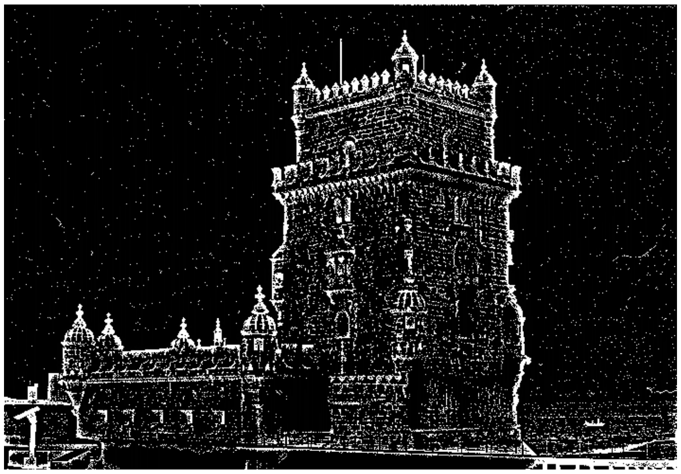


Figure 4-26 Image Processed with ilFGaussFiltImg

Functions are defined to set the value of all the parameters used in the constructors for both operators:

```
void setAlpha(float val);
void setBeta(float val);
void setGamma(float val);
void setEccent(float val);
void setTheta(float val);

void setHFgain(float val);
void setDCgain(float val);
void setMinHalf(float val);
void setMajHalf(float val);
void setTheta(float val);
```

See the reference pages for more information about these functions.

Single-input Operators

The two operators described in this section are ilFConjImg and ilFRaisePwrImg, both of which derive from ilFMonadicImg. (See “Deriving from ilFMonadicImg or ilFDyadicImg” on page 295 for more information about deriving your own operator from this class.) ilFConjImg and ilFRaisePwrImg expect a source image in the format produced by ilRFFTfImg. Typically, you’ll need to convert ilFRaisePwrImg’s output to the spatial domain by using ilRFFTiImg. (You won’t typically need to convert the result of applying ilFConjImg to an image back to the spatial domain; usually, it is used in the middle of a chain of operators in the frequency domain.)

As its name suggests, ilFConjImg computes the complex conjugate of an image; it also multiplies the complex values by a real factor:

```
ilFConjImg(ilImage* src=NULL, float scale = 1.0);
```

The *scale* argument is used to multiply or scale the values; the default value of 1.0 results in no scaling. You can change the scaling factor with **setScale()**. ilFConjImg is useful in computing the magnitude squared of the Fourier transform. For example, assume *theImg* is a pointer to a valid ilImage in the spatial domain:

```
ilRFFTfImg forwardImg(theImg);
ilFConjImg conjugateImg(&forwardImg);
ilFMultImg magSquaredImg(&forwardImg, &conjugateImg);
```

You can then display *magSquaredImg*.

The ilFRaisePwrImg operator raises the natural log of the magnitude values of a Fourier image by a power, exponentiates the result, and writes the values back in complex rectangular form:

$e^{(ln|m|)^p}$ where $|m|$ = magnitude and p = specified power

This root-filtering operation is useful for image sharpening. The constructor for this class is shown below:

```
ilFRaisePwrImg(ilImage* src, float power);
```

The log of the magnitude values of the source image, *src*, are raised by *power*, exponentiated, and converted back to complex rectangular form. The valid range for *power* is 0.0-1.0. You can set this value dynamically with **setPower()**.

Dual-input Operators

Three operators take two Fourier images as inputs:

- ilFCrCorrImg, which computes the cross-correlation of two images
- ilFMultImg, which multiplies two images
- ilFDivImg, which divides two images

These classes derive from ilFDyadicImg, which implements the basic support for dual-input Fourier operators, and they expect input images in the format produced by ilRFFTfImg. To convert the processed data back to the spatial domain, you need to apply the inverse transform implemented by ilRFFTiImg. See “Deriving from ilFMonadicImg or ilFDyadicImg” on page 295 for more information about deriving your own dual-input Fourier operator.

The constructors for ilFCrCorrImg, ilFMultImg, and ilFDivImg expect two images, which must be the same size:

```
ilFCrCorrImg(ilImage* src1 = NULL, ilImage* src2 = NULL);
ilFMultImg(ilImage* src1 = NULL, ilImage* src2 = NULL);
ilFDivImg(ilImage* src1 = NULL, ilImage* src2 = NULL,
        int ckDiv = 1);
```

To compute the cross-correlation, ilFCrCorrImg multiplies *src1* by the conjugate of *src2* and then normalizes the result using the DC (or (*0,0*)) coefficient of *src1*. One of the principal applications of cross-correlation in image processing is in prototype matching, where one tries to match a given unknown image to a known image. The closest match can be found by selecting the image that yields the correlation function with the largest value.

Multiplying two Fourier images is equivalent to convolving them in the spatial domain. Since the Fourier algorithm is very efficient, you might want to choose ilFMultImg over one of ilConvImg’s subclasses if you’re using a large kernel for the convolution.

ilFDivImg divides *src1* by *src2* and, by default, checks for division by zero according to the following rules:

- If the numerator of the real or imaginary part is positive and the denominator is zero, the result is the largest possible floating point value (3.40282346e+38).
- If the numerator of the real or imaginary part is negative and the denominator is zero, the result is the smallest possible floating point value (-3.40282346e+38).
- If both the numerator and the denominator are zero, the result is zero.

You can call **setCheck()** and pass in a 0 to prevent ilFDivImg from checking for division by zero.

You can use ilFDivImg in image restoration. Given the Fourier transform of a degraded or noisy image and the Fourier transform of the noise function (or "noise image"), you can retrieve a clean image by dividing (in the frequency domain) the degraded image by the noise image. Once converted back to the spatial domain, you can then display the clean image.

Generation of Statistical Data

It's often desirable to collect statistical information about an image, such as how frequently various pixel values occur and what the minimum and maximum pixel values are. The ilImgStat class computes this kind of information for an entire image or for a specified region within an image. More specifically, for each channel of image data, it computes:

- a one-dimensional histogram showing frequency of pixel values
- the minimum and maximum pixel values
- the mean and standard deviation of the data, calculated from the histogram

The ilImgStat class inherits from ilLink, as shown in Figure 4-27.

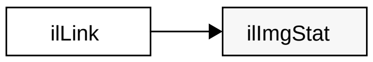


Figure 4-27 The ilImgStat Inheritance

The constructor for the ilImgStat class allows you to specify whether the data should be computed for the entire source image or for just a portion of it, as shown below. The portion is defined as a region of interest (ROI); see "Defining a Region of Interest" on page 185 for more information about the ilRoi class, which defines an ROI within an image.

```
ilImgStat(ilImage* src, ilRoi* Roi = NULL, int xoffset = 0,
          int yoffset = 0, int autoCalcEnable = TRUE);
```

The *xoffset* and *yoffset* parameters represent the offsets into the *src* image at which the ROI is placed; they're specified in the coordinate space of the input image, *src*.

You can also specify an ilRoi and its offsets for the ilImgStat with the **setRoi()** function, which accepts a pointer to an ilRoi and two integers. If no ROI is specified, ilImgStat performs its computations over the whole image.

You can use the *autoCalcEnable* parameter to enable or disable recalculation of statistics; if TRUE, the requested statistics are recalculated whenever the input image or ROI is changed or altered; if FALSE, input changes or alterations are ignored and statistics are never recalculated. Currently existing statistics are returned. If no values currently exist (for example, immediately after construction, after a reset, or if the input image has changed and the channel size of the new image is different from the previous one), then the requested values are computed based on the current input. You can set or query the autoCalc feature with the **setAutoCalc()** and **isAutoCalc()** functions. This feature is very useful if statistics from one part of an image are to be used to change other parts of an image.

Note: ilImgStat doesn't derive from ilImage, so its constructor doesn't create an ilImage. Thus, an ilImgStat object can't be passed as an image to another operator, but it might be one of an operator's input arguments. Multiprocessing on an ilImgStat object can turned on or off and queried using the **enableMP()** and **isMPenabled()** functions.

An Image’s Histogram

An image’s histogram, which is computed for each channel of image data, is defined by:

- The starting and ending pixel values. These establish the endpoints of the histogram’s range.
- The number of bins. The range is evenly divided into a specified number of bins.
- The size of each bin. The size is the range covered by each bin; this is computed by dividing the total range by the number of bins.

Once you’ve created an ilImgStat object, you can ask it to compute the histogram of the source image’s pixel values with the **getHist()** function:

```
int* getHist(int c=0, int nbins=0);
int* getHist(double start, double end, int c=0, int nbins=0);
```

As shown, this function is overloaded to allow you to specify the lower and upper endpoints of the range, as *start* and *end*. If you use the first constructor, the endpoints are the minimum and maximum values found. The other two arguments specify the channel (*c*) and the number of bins to use (*nbins*). If *nbins* has the default value of 0, the number of bins is equal to the total range indicated by *start* and *end* (in other words, the bin size is 1), up to a maximum of 4096 bins. (If *nbins*=4096, then the bin size is the range divided by 4096.) However, if the image’s data type is either ilChar or ilUChar, 256 bins are used, and if the data type is ilBit, 2 bins are used, regardless of what value is specified for *nbins*.

The **getHist()** function returns a pointer to an **int** array *nbins* long that’s allocated by ilImgStat. The values in the array correspond to the number of pixels that have values within each bin’s respective range. To normalize this data, copy the **int** array into a **float** array, and then divide each element of the array by the total number of pixels used to compute the histogram for that particular channel. You can obtain the number of pixels used with **getTotal()**:

```
int totalPixelCount = myImgStat.getTotal(1);
```

The argument for this function is an **int** that specifies the desired channel. (The number of pixels used for each of the channels might vary if you’ve specified different endpoints for the different channels.)

If the image's pixel ordering is ilSeparate, you can make multiple calls to **getHist()** for each channel and specify varying numbers of bins and starting points and endpoints. However, the histograms for all channels of ilInterleaved or ilSequential images are computed on the first call to **getHist()**, so the number of bins and the starting points and endpoints are fixed for subsequent calls. If you change limits on subsequent calls, status is set to ilUSEDOLDLIMITS, which is what **getStatus()** returns. If you need to change the histogram's attributes for subsequent calls, use **reset()**. This function deallocates the array created with **getHist()** and enables you to start over. (In general, you should call **reset()** or the ilImgStat destructor as soon as you're finished with a histogram to minimize memory usage.) If you'll need the histogram you've already computed, copy it into your own buffer before calling **reset()**.

After you've called **getHist()**, you can obtain the number of bins, the bin size, and the lower limit of the first bin for any particular channel:

```
int numBins = myImgStat.getNbins(1);
double binSize = myImgStat.getDBinSize(1);
double lowerLimit = myImgStat.getDStart(1);
```

The argument for these functions is an **int** that specifies the desired channel.

You can use **getStatus()** to check whether any errors occurred while the histogram was being computed. This function is inherited from ilLink. See "Error Codes" on page 407 for more information about the values returned by **getStatus()**.

Minimum, Maximum, Mean, and Standard Deviation

The ilImgStat class defines functions that return the minimum value, maximum value, mean, and standard deviation of a particular channel:

```
double getDMin(int c=0);
double getDMax(int c=0);
double getDMean(int c=0);
double getDStDev(int c=0);
```

These functions all return the desired number as a **double**, regardless of the data type of the image. Both **getDMean()** and **getDStDev()** perform their calculations using the most recently computed histogram, so their return values might be only approximations. (This is because a histogram represents a range of pixel values by a single value, the midpoint of the bin range. The calculations are exact for images with ilChar, ilUChar, or ilBit data types, or for ones with **int** values that use a bin size of one.) If either function is called before **getHist()**, the image's histogram is calculated first (using the minimum and maximum values as the endpoints), and then the desired statistical quantity is computed.

Other Functions

Two other support functions are provided:

```
void hwAccelerate(int enable);
void setZ(int z, int nz=1);
```

You can use the first function shown above to enable and disable hardware acceleration by passing in TRUE or FALSE, respectively. You can use the second function to limit processing in the *z* dimension of the image. The *z* argument specifies the starting *z* value, and *nz* indicates the size of the *z* tile. Thus, you can use these values to effectively create a 3-D ROI.

Radiometric Transformations

This section describes a set of operators that adjust all the pixels of an image so that together they have certain specified characteristics. Three of the operators described in this section—ilHistNormImg, ilHistEqImg, and ilHistScaleImg—modify an image's pixel values channel by channel, so that the image's histogram has certain desired properties. You can limit the area for which statistics are computed by specifying an ROI and its offsets when you create these operators; the operators then adjust all the pixels of the image so that the entire image's histogram matches that computed for the ROI. (See "Defining a Region of Interest" on page 185 for more information about ROIs.) If you've already created an image's histogram using ilImgStat as described in the previous section, you can pass a pointer to the existing ilImgStat object to speed the transformations performed by these operators.

The following radiometric operators are described in this section:

ilScaleImg	linearly scales the pixel data of an image so that it falls in a new specified range.
ilHistNormImg	transforms an image so that its histogram is normalized (Gaussian) and so that it has a specified mean and standard deviation.
ilHistEqImg	transforms an image so that its pixel values are uniformly distributed (so that the cumulative histogram is linear).
ilHistScaleImg	clamps values to a specified percentage distribution of the high- and low-intensity pixels and scales the remaining data between the clamp values.
ilThreshImg	sets each pixel to the image's minimum or maximum value, depending on whether the pixel is less than or greater than a specified threshold value.
ilLutImg	transforms a source image using a specified lookup table.
ilPiecewiseImg	transforms a source image using a lookup table created with a piecewise linear mapping function.

The operators that perform radiometric scaling, ilScaleImg and ilHistScaleImg, are accelerated on certain hardware platforms. The ilLutImg operator and the operators derived from it, such as ilPiecewiseImg, ilHistNormImg and ilHistEqImg, are also accelerated provided they meet the constraints specified in "Using Hardware Acceleration" on page 315. The ilThreshImg operator is also accelerated through the LUT mechanism, even though it's not derived from ilLutImg. All these classes derive directly or indirectly from ilMonadicImg, as shown in Figure 4-28.

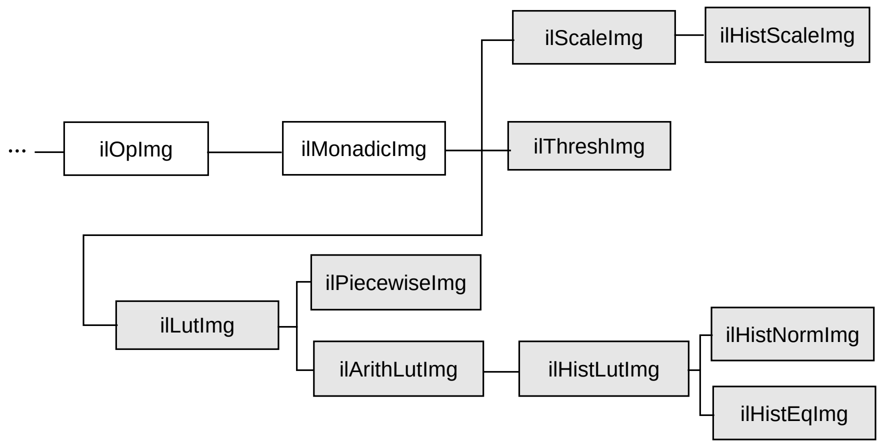


Figure 4-28 Radiometric Operator Inheritance Hierarchy

Scaling an Image

The ilScaleImg operator linearly scales the pixel data of an image so that it falls in a specified range. If you don't know the range of the input pixels, the first constructor shown below must be used. This constructor uses the minimum and maximum value fields of the input image to determine the input range, and it assumes an output range of 0 to 255. If you want to override the range of the input pixel data, you can use the second constructor and also specify an output range. The default is 0 to 255.

```
ilScaleImg(ilImage* img = NULL);
ilScaleImg(ilImage* img, double inMin, double inMax,
             double outMin=0, double outMax=255.999);
```

Pixels of value *inMin* are scaled to *outMin*, while those of value *inMax* are scaled to *outMax*. Pixels channel values lying between these extremes are scaled accordingly. Pixels outside the input domain are clamped between *outMin* and *outMax*.

The scaling function is normally computed based on *inMin* and *inMax* (the domain) and *outMin* and *outMax* (the range). To do this scaling, ilScaleImg computes the *slope* and *intercept* of a linear function of the form:

$$f(x) = (x \cdot slope) + intercept$$

Thus, an input pixel of value *x* becomes an output pixel of value *f(x)*. The *slope* and *intercept* are computed as follows:

$$slope = \frac{(outMax - outMin)}{(inMax - inMin)}$$

$$intercept = outMin - (slope \cdot inMin)$$

You can alter the operator's parameters with these member functions:

```
void setRange(double outMin, double outMax);
void setDomain(double inMin, double inMax);
```

You can control the scaling behavior with these functions:

```
void resetDomain();
void resetRange();
void resetScaling();
void setScaling(double slope, double intercept);
```

resetDomain() invalidates the current input levels and, if none are specified via **setDomain()**, the minimum and maximum values of the input images are used for the domain.

resetRange() invalidates the current output levels and, if none are specified via **setRange()**, default values are computed using the input domain and the scaling values (*slope* and *intercept*). An example image produced using ilScaleImg is shown in Figure 4-29.

resetScaling() forces the operator to forget any values explicitly set for *slope* and *intercept* and to compute them as shown above.

setScaling() allows you to explicitly set the values of the *slope* and *intercept* of the scaling function.

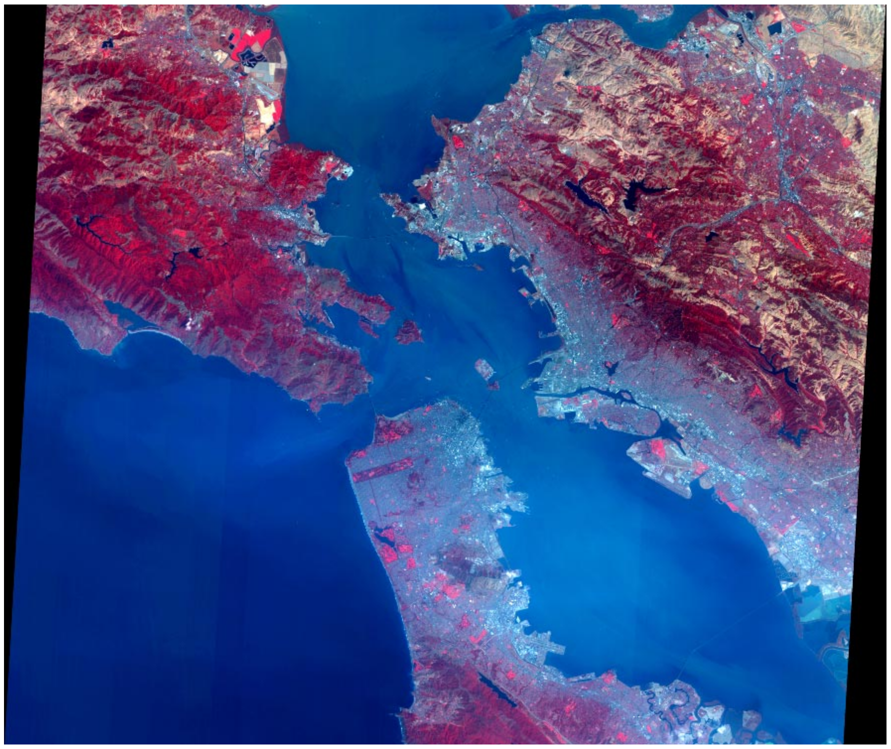


Figure 4-29 Using Scaling

Histogram Operators

Both ilHistNormImg and ilHistEqImg derive from ilHistLutImg, which itself derives from ilArithLutImg. This inheritance allows the histogram operators to use lookup tables to determine resulting values, rather than perform the computations on a per-pixel basis. As a result, the histogram operators are more efficient.

The constructors for ilHistNormImg are:

```
ilHistNormImg(ilImage* src, ilPixel& mean, ilPixel& stdev,
        ilImgStat* imgstat = NULL, ilRoi* Roi = NULL,
        int xoffset = 0, int yoffset = 0);
```

```
ilHistNormImg(ilImage* img = NULL, ilImgStat = NULL,
        ilRoi* Roi = NULL, int xoffset = 0, int yoffset = 0);
```

The first constructor below allows you to specify the source image and the desired mean and standard deviation. The second constructor takes a source image and computes default values for the mean and standard deviation. The mean for each channel is computed as the average of the minimum and maximum values of the source image for that channel. The standard deviation is set to 1.0 for each channel.

The ilPixels can use any data type, but their number of channels must match that of the source image. If you've already created an ilImgStat object (for the source or even a different image), you can pass a pointer to it. This makes ilHistNormImg more efficient. If you supply both an ilImgStat and an ilRoi, the histogram computed for the ilImgStat is used and the ilRoi is ignored.

You can dynamically change the mean, the standard deviation, the ilImgStat object, and the ilRoi and its offsets with the following functions:

```
void setMean(ilPixel& mean);
void setStdev(ilPixel& stdev);
void setImgStat(ilImgStat* imgstat);
void setRoi(ilRoi* Roi, int xoffset = 0, int yoffset = 0);
```

The **setImgStat()** and **setRoi()** functions are inherited from ilHistLutImg.

Histogram equalization and histogram scaling of an image are often performed to enhance the contrast of an image. Histogram equalization results in an image with pixel values that are more evenly distributed.

The constructor for ilHistEqImg is shown below:

```
ilHistEqImg(ilImage* src = NULL, ilImgStat* imgstat = NULL,
        ilRoi* Roi = NULL, int xoffset = 0, int yoffset = 0);
```

As shown, you specify the source image, the ilImgStat object if one exists, and an optional ROI along with its offsets. This class also inherits **setImgStat()** and **setRoi()** functions as does ilHistNormImg.

The constructor for ilHistScaleImg is more complicated:

```
ilHistScaleImg(ilImage* src = NULL,
        double lowClip=0, double highClip=0,
        double outMin=0, double outMax=255,
        ilImgStat* imgstat = NULL, ilRoi* Roi = NULL,
        int xoffset = 0, int yoffset = 0);
```

The *src* argument specifies the source image. The next four arguments specify how the source image should be transformed. The *highClip* and *lowClip* arguments indicate what percentage of the high and low intensity pixels should be clamped to the values specified by *outMax* and *outMin*, respectively. Imagine that the pixels are sorted in order of increasing intensity, as in a histogram. Then, *highClip* percent of the highest-intensity pixels are set to the *outMax* value, and *lowClip* percent of the lowest-intensity pixels are set to the *outMin* value. After the desired pixels have been clipped, the remaining pixels are scaled linearly between the clamp values. The optional ilImgStat and ilRoi objects (and offsets) each have the same meaning as with ilHistNormImg.

You can dynamically change all these arguments with the following functions:

```
void setImgStat(ilImgStat* imgstat);
void setRoi(ilRoi* Roi, int xoffset = 0, int yoffset = 0);
void setClip(double lowClip, double highClip);
void setRange(double outMin, double outMax);
```

One other useful function, **setHistLimits()**, allows you to change the limits between which the histogram is to be computed:

```
void setHistLimits(double low, double high);
```

The two arguments, *low* and *high*, define the histogram's limits.

Be careful when changing the input to any of the histogram operators by using **setInput()**. (See "Dynamically Reconfiguring a Chain" on page 75 for more information.) If an ilImgStat has already been specified in a histogram operator constructor and then **setInput()** is called, the old ilImgStat is used unless you call **setImgStat()** with a new one. You can use NULL in **setImgStat()** to force a new one to be computed.

The Threshold Operator

The ilThreshImg operator sets each pixel (on a channel by channel basis) to the image's minimum or maximum allowable value, depending on whether the pixel is less than or greater than a specified threshold value. (See "Minimum and Maximum Pixel Values" on page 44 for more information about how to set an image's minimum and maximum pixel values.)

To create an ilThreshImg operator, you can use one of the two constructors shown below. In the first constructor, the threshold is specified as an ilPixel, and a different threshold level can be applied to each channel of the source image. In the second constructor, the same threshold, *val*, is applied to all channels:

```
ilThreshImg(ilImage* src, const ilPixel& thresh);
ilThreshImg(ilImage* src = NULL, float val = 0);
```

Each channel or each pixel of the source image is compared to the threshold value, *thresh* or *val*. If the channel value is less than the threshold value, it is set to the image's minimum channel value. If the channel value is greater than or equal to the threshold value, it is set to the maximum channel value. (If *thresh* is a single-channel pixel, its value is used for all channels of the source image.)

You can query an image about its threshold value and dynamically change this value with these functions:

```
void getThresh(ilPixel& thresh);
void setThresh(float val);
void setThresh(const ilPixel& thresh);
```

getThresh() returns the threshold value by reference, and **setThresh()** sets the threshold value.

ilLutImg

The ilLutImg class transforms a source image using a specified lookup table (LUT). As mentioned previously, ilArithLutImg (see "Single-input Operators" on page 119) and ilHistLutImg (see "Histogram Operators" on page 170) derive from ilLutImg. Normally, the LUT and the image have the same number of channels. However, two other possibilities are allowed: if the LUT has only one channel, it is applied to each channel of the image; if the source image has only one channel while the LUT has *n* channels, each LUT channel is applied to the source image in turn, producing an ilLutImg with *n* channels. (For any other combination, the ilStatus value ilLUTSIZEMISMATCH is returned by any data access operations.)

The first constructor below allows you to specify the source image and the LUT. The second one lets you specify the source image and sets the LUT to NULL. You can later specify a LUT using the **setLookUpTable()** function.

```
ilLutImg(ilImage* source, const ilLut& lut);
ilLutImg(ilImage* source = NULL);
```

See “ilLut” on page 399 for more information about the ilLut class and also for an explanation of how lookup tables can be stored and retrieved using SGI image files.

You can dynamically change or retrieve the LUT with these functions:

```
ilStatus setLookUpTable(const ilLut& lut);
ilStatus getLookUpTable(ilLut& table);
```

If you change the LUT, the output number of channels and data type are updated if necessary to accommodate the new LUT. Use the following functions to set or query whether the input data is signed and to obtain its data type:

```
void setSign(int sgn);
int getSign();
ilType getInputType();
```

ilPiecewiseImg

The ilPiecewiseImg class, derived from ilLutImg, simplifies the task of constructing a lookup table when only a piecewise linear mapping is needed from the input pixels to the output data. The constructor accepts the source image, a list of breakpoints, and the length of that list:

```
ilPiecewiseImg(ilImage* inputImage = NULL,
const ilXYSfloat* bkpts=NULL, int length=0);
```

A *breakpoint* is a point on a piecewise continuous function where two continuous segments meet, as shown in Figure 4-30. The endpoints, 0 and 255, are made breakpoints by default (this does not affect the length of the breakpoints list). If a breakpoint is entered outside the range, it is clamped to the appropriate endpoint.

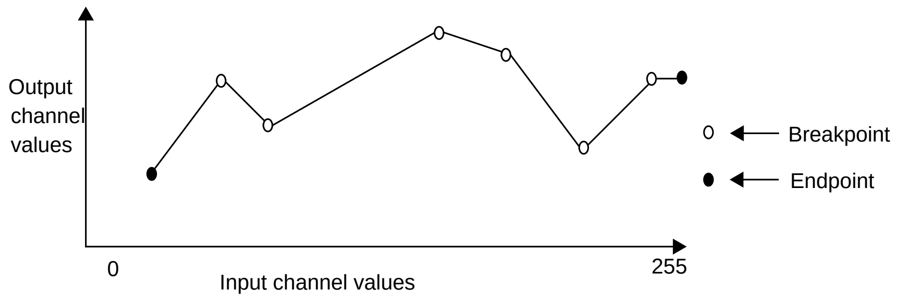


Figure 4-30 Breakpoints along a Piecewise Continuous Function

Several functions are provided to manipulate the breakpoints:

```
ilStatus setBreakpoints(const ilXYSfloat* bkpts=NULL,
          int length=0, int chan=-1);
ilStatus insertPoint(const ilXYSfloat& point, int index,
             int chan=-1);
ilStatus replacePoint(const ilXYSfloat& point, int index,
             int chan=-1);
ilStatus removePoint(int index, int chan=-1);
```

setBreakpoints()	allows you to specify a new list of breakpoints (of length *length*). You can specify a list for a specific channel with the *chan* argument; if this is minus 1 (the default), the list is used for all channels in the image.
insertPoint()	inserts a breakpoint *point* after the one at *index* in the list for channel *chan*.
replacePoint()	replaces the breakpoint at *index* in the breakpoint list for channel *chan* with *point*.
removePoint()	removes the breakpoint at *index*; you specify which channel's breakpoint list with *chan*.

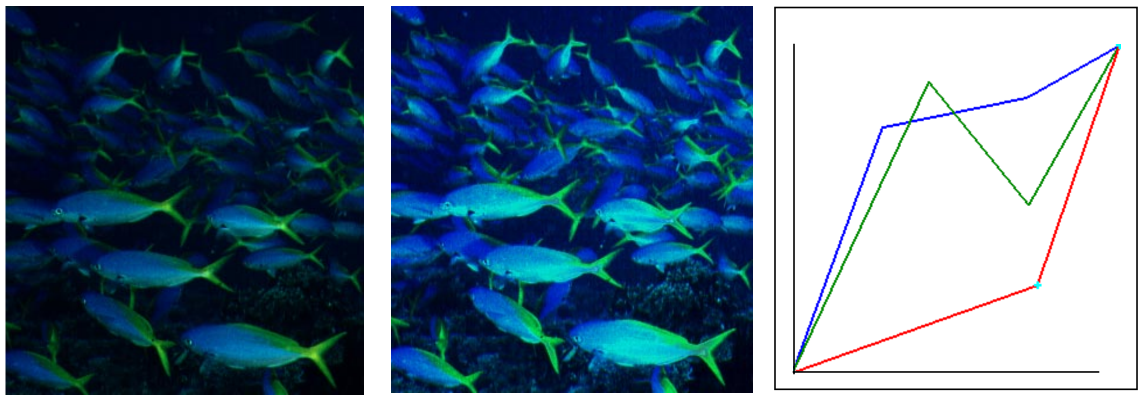
You can query an ilPiecewiseImg about its breakpoints with these functions:

```
int getBreakpoints(ilXYSfloat* bkpts, int chan=0);
int getNumBreakpoints(int chan=0);
void getPoint(ilXYSfloat& point, int index, int chan=0);
float findPoint(ilXYSfloat& loc, int& index, int forInsert=0,
              int chan=0);
```

getBreakpoints()	accepts a pointer to a list of breakpoints and returns the length of the breakpoint list for *chan* as an **int** and the breakpoint list itself through *bkpts* (you must allocate enough space in *bkpts* before this function call).
getNumBreakpoints()	returns the number of breakpoints in the breakpoint list for *chan*.
getPoint()	returns the breakpoint at *index* in the breakpoint list for *chan* by reference in *point*.
findPoint()	accepts a location (*loc*), an *index* into the breakpoints list for *chan*, and a flag specifying whether the closest breakpoint should be found (*forInsert* = 0) or whether the closest edge should be found (*forInsert* = 1). In either case, the distance between the given location and the found location is returned as a **float**, the breakpoint is returned by reference in *loc*, and the index of that breakpoint is returned in *index*.

In all of the above functions, *chan* is 0 by default, specifying the first channel of the image.

Figure 4-31 shows an example of an application with a graphical user interface (*imgview*) that can be written with ilPiecewiseImg.



Original Edited LUT Editor RGB Interface

Figure 4-31 Using a Lookup Table Editor to Set Breakpoints

Combining Images

The three operators described in this section—ilBlendImg, ilMergeImg, and ilCombineImg—combine two or more images into one using different methods:

- ilBlendImg blends two images using a specified alpha value or alpha images that indicate how to weight the images relative to each other.
- ilMergeImg merges a series of images into a single multiple-channel image.
- ilCombineImg combines two images using a mask to define which portions of the two images to use in the final combined image.

These three classes have very different pedigrees, as shown in Figure 4-32.

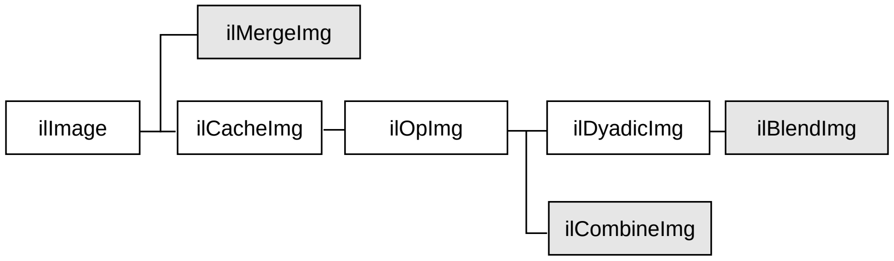


Figure 4-32 ilBlendImg, ilMergeImg, and ilCombineImg Inheritance Hierarchy

Note that ilMergeImg doesn't inherit from ilOpImg, so it's not, strictly speaking, a true operator. An ilMergeImg doesn't actually hold any image data, but all the data access functions are redefined so that the data can be treated as a single ilImage.

ilBlendImg

The constructors for ilBlendImg allow you to specify a constant alpha value or to specify third and fourth images that contain alpha values for each pixel of the foreground and background images. You can also select the way in which the foreground and background images are blended:

```
ilBlendImg(ilImage* fore, ilImage* bkgd, float alpha);
ilBlendImg(ilImage* fore = NULL, ilImage* bkgd = NULL,
          ilImage* alphaf = NULL, ilImage* alphab=NULL,
          ilCompose comp=ilAplusB);
```

The first constructor specifies one constant *alpha* value (which should fall between 0.0 and 1.0) that is used to calculate a foreground and background alpha. If the second constructor is used, the alpha values are taken from the first channel of *alphaf* (for the foreground alphas) and *alphab* (for the background alphas). The other channels, if any, are ignored. In the default mode (ilAplusB), if *alphab* is NULL, then the background alpha values for each pixel are computed from *alphaf* as 1 - *alphaf*. Figure 4-33 shows an example image produced using the ilBlendImg operator and the ilAplusB compose mode.

The second constructor also allows you to specify the composition mode. See Figure 4-34 for an explanation of these modes. The default is ilAplusB. The composition modes are defined in the header file *il/ilTypes.h*.

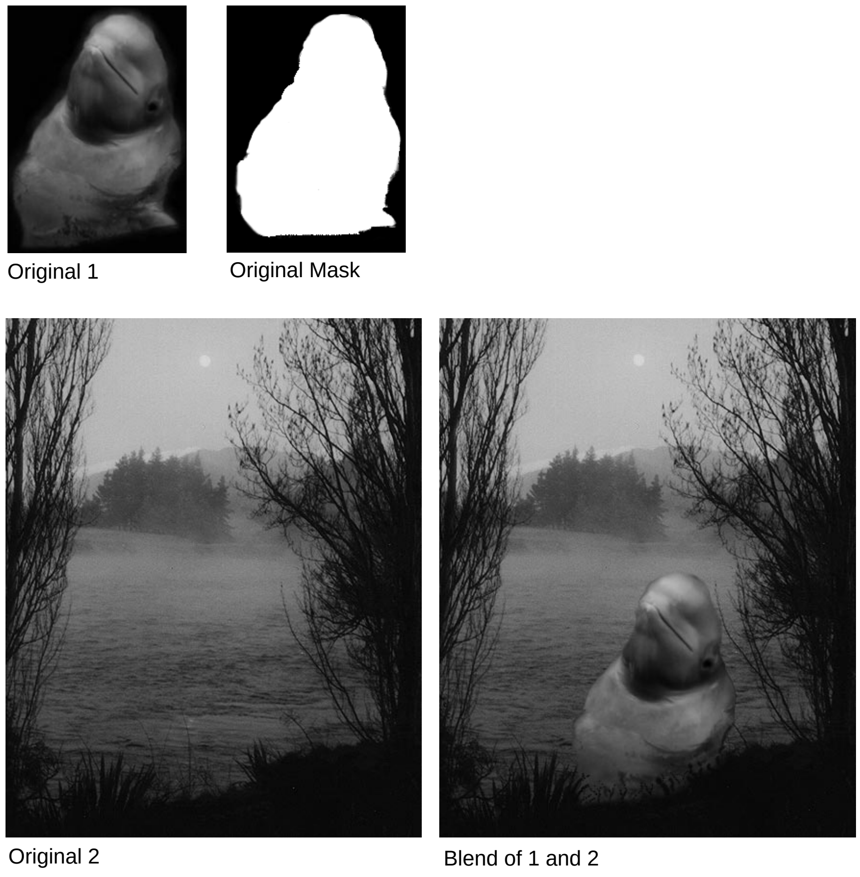


Figure 4-33 Blended Images

The foreground, background, and alpha images must all be the same size. The alpha values defined by *alphaf* and *alphab* are normalized to the range (0-1), based on the minimum and maximum allowable pixel values of *alphaf* and *alphab*. The foreground and background alphas are calculated as follows:

$fore\alpha = alpha$, $back\alpha = 1 - alpha$ or

$fore\alpha = alphaf$, $back\alpha = alphab$ (if *alphab* is not NULL), and

$$back\alpha = 1 - alphaf \text{ (if } alphab \text{ is NULL)}$$

The blending function, which is used for each pixel, is:

$$F_A \cdot \text{foreground} \cdot fore\alpha + F_B \cdot \text{background} \cdot back\alpha$$

The composition mode determines F_A and F_B. For the default composition mode (ilAplusB), they are both equal to 1.0. See Figure 4-34

If ilImgA is the foreground image and ilImgB is the background image, then

$$\alpha_A = alphaf \text{ and } \alpha_B = alphab \quad .$$

However, when alphaB=NULL, then

$$\alpha_B = alphaf$$

You may set the composition method with **setBlendMode()**. It takes one argument of type ilCompose:

```
void setBlendMode(ilCompose mode = ilAplusB);
```

You can explicitly set the minimum and maximum allowable pixel values of the alpha images *alphaf* and *alphab* using these functions:

```
void setAlphaRange(float fmin, float fmax);
void setAlphaRange(float fmin, float fmax,
             float bmin, float bmax);
```

The first function sets the normalizing values for the foreground alpha; the second sets the minimum and maximum values of the alpha for the foreground and background images.

To query an ilBlendImg about its normalizing values, use:

```
void getAlphaRange(float& fmin, float& fmax);
void getAlphaRange(float& fmin, float& fmax,
             float& bmin, float& bmax);
```

The first function returns the normalizing values for the foreground alpha, and the second function returns the normalizing values for both the foreground and background alphas. You can also dynamically change the alpha images or the constant alpha value:

```
ilStatus setAlphaPlane(ilImage* alphaImg);
ilStatus setAlphaPlane(ilImage* alphaf, ilImage* alphab)
ilStatus setConstAlpha(float val);
```

The first function shown above sets the foreground alpha image, while the second function sets both the foreground and background alpha images. The third function sets the constant alpha value. You can also use **setOffset()** (inherited from ilDyadicImg) to offset the foreground image with respect to the background image.

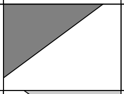
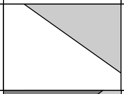
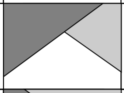
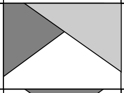
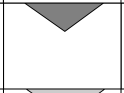
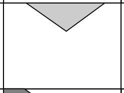
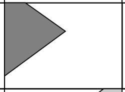
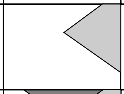
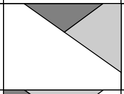
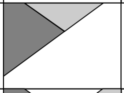
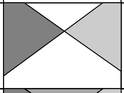
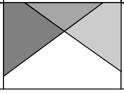
Mode	Diagram	F_A	F_B
ilImgA		1	0
ilImgB		0	1
ilAoverB		1	$1-\alpha_A$
ilBoverA		$1-\alpha_B$	1
ilAinB		α_B	0
ilBinA		0	α_A
ilAoutB		$1-\alpha_B$	0
ilBoutA		0	$1-\alpha_A$
ilAatopB		α_B	$1-\alpha_A$
ilBatopA		$1-\alpha_B$	α_A
ilAxorB		$1-\alpha_B$	$1-\alpha_A$
ilAplusB		1	1

Figure 4-34 Composition Modes for ilBlendImg

ilMergeImg

An ilMergeImg consists of a single ilImage formed by merging a number of images. The number of channels of the merged image equals the sum of the number of channels in all the individual input images. All the input images should be the same size, but you can assign a different data type or order to the final ilMergeImg as it's created:

```
ilMergeImg(ilImage** imgPtr, int nimg,
    ilOrder order=ilInterleaved,
    ilType dtype=numilTypes);
ilMergeImg(int nimg, ilImage** imgPtr);
```

In both of these constructors, *imgPtr* is an array of pointers to the ordered input ilImages. The first *nimg* ilImages in the array are merged, and the rest are ignored. (*imgPtr* should have at least *nimg* pointers.) The first constructor lets you specify an order and data type for the merged image. If the default data type numilTypes is used, the data type of the merged image is the largest data type of the ilImages. If the second constructor is used, the order and data type of the merged image are the same as those of the first ilImage pointed to in the *imgPtr* array.

ilCombineImg

An ilCombineImg takes two ilImages of the same size and uses an ROI (and its offsets) to determine which pixels to use in the final image (pixels that are inside the ROI are taken from the foreground image, and pixels that are outside the ROI are taken from the background image):

```
ilCombineImg(ilImage* bkgd = NULL, ilImage* fore = NULL,
    ilRoi* roi = NULL,int xoffset = 0, int yoffset = 0);
```

See “Defining a Region of Interest” on page 185 for more information about creating an ilRoi object. The *xoffset* and *yoffset* parameters specify the offsets at which the ROI is placed in the foreground and background images; they're specified in the coordinate space of the *fore* image. You can change the ROI and its offsets after the combined image is created, and you can obtain a pointer to it with these functions:

```
void setRoi(ilRoi* roi, int xoffset = 0, int yoffset = 0);
    ilRoi* getRoi();
```

Constant-valued Images

The ilConstImg class allows you to create an object that returns a constant value whenever its data is read. You might use this class as an input to one of the operators described in the “Dual-input Operators” on page 121—for example, to multiply each pixel in an image by a constant value. Remember that ilConstImg isn’t an operator since it derives directly from ilImage.

The ilConstImg class defines only one function, its constructor:

```
ilConstImg(const ilPixel& fillPix);
```

The specified ilPixel is the value returned whenever the image’s data is read, regardless of how much data is read. Since an ilConstImg stores only one ilPixel, it uses much less memory than, for example, an ilMemoryImg filled with pixels. To change an ilConstImg’s pixel value after you’ve created an ilConstImg object, use the **setFill()** function defined in ilImage and described in “Fill Value” on page 43.

Using a Null Operator

As its name suggests, the ilNopImg operator performs no operation at all. It is useful for caching the results defined by a non-cached class, such as ilMemoryImg (described in “Importing and Exporting Image Data” on page 101) or ilSubImg (described in “Defining a Region of Interest” on page 185). It’s also useful if you just want to change some of the attributes of any image (for example, data type, data ordering, or page size) and need to cache the result. Note that this class is a real operator, as it derives from ilMonadicImg.

The ilNopImg class defines one public member function, its constructor:

```
ilNopImg(ilImage* inputImage = NULL);
```

An image stored as an ilMemoryImg cannot take advantage of the IL’s on-demand paging mechanism, since it does not derive from ilCacheImg; however, ilNopImg is derived (indirectly) from ilCacheImg. Thus, storing that ilMemoryImg as an ilNopImg allows you to page that image.

Defining a Region of Interest

Some IL programs, especially those that deal with large images, may need to apply an operator to only a portion of an entire image. When this is the case, you can restrict the processing area to a region of interest (ROI). An ROI allows you to modify irregular regions of an image. The IL provides two principal classes that let you restrict the data that can be accessed:

- ilRoiImg, which associates an ROI with an image so that subsequent operations on the image affect only the data inside the ROI
- ilSubImg, which allows a rectangular portion of an image to be treated as if it were an independent image

In some situations, these two classes might appear to have similar effects, but they actually achieve their results through very different means, and they have different uses. An ilRoiImg is the same size as the initial image; the difference is that portions of the ilRoiImg are "masked out"—set to a specified background value—so that they won't be affected by processing. You use an ilRoiImg when you wish to modify a portion of an image while leaving the rest of the image intact. This is the traditional masking, or ROI, concept.

An ilSubImg doesn't actually hold any data itself; it merely implements the standard data access functions—**getSubTile3D()**, **setSubTile3D()**, and **copyTileCfg()**—so that they access only the data in the subimage. When you call one of the access functions, you specify the origin and size of the desired tile in the subimage. The ilSubImg maps the coordinates of the desired tile to the source image so that the correct data is accessed. An ilSubImg can be used as a rectangular ROI, but it's most useful for manipulating the input images to an operator to achieve particular results. For example, you can use an ilSubImg to offset two images relative to each other before they're fed into an ilAddImg operator to be added together. (You can also do this with the **setOffset()** function in ilDyadicImg.) Or you can select the red and blue channels of an image using two ilSubImgs and then add them together.

Once you've created either an ilSubImg or an ilRoiImg, you can use it in an operator chain just as you would any other ilImage. You can also write data back into ilSubImg or ilRoiImg, which you cannot do with an operator (since all operators are read-only). When you do write data back into an ilSubImg or an ilRoiImg, the input image is modified appropriately. The next sections describe how to use these two classes.

Creating an ilRoiImg

Typically, you'll use an ilRoiImg when you're displaying processed data or writing it to a file. By restricting the area that needs to be processed, you can prevent data from being processed unnecessarily.

Before you can create an ilRoiImg, you need to create the following:

- the source ilImage that's to be masked with the ROI
- the actual ROI itself, in the form of an ilRoi object
- the *x* and *y* offsets for placing the ROI into the source image
- the background value, an ilPixel, that's used to fill areas outside the ROI

The source image can be any ilImage, and it can be part of an already existing operator chain. The background value defines the ilImage's values outside the ROI. As shown below, the constructor for the ilRoiImg class takes pointers to all three of these objects:

```
ilRoiImg(ilImage* src, ilRoi* roi, ilPixel& bkgd,
          int xoffset = 0, int yoffset = 0);
```

This constructor associates the ilRoi with the source ilImage and sets the ilRoiImg's background value. The *xoffset* and *yoffset* values determine where the ROI is placed; they're specified in the *src* image's coordinate space. Subsequent operations to the ilRoiImg affect only the image data inside the specified ilRoi. Any attribute of an ilRoiImg that's not explicitly set is inherited from its source image.

Once an ilRoiImg is created, you can modify its associated ilRoi or the background value by calling **setRoi()** or **setBkgd()**. These functions take a pointer to the desired ilRoi or ilPixel:

```
void setRoi(ilRoi* roi, int xoffset = 0, int yoffset = 0);
void setBkgd(ilPixel& bkgd);
```

You can also query an ilRoiImg about its ROI or background value:

```
ilRoi* getRoi();
void getBkgd(ilPixel& bkgd);
```

The ilRoi base class defines the basic concept of a region of interest in the IL; it's an abstract class, so you must use one of the classes that derive from it to create an ROI. (You can also derive your own class to define an ROI that more specifically matches your needs. See "Deriving from ilRoi" on page 300 to learn more about deriving from ilRoi.) An ilRoi is a two-dimensional object with its own *x* and *y* dimensions and its own coordinate space. If you imagine the ilRoi placed on top of the image and yourself viewing it from above, you would see regions of the image inside and outside the ilRoi. The regions inside are considered valid and are accessible for image processing operations; those outside the ilRoi are invalid and are typically set to a background value for processing. The same ilRoi can be associated with different images (which can be different sizes), and it can be placed at different offsets within each image. This functionality is achieved through the ilRoiMap class, which is described later. You manage the ilRoi's coordinate space with **setCoordSpace()** and **getCoordSpace()**.

Currently, the IL provides two classes derived from ilRoi, as shown in Figure 4-35.

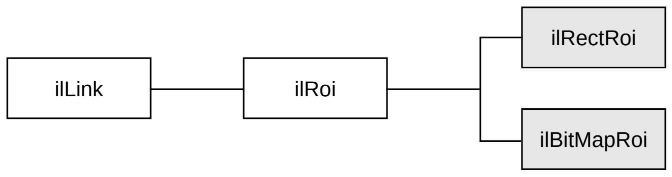


Figure 4-35 ilRoi's Subclasses

An ilRectRoi defines a rectangular ROI, and an ilBitMapRoi defines a bitmap of any shape that can be used as an ROI.

A Rectangular ROI

As its name suggests, ilRectRoi allows you to define a rectangular ROI:

```
ilRectRoi myRoi(20, 30, 1);
```

All the arguments for the ilRectRoi constructor are of type **int**. The first two specify the sizes in the *x* and *y* dimensions (*20* and *30*) of the rectangle to be used as the ROI. The optional last argument, which can be either 1 or 0, indicates whether the area inside or outside the rectangle should be considered the valid area. The default value is 1, which defines the inside of the rectangle as the valid area. You specify the image that the ilRectRoi is associated with and the offsets into the image later so that the same ilRectRoi can be used for different images at different offsets. In addition, operators that take an ROI as an input also take the offsets as arguments.

The ilRectRoi class defines functions that allow you to change the *x* and *y* dimensions of the rectangle and to retrieve this information:

```
ilStatus setRSize(int nxr, int nyr);
void getRSize(int& nxr, int& nyr);
```

You can also determine which is the valid area (the inside or the outside of the rectangle) and change the current designation:

```
int getValidValue();
ilStatus setValidValue(int val);
```

The first function returns either a 1 or a 0 to indicate that the inside or the outside is valid, and the second function sets the valid area.

A Bitmap ROI

Since it allows you to define an ROI of any arbitrary shape (it might even have disjoint regions), ilBitMapRoi is more versatile than ilRectRoi. As its name suggests, ilBitMapRoi uses a bitmap to define an ROI. A bitmap ROI is a data array in which each bit corresponds to a pixel in the image that's to be masked with the ROI. The value of the bit specifies whether a pixel is considered valid or invalid. Look in:

- */usr/people/4Dgifts/examples/ImageVision/ilguide /bitmapRoiEx.c++*
- */usr/people/4Dgifts/examples/ImageVision/iltutorial/ex3.c++*

for sample programs that define and use an ilBitMapRoi.

Like an ilRectRoi, an ilBitMapRoi is a two-dimensional object with *x* and *y* dimensions and a coordinate space that don't have to match those of the image with which it's associated. Also, you can specify the images and offsets to be associated with an ilBitMapRoi after you've created it, as described earlier for ilRectRoi.

You can create an ilBitMapRoi using either an ilImage or an existing data array:

```
ilBitMapRoi(ilImage* bitmapimg, int valid=1);
ilBitMapRoi(int xsize, ysize, void* data=NULL,
      ilCoordSpace spc=ilCoordSpace(0), int valid=1);
ilBitMapRoi();
```

The first constructor takes a pointer to an ilImage as the definition of the bitmap ROI; *bitmapimg*'s data should consist of a series of 1s and 0s. (Actually, any value other than 0 is treated as a 1.) The coordinate space of the ilBitMapRoi is taken from that of *bitmapimg*.

With either constructor, you can choose to use either a 1 or a 0 to indicate valid data in the bitmap. By default, a value of 1 denotes valid data and a 0 denotes invalid data. You can also reverse the meaning of the bit values after an ilBitMapRoi is created by using **setValidValue()**. This function sets the valid value to the **int** argument passed in. If you need to check what the valid value is, call **getValidValue()**, which returns the value used to indicate valid data.

In the second constructor shown above, *xsize* and *ysize* are the dimensions of the bitmap. The next argument, *data*, is a pointer to the bitmap, which should consist of

$$\frac{xsize \cdot ysize + 7}{8}$$

bytes of data. If *data* is NULL, a bitmap of this size is allocated. The *spc* argument is the bitmap's coordinate space; by default, it's ilLowerLeftOrigin.

The third constructor creates an instance of an ilBitMapRoi object, initializing its valid value to 1 and *bitmapimg* and *data* to NULL. You must later use either **setImg()** or **setData()** described below to actually associate an ROI with this instance of ilBitMapRoi.

Several functions are provided to set and obtain the bitmap data and the bitmap image:

```
void setData(void* data, int nx, int ny, ilCoordSpace spc);
void* getData();
void setImg(ilImage* img);
```

Also, you can use **getRSize()** as described earlier for ilRectRoi to obtain the bitmap's dimensions.

You can write to the bitmap using ilBitMapRoi's **setTile()** function:

```
ilStatus setTile(int x, int y, int xsize, int ysize,
          void* data, const ilConfig* config=NULL);
```

This function writes the tile of data pointed to by *data* to the ilBitMapRoi, starting at the location indicated by *x* and *y*. The tile's size is specified by *xsize* and *ysize*. The *config* argument describes the configuration of *data*; if it's NULL, the tile is assumed to have the same configuration as the bitmap.

You can also retrieve a tile of bitmap data into a buffer you allocate using **getTile()**:

```
ilStatus getTile(int x, int y, int xsize, int ysize,
      void* data, const ilConfig* config=NULL);
```

This function retrieves a tile of data from the ilBitMapRoi and puts it in the buffer pointed to by *data*. The tile's size is specified by *xsize* and *ysize*, and its origin in the ilBitMapRoi is indicated by *x* and *y*. The optional *config* argument allows you to reconfigure the data before it's written to the buffer.

Creating an ilSubImg

The ilSubImg class defines three constructors that let you create a subimage that's a different size from the source image. The first constructor is for two-dimensional images, the second for three-dimensional images, and the third for four-dimensional images.

```
ilSubImg(ilImage* src, int xs, int ys, int xsz, int ysz,
         ilConfig* config = NULL);
ilSubImg(ilImage* src, int xs, int ys, int zs,
         int xsz, int ysz, int zsz, ilConfig* config = NULL);
ilSubImg(ilImage* src, int xs, int ys, int zs, int cs,
         int xsz, int ysz, int zsz, int csz,
         ilConfig* config = NULL);
```

The first argument in all of these functions is a pointer to the source image. The next arguments specify the location of the origin of the subimage (*xs*, *ys*, zs, and *cs*), measured in pixels in the source image, and the dimensions of the subimage (*xsz*, *ysz*, *zsz*, and *csz*), as shown in Figure 4-36. (This figure assumes that the subimage's coordinate space is ilLowerLeftOrigin.) If the dimensions are larger than the source image, the subimage is padded with the source image's fill value.

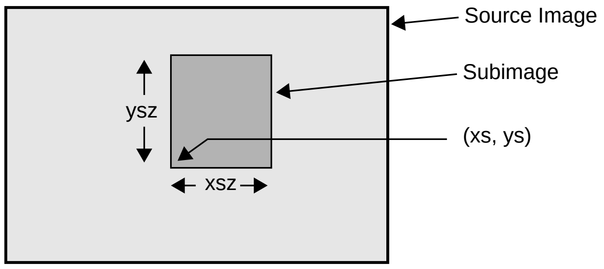


Figure 4-36 Source Image and Subimage

The last, optional argument for these constructors is a pointer to an ilConfig object that specifies the configuration of the subimage. If this argument isn't supplied, the subimage inherits its configuration from the source image.

A fourth constructor is provided for convenience when the subimage has the same size as the source image but a different configuration:

```
ilSubImg(ilImage* src, ilConfig* config);
```

You can use the ilConfig argument for any of these constructors to select a subset of the source image's channels and to reorder them; you can also use it to set the coordinate space, data type, and pixel ordering of the subimage.

Once you've created an ilSubImg, you can modify several of its attributes—size, data type, order, color model, and coordinate space—using the functions defined in ilImage. To change an ilSubImg's configuration after you've created it, use **setConfig()**. This function takes a pointer to an ilConfig and modifies the ilSubImg accordingly. Any attribute of an ilSubImg that's not explicitly set is inherited from its source image.

You can also translate the origin of a subimage after it's been created:

```
const int xorigin = 20;
const int yorigin = 20;

mySubImg.setStart(xorigin, yorigin);
```

As shown, **setStart()** expects const **int** arguments. For a three-dimensional image, supply a third argument for the *z* dimension. For a four-dimensional image, supply a fourth *c* dimension. The ilSubImg's origin, not its size, is affected by **setStart()**, as shown in Figure 4-37.

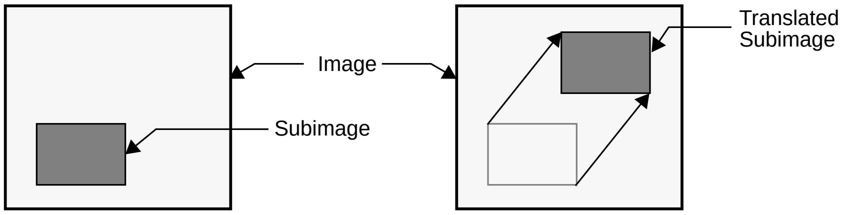


Figure 4-37 Translated Subimage

You can also query a subimage about its origin:

```
int xorigin, yorigin, zorigin, corigin;
mySubImg.getStart(xorigin, yorigin, zorigin);
```

or

```
mySubImg.getStart(xorigin, yorigin, zorigin, corigin);
```

As shown, the overloaded **getStart()** retrieves the origin by reference.

The virtual method **hasPages()**, inherited from ilImage, indicates whether a class implements paging and is defined by ilSubImg. It returns TRUE if its parent implements paging and FALSE otherwise.

Chapter 5

Displaying an Image

This chapter explains how to use the IL's display facility to display and manage a set of images on the screen.

Chapter 5

Displaying an Image

This chapter describes how to display and manage a set of images on the screen using the IL's display facility. As part of this facility, numerous functions are provided to help you develop an interactive image processing program. You can use these functions to move images, perform wipes, roam around an image, and create split views of multiple images.

The chapter describes the IL's display facility in the following major sections:

- "Overview of the Display Facility" on page 196 describes the sequence of operations you must perform to display an image.
- "A Simple Interactive Display Program" on page 201 lists and describes a program that opens an image file, performs an operation on it, and allows interactive viewing of both images.
- "Creating an ilDisplay" on page 208 describes in detail how to open a window and create an ilDisplay.
- "View and Display Basics" on page 211 describes basic concepts such as setting background color and page borders and deferring drawing of a view.
- "Managing Views" on page 216 describes how to manage the view stack and how to retrieve information from views.
- "Applying a Display Operator" on page 223 tells you how to use display operators to draw views, relocate or resize them, and update them.
- "A More Complicated Interactive Display Program" on page 235 contains a program illustrating control of a display.

Overview of the Display Facility

The IL display classes described in this chapter are shown shaded in Figure 5-1.

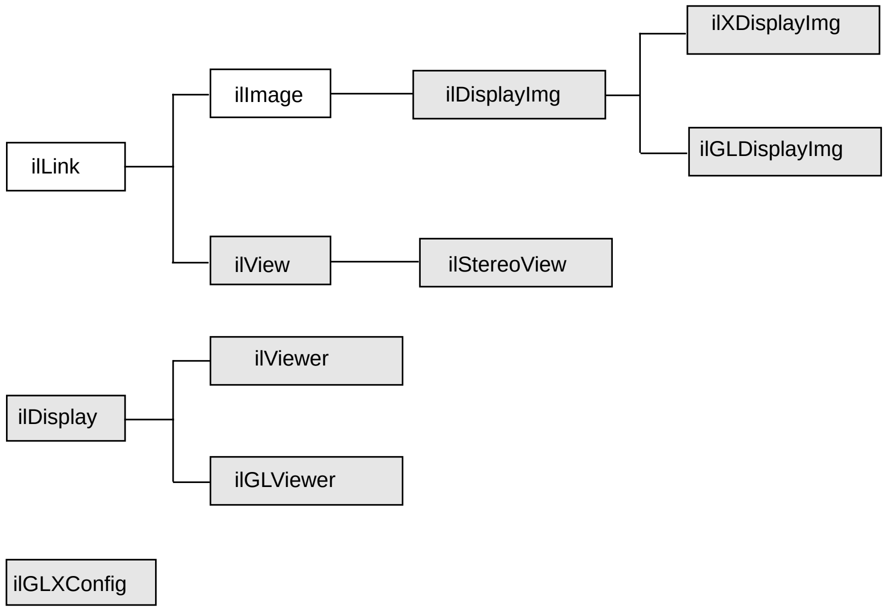


Figure 5-1 IL Display Classes

With the IL's display facility, you can display any combination of IL or X images in an X or GL window.[1] These images can be positioned anywhere within the window and can overlap each other. Overlapped regions are displayed based on a stacking order such that the image on top is visible, as shown in Figure 5-2.

[1] In the future, the GL will no longer support window and event management. You are encouraged to use mixed-model programming instead. A mixed-model program is an X program that uses GL to handle graphics.

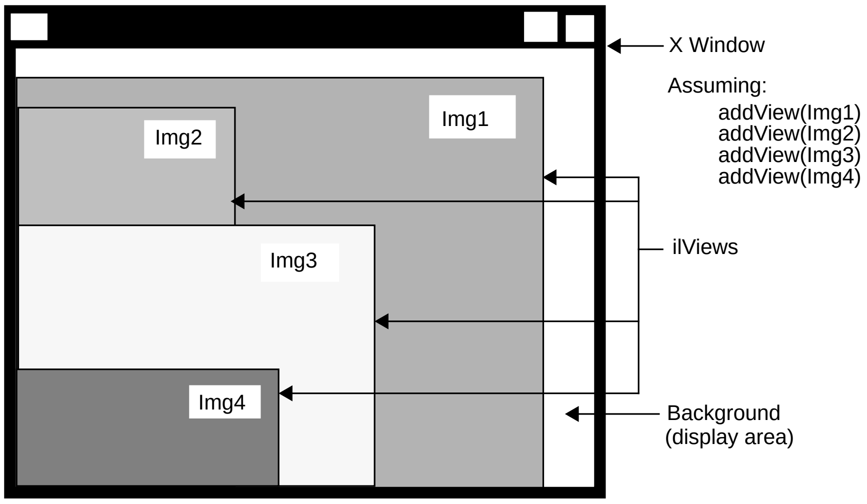


Figure 5-2 Stacked Images in an X Window

In order to assemble such a display, you must:

1. Create or open the images.

   You can display any combination of ilImage, ilXImage, or XImage using either GL or X to render them. (The ilXImage class is described in "X Window Images" on page 103. An XImage is an X Window **struct**.) Often, displayed images are the product of an image processing chain. For example, you might want to display the original unprocessed image, an intermediate stage of the chain, and the final image. In some cases, you might want to display only a portion of an image.

2. Configure and open a window.

   To open an X window, you can use the standard X calls or you can use the ilGLXConfig object, as explained in "Creating an ilDisplay" on page 208. To open a GL window, you can use the GL calls explained in "A Sample Program in C++" on page 4.

3. Create a display.

   Use calls to ilDisplay functions.

4. Add the images to the display.

   Use calls to ilDisplay functions.

5. Cause the images to be displayed.

   Use calls to ilView and ilDisplay functions.

Note: You should assume that any function discussed in this chapter is an IL function, unless it's explicitly identified as a GL or X function.

The three principal classes within the IL display facility are:

- ilDisplay—Manages one or more ilViews in an X or GL window. The entire window is used for display. An ilDisplay object maintains a stack of ilViews and provides functions to manipulate them. Two classes derive from ilDisplay: ilViewer, which manages the display of images in an X window with X event handling, and ilGLViewer, which manages the display of images in a GL window with GL event handling.
- ilView—Maps an ilImage or XImage to a region within the ilDisplay. It has various attributes such as view position, view size, image position, border color, and border width.
- ilDisplayImg—Acts as a base class for images that reside in the frame buffer. There are two derived classes:
  - ilXDisplayImg—implements reading and writing using the X Window library
  - ilGLDisplayImg—implements reading and writing using the GL

When an ilDisplay is created, it creates an ilGLDisplayImg for GL rendering or an ilXDisplayImg for X rendering. The display image is configured to occupy the entire window specified by the application. As Figure 5-3 illustrates, the creation of an ilDisplay object defines a display area in which views will be drawn.

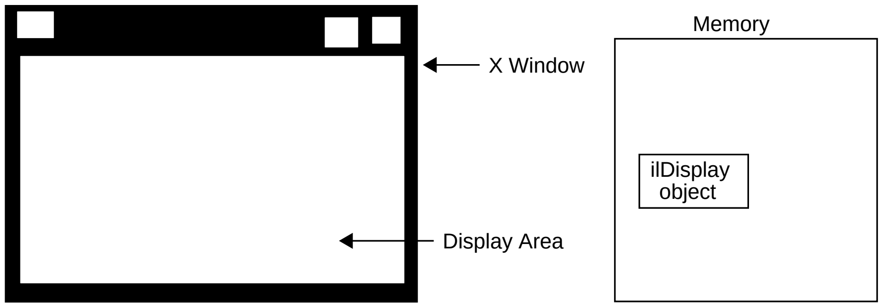


Figure 5-3 ilDisplay Object Creates a Display Area

When you want to add an image to ilDisplay, create an ilView to control where the image is to be displayed. This view is pushed onto an indexed view stack and a pointer to it is returned to your application. As you add more images, you must create an ilView for each image. These views are pushed onto the view stack. When a view is added to the stack, it is pushed onto the top by default. However, you can specify a particular index to control where the view is put in the view stack. An ilView has various attributes such as:

- view position that controls where in the window the image is displayed
- view size that controls how much of the image is displayed
- image position that controls what part of the image is displayed

The position of an ilView in the view stack controls its visibility on the screen, as shown in Figure 5-2. The view on the top of the stack is fully visible. A view at the bottom of the stack is obscured by the views above it.

In Figure 5-4, two ilView objects have been created and the positions of the corresponding views in the display defined. Two images to be displayed have been added to the view stack.

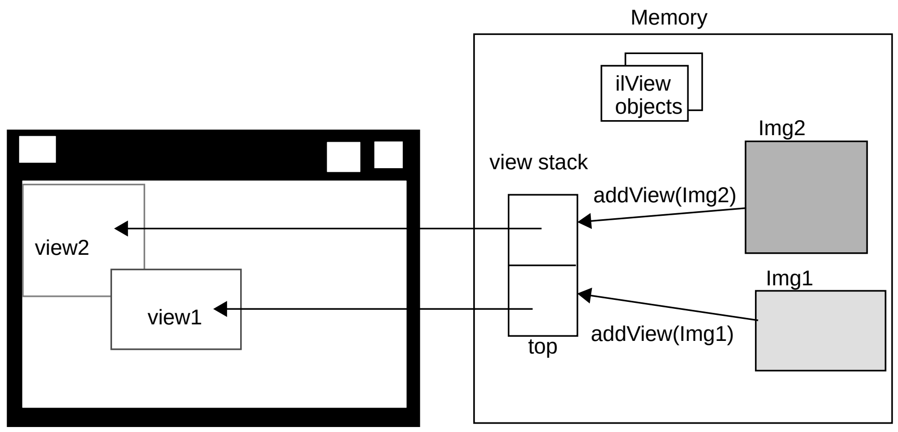


Figure 5-4 ilView Objects Map Images to Display Regions

When ilDisplay draws its contents, the position and size of each ilView, as well as the stacking order, are used to determine what portion of each view is visible. ilDisplayImg (GL or X) is called to render the images into the frame buffer. Each image is converted to the proper data type, order, color model, and coordinate space as necessary for displaying. Figure 5-5 shows the display after the views have been drawn.

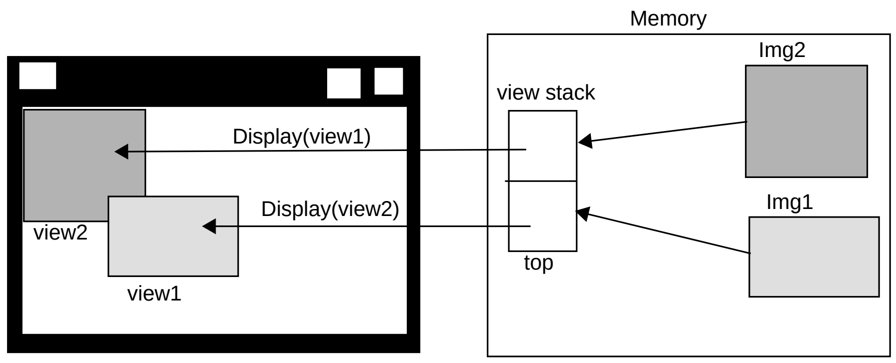


Figure 5-5 Display Area After Views Are Drawn

In addition to the views added by an application, ilDisplay creates a background view. This background view is the size of the window and is always at the bottom of the view stack. You can't control it other than to change its color from the default, which is black.

ilDisplay provides several operators to manipulate ilViews as well as functions to facilitate interactive display. The display operators enable you to move a view, change its size, or move the image within the view. The display operators are discussed in detail in "Applying a Display Operator" on page 223. By default, view manipulation also causes the display to be redrawn; however, a sequence of display operations can be performed with drawing deferred. In addition, an application can explicitly control drawing. ilDisplay is optimized to draw only the areas that have changed or that have been exposed.

A Simple Interactive Display Program

Now let's look at a simple interactive program that shows the IL display facility in action. This program opens an image file and applies a threshold operator to it. Both the original image and the processed image are displayed in a window, stacked on top of each other. You can *wipe* the original image away gradually so that you can see the processed image beneath it. Wiping changes the view size. The best way to understand wiping, of course, is to compile and run the display program. It's available online in:

/usr/people/4Dgifts/examples/ImageVision/ilguide/displayEx.c++

When you run the program, you'll see a window displaying the original image. The processed image is actually underneath the original one, but you can't see it since the images are opaque and of the same size. If you click in the window with the left mouse button, a red highlight border appears around the image, indicating that it's ready to be wiped. To wipe, click the left mouse button near any edge of the image and drag toward the center of the image. As you drag, the processed image becomes visible as the original image is wiped away; release the button to stop the wiping. You can wipe any edge or corner of the original image. To exit the program, use the normal window manager menu command.

Sample Program Code

The code for the sample program is shown in Example 5-1 and discussed in the paragraphs following that. All the ilDisplay functions used in the program are explained in more detail in the appropriate sections in this chapter.

Example 5-1 A Simple Interactive Display Program

```
/* Example program showing IL display facility.

#include <stdlib.h>
#include <stdio.h>
#include <il/ilGLXConfig.h>
#include <il/ilGenericImgFile.h>
#include <il/ilThreshImg.h>
#include <il/ilDisplay.h>

const int Border = 10;// Threshold (in pixels)
// for edge finding operation

void main (int argc, char* argv[])
{
  if (argc < 2) {
    printf ("Usage: %s in-image1\n", argv[0]);
    exit(0);
  }

// Open input image file
  ilFileImg* in = ilOpenImgFile(argv[1], "r");
  if (in == NULL) {
    printf (“Couldn’t open image file: %s\n”, argv[1]);
    exit(0);
  }

// Create threshold image
  float threshVal = 100.0;
  ilPixel threshPix(ilFloat, 1, &threshVal);
  ilThreshImg thresh(in, threshPix);
```

Step 1: Open an image file and create a threshold image.

Step 2: Open an X window.

```
// Set up window configuration
  int rgbMode = getgdesc(GD_BITS_NORM_SNG_RED) > 0;
  int doubleBuffer = getgdesc(GD_BITS_NORM_DBL_RED) >= 8;
  ilGLXConfig glx;
  glx.addEntry(GLX_NORMAL, GLX_RGB, rgbMode);
  glx.addEntry(GLX_NORMAL, GLX_DOUBLE, doubleBuffer);
  glx.addEntry(0, 0, 0); // terminator

// Get Connection to X server and open X window
  ilSize size;
  in->getSize(size);

  Display* dpy = XOpenDisplay(NULL);
  Window win = glx.createWindow(dpy,
            RootWindow(dpy, DefaultScreen(dpy)),
            0, 0, size.x, size.y, 0);

  XSelectInput(dpy, win, ExposureMask | KeyPressMask |
            PointerMotionMask | PointerMotionHintMask |
            ButtonPressMask | ButtonReleaseMask);

//Set window title

  XStoreName(dpy, win, argv[0]);

  XMapWindow(dpy, win);
  glx.winset();
```

Step 3: Create an ilDisplay object and add the images.

```
// Create ilDisplay object and add the images
  ilDisplay disp(dpy, win);
  disp.addView(&thresh);
  ilView* inView = disp.addView(in);
  disp.setBorders(TRUE);
```

Step 4: Process events.

```
// Process events, allowing wipe between original & processed
// images using the left mouse button
  int active = TRUE;
  int wipemode = 0;

  while (active) {
    ilXYint winSize;
    XEvent event;
    XNextEvent(dpy, &event);
    switch (event.type) {
    case MotionNotify:
      // flush the event queue
      Window rw, cw;
      int rx, ry, x, y;
      unsigned int state;
      XQueryPointer(dpy, win,
          &rw, &cw, &rx, &ry, &x, &y, &state);
      if (event.xmotion.state&Button1Mask)
          inView->wipe(x, y, wipemode|ilClip);
      else
           wipemode = inView->findEdge(x, y, Border);
      break;

    case ButtonPress:
      disp.setStart(event.xbutton.x, event.xbutton.y);
      break;

    case Expose:
      disp.display(NULL, ilDefer|ilCenter);
      disp.redraw();
      disp.getSize(winSize.x, winSize.y);
      break;

    case DestroyNotify:
      active = FALSE;
      break;
    }
  }

  XCloseDisplay(dpy);
}
```

Sample Program Comments

The first half of this program should be familiar to you; it's very similar to the sample program in Chapter 1, "Writing an ImageVision Library Program." The first several lines of code include the necessary header files. If the user specifies fewer than two arguments (the name of the program and the name of the image file), the program prints an error message and then exits.

Step 1: Open an Input Image File and Create a Threshold Image

The specified image file is opened as an ilFileImg object called *in*. Next, an ilPixel object is created for use by the threshold operator; the threshold value chosen for this example is 100.0. The ilThreshImg operator *thresh* sets each pixel to its maximum possible value if the pixel value is greater than or equal to the threshold value, or to its minimum possible value if it's less than the threshold value.

Step 2: Open an X Window

After the threshold image is created, the window to be created must be configured. This is accomplished with the ilGLXConfig object *glx*. The **getgdesc()** GL function, which returns information describing the graphics system, avoids RGB mode if the machine doesn't support it. The necessary information about the RGB and double buffering modes are set with ilGLXConfig's **addEntry()** function.

After the configuration of the window is specified, the size of the threshold image is determined with **getSize()**, and this information is used to create a window. The X server connection is created with the X function **XOpenDisplay()**.

The **createWindow()** member function of ilGLXConfig accepts arguments specifying the X server connection, parent window, origin, and size of the X window to create. This new window is saved as *win*. The X macro **RootWindow()** returns the root window of the system; here it takes as its second argument the result from the X macro **DefaultScreen()**, which returns the screen number referenced by the previous call to **XOpenDisplay()**.

The X function **XSelectInput()** tells the window system to monitor the input events corresponding to the specified event masks. When the user triggers an event (for example, moving the mouse), the event is added to the X event queue. In this case, the user can perform one of three actions:

- quit by selecting the quit item in the window menu (event DestroyNotify)
- press the left mouse button to initiate dragging (event ButtonPress)
- drag the mouse to perform a wiping (event MotionNotify).

The X function **XMapWindow()** maps the window to the screen, and ilGLXConfig's **winset()** function creates a GL context to allow GL rendering in that window.

Step 3: Create an ilDisplay Object and Add the Images

Next, an ilDisplay object is created by passing the appropriate X window and display ids. As shown, the processed image *thresh* is added first, and then the original image *in* is added using **addView()**. The **addView()** function creates an ilView for the specified image, adds it to the display's view stack, and returns a pointer to the ilView. The pointer to the ilView associated with *in* is used later in the program.

The order in which images are added determines their stacking order when they're displayed; the last view added is on top. In this case, the original image is displayed on top of (and completely obscuring) the processed image. You can reorder the views as needed.

The **setBorders()** function is used with default arguments in this example to highlight all views in the view stack. You can also specify the border color and width.

Step 4: Process Events

Processing events is a critical task for interactive programs. All of the previously identified inputs (events) must be handled. The event loop in this example processes events continuously while the program is active.

The code in the event loop uses the following variables:

active	indicates that the window is still active. It becomes FALSE when the user selects “Quit” from the window menu.
event	holds the event read from the X event queue with **XNextEvent()**.
winSize	indicates the current size of the window.
wipemode	indicates which edge of the image to wipe.
x and *y*	hold the *x* and *y* positions of the mouse, respectively, as the user drags to perform a wipe.

The code in the event loop takes the following actions in response to use actions:

- MotionNotify. As the mouse is moved, the *x* and *y* values are accessed with the X function **XQueryPointer()**. The current *xy* location of the mouse is passed to **wipe()**, which changes the size of the view by moving one or more edges. While the mouse button is pressed, motion causes a wipe to be performed. If the mouse button is *not* pressed, then **findEdge()** is called. The **findEdge()** function returns the edge(s) near the *xy* location specified and is saved in *wipemode*. (In this program, the user must click within 10 pixels of the edge.) When **wipe()** is called, *wipemode* specifies which edges to wipe.
- Button Press. When the user presses the left mouse button to start wiping, the **setStart()** function is called to save the current *x* and *y* mouse positions.
- Expose. The first time through the loop, an Expose event is processed, causing the entire display to be drawn on the screen with the **redraw()** function.
- DestroyNotify. When the user quits, the active flag is set to FALSE.

Creating an ilDisplay

To incorporate the IL's display facility in your program, you must:

1. Open and configure a window.
2. Create an ilDisplay object to manage the window.
3. Add and manipulate images you want to display.
4. Apply the desired display operator(s) to one or more of the views.

This section discusses the first two of these items. The remaining two items are covered in detail in following sections.

Opening and Configuring a Window

Before creating an ilDisplay object, you must open a window and configure it. To do this, use standard X calls for rendering or use the ilGLXConfig object as follows:

```
int rgbMode = TRUE, doubleBuffer = TRUE;

// set up configuration: rgb mode, double buffering
ilGLXConfig glx;
glx.addEntry(GLX_NORMAL, GLX_RGB, rgbMode);
glx.addEntry(GLX_NORMAL, GLX_DOUBLE, doubleBuffer);
glx.addEntry(0, 0, 0); // terminator - end of config list

// connect to X server
Display* dpy = XOpenDisplay(NULL);

// open X window
Window win = glx.createWindow(dpy, RootWindow(dpy,
    DefaultScreen(dpy)),
    origin.x, origin.y,
    winsize.x, winsize.y, 0);
```

After you create an ilGLXConfig object, the configuration of the window it creates is set with calls to the **addEntry()** function. In the example above, three calls are required. The first turns on RGB mode; the second turns on double buffering. The last call terminates the list. After a connection to the X server is made with the X call **XOpenDisplay()**, the ilGLXConfig object can create a new X window with **createWindow()**. See the X call **XCreateWindow()** for similar arguments. Here, the parent window is the root window. The next several arguments specify the origin, width, and height of the window. The last argument specifies the width of the window's border, zero in this case.

Creating an ilDisplay Object

An ilDisplay object manages views of images within the window passed to it. If an X window is passed, render mode specifies whether X or GL should be used to render the images. Only GL rendering can be used if a mixed model window is passed (ilGLXConfig is used to create the window). If a GL window is passed, GL rendering is automatically selected. Hardware acceleration is only available with GL rendering. The constructor for ilDisplay is shown below:

```
ilDisplay (Display* display, Window win, ilRender
    rendmode=ilGLRender, int mode = ilDefault);
```

The statement below creates an ilDisplay object for the window created in the preceding section:

```
ilDisplay myDisp(dpy, win);
```

The ilDisplay object created has the GL graphics configuration (single versus double buffer and RGB versus color map mode) that was established by the ilGLXConfig object that created the window. When ilDisplay is created, a render mode of ilGLRender (default) or ilXRender can be specified. If the render mode is GL, then the display origin is the lower left corner. If the render mode is X, then the display origin is the upper left corner. The entire window is used for drawing, but this window may be a subwindow within an application.

Note: The ilDisplay class uses many enumerated types, which are listed in "Enumerated Types and Constants" on page 408 and the header file *il/ilDisplayDefs.h*.

Changing a Graphics Configuration

To change an ilDisplay's graphics configuration, you need to create a new X window with the desired configuration, then switch to the new window with the **setWindow()** function:

```
ilGLXConfig nextglx;

// set up configuration: non-rgb, non-double buffered
nextglx.addEntry(GLX_NORMAL, GLX_RGB, FALSE);
nextglx.addEntry(GLX_NORMAL, GLX_DOUBLE, FALSE);
nextglx.addEntry(0, 0, 0); // terminator

Window win2 = nextglx.createWindow(dpy, // open X window
             RootWindow(dpy, DefaultScreen(dpy)),
             origin.x, origin.y,
             winsize.x, winsize.y, 0);

myDisp.setWindow(win2);
```

In this example, a new configuration is created with *nextglx*. The new window created, *win2*, has RGB mode turned off and double buffering disabled. The *myDisp* display switches to this new window. Since an X window cannot change its configuration after it has been created, this is the only way to change the configuration of a display.

Until an X window is deallocated (with the X call **XDestroyWindow()**), it can be used again; you can switch back to the original configuration by using **setWindow()**:

```
myDisp.setWindow(win);
```

You can query an ilDisplay about its graphics configuration with **isDoubleBuffer()** or **isRGBMode()**. Each function returns a TRUE if the feature is on and a FALSE otherwise.

Note: If you no longer need the previous window, you should free or delete it to save or recapture memory.

View and Display Basics

Once you have created a display object, the next step is to add views to this display and then apply display operators to these views. Before learning more about views, however, you'll need to be familiar with some basic concepts that apply to both displays and the views contained within them.

Background Color

If the images being displayed don't cover the entire display area, the ilDisplay's background view is seen in the uncovered areas. The background may also be revealed if images are dragged around or resized by the user. By default, an ilDisplay uses black as the background color. You can set the color to any pixel value with **setBackground()**:

```
unsigned char bgdColor[] = {0, 255, 0};
myDisp.setBackground( ilPixel(ilUChar, 3, bgdColor) );
```

In this example, the background color is set to green.

You can also retrieve an ilDisplay's current background color:

```
ilPixel pix;
myDisp.getBackground(pix);
```

Note: The color is always returned as a three-channel (RGB) pixel, whether the rendering occurs in X or GL.

Borders

All ilViews have borders, but by default they're not drawn (that is, they're turned off). You can use ilView's **setBorders()** to turn borders on (TRUE) or off (FALSE).

```
void setBorders(int flag);
```

When borders are turned off, the highlight flag (see "Finding a View" on page 220) is also turned off. The borders are painted or erased immediately unless painting is deferred. Note that borders are painted inside the view.

In addition, both the borders and the nop flag can be controlled using the select functions on ilView (see “Preventing View Operations” on page 213 to learn more about the nop flag). When **select()** is called, borders are turned on and its nop flag is turned off. When **unselect()** is called, borders are turned off and its nop flag is turned on. The **isSelected()** function returns TRUE if the view is selected or FALSE otherwise:

```
void select();
void unselect ();
int isSelected();
```

You can also specify the width and color of the borders:

```
void setBorderWidth(unsigned int bordWidth);
void setBorderColor(ilPixel *pix);
```

The first function sets the width of the border in pixels to *bordWidth*; by default, a border has a width of two pixels. (*bordWidth* should be a number greater than or equal to zero.) The second function sets the color of the border to the specified pixel value; borders are red by default.

You can query an ilView about its border width or color:

```
unsigned int getBorderWidth();
ilPixel *getBorderColor();
```

Note that the color is always returned and set as a three-channel (RGB) pixel, whether rendering occurs in GL or X.

For convenience, you may set border parameters on all the views in an ilDisplay’s view stack by calling the corresponding functions on ilDisplay. (You can exclude particular views in the stack from being acted upon by these functions by setting a nop flag in each view you wish to exclude. See “Deferring Drawing” on page 213.) For example:

```
float *bordColor = {0.0, 0.0, 255};

myDisp.setBorders(TRUE);
myDisp.setBorderWidth(5);
myDisp.setBorderColor(ilPixel(ilFloat,3,bordColor));
```

There are no convenience functions for **getBorders()**, **getBorderWidth()**, or **getBorderColor()** in the ilDisplay class since the information may vary from view to view.

Preventing View Operations

To keep any view in the stack from being operated upon, use the **setNop()** function to set the nop flag:

```
void ilDisplay::setNop(int nop, ilView* view);
void ilView::setNop(int nop);
```

If the *nop* argument is TRUE, then the view won't be operated on. To allow operations to take place on a view, *nop* should be FALSE. You can use the function **isNop()** to determine the state of the nop flag:

```
int ilDisplay::isNop(ilView* view);
int ilView::isNop();
```

If you need to perform an operation on each view in the stack regardless of the value of each view's nop flag, pass the ilDop flag in the *mode* for that operation.

If an operation is called on a view, the nop flag is overridden. For example, the statement below ignores the nop flag on the specified view:

```
view->wipe();
```

Deferring Drawing

Drawing can be deferred by calling **setDefer()** on ilDisplay or ilView. When used to defer the display, nothing is drawn; however, each view can be individually deferred as well. These calls are shown below:

```
void ilDisplay::setDefer(int def, ilView *view=NULL);
void ilView::setDefer(int def);
```

In the ilDisplay version, you specify the view in which you wish to defer drawing (the default is all views) by setting the ilView pointer argument to:

- NULL, which causes all views in the view stack to be affected.
- A pointer to an ilView in the view stack.

You might want to defer drawing until you've made a series of changes to an ilDisplay's attributes (or to those of its views) so that they all take effect simultaneously. You might also want to defer drawing while you apply more than one display operator to avoid drawing intermediate results. In addition, most of the display operators allow you to pass the ilDefer flag (see “Mode Flags” on page 215) to defer drawing. (Display operators are described in more detail in “Applying a Display Operator” on page 223.)

To defer drawing, call **setDefer()** and pass TRUE as its *def* argument. After that, the display won’t be redrawn until you call **setDefer()** with FALSE as its *def* argument. You can check whether drawing is deferred with **isDefer()**:

```
int ilDisplay::isDefer(ilView *view=NULL);
int ilView::isDefer();
```

This function returns TRUE or FALSE to indicate whether drawing has been deferred or not.

The Drawing Area

An ilDisplay assumes that it can draw anywhere in the window that’s been passed to it. You can retrieve the current size of the drawing area with **getSize()**, which returns the *x* and *y* dimensions by reference:

```
void getSize(int& x, int& y);
```

Managing the Cache

With global cache management, using ilView to manage the cache on its input is unnecessary. The various cache management methods on ilView have no effect and will be removed in the future. Instead refer to “The Cache” on page 46 and “Optimizing Use of Cache” on page 308 for a discussion of the global cache management scheme.

Automatic Seek-ahead

The **isAutoSeek()** and **setAutoSeek()** functions have no effect and will be removed in the future. When hardware acceleration is used, the auto-seek feature can become disabled. For this reason, the auto-seek feature has been replaced by the prefetcher feature (see “Prefetching” on page 62).

Mode Flags

All the display operators use a mode argument to control the display of views. This mode is a bitwise-ORed combination of flags that control the operator. The flags are defined as enumerated values (see *il/ilDisplayDefs.h* or “Controlling the Display Facility” on page 413). Some flag types are described below:

Display Flags

Display flags specify various display modes. Examples are:

- ilClip to clip an image to the edge of the display or view
- ilDefer to defer painting
- ilDop to override the nop flag

Coordinate Flags

Coordinate flags specify how the resizing, relocating, and update operators are to interpret coordinate values. Examples are:

- ilDelVal where *x,y* is interpreted as delta relative to the current values
- ilRelVal to interpret the *x,y* coordinates relative to the starting *x,y* (starting *x,y* is updated)
- ilAbsVal to interpret the *x,y* coordinates as absolute values
- ilOldRel to interpret the *x,y* coordinates relative to the starting *x,y* (starting *x,y* is not updated)

Wipe Mode Flags

Wipe mode flags specify the edges in a wipe operation. Some examples are:

- ilTopEdge to do the wipe from the top edge
- ilLeftEdge to do the wipe from the left edge

Align Mode Flags

Align mode flags specify image alignment. Some examples are:

- ilTopLeft to align the view from the top left corner
- ilCenter to align the view to the center of the window or the image to the center of the view

The sample program shown at the beginning of this chapter contains an example of the use of the mode argument. In this example, the display operator initializes all views in the view stack, aligns the views to the center of the image, and defers the painting of the view until later.

```
disp.display(NULL, ilDefer|ilCenter);
```

Managing Views

Once an ilDisplay has been created, you can create views of the images you want displayed. As views are created, they are pushed onto the view stack. You can also retrieve views from the stack, replace the images within the views with other images, remove views, and reorder the views in the stack. This section explains how to perform these tasks.

Note: If an error occurs while rendering part of a view, the offending tile is painted with the error color, and the status is set on ilDisplay. The error color defaults to yellow, but can be set per view with:

```
ilView::setErrorColor(const ilPixel& pixel);
```

Adding Images

The **addView()** function creates an ilView and adds it to the view stack. The image is drawn when **addView()** is called unless ilDefer is passed in *mode*. It returns a pointer to the ilView for the ilImage (or XImage) pointer passed in:

```
ilView* addView(ilImage* img, int index, int mode);
ilView* addView(ilImage* img, int mode=ilCenter);

ilView* addView(XImage* img, int index, int mode);
ilView* addView(XImage* img, int mode=ilCenter);
```

You can call **addView()** with just the image or the image and the display mode. In this case, the view index defaults to 0 (top of the stack). If you use the version of **addView()** that takes an index, you can specify the location in the view stack where the image is to go.

The mode parameter controls the creation and position of the ilView. By default, the view is centered, not clipped to the display window, and is painted after being added. However, this behavior can be modified using various display mode flags such as ilClip and ilDefer. See “Mode Flags” on page 215 and the ilDisplay reference page for more details.

If an image has a *z* dimension that’s greater than one, you can choose which *xy* plane of the image to display. By default, the first plane (z = 0) is displayed. To display a different plane, call **setZ()** on ilView:

```
void setZ(int startZ);
```

The *startZ* argument specifies the desired plane of the image in the view that the function is called on. ilView’s **getZ()** function takes no arguments and returns the current *z* plane being displayed of the corresponding image.

Stereo Viewing

If your machine is capable of stereo and stereo is supported by IL on that machine, you can turn on stereo viewing mode. Currently, stereo is supported only on the RealityEngine. A stereo view can be created as shown below:

```
ilStereoView* addStereoView(ilImage* imgL, ilImage* imgR,
    int index=0, int mode=ilCenter);

ilStereoView* addStereoView(ilImage* zImg,
    int zLeft = 0, int zRight = 1,
    int index=0, int mode=ilCenter);
```

Using the first version, pointers to the left and right images are passed to this method on ilDisplay. The last two arguments specify where to add the view to the view stack and the display mode for the view. In this case, an IL chain must be set up for each image.

The second version takes a single image with the left and right images stored in the *z* dimension. The parameters **zLeft** and **zRight** specify the index in the *z* dimension corresponding to the left and right images. The benefit of this approach is that you can use a single IL chain to process both images.

In either case, the relative screen positions of the left and right images can be adjusted (see the ilStereoView reference page). If the hardware doesn't support stereo, or if stereo mode is disabled, only the left image is displayed. Also note that the IL can display a mixture of monoscopic and stereoscopic views in the same stereo window.

The application must allocate a stereo buffer using ilGLXConfig or GLXconfig similar to the way double buffer is done. The application must also configure stereo video mode by calling **setmonitor()** or **setmon()**. For more information, see the setmon and setmonitor reference pages or the example in *~4Dgifts/examples/ImageVision/ilapps/ilstereoview.c++*.

Retrieving Views

You can obtain a pointer to any view in the stack with **getView()**. There are two versions of this function, one that takes an index and another that takes a pointer to an ilImage:

```
ilView* getView(int index = 0);
ilView* getView(ilImage* img);
```

Both functions return a pointer to the corresponding ilView. If the image appears in more than one view, the view that's nearest the top of the stack is returned.

You can also retrieve the index corresponding to a particular view:

```
int theIndx = myDisp.getViewIndex(someView);
int theIndx = myDisp.getViewIndex(someImg);
```

The **getViewIndex()** function takes a pointer to an ilView (*someView*) or a pointer to an ilImage (*someImg*) and returns its index as an **int** (*theIndx*).

To determine how many views are in the view stack, call **getNumViews()**.

Retrieving Images

You can obtain a pointer to the image in a particular view with **getImg()** or **getXImg()**:

```
ilImage* myImg = someView->getImg();
XImage* myXImg = someOtherView->getXImg();
```

A pointer to the ilImage or XImage in the view is returned. (Here, *someView* and *someOtherView* are ilView pointers.)

To obtain pointers to the images in a stereo view, use **getLImg()** and **getRImg()**:

```
ilImage* myLeft = someStereoView->getLImg();
ilImage* myRight = someStereoView->getRImg();
```

A pointer to the left ilImage is returned from **getLImg()** and a pointer to the right from **getRImg()**. (Here, *someStereoView* is an ilStereoView pointer.)

Removing Views

You can remove a view from the stack by deleting the view or by calling **deleteView()** on ilDisplay. This function removes the specified view from the stack and deletes it:

```
void deleteView(ilView* view);
```

Replacing Images

An ilView object allows you to replace its image:

```
void setImg(ilImage* ilInImg);
void setXImg(XImage* xInImg);
```

The argument is a pointer to the image you want the view to hold. This image replaces the image mapped to the view.

Reordering the View Stack

Several functions are provided by ilDisplay to reorder the view stack. The **push()** function pushes the specified view down *count* places in the stack. By default, it pushes the view to the bottom. Similarly, the **pop()** function pops the specified view up *count* places in the stack. By default, it pops the view to the top. On both push and pop, when *count* is 1, the view is moved one position in the view stack. In addition, the **swap()** function swaps two views in the stack. These functions are shown below:

```
push(ilView *view, int count=0);
pop(ilView *view, int count=0);
swap(ilView *view1, ilView *view2);
```

Finding a View

Sometimes you need to find the view at a specified location. In an interactive program, the mouse is typically used to select a view. To find the view at a given *x,y* location, **findView()** can be called on ilDisplay as shown below:

```
ilView* findView(int x, int y, int mode = ilDspCoord);
```

This function returns a pointer to the topmost ilView found at location *xy* within the display. If there is no view at *xy*, it returns NULL. If ilHighlight is passed in *mode*, the view is highlighted if found. When a view is highlighted, its borders are turned on. However, only one view at a time can be highlighted. If ilDspCoord is passed in *mode* (the default), the *xy* coordinates are interpreted relative to the origin of the display area (display coordinates). If ilScrCoord in passed in *mode*, then the *xy* coordinates are interpreted relative to the screen (screen coordinates). Recall that the origin of the display area coincides with that of the window.

Finding an Edge

You may need an edge of a view for certain operations. Sometimes, you'll want to determine which edge of a view the cursor is near. This is especially useful for wiping, as described in "Applying a Display Operator" on page 223. For this, use ilView's **findEdge()** function:

```
int findEdge(int x, int y, int margin = -1,
             int mode = ilDspCoord);
```

This function determines which edge of the view is nearest to the specified *xy* coordinates. If the specified point is within *margin* pixels from an edge, that edge is returned. By default, the margin is either the default margin (15) or the current border width, whichever is greater. The *mode* argument can be either ilDspCoord (the default) or ilScrCoord to indicate whether *x* and *y* are specified in display or screen coordinates.

The value returned by **findEdge()** is a bitwise-ORed combination of the following values:

ilNoView	The coordinates lie outside *margin* pixels of all views.
ilRightEdge	The coordinates are within *margin* from the right edge.
ilLeftEdge	The coordinates are within *margin* from the left edge.
ilTopEdge	The coordinates are within *margin* from the top edge.
ilBottomEdge	The coordinates are within *margin* from the bottom edge.
ilAllEdge	The coordinates are within *margin* from all edges; this is an unusual case since it implies that *margin* is very large relative to the image. The ilAllEdge value is used primarily as an argument for **wipe()**, which is described in “Applying a Display Operator” on page 223.
ilNoEdge	The coordinates don’t lie within *margin* from any edge.

If a combination of two intersecting edges is returned—for example, ilRightEdge | ilTopEdge—you can treat the value as corresponding to a corner, in this case the upper-right corner. Note that you can also receive a value such as ilTopRight, which is equivalent to ilTopEdge | ilRightEdge.

ilDisplay also defines a **findEdge()** function, which finds the edge on all views. For each view, it saves the edge for later use with **wipeSplit()**.

Operating on a Pixel

You can obtain the actual pixel data at a specified point in a view with **getPixel()** (defined by both ilDisplay and ilView):

```
ilStatus getPixel(int x, int y, ilPixel& pix, int mode = 0);
```

In ilView's version, this function copies the pixel data at the point *x*,*y* into *pix*. If the point lies outside the view, the fill value is returned by reference. In ilDisplay's version, the topmost view pointed to by the point *x*,*y* is found with **findView()**; the pixel data from the point in that view is copied into *pix*. If the point refers to no view, no pixel data is returned by reference. (The *x*,*y* point is specified in display coordinates.)

You can also set a pixel value with **setPixel()**:

```
void setPixel(int x, int y, ilPixel pix, int mode = 0);
```

Locating a Point

You can find out where you are in an image by passing the display coordinates to **getLoc()** (defined by both ilDisplay and ilView):

```
void getLoc(int x, int y, int& ix, int& iy,
           int mode = ilLocIn);
void getLoc(float x, float y, float& ix, float& iy,
           int mode = ilLocIn);
void getLoc(float& ix, float& iy,
           int mode = ilLocIn|ilCenter);
```

The **getloc()** function returns the location in the image corresponding to *x* and *y*. The location in the image is returned in *ix* and *iy*. If *ilLocIn* is passed in mode, the location is returned in the input space of the image. If *ilLocOut* is passed in mode, the location is returned in the output space of the image. For example, if an ilRotZoomIng is mapped to the view and *ilLocIn* is specified, *ix* and *iy* correspond to the location in the unzoomed image. However, if ilLocOut is specified, *ix* and *iy* correspond to the location in the zoomed image. If ilLocImg is specified (default), then the image is moved to the specified location. If ilLocView is specified, then the view is moved to the specified location.

The second version uses floating point values for more accuracy. The third version determines the desired location based on mode. For example, if ilCenter is specified, the location corresponding to the center of the view returned.

When called on ilDisplay, the topmost view pointed to by *x, y* is found with **findView()**. Then the location is returned for that view. On both ilDisplay and ilView, a version is provided that doesn't require an *xy* location to be specified. Instead, the *mode* is used to specify the center or a corner.

Similarly, you can set the location of an image within the display by calling **setLoc()** on ilDisplay or ilView. This allows you to move a point within the image to a specific location within the display as show below:

```
void setLoc(int ix, int iy, int x, int y,
          int mode = ilLocIn);
void setLoc(float ix, float iy, int mode = ilLocIn|ilCenter);
void setLoc(float ix, float iy, float x, float y,
          int mode = ilLocIn);
```

The relocation can be accomplished by moving the image or the view. If ilLocView is specified, then the view is moved, otherwise the image is moved (ilLocImg).

Applying a Display Operator

Display operators alter views, typically in response to input from the user. These operators may draw all or portions of a view. Also, they can change the size and / or location of all or some of the displayed views and then update the display accordingly. These are the IL's display operators; they can be called on both ilDisplay and ilView (except for **display()**, which may be called only on ilDisplay):

- Drawing operators—Operators whose primary purpose is to draw all or part of the display. This group includes **display()**, **paint()**, **redraw()**, **setStaticUpdate()**, and **save()**.
- Relocating operators—Operators whose primary purpose is to change the location of views or images. This group includes **alignView()**, **alignImg()**, **moveView()**, **moveImg()**, and **split()**.
- Resizing operators—Operators whose primary purpose is to change the size of views or images. This group includes **wipe()**, **wipeSize()**, **wipeSplit()**, and **resize()**.
- **update()**—Generalized operator that combines the capabilities of **moveView()**, **moveImg()**, and **wipe()**. However, because it is a generalized operator, it is not as optimized as some of the other operators.

There is only one difference between calling a function on ilDisplay and calling it on ilView. When called on ilView, the function only operates on that view regardless of the state of the nop flag. In contrast, when called on ilDisplay, a view must be specified. If NULL is passed, then all views in the stack are operated on (except those with the nop flag set). If a pointer to a view is passed, the function only operates on that view.

In this section, all operators are given in their ilDisplay forms. The ilView versions are easily derived by leaving out the argument specifying the view.

Drawing Views

The functions used primarily for drawing are described in this section:

- **display()** reinitializes the specified view and optionally aligns the view and image. The specified view is then painted. If NULL is specified, then all views are initialized (except those with nop flag set).
- **paint()** doesn't resize or reposition the view. It simply paints the specified view if it needs to be painted. If *ilPaintExpose* is passed, then the view is forced to be painted.
- **redraw()** resizes the display image and background view to occupy the entire window. It then paints all views regardless of the nop flag. It doesn't resize or reposition any views.
- **save()** paints the specified region of the display to an ilImage. A starting location within the display and a pointer to an ilImage are passed. The save region is specified by the starting location and the size of the image.
- **setStaticUpdate()** sets the staticUpdate mode to paint a rectangular region as one tile rather than many smaller ones.

display()

The **display()** function takes three arguments, all of which have default values as shown below:

```
void ilDisplay::display(ilView* view = NULL,
                  int vmode = ilCenter,
                  int imode = ilCenter);
```

view	Reinitializes the specified view. If NULL is passed, then it reinitializes all views (except those with nop flag set). If the ilDop flag is passed in mode, the nop flag is ignored.
vmode	Specifies how to align the view within the display.
imode	Specifies how to align the image within the view. As explained above, only the visible portion of each view is drawn.

Both *vmode* and *imode* are a bitwise-ORed combination of values that allow you to specify alignment. You can align to any corner or edge using any combination of ilTopEdge, ilBottomEdge, ilLeftEdge, or ilRightEdge. In addition, ilTopLeft, ilBottomLeft, ilTopRight, or ilBottomRight can be used to specify a corner. By default, ilCenter is used. If no alignment is desired, ilNoEdge or ilNoAlign can be passed instead. See “Relocating Views and Images” on page 227 for more information about **the alignView()** and **alignImg()** functions.

By default, a view is the size of its image; however, if ilClip is passed in *vmode*, then the view is clipped to the size of the display or window.

paint()

The **paint()** function is typically used when a view needs to be redrawn after several deferred operations. This function takes a view pointer and a mode as arguments:

```
void paint(ilView* view = NULL, int mode = 0);
```

The *view* argument has the same meaning as that for **display()**. The *mode* argument can include any of the generic display flags.

redraw()

The **redraw()** function is called when a REDRAW (GL) or Expose (X) event occurs (for example, if the window is exposed or resized):

```
void redraw(int mode = ilDefault);
```

The **redraw()** function resizes the drawing area (display image) and the background view to match the new size of the window, and paints all views.

save()

The **save()** function saves a region of the display by painting to an ilImage. The region saved is specified by the origin *x, y* and is the size of the image passed in:

```
ilStatus save(ilImage* img, int x = 0, int y = 0,
                         int mode = ilDefault);
```

By default, borders are not painted. However, if ilPaintBorder is passed in mode, the borders are painted. Note that on 8-bit graphics systems, displayed images may be dithered. Therefore, the save function provides a higher quality result than copying from the screen.

setStaticUpdate()

The **setStaticUpdate()** function allows you to enable or disable the staticUpdate mode. Static update paints a rectangular region as one large tile rather than as many smaller tiles. When staticUpdate mode is enabled, it forces a static update to occur whenever the view is painted.

```
void setStaticUpdate(int enable)
```

The **setAutoStaticUpdate()** function forces a static update after a reset has occurred. A reset is caused by changing inputs or processing parameters in the chain. In this case, since the entire exposed region of the view must be painted, the performance can be improved by painting the region as one large tile. After the static update has been completed, normal tiled painting resumes. By default, automatic static update is enabled.

```
void setAutoStaticUpdate(int enable)
```

The **isStaticUpdate()** function allows you to retrieve the current staticUpdate mode:

```
isStaticUpdate()
```

Note: Static update mode only has effect for hardware acceleration.

Relocating Views and Images

The functions used to relocate views and images are described in this section:

- **alignImg()** aligns an image within its view.
- **alignView()** aligns a view with a reference view.
- **moveImg()** moves an image within a view.
- **moveView()** moves a view within the display area.
- **split()** repositions all views into rows and / or columns and resizes the views to fit.

The following mode flags are also used in conjunction with the functions discussed in this section:

ilAbsVal	The *xy* pair represents absolute values. In other words, the view is simply moved to the location specified.
ilDelVal	The coordinates represent a change (delta) in the current view or image position. For example, if moveView is called with (*2*,*5*) and the specified view is located at (*1*,*1*), then the view is moved to (*3*,*6*).
ilRelVal	The *xy* pair is interpreted relative to the starting *xy* set by calling **setStart()**. The starting *x,y* values are updated. The **setStart()** function must have been called previously to initialize ilDisplay's coordinate values. This is the default mode for most functions.
ilOldRel	Same as ilRelVal except that the starting *xy* values aren't updated.

alignImg()

The **alignImg()** function is defined on both ilDisplay and ilView. This function aligns the image in *view*. If *view* is NULL (the default), the function aligns the images in all the views in the view stack (except those with the nop flag set). It is called as shown below:

```
void alignImg(ilView* view=NULL, int mode=ilCenter);
```

Alignment means that an edge, corner, or center of an image is aligned within the view, as shown in Figure 5-6. The *mode* argument specifies how to align the image. For example, the default, ilCenter, indicates that the image is to be centered in the view. In Figure 5-6, ilBottomLeft is passed in *mode*, causing the lower left corner of the image to be aligned to the lower left corner of the view.

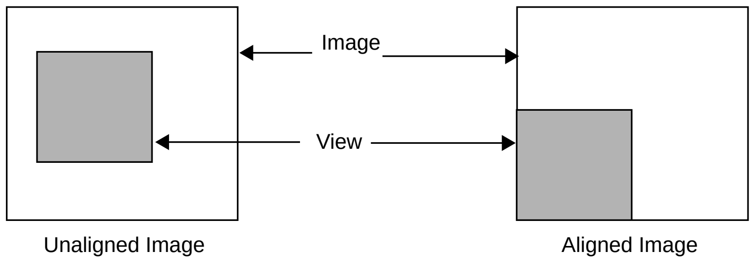


Figure 5-6 Aligning an Image to Bottom Left Corner

alignView()

The **alignView()** function is defined on both ilDisplay and ilView. This function aligns the specified view with a reference view. If NULL is passed, all views are aligned (except those with the nop flag set). The function is called as shown below:

```
void alignView(ilView* view = NULL, int mode = ilCenter,
   ilView* rView = NULL);
```

The reference view is specified by *rview*. If it is NULL, then the back view is used. Alignment means that edges, corners, or centers of the views are aligned, as shown in Figure 5-7. The *mode* argument specifies how to align the views. By default, ilCenter causes views to be aligned by their centers. In Figure 5-7, the views are aligned by their lower left corners with ilBottomLeft.

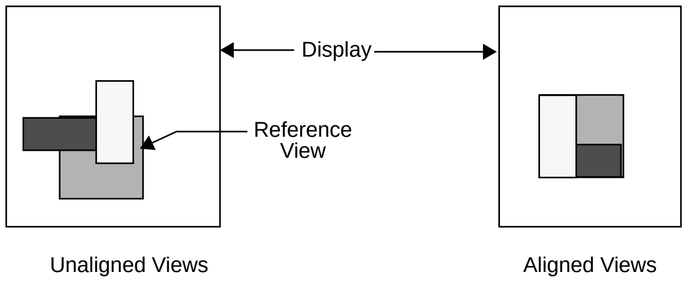


Figure 5-7 Aligning Views

moveImg()

The **moveImg()** function changes the location of images within their respective views. To use this function, you need to specify the desired location and the view to which the image corresponds:

```
void moveImg(int x, int y, ilView* view = NULL,
   int mode = ilRelVal);
```

This function moves the image within the specified view. In other words, the view remains fixed relative to the display while the corresponding image moves within the view. This function allows a user to roam around an image. This is particularly useful for large images that are bigger than the screen. Thus, the coordinate values *x*,*y* specify the desired location of the image's origin. They're interpreted according to the relevant flags passed in *mode* (such as ilDelVal, ilRelVal, and so on). The *mode* argument can also include flags indicating that drawing should be deferred (ilDefer) and that the image shouldn't be moved beyond its edge (ilClip). By default, the image can be moved beyond its view, in which case the image's fill value is used to paint the view.

moveView()

The **moveView()** function changes the location of views within the display. You might use this function to allow a user to drag a view around the display area using one of the mouse buttons. To use this function, you need to specify the desired location and the view to be moved:

```
void moveView(int x, int y, ilView* view = NULL,
    int mode = ilRelVal);
```

The view pointer argument *view* specifies which view to move (or all the views if NULL, the default). The *x* and *y* arguments indicate where to move the view, and *mode* specifies how these arguments should be interpreted (with ilDelVal, ilRelVal, and so on).

You can include ilDefer in the *mode* argument if you don't want the display updated. Also, by default, you can move the views out of the window. For example, a user can continue dragging a view past the edge of the window; the view won't be visible, but ilDisplay keeps track of its location so that if the user drags it in the opposite direction, eventually the view becomes visible in the window. You can prevent a view from being moved past the window's edge by specifying ilClip as part of the mode argument.

split()

The **split()** function allows you to display all views next to one another in rows and / or columns rather than randomly overlapping one another. All views are resized and repositioned based on the number of views in the view stack. Starting at the bottom of the stack, views are positioned starting at the lower left corner of the display. The **split()** function is called as shown below:

```
void split(int mode = ilAbsSplit|ilRowSplit|ilColSplit)
```

The *mode* argument controls the layout. It can be a combination of the following modes:

ilAbsSplit	Aligns images to the origin regardless of the view position. (See Figure 5-8.)
ilRelSplit	Positions images relative to view position. (See Figure 5-9.)
ilRowSplit	Divides the drawing area into rows. (See Figure 5-8.)
ilColSplit	Divides the drawing area into columns.
ilPackSplit	Clips views to an image if needed and packs them together.

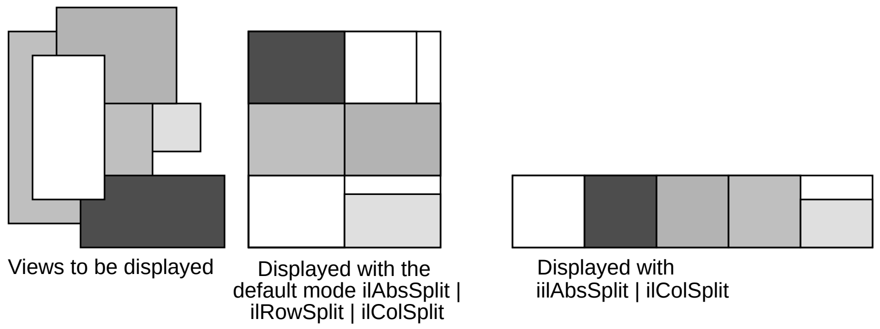

Views to be displayed
Displayed with the default mode ilAbsSplit | ilRowSplit | ilColSplit
Displayed with iilAbsSplit | ilColSplit

Figure 5-8 **split()** with ilAbsSplit | ilRowSplit | ilColSplit

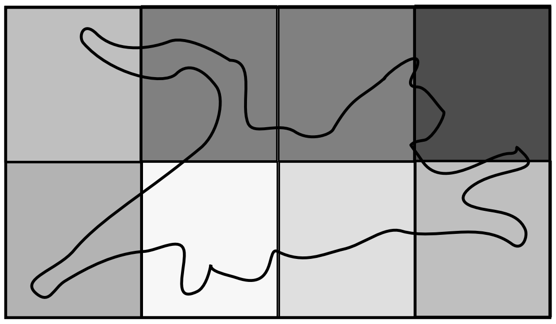


Figure 5-9 **split()** with ilRelSplit | ilRowSplit | ilColSplit

If both ilRowSplit and ilColSplit are specified, **split()** divides the drawing area into equal-sized rectangles such that the number of rows and columns is nearly equal. (See Figure 5-9.) Note that if both ilAbsSplit and ilRelSplit are specified, split defaults to ilAbsSplit. In addition, an alignment mode can be specified with ilAbsSplit, such as ilCenter.

Resizing Views

The functions used to resize one or more views are shown below and are described in this section:

- **resize()** resizes a view (defined only on ilView).
- **wipe()** moves one or more edges of a view.
- **wipeSplit()** wipes the nearest edge of all views.
- **wipeSize()** wipes an edge or corner and the opposite edge or corner.

As with the relocating functions, if ilAbsVal is passed in *mode*, the *xy* values specify the new size of the view. For ilDelVal, the *xy* values represent changes to the current size of the view. ilRelVal means that the *xy* values are interpreted relative to the start values previously set with **setStart()**. The start values are then updated by ilDisplay unless ilOldRel is specified.

resize()

The **resize()** function reinitializes the size of a view to the size of the image it displays. This useful after setting the image in ilView. The **resize()** function is called as shown below:

```
void resize(int mode = 0);
```

If ilClip is passed in *mode*, then the view is clipped to the size of the display. After the view is resized, it is painted unless ilDefer is passed.

wipe()

The **wipe()** function moves one or more edges on the specified view. It is called as shown below:

```
void wipe(int x, int y, ilView* view = NULL,
             int mode = ilRelVal);
```

The values *x* and *y* specify how to move the specified edge of the view. They’re interpreted according to the flags passed in *mode* (such as ilRelVal, ilDelVal, and so on). The default is ilRelVal. If NULL is passed for *view*, then all views are wiped (except those with nop flag set).

The edge to wipe is specified in *mode*. Any combination of the following edge modes can be used: ilRightEdge, ilLeftEdge, ilTopEdge, or ilBottomEdge. For example, ilTopEdge | ilRightEdge (or ilTopRight) allows the user to wipe the upper-right corner, thus resizing the view. In addition, the value returned by **findEdge()** can be used directly. (See "Finding an Edge" on page 220.)

If ilAllEdge (or a bitwise OR of all four edges) is used, the effect is slightly different from a normal wipe. In this mode, called an *inset*, the view moves while the image remains fixed (opposite of **moveImg()**). This mode is useful to move a processed view of an image around on top of the original image for comparison.

By default, the view is painted after it is wiped unless ilDefer is passed in *mode*. Also by default, the edge of a view can be moved beyond the edge of the image, unless ilClip is passed. When the view is allowed to be wiped beyond the edge of the image, the image's fill value is used to paint the exposed region. Note that the wipe function is optimized to paint only the wiped region.

wipeSplit()

The **wipeSplit()** function is used in conjunction with **findEdge()** on ilDisplay to wipe the nearest edge of all views. It is called as shown below:

```
void wipeSplit(int x, int y, int mode = ilRelVal);
```

The *x* and *y* parameters control how the edges are moved. No view is specified because it operates on all views in the view stack. The *mode* parameter specifies only how to interpret *x* and *y*. Note that the edge on each view is not specified by *mode*. Instead, **findEdge()** must be called on ilDisplay first to find the edge on all views. If no edge is found for a particular view, then that view is not wiped.

This function is useful after a split operation. For example, if the display is split to show two views side by side, it allows you to wipe the right edge of the left view and the left edge of the right view simultaneously. This is useful when comparing two or more images. In general, adjacent views can be wiped using this function.

wipeSize()

The **wipeSize()** function wipes the specified edge and the opposite edge to resize the view. It is called as shown below:

```
void wipeSize(int x, int y, ilView* view = NULL,
              int mode = ilDelVal | ilTopRight);
```

The *x* and *y* parameters control which way to move the edge specified in *mode*. In addition, the opposite edge is moved in the opposite direction, causing the view to grow or shrink in size. For example, if the right edge is moved to the right, then the left edge is moved to the left as well. In this case, the view would grow in width, as show in Figure 5-10.

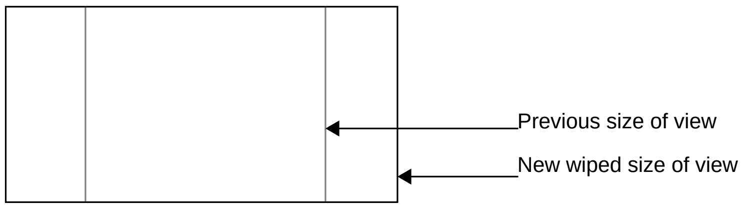


Figure 5-10 Using **wipeSize()**

Updating Views

The **update()** function can change the view position, view size, and image position as shown below:

```
void update(int x=0, int y=0, int nx=0, int ny=0,
              int imgX=0, int imgY=0,
              ilView* view=NULL, int mode=ilRelVal);
```

The view is moved to the position specified by *x* and *y* and is resized to *nx* and *ny*. The image within the view is moved to the position specified by *imgX* and *imgY*. If *view* is NULL, then all views in the view stack are updated (except those with nop flag set).

The first six of these parameters are interpreted as specified by *mode*. For example, if ilDelVal is specified, then all six parameters are interpreted as changes from the current configuration. In addition, the parameters are used as specified. However, if ilClip is passed in *mode*, then the view position, size, and image position are clipped. After the view has been updated, it is painted unless ilDefer is passed in *mode*. The update function combines the functionality of **moveView()**, **wipe()**, and **moveImg()**.

Using setStart()

A display support function that you might find useful as you apply display operators is **setStart()**:

```
void setStart(int x, int y, int mode = 0);
```

This function is typically used in an interactive loop to initialize the starting *x* and *y* coordinate values that the ilDisplay keeps track of. The coordinates passed to any function with ilRelVal or ilOldRel are interpreted relative to the current start values. If ilRelVal is specified, the old start values are updated; however, if ilOldRel is specified, the start values aren’t updated. This is useful if several operations are needed and you don’t want to update the start values until you are done. This model is used in the program presented in “A Simple Interactive Display Program” on page 201. To retrieve the previously set start values, use **getStart()**. This function returns the start values by reference:

```
void getStart(int& x, int& y);
```

You can achieve many different effects by judiciously deferring drawing while you apply a combination of these and / or any of the other display operators.

A More Complicated Interactive Display Program

The *imgview* interactive display program (which is installed in */usr/sbin* when you install the Image Tools) allows a user to drag, roam, and wipe several images in a display window. (See its reference page for more information.) A simplified version of its source code is provided online in:

/usr/people/4Dgifts/examples/ImageVision/ilapps

The C++ version of the simplified program is *ilview.c++*, and the C version is *ilcview.c*.

The portion of this program that processes events and calls display operators is shown below for your convenience. It uses an ilViewer to handle events. The ilViewer class is a higher-level object derived from ilDisplay. It calls ilDisplay functions and operators based on X events. It calls **moveView()** for left mouse button movement and **moveImg()** for middle mouse button movement. The cursor changes shape near the edges and corners to indicate that wipe mode is enabled on the left mouse button. If you press the left mouse button and perform a wipe, this changed cursor remains for the duration of the wipe. See the ilViewer reference page and the header file *il/ilViewer.h* for details. The source code for ilViewer is provided in:

/usr/people/4Dgifts/examples/ImageVision/ilsrc

Example 5-2 A More Complicated Interactive Display Program

```
// Create X window for GL or X rendering

Display* dpy = XOpenDisplay(NULL);

ilGLXConfig glx;
glx.addEntry(GLX_NORMAL, GLX_RGB, rgbMode);
glx.addEntry(GLX_NORMAL, GLX_DOUBLE, doubleBuffer);
glx.addEntry(0, 0, 0); // terminator

Window win = glx.createWindow(dpy,
        RootWindow(dpy, DefaultScreen(dpy)),
        origin.x, origin.y, winsize.x, winsize.y, 0);

XMapWindow(dpy, win);
glx.winset();

// Revert to default error handler
XSetErrorHandler(0);

// Set window title
XStoreName(dpy, win, argv[optind]);

// Select the input events that will be acted on
XSelectInput(dpy, win, ExposureMask | KeyPressMask |
       PointerMotionMask | PointerMotionHintMask |
       ButtonPressMask | ButtonReleaseMask);

// Add the images to be viewed
ilViewer viewer(dpy, win, render);
```

```
for (idx = 0; idx < nimg; idx++)
      viewer.addView(img[idx], ilLast,
      ilClip|ilCenter|ilDefer);

viewer.setStop(TRUE);

// Execute the UI event loop

XEvent event;
int ever = 1;
for (;ever;) {

  XNextEvent(dpy, &event);
  switch (event.type) {

    case KeyPress:
      switch(XLookupKeysym(&event.xkey, 0)) {
        case XK_Home:
          viewer.display(NULL, ilCenter|ilClip);
          break;

        case XK_Escape:
          ever = 0;
          break;

        case XK_Up:
          viewer.raise();
          break;

        case XK_Down:
          viewer.lower();
          break;

        default:
          break;
      }
      break;

    case DestroyNotify:
      ever = 0;
      break;

    default:
      viewer.event(&event);
      break;
  }

}
```

```
  glx.destroyWindow();
  XCloseDisplay(dpy);
  exit(0);
}
```

Chapter 6

Extending the IL

This chapter explains how you can extend the ImageVision Library by deriving new classes to support capabilities unique to your applications.

Chapter 6

Extending the IL

Since the IL is implemented in C++, you can easily extend it by deriving new classes that provide support for the capabilities you need, for instance, to include another file format or image processing algorithm. You can derive from any C++ class, but you're most likely to want to derive from the foundation classes. Figure 6-1 shows the types of classes you're most likely to derive.

Note: If you're using the C or the Fortran interfaces to the IL, extending the library isn't quite so simple. You have to implement a new class in C++ and then generate a C or Fortran interface for it.

This chapter contains the following major sections:

- "Deriving from ilImage" on page 244 tells you how to derive new classes from ilImage.
- "Deriving from ilCacheImg" on page 255 tells you how to derive new caching classes to manage data.
- "Deriving From ilMemCacheImg" on page 255 tells you how you can derive from ilMemCacheImg to manage images in main memory.
- "Implementing Your Own File Format" on page 258 describes adding classes to support new file formats.
- "Implementing an Image Processing Operator" on page 273 tells you how to define operators that implement new image processing algorithms.
- "Deriving from ilRoi" on page 300 describes how you define new regions of interest in your images.

The IL classes from which you might want to derive you own new classes are shown in Figure 6-1.

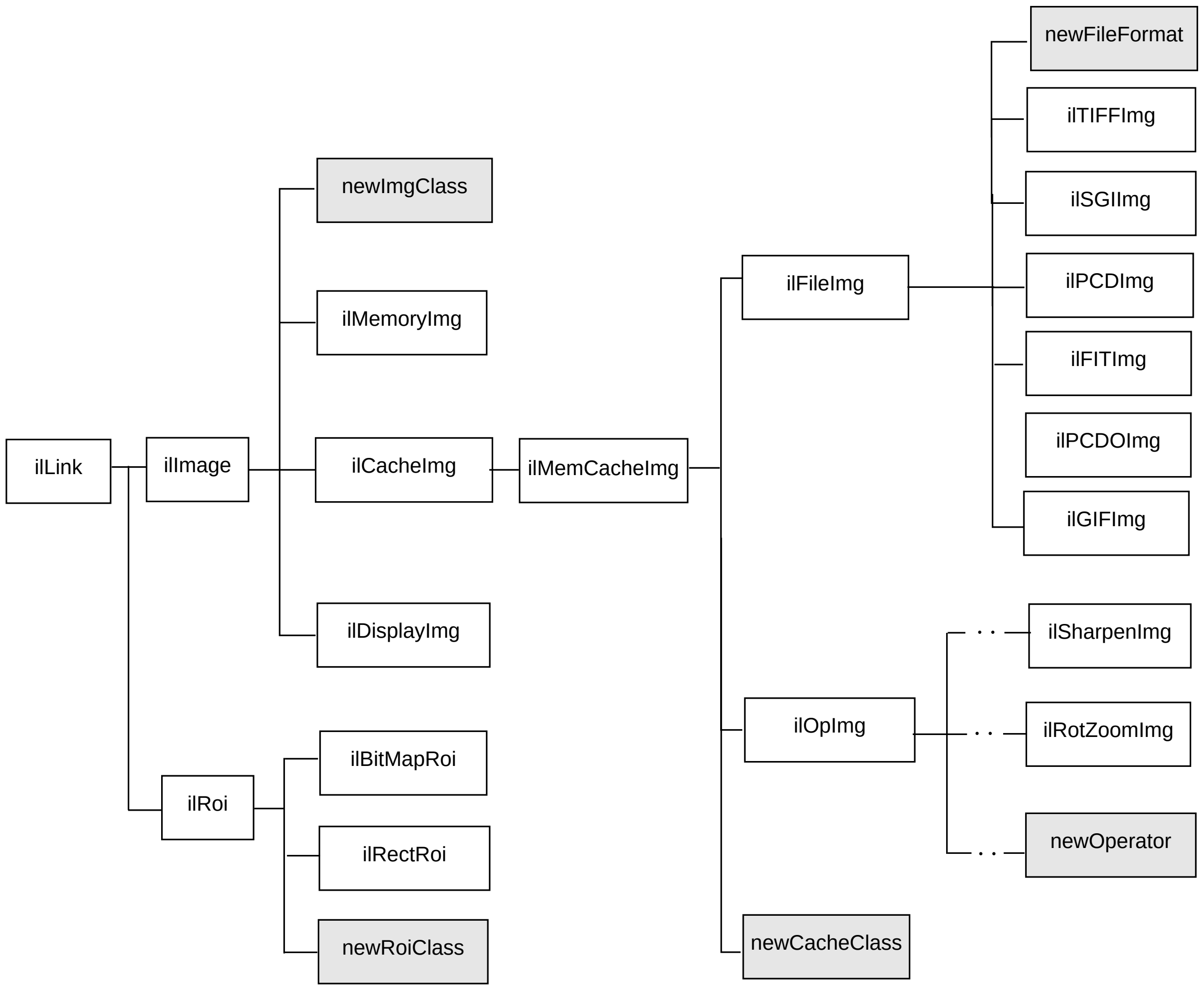


Figure 6-1 User-defined Classes in the IL

Each extension to the IL can be designed to provide a certain set of capabilities and require the implementation of a matching set of functions, as described below:

- newImgClass—A class derived from ilImage inherits all of its functions for handling an image's attributes; it needs to implement ilImage's pure virtual functions for reading and writing data. More information on deriving from ilImage is provided in "Deriving from ilImage" on page 244.
- newCacheClass—A class derived from ilCacheImg inherits its caching mechanism; such a class is useful for managing a large amount of data that's accessed a portion at a time. More information on deriving from ilCacheImg is provided in "Deriving from ilCacheImg" on page 255.
- new ilMemCacheImg class—A class derived from ilMemCacheImg inherits its main memory caching mechanism. Pure virtual functions for storing and retrieving pages of image data must be implemented. More information on deriving from ilMemCacheImg is provided in "Deriving From ilMemCacheImg" on page 255.
- newFileFormat—To add support for your file format, you need to derive from ilFileImg and implement functions that create a new file or open an existing one in the desired format. You must also create functions that read and write data from and to the file. More information on deriving from ilFileImg is provided in "Implementing Your Own File Format" on page 258.
- newOperator—To define a new operator, you need to implement the desired image processing algorithm and ensure that the processed image has the correct attributes. You can derive directly from ilOpImg or from one of its generalized subclasses. See "Implementing an Image Processing Operator" on page 273 for more information.
- newRoi—To define a new ROI, you need to derive from ilRoi and implement functions that describe valid and invalid regions with respect to this new ROI. See "Deriving from ilRoi" on page 300 for more information.

The classes ilImage, ilCacheImg, ilMemCacheImg, ilFileImg, ilOpImg, and ilRoi are abstract classes; they declare pure virtual functions that subclasses need to implement. These virtual functions, and the class constructor and destructor, represent the minimum set that a derived class needs to implement. Other functions can be added as necessary to provide the desired capabilities of the class.

The remaining sections in this chapter explain how to derive from ilImage, ilCacheImg, ilMemCacheImg, ilFileImg, ilOpImg, or ilRoi (or one of their generalized subclasses). Remember that when you derive from a class, you inherit all its public and protected data members and member functions. as well as the public and protected members from its superclasses. You should review beforehand the header files and the reference pages for any class you plan to derive from in order to become familiar with its data members and member functions. Many of the functions described in the following sections are protected, so they're available for use only by derived classes.

Deriving from ilImage

A class derived from ilImage must assign values to the image's attributes and implement ilImage's virtual functions. The image's attributes (data members) are listed in Table 6-1; they're generally initialized in the constructor.

Table 6-1 Image Attributes Needing Initialization in ilImage Subclass

Name	Data Type	Meaning
size	ilSize	size of the image in pixels
dtype	ilType	pixel data type
order	ilOrder	pixel data ordering
cm	ilColorModel	image's color model
space	ilCoordSpace	location of origin and orientation of axes
imtype	ilImageType	type of image (memory, file, display, operator)
fillValue	ilPixel	value used to fill pixels beyond the image's edge
minValue, maxValue	ilPixel	minimum and maximum allowable pixel values
status	ilStatus	image's status (for example, ilOKAY)[a]

a. Inherited from ilLink

Typically, you'll just set these attributes directly. However, there are convenience functions—for setting **minValue**, **maxValue**, **cm**, and **status**—that you might want to use (these functions are protected, so they're available only to classes derived from ilImage):

```
void initMinMax(int force=0);
void initColorModel(int noABGR=0);
ilStatus setStatus(ilStatus val); //inherited from ilLink
void clearStatus();              // inherited from ilLink
```

The **initMinMax()** function simultaneously sets both the minimum and maximum allowable pixel values. They're set to the smallest and largest possible values, respectively, allowed by the image's data type. Therefore, you must set the image's data type before you call **initMinMax()**. By default, this function's argument is 0, which means that the minimum and maximum values won't be changed if they've already been explicitly set; if you pass in 1 as the argument to this function, both values will be set regardless of whether they've been set before.

The **initColorModel()** function sets the color model based on the channel dimension of the image. If the channel dimension is 1, the color model is ilMinBlack; if it's 3, the color model is ilRGB. If the channel dimension is 4 and the default value of 0 is used for the *noABGR* argument, the color model is ilABGR. Otherwise, the color model is ilMultiSpectral.

The **setStatus()** function simply sets and returns the image's status. The **clearStatus()** function sets the image's status to ilOKAY. (Both of these functions are inherited from ilLink.) See "Error Codes" on page 407 for a list of the error codes that the IL defines as being of type ilStatus.

Another function you may want to use in a constructor is **setNumInputs()**. This function sets the maximum possible number of inputs to an image. Typically, you'll use this function only when deriving an operator. See "Implementing an Image Processing Operator" on page 273 for more information about doing this.

Data Access Functions

Most of the functions declared virtual in ilImage are data access functions:

```
virtual ilStatus getSubTile3D(int x, int y, int z,
       int nx, int ny, int nz,
       void* data, int dx, int dy, int dz,
       int dnx, int dny, int dnz,
       const ilConfig* config=NULL) = 0;

virtual ilStatus setSubTile3D(int x, int y, int z,
       int nx, int ny, int nz,
       void* data, int dx, int dy, int dz,
       int dnx, int dny, int dnz,
       const ilConfig* config=NULL) = 0;

virtual ilStatus copyTileCfg(int x, int y, int z,
       int nx, int ny, int nz,
       ilImage* other, int ox, int oy, int oz,
       const ilConfig* config=NULL, int from=1);

virtual ilStatus fillTile3D(int x, int y, int z,
       int nx, int ny, int nz,
       void* data, const ilConfig* config=NULL,
       const ilTile* fillMask=NULL);

virtual ilStatus seekTile3D(int x, int y, int z,
       int nx, int ny, int nz,
       const ilConfig* config=NULL,
       ilSemaphore* sem=NULL);

virtual ilLockedTile* lockTile3D(int x, int y, int z,
       int nx, int ny, int nz,
       const ilConfig* cfg=NULL, int mode=ilLMread);
```

Note that two of these—**getSubTile3D()** and **setSubTile3D()**—are pure virtual functions; in other words, they have no default implementation, so they must be defined by derived classes. The other functions—**copyTileCfg()**, **fillTile3D()**, **seekTile3D()**, and **ilLockedTile()**—have a default implementation that may or may not meet your needs. You can choose whether or not to override these functions. The rest of this section explains how to implement all of these functions.

Implementing getSubTile3D()

You should implement **getSubTile3D()** so that it retrieves an arbitrary tile of data from the source image and puts it into the location indicated by *data*. The tile is located at position (*x*, *y*, *z*) in the source image and has the size indicated by *nx*, *ny*, and *nz*. The *dx*, *dy*, and *dz* parameters specify the data buffer's origin relative to the image; *dnx*, *dny*, and *dnz* specify the buffer's size. The optional *config* argument indicates how the data should be configured in the buffer. See "Three-dimensional Functions" on page 63 for more information about **getSubTile3D()**.

Implementing setSubTile3D()

Your version of the **setSubTile3D()** function should write the tile of data pointed to by *data* into the destination image. The arguments for **setSubTile3D()** have analogous meanings to those for **getSubTile3D()**: (*x*,*y*,*z*) and (*nx*, *ny*, *nz*) indicate the desired origin and size of the tile in the destination image; *dx*, *dy*, and *dz* specify the data buffer's origin relative to the image; and *dnx*, *dny*, and *dnz* specify the size of the data buffer. The optional *config* argument describes the configuration of the tile being passed or written; if it's NULL, assume that the tile's configuration matches that of the destination image. See "Three-dimensional Functions" on page 63 for more information about **setSubTile3D()**.

Implementing Other Data Access Functions

Several other data access functions have default implementations that you may choose to override. These include:

copyTileCfg() The default implementation of **copyTileCfg()** copies a tile of data from one image to another. This implementation isn't as efficient as possible, since it allocates a temporary buffer for holding the data as it performs the copy and then deletes the buffer when it completes the copy; however, this implementation is still more efficient than **getTile()** or **setTile()**. You might want to override this function to provide a more efficient version.

fillTile3D()	The default version of **fillTile3D()** does nothing; you'll need to override it if you want its functionality. Your implementation of **fillTile3D()** should fill a specified tile with the fill value of the image.
seekTile3D()	The default version of **seekTile3D()** does nothing; you'll need to override it if you want its functionality. Your implementation of **seekTile3D()** merely needs to load the specified tile of data into the image's cache.
lockTile3D()	The default implementation of **lockTile3D()** requests locking of the pages spanning the requested tile. The pages are locked as read-only by default.

For more information about these three functions and their arguments, see "Three-dimensional Functions" on page 63.

Support Functions

The **outOfBound()** support functions are provided to help implement the data access functions:

```
int outOfBound(int x, int y);
int outOfBound(int x, int y, int z);
```

These functions return TRUE if the specified point lies outside the image.

You might also want to use the functions that help convert from one coordinate space to another. These functions are discussed in "Coordinate Space Support" on page 66 and described in more detail in the ilImage reference page.

If you implement any of the data access functions, you need to hook them into the reset mechanism, which is described next.

Color Conversion

The **checkColorModel()** function matches the color model of an image with the number of channels. If there is a mismatch, the number of channels is updated to match the color model. However, if the number of channels was set and there is a mismatch, a status of ilBADCOLFMT is set.

```
checkColorModel();
```

The **needColorConv()** function returns TRUE if the image's color model does not match the color model of *other*. The *from* flag indicates the direction that data is copied.

```
needColorConv(ilImage* other, int from, const ilConfig* cfg);
```

The **copyConverted()** function uses **ilCopyTileCfg()** to copy a tile of color-sensitive data from one image to another.

```
ilStatus CopyConverted(int x, int y, int z,
       int nx, int ny, int nz, ilImage *other,
       int ox, int oy, int oz,
       const ilCOnfig* cfg, int from);
```

Managing Image Attributes

An image has numerous attributes associated with it that describe the image. You can change some attributes; some change as a result of being operated on by a display function. This section describes functions you can use to manage attribute values in a class derived from ilImage.

The reset() Function

The only other virtual function in ilImage that you should be concerned with is **reset()**:

```
virtual void reset(); // inherited from ilLink
```

This function is designed to adjust or validate an image's attributes if they've been altered, for example, by applying an operator or by setting an attribute explicitly. This function plays a key role in the IL's execution model, which propagates image attribute values down an operator chain. (See

"Propagating Image Attributes" on page 78 for more information on propagating image attributes.)

The reset mechanism is triggered whenever an image is queried about its attributes or when its data is accessed. The query and access functions all call **resetCheck()** (which is inherited from ilLink) to initiate the reset process. If you implement **getSubTile3D()**, **setSubTile3D()**, **copyTileCfg()**, **fillTile3D()**, **seekTile3D()**, **lockTile3D()**, or any **getAttribute()** function, you need to call **resetCheck()** before you do anything else in your versions of these functions. This ensures that correct information about an image's attributes is returned and that image data is always valid before it's read, written, copied, filled, or updated.

Example 6-1 contains ilLink's implementation of **resetCheck()**. As shown in this example, **resetCheck()** first checks whether any attributes have been altered since the last reset operation.

Example 6-1 ilLink's Implementation of **resetCheck()**

```
ilStatus ilLink::resetCheck()
{
  if (anyAltered()) {                   // is a reset needed?
    mutex.set();                        // reentrant lock
      //Have any attributes been altered?
      if (anyAltered() && !inProgress()) {
      disableAltered(); // prevent recursion during reset
      calcDepth();      // determine depth of this image
      reset();          // do the reset operation
      resetAltered ();  // clear the altered flags

      // notify user of alteration
      if (isResetCallbackEnabled(ilResetCbAll)) {
        if(isResetCallbackEnable(ilResetCbOn|ilResetCbAlter)&&
           neverReset())
           rcbFunc(this, ‘i’, rcbArg);
        else if (isResetCallbackEnabled(ilResetCbAlter))
           rcbFunc(this, ‘a’, rcbArg);
      }

      clearNeverReset;
   }
   mutex.unset();        // reentrant free
 }
  return status;
}
```

The **anyAltered()** function (inherited from ilLink) returns TRUE if any attributes have changed. If any have, a reset operation must be performed. First, the **disableAltered()** function (inherited from ilLink) is called to avoid recursion that otherwise would occur during a reset operation (when attributes are adjusted and checked to see if they've been adjusted).

The **reset()** function must be defined by derived classes to perform any necessary reset tasks. For example, the ilMemCacheImg class's version of **reset()** throws out any existing data in the cache since it's invalid; ilOpImg performs several chores in its **reset()** function and then calls **resetOp()**, which needs to be implemented by derived classes to perform more specific reset tasks. The **resetAltered()** function (inherited from ilLink) clears the flags indicating that attributes have been altered and that a reset is needed.

Allowing Attributes to Change

Not every image attribute can be changed; by default, the fill value and the maximum and minimum pixel values are allowed to change. Each ilImage derived class can choose which attributes it allows to be modified by using the **setAllowed()** function (inherited from ilLink), typically in the constructor:

```
myImg.setAllowed(ilIPcolorModel|ilIPcoordSpace);
```

The argument passed to **setAllowed()** is a mask composed of a logical combination of the enumerated type, ilImgParam, which is defined in the header file *il/ilImage.h*. The ilImgParam constants defined in the IL are listed in Table 6-2. Each image attribute listed in the table is described elsewhere in this guide. Derived classes can add members to this structure to trace whether particular parameter values have changed and to control whether they can be explicitly modified.

Table 6-2 ilImgParam Constants

Defining Class	ilImgParam	Image Attribute
ilImage	ilIPdataType	data type
“	ilIPorder	pixel ordering
“	ilIPpageSize	page size
“	ilIPxsize	x image size

Table 6-2 ilImgParam Constants **(continued)**

Defining Class	ilImgParam	Image Attribute
“	ilIPysize	y image size
“	ilIPchans	number of channels
“	ilIPcoordSpace	coordinate space
“	ilIPcolorModel	color model
“	ilIPminPixel	minimum pixel value
“	ilIPmaxPixel	maximum pixel value
“	ilIPscale	color scaling value
“	ilIPfill	fill value
“	ilIPcompression	compression
“	ilIPcmap	look-up table color map
“	ilIPpageBorder	page border for overlapping pages
“	ilIPdepth	image depth (size of *z* dimension)
ilFileImg	ilFPimageIdx	image index
ilOpImg	ilIPbias	bias value
“	ilIPclamp	clamp value
“	ilIPworkingType	working data type
ilSubImg	ilIPconfig	configuration
ilImgStat	ilISPzBounds	*z* dimension bounds
ilRoi	ilROIcoordspace	coordinate space
ilDisplayImg	ilDPcontext	

Preventing Attributes from Changing

An image can explicitly disallow any of these attributes to be modified. For this, it uses the **clearAllowed()** function (from ilLink) and passes in a logical combination of the ilImgParam parameters that should be disallowed.

Another function, **unalterable()** (inherited from ilLink), checks whether a particular attribute can be modified:

```
canNotChange = myImg.unalterable(ilIPsize);
```

This function takes the same sort of argument as **clearAllowed()** and returns TRUE if the attributes specified aren't allowed to be modified.

Setting Altered and Stuck Flags

When an attribute's value is changed by the user (by calling the appropriate **setAttribute()** function), **setAltered()** (from ilLink) should be called to set a flag indicating that a reset is needed. Thus, you must call **setAltered()** within any **setAttribute()** functions you define. This function takes a mask of ilImgParam parameters as an argument and sets the altered flags for the specified attributes.

You can check whether any particular attributes have been altered with **isAltered()** (inherited from ilLink). This function takes an ilImgParam mask as an argument and returns TRUE if any of the specified attributes have been altered.

As explained in "Propagating Image Attributes" on page 78, IL programs need to keep track of attributes that have been explicitly set by the user so that they remain fixed during the reset process. To keep track of these attributes, you should call **markSet()** (inherited from ilLink) with an ilImgParam mask as an argument. This function marks the specified attributes with a *stuck* flag (yet another item inherited from the ilLink class), which indicates that their values shouldn't be changed during a reset operation. **markSet()** is invoked automatically for you when **setAltered()** is called, so generally you don't need to call **markSet()** yourself.

You can determine whether any attributes are fixed with **isSet()** (inherited from ilLink). This function returns TRUE if any of the attributes specified in the mask passed in have been explicitly set.

Setting Attributes Directly

Sometimes within a derived class's implementation, you may want to change an attribute's value without triggering the reset mechanism and without causing the value to become fixed. You've already seen one situation where you want to do this: within a constructor, when attributes are being initialized. Another case is when you're computing attribute values during the reset operation itself. In these situations, you don't use a **setAttribute()** function since it calls **setAltered()**, which in turn calls **markSet()**. Since derived classes have access to protected data members, simply set the value of the desired attribute directly:

```
dtype = ilFloat;     // changes value; no flag set
```

You should bypass the reset mechanism in this way only when necessary, since doing so prevents attribute values from being automatically propagated down the operator chain. In particular, since the IL has a parallel implementation, it should never be done anywhere that code is required to be reentrant (such as within the implementations of any tile access functions like **getPage()** or **setPage()**). The **initMinMax()**, **initColorModel()**, and **setStatus()** functions described earlier in this section all set attributes directly.

Adding New Attributes

It is quite easy to add attributes to a newly derived class. You can use the header files for the already existing IL classes for examples. Here's an example from the *il/ilOpImg.h* header file:

```
enum ilOpImgParam {
  ilIPbias = ilImgParamLast<<1,
  ilIPclamp = ilImgParamLast<<2,
  ilIPworkingType = ilImgParamLast<<3,
  ilOpImgParamLast = ilIPworkingType
};
```

The pattern is simple. Suppose you were to derive a new class from ilOpImg and add parameters to it. You might do the following:

```
enum ilMyClassParam {
  ilIPparam1 = ilOpImgParamLast<<1,
  ilIPparam2 = ilOpImgParamLast<<2,

  ilIPparam5 = ilOpImgParamLast<<5,
  ilMyClassParamLast = ilIPparam5
};
```

Deriving from ilCacheImg

The ilCacheImg class implements an abstract model of cached image data. The main purpose of this class is the definition of a common API for cached image objects. You can implement your own caching mechanism by deriving from ilCacheImg. The ilMemCacheImg class, derived from ilCacheImg, provides an example of the implementation of a caching mechanism.

If you derive from ilCacheImg, you must implement the **getTile()**, **setTile()**, **copyTile(),** and **copyTileCfg()** methods to retrieve data. You must also implement the **flush()**, **getCacheSize()**, and **listResident()** functions if you derive from ilCacheImg.

The **flush()** function causes any modified data in the cache to be written out. Derived classes that access an image file can call this function in their destructor before they close the file to ensure that all data is written.

```
virtual ilstatus flush(int discard=FALSE);
```

The **getCacheSize()** function returns the amount of cache memory, in bytes, currently allocated by this image object.

```
virtual int getCacheSize();
```

The **listResident()** function returns a list of all the resident pages. The page callback specified by *func* is called once for each resident page of the image.

```
virtual ilStatus listResident(ilCacheImgPagingCB* func,
              void* arg=NULL);.
```

Deriving From ilMemCacheImg

The ilMemCacheImg class implements a caching mechanism for efficiently manipulating image data in main memory. In managing the interface to an image's cache, ilMemCacheImg implements several ilImage virtual functions —**getSubTile3D()**, **setSubTile3D()**, **copyTileCfg()**, **fillTile3D()**, and **seekTile3D()**. The ilMemCacheImg class also implements the virtual functions **hasPages()**, **lockPage()**, and **unlockPage()** defined in ilImage. **hasPages()** should return TRUE only for classes that implement the IL's paging mechanism (ilMemCacheImg does). **lockPage()** returns a pointer to the page containing the specified pixel. **unlockPage()** unlocks the specified page.

Classes that derive from ilMemCacheImg don't need to implement these functions; instead, they need to implement ilMemCacheImg's pure virtual functions:

```
virtual ilStatus getPage(void* data, ilPgCB& cb) = 0;
virtual ilStatus setPage(void* data, ilPgCB& cb) = 0;
```

getPage()	is called when a page of image data needs to be retrieved from the image and put into the cache. The *data* argument is a pointer to a page-sized buffer into which data will be read.
setPage()	is called when the cache is full and a page needs to be written back to the image. The *data* argument is a pointer to a page-sized buffer that contains data that will be written to disk.

The ilPgCB **struct** (defined in the header file *il/ilMemCacheImg.h*) is a control block that defines the location within the buffer and the amount of data to be read or written:

```
struct ilPgCB {
// C++ constructor:
  ilPgCB(ilImage* img, int x, int y, int z, int c);

  int x, y, z, c;        // origin of page
  int nx, ny, nz, nc;    // size of page
};
```

Since an image's size isn't generally an exact multiple of the page size, you're likely to encounter pages that are only partially full of data. The *nx*, *ny*, *nz*, and *nc* parameters define the actual limits of the data that you need to read or write within a given page buffer. You might want to use the **getStrides3D()** function to help you step through a page buffer. See "Data Access Support Functions" on page 65 for more information about **getStrides3D()**.

Table 6-3 lists additional attributes you might need to initialize for a class derived from ilMemCacheImg.

Table 6-3 Additional Attributes Needing Initialization in ilMemCacheImg Derived Classes

Name	Data Type	Meaning
pageSize	int	size of a page in bytes
xPageSize, yPageSize, zPageSize, cPageSize	int	pixel dimensions of the pages used to store data on disk
xPageBorder, yPageBorder, zPageBorder	int	pixel dimensions of page borders as stored on disk (default is zero)

If you derive from ilMemCacheImg, you must implement the virtual function **listResident()**. The callback function specified in *func* is invoked once for each page resident in memory. The callback function should have prototype as defined in **setPagingCallback()**.

```
virtual ilStatus listResident(ilCacheImgPagingCB* func,
        void* arg=NULL);
```

You can also implement the **allocPage()** and **freePage()** functions. These functions allocate or free a page in main memory whose pixel includes (*x*,*y*,*z*,*c*). If you implement the function **allocPage()**, you must also call the function **doUserPageAlloc()** in the function that calls **allocPage()** to notify the IL that the pages need to be defined.

The **flush()** function (defined by ilMemCacheImg) flushes data from an image's cache; it calls **setPage()** to ensure that the data is written to the proper place:

```
virtual ilStatus flush(int discard=FALSE);
```

This function takes one optional argument and returns an ilStatus to indicate whether the flush was successful. Calling **flush()** with a TRUE argument will discard all data in the cache. This is useful for freeing up memory if you know you're never going to use the cached data again. When discard is FALSE, **flush()** writes any modified data from the cache to the image. The destructor for any class derived from ilMemCacheImg must call ilMemCacheImg's **flush()** (with *discard* equal to FALSE) before the class object is deleted to ensure that any modified data is written back to the image.

For more information about deriving from either of ilMemCacheImg's derived classes (ilFileImg and ilOpImg), see "Implementing Your Own File Format" below and "Implementing an Image Processing Operator" on page 273.

Implementing Your Own File Format

The IL is designed so that you can easily extend it to support any particular file format. In fact, the FIT format supplied with the IL was originally designed in part to exercise the robustness of extending the IL's file input and output abilities; it's used as an example in the paragraphs that follow. The code for the FIT format is also available online in:

```
/usr/people/4Dgifts/examples/ImageVision/ilsrc/ilFITImg.c++
```

The online code may differ slightly from the excerpts shown in the following paragraphs, but the functionality is the same.

To implement IL support for your file format, you need to derive a new class from ilFileImg and provide versions of functions that:

- create a new file or open an existing one
- read data from a file into the cache, one page at a time, decompressing it if necessary
- write data from the cache into a file, one page at a time, compressing it if necessary
- close a file
- allow your format to be registered

Typically, the constructors for the class you derive from ilFileImg create and open files, and **getPage()** and **setPage()** read and write data, respectively. These functions (and a class destructor) are the bare minimum that you need to override in order to derive from ilFileImg. You might also want to provide your own version of **reset()** (declared in ilLink and ilImage) so that altered parameters are handled properly. In addition, you may want to implement other functions of your own design to provide more capabilities. The rest of this section describes in detail the tasks that the minimal set of functions needs to perform.

Creating and Opening a File

Most image data file formats consist of a file header and the data itself. The file header specifies such information as the size of the image, the data type of the image data, and the color model that should be used to interpret the data. When you open an existing file for reading, the file header is generally read first so that the characteristics of the data are known. When you create a new file and write data into it, the header needs to be written before you close the file.

Since existing and new files need to be treated slightly differently, it makes sense to have two corresponding types of functions, typically constructors, in a class derived from ilFileImg. The ilFITImg class, which can serve as a model for this discussion, provides two constructors for existing files and one for creating new image files.

Opening an Existing File

The two constructors in the ilFITImg class that open an existing file differ in how they allow the file to be specified; one takes a filename as an argument, and the other takes a file descriptor. The constructor shown in Example 6-2 demonstrates how, given a filename and a mode (indicating whether the file is readable or writable), the corresponding file might be opened. The **init()** function, which performs important initialization tasks such as reading the file's header, is described in the paragraphs that follow Example 6-2.

Example 6-2 ilFITImg Constructor to Open an Existing File

```
ilFITImg::ilFITImg(const char* name, const char* mode)
{
   // decipher mode
   fmode = mode[0] == 'r' && mode[1] != '+'? O_RDONLY:O_RDWR;

  // open the file
    if ((fd = open(name, fmode)) == -1)
     { setStatus(ilBADFILEOPEN); return; }
             fname = strdup(name);

  // do common initializations
   init();
}
```

In addition to opening the file, the constructor needs to read the header information and initialize the values of several variables related to the image data. These variables need to be initialized so that query functions inherited from ilFileImg and ilImage return the proper values. (ilFileImg inherits from ilMemCacheImg, but ilMemCacheImg doesn't define any variables or query functions that require initialization.)

Table 6-4 lists the variables that need to be initialized (in addition to those listed in Table 6-1 and Table 6-3). Some variables may not need to be changed, depending on the requirements of the particular file format and on what default value they're given. For example, formats that don't provide compression don't need to set the *compress* variable since by default it's ilNoCompression; similarly, *imgidx* doesn't need to be changed if the default first image in the file (0) is appropriate.

Table 6-4 Additional Image Attributes Needing Initialization during File Opening

Name	Data Type	Declared In	Comments
fname	char*	ilFileImg	filename
fmode	int	“	indicates read-only or read-write
fd	int	“	file descriptor, if available
format	char*	“	file format (for example, “FIT”)
nimgs	int	“	number of images in file; default is 1
imgidx	int	“	initialize image index; default is 0
compress	ilCompress	ilImage	data compression, if supported; default is ilNoCompression
allowed	int	ilLink	defines which image parameters can be changed; default is ilIPfill, ilIPminPixel, and ilIPmaxPixel
status	ilStatus	“	status; default is ilOKAY

Values for some of these variables are typically stored in a file’s header and can therefore be set by reading the header. The remaining variables need to be set explicitly. Example 6-3 shows how a header for a file in the ilFIT format might be read and how all the needed variables might be initialized.

Example 6-3 Reading a FIT Header File and Initializing Variables

```
struct FIThead {
  unsigned short magic;      // file identifier
  unsigned short version;    // file version
  unsigned int xSize;        // image size
  unsigned int ySize;
  unsigned int zSize;
  unsigned int cSize;
  int dtype;                 // data type
  int order;                 // RGBRGB.. or RR..GG..BB..
  int space;                 // coordinate space
  int cm;                    // color model
  unsigned int xPageSize;    // page size
  unsigned int yPageSize;
  unsigned int zPageSize;
  unsigned int cPageSize;
  double minValue;           // min/max pixel values
  double maxValue;
  unsigned dataOffset;       // offset to first page of data
// user extensible area...
};

void ilFITImg::init()
{

  needHeader = 0;   // must be initialized for destructor
  dataWritten = 1;  // so extensions can't be reserved

   // read the header
   if (readHeader() != ilOKAY) return;

   // fill in other info
   format = "FIT";
   calcPageParams();

   setStatus(ilOKAY);
}

void ilFITImg::calcPageParams()
{

   pageCount.x = (size.x-1)/xPageSize + 1;
   pageCount.y = (size.y-1)/yPageSize + 1;
   pageCount.z = (size.z-1)/zPageSize + 1;
   pageSize = ilDataSize(dtype,
             xPageSize*yPageSize*zPageSize*cPageSize);
}

ilStatus ilFITImg::readHeader()
{
    FIThead head;

   lseek (fd, 0, SEEK_SET);

   if (read(fd, &head, sizeof head) == -1)
      return setStatus(ilBADFILEREAD);
   if (head.magic != ilFITmagic)
      return setStatus(ilBADMAGIC);
```

```
// extract the image attributes from the file header
   if (head.version == '02') {
      size.x = head.xSize;
      size.y = head.ySize;
      size.z = head.zSize;
      size.c = head.cSize;
      dtype = ilType(head.dtype);
      order = ilOrder(head.order);
      space = ilCoordSpace(head.space);
      cm = ilColorModel(head.cm);
      xPageSize = head.xPageSize;
      yPageSize = head.yPageSize;
      zPageSize = head.zPageSize;
      cPageSize = head.cPageSize;
      setMinValue(head.minValue);
      setMaxValue(head.maxValue);
      dataOffset = head.dataOffset;
      userOffset = sizeof(FIThead);
   }
   else if (head.version == '01') {
      // similar code omitted for older version
   }
   else
       return setStatus(ilBADIMGFMT);
   return setStatus(ilOKAY);
}
```

ilFITImg's initializing method **init()** sets the needHeader flag to indicate that the header doesn't need to be written out when the file is closed (since the file is just being read). The flag is set before reading the header to avoid problems in the destructor if there is a failure reading the header. Another flag, dataWritten, is also set to indicate that the extension area can't be written (it has to be written before any image data).

The **init()** method then calls **readHeader()** to read all the header data. Notice that a special **struct** is defined for the header to make the code more legible. The header is read into this **struct** using the UNIX® system calls **lseek()** and **read()**. Once the header information is obtained, the variables can be initialized.

For most file formats, the values read from the header would need to be translated to match the enumerated values used by IL; since the FIT format was developed for IL, these values are conveniently matched. Your code will almost certainly be more involved.

After the values are filled in, **readHeader()** returns to **init()**, where the page dimensions are set with **calcPageParams()**. Note that FIT supports paging in the channel dimension, with **cPageSize**. This is an unusual feature that's useful for multispectral data.

Creating a New Image File

When you create a new image file, you must supply much of the same information as when you open an existing file. The constructor that creates a new file of image data typically takes several arguments that describe the data that will be written into the file. Example 6-4 shows the constructor that creates a new file for data in the FIT format.

Example 6-4 Creating a File for FIT Data

```
ilFITImg::ilFITImg(const char* name, const ilSize& sz,
                  ilType type, ilOrder o,
                  int xpgsz, int ypgsz, int zpgsz,
                  int cpgsz)
{
    // create the file
    if ((fd = open(name, O_RDWR|O_CREAT|O_TRUNC)) == -1)
        { setStatus(ilBADFILEOPEN); return; }

    // identify what attributes are settable
    setAllowed(ilIPcolorModel|ilIPcoordSpace|ilIPdepth);

    // compute defaults
    if (cpgsz == 0) cpgsz = o==ilSeparate? 1:sz.c;

    // fill in info
    fname = strdup(name);
    fmode = O_RDWR;
    format = “FIT”;
    size = sz;
    dtype = type;
    order = o;
    space = ilLowerLeftOrigin;
    initColorModel();
    initMinMax();
    xPageSize = xpgsz;
    yPageSize = ypgsz;
    zPageSize = zpgsz;
    cPageSize = cpgsz;
    dataOffset = userOffset = sizeof(FIThead);

    // flag the header/data as not written
    needHeader = 1;  // header needs to be written
    dataWritten = 0; // still ok to reserve an extension
    calcPageParams();
    setStatus(ilOKAY);
}
```

The first four arguments specify the file name, the size of the image being written into the file, the data type, and the pixel ordering. The remaining arguments specify the dimensions of the pages as they're stored on disk. Although it's not explicitly shown here, these remaining arguments are given default values in the *il/ilFITImg.h* header file, so they don't need to be specified when the constructor is called.

The first task of this constructor is to create the image data file using the UNIX system call **open()**. Next, those image attributes that a programmer is allowed to change—color model and coordinate orientation—are specified. After that, image attributes are initialized. Note that a flag (needHeader) is set indicating that the header needs to be written out before the file is closed, as discussed in the next section. Another flag, dataWritten, is set to indicate that the user can reserve an extension area before image data is written.

If you allow any attributes to be modified, you need to provide a version of **reset()** (inherited from ilImage) that checks for such modification. As shown in Example 6-5, the ilFITImg version of **reset()** uses the keepCacheData flag (inherited from ilMemCacheImg) to prevent the cache data from being discarded when the file attributes such as depth (*z* size) are changed. If any attributes have been altered**, reset()** sets the flag needHeader to indicate that the header should be written. It then calls ilFileImg's version of **reset()**, which updates any other attributes that have changed.

Example 6-5 An Implementation of **reset()** for ilFITImg

```
void ilFITImg::reset()
{
    // don't discard any cached data
    keepCacheData = TRUE;

    // flag the header to be updated if params have been
    // altered
    if (isAltered(ilIPcolorModel|ilIPcoordSpace|ilIPdepth))
        needHeader = 1;
    ilFileImg::reset();
}
```

The flag needHeader is checked in ilFileImg's destructor, as discussed below. The IL's reset mechanism is discussed in detail in "Deriving from ilImage" on page 244. If there are multiple images in a file, you need to make sure the proper image is being read or written; this might involve seeking to the image's index.

Closing a File

In addition to writing constructors for a class derived from ilFileImg, you need to write a destructor. This destructor needs to:

- write out the header if necessary (generally, if the file is newly created or if any attributes were modified)
- finish writing out any modified pages of image data to disk
- close the file and release the file descriptor
- free any temporary buffers that were allocated

The destructor shown in Example 6-6 is an example of how the ilFITImg destructor might perform these tasks. It uses the same FIThead **struct** defined in Example 6-3.

Example 6-6 Destructor for ilFITImg

```
ilFITImg::~ilFITImg()
{
    if (fname != NULL) free(fname);
    if (fd != -1) {
        flush();
        close(fd);
    }
}

ilStatus ilFITImg::flush(int discard)
{
    // update the header if necessary
    if (!discard && needHeader) {
        setStatus(writeHeader());
        if (status != ilOKAY) return status;
        needHeader = 0;
    }
    return setStatus(ilMemCacheImg::flush(discard));
}

ilStatus ilFITImg::writeHeader()
{
    FIThead head;

    // fill in the file header
    head.magic = ilFITmagic;
    head.version = ilFITversion;
    head.xSize = size.x;
    head.ySize = size.y;
    head.zSize = size.z;
    head.cSize = size.c;
    head.dtype = dtype;
    head.order = order;
    head.space = space;
    head.cm = cm;
    head.xPageSize = xPageSize;
    head.yPageSize = yPageSize;
    head.zPageSize = zPageSize;
    head.cPageSize = cPageSize;
    head.minValue = getMinValue();
    head.maxValue = getMaxValue();
    head.dataOffset = dataOffset;

    lseek (fd, 0, SEEK_SET);
    if (write(fd, &head, sizeof head) == -1)
        return setStatus(ilBADFILEWRITE);

    return ilOKAY;
}
```

In this example, the destructor calls ilFITImg’s own version of **flush()**, which updates the header with **writeHeader()**. The header information is first written into the FIThead **struct** and then into the file using the UNIX calls **lseek()** and **write()**. After that, ilFITImg’s **flush()** calls the **flush()** member function inherited from ilMemCacheImg to write any pages that have been modified out to disk. Storage associated with the filename and descriptor is released through the UNIX calls **free()** and **close()**.

Reading and Writing Formatted Data

In addition to providing functions that open and close files, you need to override **getPage()** and **setPage()** to read and write image data in your particular file format. These two functions are declared as protected, pure virtual member functions in ilFileImg:

```
virtual ilStatus getPage(void* data, ilPgCB& cb) = 0;
virtual ilStatus setPage(void* data, ilPgCB& cb) = 0;
```

The *data* argument for **getPage()** is a pointer to an already allocated, page-sized buffer into which data is read. This buffer must be large enough to hold a page of data. For **setPage()**, *data* is a pointer to a page-sized chunk of memory that contains data that will be written to disk. The ilPgCB **struct** is discussed in “Deriving From ilMemCacheImg” on page 255. The ilFITImg versions of **getPage()** and **setPage()** use the information in the ilPgCB **struct** to determine the offset to pass to the UNIX system call **lseek()**. Then, the actual reading or writing is performed using the system calls **read()** and **write()**. Example 6-7 shows the implementation of **getPage()** and **setPage()** for ilFITImg.

Example 6-7 Reading and Writing Data in the FIT format

```
ilStatus ilFITImg::getPage(void* data, ilPgCB& cb)
{
    beginFileIO();
    lseek (fd, pageOffset(cb.x,cb.y,cb.z,cb.c), SEEK_SET);
    int sts = read(fd, data, pageSize);
    endFileIO();
    return setStatus(sts == -1? ilBADFILEREAD:ilOKAY);
}
ilStatus ilFITImg::setPage(void* data, ilPgCB& cb)
{
    beginFileIO();
    lseek (fd, pageOffset(cb.x,cb.y,cb.z,cb.c), SEEK_SET);
    int sts = write(fd, data, pageSize);
    endFileIO();

    dataWritten = 1;
    return setStatus(sts == -1? ilBADFILEWRITE:ilOKAY);
}
```

Both **getPage()** and **setPage()** must be reentrant so that IL programs can be multi-threaded. In this example, the functions **beginFileIO()** and **endFileIO()** are used to “lock” file access during the seeks, reads, and writes. These two protected member functions are inherited from ilFileImg to ensure that reads and writes can be made safely in a multi-threaded environment.

A file image’s cache holds uncompressed data. Therefore, if the particular file format being used supports compression, **getPage()** needs to decompress the data before putting it into the cache. Similarly, **setPage()** must compress data before it writes it to disk.

Registering Your File Format

The IL provides an optional mechanism for registering your file format so that other programmers can manipulate files of image data in that format. The FIT, TIFF, SGI, and Photo CD file formats are all registered this way. Registration also allows you to write file format-independent programs using the convenience functions **ilCreateImgFile()** and **ilOpenImgFile()**. These two functions are declared in the header file *il/ilGenericImgFile.h* and discussed in more detail in their own reference pages and in “Opening an Existing File” on page 92 and “Creating a TIFF, SGI, or FIT File” on page 94. As their names suggest, these functions create new files or open existing ones.

Deriving from ilFileFormat

To register your file format, you must first create a class derived from the ilFileFormat class. The example shown below is for the FIT format; an online version is also available in:

/usr/people/4Dgifts/examples/ImageVision/ilsrc/ilFITformat.c++

The example starts with the ilDeclareFileFormat macro to declare ilFITformat as a derivative of ilFileFormat:

```
ilDeclareFileFormat(ilFITformat);
```

The next line defines the constructor for the ilFITformat class:

```
ilFITformat::ilFITformat() : ilFileFormat(“FIT”, “fit”) {}
```

The arguments passed to the base class constructor define the format name to be “FIT”, and the standard file extension for this format to be “.fit”. The format name is used with **ilCreateImgFile()** to determine what file format is being created. The file extension is used to guide the IL in choosing which of the registered formats to try first when opening a file of unknown type with **ilOpenImgFile()**. If that format fails, or the extension doesn’t match any of the registered file formats, then all of the formats will be tried in succession, in an attempt to open the file. Additional extensions can be registered in the body of the constructor using the **addExtension()** method:

```
void addExtension(const char* formatExt);
```

Example 6-8 continues with the definition of the **openFile()** virtual method.

Example 6-8 Implementation of **OpenFile()**

```
ilFileImg*
ilFITformat::openFile(const char* name, const char* mode,
               const ilSize* size, ilType type,
               ilOrder ord, const ilSize* pgSize)
{
    ilFITImg *img;

   if (size==NULL)
        // try opening an existing file
        img = new ilFITImg(name, mode);
    else {

        // create a new file
        if (pgSize == NULL)
               img = new ilFITImg(name, *size, type, ord);
        else
               img = new ilFITImg(name, *size, type, ord,
               pgSize->x, pgSize->y, pgSize->z, pgSize->c);
    }

    // if anything went wrong then delete the object
    if (img != NULL && img->getStatus() != ilOKAY)
            { delete img; return NULL; }
    return img;
}
```

This function is used by the IL to either open or create a file of the FIT format. As you can see, the **openFile()** function is mostly a wrapper for the constructors defined in the ilFITImg class. The *size* parameter distinguishes opening from creation. When NULL is passed, the file is opened; otherwise the file of the indicated size is to be created.

Creating a Dynamic Shared Object

The simplest way to actually register your file format with IL is to construct an object of the class you have derived from ilFileFormat. The constructor for the ilFileFormat base class performs all the required bookkeeping. However, the preferred method of registering with IL is to create a dynamic shared object (DSO) that allows programs that use IL to recognize your new file format, without any need to relink those programs.

The DSO contains the code for both your file format and the registration object. In the case of FIT, this is the code for ilFITformat and ilFITImg. In the DSO you must also define a global function, **ilNewFileFormat()**, with the C linkage. This function is defined in the example FIT source as:

```
void ilNewFileFormat() { new ilFITformat(); }
```

At run time, the IL searches the directory */usr/lib/ImageVision/filefmts* for DSOs that define file formats. It looks up the symbol **ilNewFileFormat()** in each DSO and calls it to register the file format(s) defined in that DSO. The IL can be told to look in other directories by setting the IL_FILE_FORMAT_PATH environment variable to a colon (:) separated list of directories to be searched. Be sure to include the default directory mentioned above in this list if you want IL to continue to recognize the standard file formats.

To actually create the DSO for the FIT format, you would issue a command of the form:

```
ld -shared -all ilFITformat.o ilFITImg.o -lil -o libilFIT.so
```

This loads the two object files for the FIT example source into a DSO named libilFIT.so. The name of the DSO is not important, but the DSO must be placed in a directory that IL will search, as described above. Refer to the **ld()** reference page for more details on creating DSOs.

Format-independent File Access

Once **ilNewFileFormat()** is called for each of the file formats to be recognized (establishing a means of opening and creating files), programmers can call **ilCreateImgFile()** and **ilOpenImgFile()** to create and open files regardless of which file format is being used. As shown below, **ilOpenImgFile()** takes two arguments, a filename and a string indicating whether the file is to be opened for reading or for writing:

```
ilFileImg* ilOpenImgFile(const char* name, const char* mode);
```

This function extracts the extension from the filename and uses it to try to determine the file format to use when opening the file. If this fails, it iterates through the known formats until a valid ilFileImg pointer is returned or the list is exhausted. It tries to open the desired file by passing the filename and mode to each **openFile()** function. Note that these arguments match those accepted by the ilFITImg constructors.

The **ilCreateImgFile()** function takes several arguments that describe the data to be written into the file:

```
ilFileImg* ilCreateImgFile(const char* name,
    const ilSize& size, ilType type, ilOrder order,
    char* format=NULL, const ilSize* pageSize=NULL);
```

This function uses the *format* argument to find the correct **openFile()** to call. If the format is NULL, it uses the file extension to determine the appropriate file format. If there is no match on extension, then it uses the default (TIFF) format. All the information describing the data to be written is passed to the routine so that the file can be created with the proper attributes.

Consult the reference pages for **ilOpenImgFile()** and **ilCreateImgFile()**, and the ilFileFormat class for more details on this topic.

Implementing an Image Processing Operator

The IL is designed to be easily extendable in C++ to include image processing algorithms you implement. You can derive a new operator directly from ilOpImg, or you can take advantage of the support provided by its subclasses, some of which are specifically designed to be derived from. This section explains in detail how to derive your own operator; it contains these sections:

- “Deriving from ilOpImg” on page 275
- “Deriving from ilMonadicImg or ilDyadicImg” on page 285
- “Deriving from ilSpatialImg” on page 290
- “Deriving from ilWarpImg or ilPolyWarpImg” on page 294
- “Deriving from ilFMonadicImg or ilFDyadicImg” on page 295
- “Deriving from ilFFiltImg” on page 299

The subclasses of ilOpImg handle the tasks of reading raw data from the cache and writing processed data back to the cache; if you derive from these classes, you’re responsible for writing only the function that processes the data in a given input buffer and writes it to a given output buffer. If you derive directly from ilOpImg, you need to supply your own interface to the cache as well as your processing algorithm. Figure 6-2 shows the operator classes you're most likely to derive from.

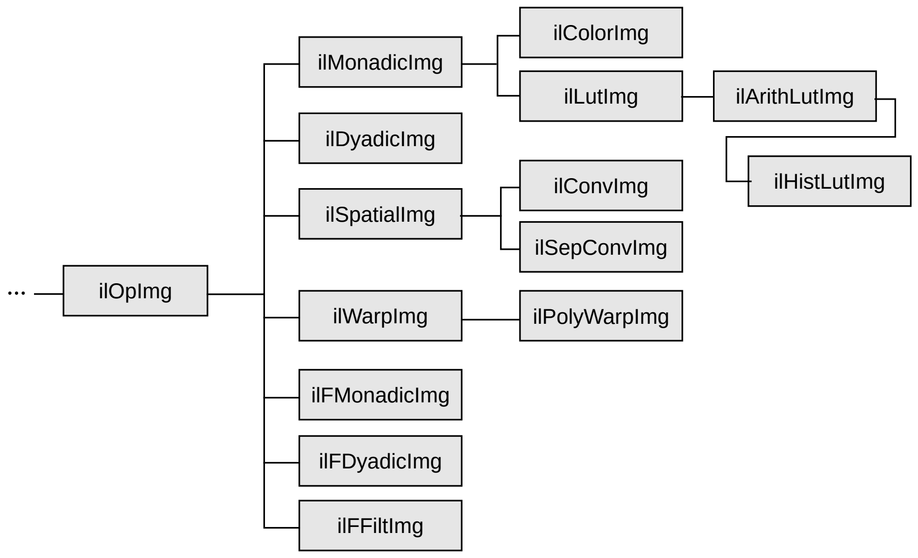


Figure 6-2 ilOpImg and Its Subclasses for Deriving

Remember that when you derive from a class, you inherit all of its public and protected data members and member functions. You also inherit members from its superclasses. You should review the header file and the reference page for any class you plan to derive from (as well as the header file and reference pages of its superclasses) to become familiar with its data members and member functions. It's also a good idea to look at a few of its subclasses to see what general tasks they perform and what functions they implement. Finally, you might want to take a look at the selected IL source code that's provided online in:

```
/usr/people/4Dgifts/examples/ImageVision/ilsrc
```

The next section contains information that's useful whether you derive directly from ilOpImg or from one of its subclasses. The sections that follow contain more detailed information about deriving from each of ilOpImg's subclasses shown in Figure 6-2.

Deriving from ilOpImg

A class derived from ilOpImg needs to implement these member functions:

- the constructor, which creates the object, declares which data types and pixel orders are valid for the output, and sets the working data type
- **getPage()**, which reads data from the cache into an internal buffer, processes it, and writes the processed data back to the cache (often a separate function is defined and called from **getPage()** to process the data)
- **resetOp()**, which sets any image parameters that have been altered as a result of processing the data
- any public **setParam()** and **getParam()** functions provided to control the operator's algorithm

You also need to implement a *destructor* if you allocate any memory or change state within the constructor or any other function you implement. Note that you don't implement **setPage()** since it's not available for use by operators. This is because operators by definition compute new data and thus are read-only. Example 6-9 shows a typical header file for an ilOpImg subclass.

Example 6-9 Typical Header for a Class Derived from ilOpImg

```
#include <il/ilOpImg.h>

class myOperator : public ilOpImg {
private:
    float param1;
    void resetOp();
    ilStatus getPage(void* data, ilPgCB& cb);

public:
    myOperator(ilImage* img, float param1);
    void setParam1(float val)
               { param1 = val; setAltered(); }
    float getParam1()
               { resetCheck(); return param1; }
};
```

The **resetOp()** function can be declared protected if other programmers are likely to want to derive a class from the myOperator class.

The Constructor

The constructor takes a pointer to the source ilImage(s) and additional arguments as needed to provide parameters to control the operator's processing algorithm (for example, *param1*). If you do use additional parameters, you might want to define corresponding functions that allow the user to alter and retrieve the value of those parameters (such as **setParam1()** and **getParam1()**). These functions should probably take advantage of the IL's reset mechanism by calling **setAltered()** and **resetCheck()**, respectively. (See "The reset() Function" on page 249 for more information about how the IL's reset mechanism works.) Example 6-10 shows you what a simple constructor might look like.

Example 6-10 Typical Constructor for a Class Derived from ilOpImg

```
myOperator::myOperator(ilImage* img, float param1)
{
    setValidType(ilFloat|ilDouble);
    setValidOrder(ilInterleaved|ilSequential|ilSeparate);
    setWorkingType(ilDouble);
    setNumInputs(1);
    setInput(img);
    setParam1(param1);
}
```

In this example, myOperator can produce output of either ilFloat or ilDouble data type; the output will have the same pixel ordering as the input image. Input image data that's of type ilFloat will be cast to ilDouble before it's processed; this is the meaning of an operator's *working type*. Some operators can handle multiple inputs, but the **setNumInputs()** function is used here to limit myOperator to one input. The **setInput()** function sets the input to be the ilImage passed in; this step chains myOperator to the input image. Finally, *param1*'s value is initialized.

The **setValidType()**, **setValidOrder()**, and **setWorkingType()** functions are all defined as protected in ilOpImg. They're discussed in more detail in ilOpImg's reference page. The ilImage class defines **setNumInputs()** (protected) and **setInput()**.

The constructor shouldn't contain any calculations that are based on the value of arguments passed in, since these arguments might change. Most operators that require arguments other than the input image in their constructors define functions for dynamically changing the value of those arguments (like **setParam1()**). Such calculations should be done in the **resetOp()** function described below. The **resetOp()** function is declared in ilOpImg, but its implementation is left to derived operators. Note that when any ilImage is created, it's considered "altered," so **resetOp()** will always be called before any data is read (with **getPage()**).

The resetOp() Function

Since **resetOp()** is guaranteed to be called before **getPage()**, it can—and should—be used to calculate the values of variables needed by **getPage()**, particularly if those variables depend on arguments passed in the operator's constructor. The **resetOp()** function also needs to reset any image attributes that change as a result of the image's data being processed, so that the proper attribute values can be propagated down an operator chain. As an example, imagine an operator that defined the following variables (probably as protected) in its header file (ilMonadicImg defines these variables):

```
ilXYZCint str;         // output (page) buffer strides
ilXYZCint istr;        // input image strides

int bufferSize;        // size of input buffer in bytes
int cBuffSize;         // number of channels in input buffer
```

As you might expect, these variables are used to determine the size of the internal buffer needed for reading in the image's data that's to be processed. This buffer is actually allocated in **getPage()**, but the values for these variables are calculated in **resetOp()**, since they depend on the input image's page size and data type attributes. Example 6-11 illustrates this with ilMonadicImg's implementation of **resetOp()**. (The ilXYZCint **struct** holds four integers, one for each of an image's dimensions; see "Convenient Structures" on page 403 for more information.)

Example 6-11 The **resetOp()** Function of ilMonadicImg

```
void ilMonadicImg::resetOp()
{
    // make sure we have a valid input
    ilImage* img = getInput();
    if (img == NULL ||
        getOrder() == ilSeparate &&
        getCsize() != img->getCsize())
    { setStatus(ilBADINPUT); return; }
    // get buffer strides
    ilXYZint pgSize, pgDel;
    int cps, nc;
    getPageSize(pgSize.x, pgSize.y, pgSize.z, cps);
    getPageDelta(pgDel.x, pgDel.y, pgDel.z, cps);
    getStrides3D(pgSize.x, pgSize.y,
        str.x, str.y, str.z, str.c, nc);

    ilXYZint opgSize, opgDel;
    int ocps;
    img->getPageSize(opgSize.x, opgSize.y, opgSize.z, ocps);
    img->getPageDelta(opgDel.x, opgDel.y, opgDel.z, ocps);
    ilOrder inord = img->getOrder();
    usesIstr = 0; // not supported yet;

    // true if output can use input strides
    useLock = (usesIstr || pgSize==opgSize && pgDel==opgDel)
    && (cps==ocps || cps==size.c && ocps==img->getCsize())
        && img->getDataType()==wType
        && img->getCoordSpace()==space
        && (order == inord || usesIstr
        && (order==ilSeparate) == (inord==ilSeparate));

    if (useLock) {
        img->getStrides3D(opgSize.x, opgSize.y,
               istr.x, istr.y, istr.z, istr.c,
               cBuffSize, img->getOrder());
        bufferSize = 0;
 }

     else {
        img->getStrides3D(pgSize.x, pgSize.y,
               istr.x, istr.y, istr.z, istr.c,
               cBuffSize, getOrder());

    // determine input buffer size
    bufferSize = ilDataSize(wType,
            pgSize.x*pgSize.y*pgSize.z*cBuffSize);
    }

// reset any attributes that change (none in this example)
setStatus(ilOKAY);
}
```

As shown, the **resetOp()** function performs three tasks:

- it makes sure the working data type is set
- it determines the size of the internal buffer
- it resets any attributes that change as a result of processing

The size of the internal buffer depends on the operator's working data type, on its page size, and on the input image's channel stride. Note that for this operator, the input and output buffers are the same size. (All the functions used in this example are described in Chapter 2, "The ImageVision Library Foundation," except for **ilDataSize()**, which is described in the reference pages.) In this example, none of the image's attributes change as a result of this operator's image processing algorithm. An example of an operator that does change attributes is ilRotZoomImg, which changes the image's size, unless the user has explicitly specified a desired size:

```
if (!isSet(ilIPsize)) {
    // calculate newXsize and newYsize
    size.x = newXsize;
    size.y = newYsize;
}
```

Notice that the attributes are set directly; the **setSize()** function isn't used since it would flag the size attribute as having been altered. You can use **isDiff()** to determine whether any parameters changed as a result of propagation. This function takes a mask of ilImgParam values and returns TRUE if any of the specified attributes changed.

The getPage() Function

The **getPage()** function is automatically called when another page is needed in an operator's cache. (See "The Cache" on page 46 for more information about the IL's caching mechanism.) Since operators hold processed data in their caches, the **getPage()** function needs to process data before writing it into the buffer provided.

Here are the arguments passed to **getPage()**:

```
ilStatus getPage(void* data, ilPgCB& cb);
```

The *data* argument indicates the location of the page buffer into which processed data should be written. The ilPgCB argument *cb* is a page control block that defines the amount of data needed. See “Deriving From ilMemCacheImg” on page 255.

Typically, the **getPage()** function handles the interface with the cache and then leaves the actual processing to another function (usually **calcPage()**), which is called by **getPage()**. This is how classes that derive from ilOpImg and that act as superclasses for other operators work; the other classes described in this section—for example, ilMonadicImg and ilDyadicImg—follow this model. The classes you derive do not have to follow this model, but it lends consistency to the library to do so. The name of the function that implements the image processing algorithm varies (as do the arguments it takes), depending on the class, as shown Table 6-5.

Table 6-5 ilOpImg Subclasses and Their Algorithm Functions

ilOpImg Subclass	Function That Implements the Image Processing Algorithm
ilMonadicImg	ilStatus calcPage(void* inBuf, void* outBuf, ilPgCB& cb)
ilArithLutImg	void calcRow(ilType inType, void* ilBuf, void* outBuf, int sx, int lim, int idx);
ilHistLutImg	ilStatus brpCalc(ilImage *src, ilImgStat *imgstat, double **brPoints);
ilDyadicImg	ilStatus calcPage(void* inBuf1, void* inBuf2, void* outBuf, ilPgCB& cb)
ilSpatialImg	ilStatus calcPage(void* inBuf, void* outBuf, ilXYZCint start, ilXYZCint end)
ilWarpImg, ilPolyWarpImg	void addrGen(int xpos, int ypos, int count, int xstep, int ystep, ilXYfloat* addrs)
ilFMonadicImg	void cmplxVectorCalc(float* vect, int rr, int ri, int size)
ilFDyadicImg	void cmplxVectorCalc(float* vect1, int rr1, int ri1, float* vect2, int rr2, int ri2, int size, int ch)
ilFFiltImg	float freqFilt(int u, int v)

Example 6-12 shows what a **getPage()** implementation might look like under this model.

Example 6-12 A **getPage()** Implementation for a Class Derived from ilOpImg

```
ilStatus myOperator::getPage(void* data, ilPgCB& cb)
{
    if (status != ilOKAY) return status;

    // get and check input
    ilImage* inImg = getInput(0);
    if (inImg == NULL) return setStatus(ilBADINPUT);

    if (useLock) {// if can multithread
      // lock the page down in the input
      ilPage* page = inImg->lockPage(cb.x, cb.y, cb.z, cb.c,
               status);
      if (status != ilOKAY) return status;
       if (page == NULL) return setStatus(ilBADINPUT);

    // call calcPage defined by derived class
    status = calcPage(*page, data, cb);

    // done with the input data, now release it
    inImg->unlockPage(page);
    return status;
  }
    else {/ / can’t multithread
        // allocate buffer
        ilStackBuffer inBuf(bufferSize);

       // read input data into inBuf
       // amount of data may vary but buffer size is constant
       ilConfig cfg(wType, order, cBuffSize, NULL, cb.c,
             getCoordSpace());

       status = inImg->getSubTile3D(cb.x, cb.y, cb.z,
               cb.nx, cb.ny, cb.nz,
               inBuf.data, cb.x, cb.y, cb.z,
               xPageSize, yPageSize, zPageSize, &cfg);
       if (status != ilOKAY) return status;

       // call calcPage defined by derived class
       return calcPage(inBuf.data, data, cb);
  }
}
```

As discussed above, **getPage()** can rely on calculations made in **resetOp()**. In this case, **resetOp()** determines the size of the internal buffer allocated by **getPage()**. If *uselock* is set in **resetOp()**, then the requested page is locked down in the input (using ilImage’s method **lockPage()**) and is passed to

calcPage(), which writes the results of its operation into *data*. When you are finished with the input, you unlock the page (using ilImage's method **unlockPage()**).

If this class is not using a **lockPage()** to access the input, then the ilStackBuffer class is used to allocate an input buffer. (See "Auxiliary Classes" on page 396 and the ilStackBuffer reference page for more information about this class.) The data to be processed is read into this buffer, and then both the input and output buffers (and the page control block) are passed to **calcPage()**, which performs the actual processing. After **getPage()** returns, the ilStackBuffer is immediately deallocated.

The **calcPage()** function implements the image processing algorithm, taking care to handle each valid data type appropriately. For example, Example 6-13 shows how ilAddImg computes the pixelwise sum of two images.

Example 6-13 Computing the Pixelwise Sum of Two Images

```
#define doAdd (type) \
if (1) { \
    type tb = type(bias); \
    for (; idx < lim; idx += sx) \
        ((type*)outBuf)[idx] = \
        ((type*)inBuf1)[idx]+((type*)inBuf2)[idx] + tb; \
} else

ilStatus
ilAddImg::calcPage(void* inBuf1, void* inBuf2, void* outBuf,
               ilPgCB& cb)
{
 // for interleaved: combine x/c loops to improve performance
 int nc = cb.nc, sc = str.c, nx = cb.nx, sx = str.x;
 if (sc == 1 && sx == nc) { nx *= nc; nc = 1; sx = 1;
              sc = 0; }

for (int z = 0; z < cb.nz; z++) {
    for (int y = 0; y < cb.ny; y++) {
       for (int c = 0; c < nc; c++) {
       int idx = z*str.z + y*str.y + c*sc, lim = idx + nx*sx;
       switch (dtype) {
                   case ilUChar:    doAdd(u_char); break;
                   case ilChar:     doAdd(char); break;
                   case ilUShort:   doAdd(u_short); break;
                   case ilShort:    doAdd(short); break;
```

```
                    case ilULong:    doAdd(u_long); break;
                    case ilLong:     doAdd(long); break;
                    case ilFloat:    doAdd(float); break;
                    case ilDouble:   doAdd(double); break;
                    case ilBit:      return ilBADPIXTYPE;
                   }
        }
      }
 }

 return ilOKAY;
}
```

Since ilAddImg is derived from ilMonadicImg, this function uses ilMonadicImg's stride data members—*str.x*, *str.y*, *str.z*, and *str.c*—to step through the data. Since the function used to compute the square root varies with the data type (**int** or **float**), a macro is used to apply the correct version. This macro computes the square root of each pixel value and writes the result in the output buffer.

Because IL programs can be multi-threaded, the **getPage()** and **calcPage()** functions shouldn't alter any member variables or do anything else that would make the algorithm non-reentrant. For example, the input buffer is allocated locally in **getPage()** rather than as a member in **resetOp()** so that concurrent execution of **getPage()** uses unique buffers for the different portions of the input image at the same time.

Clamping Processed Data

Some operators might trigger overflow or underflow conditions as they process data. To solve this potential problem, you should set clamp values that will then be used automatically when overflow or underflow arises, as described below.

In your implementation of **resetOp()**, call **setClamp()**:

```
void setClamp(ilType type = numilTypes);
void setClamp(double min, double max);
```

This function sets the values that pixels will be clamped to if underflow or overflow occurs. The first version sets the clamp values to be the minimum and maximum values allowed for the data type *type*; the default value of numilTypes means to use the operator’s current data type. The second version allows you to specify actual clamp values.

In the **calcPage()** function, use the **initClamp()** macro, passing in the operator’s data type (for example, **int** or **float**). This macro initializes two temporary variables to hold the minimum and maximum clamp values. Then, after you process each pixel of data, call the **clamp()** macro and pass in the processed pixel value. This function clamps the pixel value, if necessary, to the minimum or maximum clamp value.

To allow a user to set clamp values, you need to add ilIPclamp to the ilImgParam mask passed to **setAllowed()** in the constructor.

Setting Minimum and Maximum Pixel Values

Another problem that might arise as a result of processing data is that the processed values might exceed the range of values. For example, if you multiply two images (the pixel values of which fall in the 0 to 255 range) and then display the result, you might end up with pixel data that appears to be invalid if the pixel values exceed 255. To solve this potential problem, operators that alter the data range of their inputs need to set the **minValue** and **maxValue** data members (inherited from ilImage) to ensure that the processed data can be displayed. When the data is displayed using ilDisplay, it’s automatically scaled between these values so that a meaningful display is produced.

Here’s how ilAddImg computes **minValue** and **maxValue** in its **resetOp()** function (ilAddImg performs pixelwise addition on two images; a user-specified bias value can also be added to each pixel of the output):

```
// compute worst case min/max values
double min = getInputMin(0) + getInputMin(1);
double max = getInputMax(0) + getInputMax(1);

setStatus(checkMinMax(min+bias, max+bias));
```

The **getInputMin()** and **getInputMax()** functions return the minimum and maximum pixel value attributes of the input image. The argument for these functions is the index of the desired image in the list of inputs (the first input is at index 0). These values are added (since that’s what ilAddImg does), combined with the bias value, and then passed to **checkMinMax()**. This function first attempts to set the operator’s data type to the smallest supported data type that can hold the range specified by its arguments. If the data type is explicitly set by the user, however, it won't be changed. Then, if **minValue** and **maxValue** aren't explicitly set, they're set to the values passed to **checkMinMax()**. If **checkMinMax()** returns ilUNSUPPORTED, it isn't able to change the data type to support the range; in this case, **minValue** and **maxValue** are set to the maximum range of the current data type.

Deriving from ilMonadicImg or ilDyadicImg

Both ilMonadicImg and ilDyadicImg follow the **getPage()/calcPage()** model described above. These two classes provide support for operators that take a single input image (ilMonadicImg) or two input images (ilDyadicImg) and that operate on all pixels of the input image data. The classes that derive from ilMonadicImg and ilDyadicImg are listed below:

Classes That Derive from ilMonadicImg	Classes That Derive from ilDyadicImg
ilAbsImg	ilAddImg
ilFalseColorImg	ilANDImg
ilInvertImg	ilBlendImg
ilNegImg	ilDivImg
ilThreshImg	ilMaxImg
ilColorImg (& subclasses)	ilMinImg
ilLutImg (& subclasses)	ilMultiplyImg
ilScaleImg (& subclasses)	ilORImg
	ilSubtractImg
	ilXorImg

Here are some things you need to keep in mind if you derive from either of these classes:

- Don't redefine **getPage()**; use the version defined in ilMonadicImg or ilDyadicImg. Just implement your algorithm in **calcPage()**.
- If you redefine **resetOp()**, call the superclass version in your **resetOp()** (so that buffers and page sizes are reset appropriately):

  ```
  // either
  ilMonadicImg::resetOp();
  // or
  ilDyadicImg::resetOp();
  ```
- Use **setWorkingType()** if you want the input buffer to be read in as a different type than the operator image's data type. Note that the output buffer always uses the operator's data type.

Example 6-13 shows that ilAddImg's implementation of **calcPage()** takes three arguments. Similarly, ilMonadicImg's **calcPage()** function takes three arguments:

```
virtual ilStatus calcPage(void* inBuf, void* outBuf,
               ilPgCB& cb) = 0;
```

inBuf is the input buffer of data that needs to be processed, *outBuf* is the output buffer into which the processed data should be written, and *cb* is the page control block that describes the page of data being processed. Your implementation of **calcPage()** (for any class derived directly or indirectly from ilMonadicImg) must accept this argument list.

Since ilDyadicImg processes two input images at once, its **calcPage()** function supplies an extra input buffer of data. As above, your implementation of **calcPage()** must accept this argument list:

```
virtual ilStatus calcPage(void* inBuf1, void* inBuf2,
               void* outBuf, ilPgCB& cb) = 0;
```

When you derive from a class, you inherit all of its public and protected data members and member functions. All the public members for ilMonadicImg and ilDyadicImg have been discussed in previous sections. The protected member functions are **resetOp()**, **getPage()**, and **calcPage()**. For reference purposes, here are ilMonadicImg's protected data members:

```
ilXYZCint str;     // output (page) buffer strides
ilXYZCint istr;    // input image strides

int bufferSize;    // size of input buffer in bytes
int cBuffSize;     // number of channels in input buffer
```

An example using these members is shown in "The getPage() Function" on page 279.

The protected data members defined in ilDyadicImg are similar:

```
ilXYZCint str;              // output buffer strides
ilXYZCint istr1, istr2;     // input image strides

int buffSize1, buffSize2;   // size of input buffers in bytes
int cBuffSize1, cBuffSize2; // number of channels in input
//  buffers
```

Deriving from ilArithLutImg

As an abstract class, ilArithLutImg defines how to use look-up tables when performing arithmetic or radiometric operations. To derive from it, you implement your algorithm in **calcRow()** rather than in **calcPage()**:

```
void calcRow(ilType intype, void *inBuf, void *outBuf,
              int sx, int lim, int idx);
```

The *intype* parameter indicates the input image's data type. The next two arguments are the input buffer of data that needs to be processed and the output buffer into which processed data should be written. The next three arguments specify how to step through the data: *sx* is the *x* stride of the output buffer, *lim* is the maximum *x* stride, and *idx* is the starting index. The **calcRow()** function contains the algorithm for processing one row of input data. For efficiency, you can use the defined macro **doRow()** to obtain the proper data type and feed it to the macro **doCalc()**. (The **doRow()** macro is defined in ilArithLutImg's header file.) If you use these macros, your **calcRow()** definition would be just a call to **doRow()**:

```
ilMyOpImg::calcRow(ilType inType, void* inBuf,
               void* outBuf,int sx, int lim, int idx)
{ doRow(); }
```

and you'd actually implement the computation algorithm in the macro **doCalc()**, as ilPowerImg does, for example, as shown in Example 6-14.

Example 6-14 Implementation of **doCalc()** in ilPowerImg

```
#define doCalc(outype, intype) \
if (1) {\
    if (inType == ilDouble || dtype == ilDouble) { \
        for (; idx < lim; idx += sx) \
              ((outype*)outBuf)[idx] = (outype)pow((double) \
                  (((intype*)inBuf)[idx]*scale+bias), \
                  (double)power);\
    }\
    else {\
        for (; idx < lim; idx += sx) \
           ((outype*)outBuf)[idx] = (outype)powf((float) \
           (((intype*)inBuf)[idx]*scale+bias),(float)power);\
    }\
}
```

You also need to implement **loadLut()** to compute and load the appropriate values into the LUT. Example 6-15 shows ilPowerImg's version of **loadLut()**.

Example 6-15 Implementation of **loadLut()** in ilPowerImg

```
void
ilPowerImg::loadLut()
{
  for (int i=0; i<lut->getLength(); i++) {
  if (wType == ilDouble || dtype == ilDouble)

    lut->setVal(pow((double)(i*scale+bias),(double)power),i);
  else
    lut->setVal(powf((float)(i*scale+bias),(float)power),i);
   }
}
```

For your convenience, ilArithLutImg has functions for scaling and biasing the input data before the LUT is applied:

```
void setScale(double scale);
double getScale();
void setBias(double bias);
double getBias();
```

Deriving from ilHistLutImg

The ilHistLutImg class provides support for operators that compute a look-up table from the histogram of the source image and then apply this table to the source image. It derives from ilArithLutImg and implements its own versions of **calcPage()**, **calcRow()**, and **loadLut()**. The only pure virtual function in ilHistLutImg is **brpCalc()**, which all derived classes must implement:

```
virtual ilStatus brpCalc(ilImage *src, ilImgStat *imgstat,
    double **brPoints) = 0;
```

This function computes the breakpoints (*brPoints*) of a piecewise LUT. You can think of it as a pointer to a two-dimensional array whose members can be accessed by:

```
double val = brPoints[i][j] where:

    i = 0,1,2,...,nc-1
    j = 0,1,2,...,nbins_i
    nc = number of channels in the source image
    nbins_i = number of bins in the histogram of channel i
```

You can obtain the number of bins by using *imgstat*'s **getNbins()** function. The variable *val* in the example shown above represents what the pixel intensity represented by the *j*th bin of the histogram for channel *i* maps to. For example, to invert pixel intensities of an image containing **unsigned char** data, you can use:

```
brPoints[i][j] = 255-j;
```

All the members of *brPoints* need to be evaluated in **brpCalc()**, using both the source image and a pointer to its associated data as inputs. Derived classes don't need to allocate and manage memory for *brPoints*, since ilHistImg does this for them. In addition, ilHistImg provides convenience functions for setting the ilImgStat and ilRoi objects:

```
void setImgStat(ilImgStat *imgstat);
void setRoi(ilRoi *roi, int xoffset=0, int yoffset=0);
```

If you implement **resetOp()** in a derived class, be sure to explicitly call ilHistLutImg's version of **resetOp()**.

An example of a class derived from ilHistLutImg might be an operator called ilPixelCountImg, which replaces each pixel intensity by the number of times it occurs in that particular channel. Such an operator might be implemented as shown in Example 6-16.

Example 6-16 A Class Derived from ilHistLutImg to Count Pixels

```
class ilPixelCountImg:public ilHistLutImg {
    private:
        ilStatus brpCalc(ilImage *src, ilImgStat *imgstat,
               double **brPoints);
    public:
        ilPixelCountImg(ilImage *src);
}

ilPixelCountImg::ilPixelCountImg(ilImage *src)
             :ilHistLutImg(src)
{
}

ilStatus brpCalc(ilImage *src, ilImgStat *imgstat,
        double **brPoints)
{
    if (src==NULL) return ilBADINPUT;
    int nch=src->getNumChans();

    for (int i=0; i<nch ; i++) {
        int *hist = imgstat->getHist(i);
        int nbins = imgstat->getNbins(i);
        int total = imgstat->getTotal(i);
        double max = src->getMaxValue(i);

        for (int j=0; j<nbins; j++) {
            brPoints[i][j]=(hist[j]*max)/total;
        }
    }
    return ilOKAY;
}
```

Deriving from ilSpatialImg

The ilSpatialImg class provides basic support for operators that adjust a pixel’s value based on a weighted sum of its surrounding pixels. The kinds of operators that can use this support perform convolutions for particular purposes—for example, they calculate gradients or perform rank filtering. ilSpatialImg's subclasses are listed below.

ilSepConvImg	**ilSepConvImg Subclasses**	**RankFltImg Subclasses**
ilLaplaceImg	ilBlurImg	ilMaxFltImg
ilRobertsImg	ilCompassImg	ilMedFltImg
ilSobelImg	ilSharpenImg	ilMinFltImg

The ilSpatialImg class follows the same **getPage()/calcPage()** model as ilMonadicImg does. All the following hints are also true about deriving from ilSpatialImg (and any of its subclasses):

- Don't redefine **getPage()**, just implement your algorithm in **calcPage()**.
- If you redefine **resetOp()**, call the superclasses in your **resetOp()** (so that buffers and page sizes are reset appropriately):

  ```
  ilSpatialImg::resetOp();
  ```
- Use **wType** as the working data type, but be sure the data you write into the output buffer is of type **dType**.

The **calcPage()** function for ilSpatialImg takes these arguments:

```
virtual ilStatus calcPage(void* inBuf, void* outBuf,
        ilXYZCint start, ilXYZCint end) = 0;
```

The input buffer *inBuf* points to a buffer containing the data that needs to be processed, and *outBuf* points to a page in the cache where the processed data should go. Depending on the edge mode, some of the data in *inBuf* may have been set to the image's fill value. (Refer to "Spatial Domain Transformations" on page 133 for further explanation of the possible edge modes.) *start* and *end* demarcate the beginning and the end of source data in *inBuf* that needs to be computed, so you should use them to delimit the computation.

ilSpatialImg provides several protected member variables that are likely to be useful as you implement your algorithm. These include strides, for use in stepping through the input and output buffers:

```
ilXYZCint inStr;    // input strides
ilXYZCint outStr;   // output strides
```

The ilXYZCint **struct** holds four integers; for more information about it, see "Convenient Structures" on page 403. ilSpatialImg also constructs a kernel offset table and a kernel value table based on the data in the kernel. The offset table contains offsets into the input buffer to access data corresponding to nonzero kernel elements. The value table contains the nonzero elements and corresponds to the offset table. These data members are shown below:

```
ilKernel* kernel;   // kernel object
int kernSz;         // number of nonzero kernel elements
int* kernOff;       // kernel offset table
void* kernVal;      // kernel value table
```

You can use these tables to improve the efficiency of your algorithm—for example, by avoiding multiplications by 0. A related function, **setKernFlags()**, allows you to set flags indicating that the offset table and / or value table must be created:

```
void setKernFlags(int of=0, int vf=0);
```

If you pass in a 1 for either the offset flag *of* or the value flag *vf*, the corresponding table will be created to match the current kernel. You should call this function in the constructor of your class (with 1's as arguments) so that the tables get built.

The following code might be part of a **calcPage()** implementation for a convolution. It shows how kernel values multiply data values and how this result is accumulated. It also demonstrates how *inBuf*, *outBuf*, and the kernel are offset with respect to one another. This example is a bit simplified in that it assumes both wType and dtype are ilFloat, and it assumes that the kernel weights sum to 1.0 so that no clamping is necessary. Also, if you actually need to implement a convolution-based algorithm, consider deriving from ilConvImg, as described in Example 6-17

Example 6-17 A Class Derived from ilConvImg to Multiply and Accumulate Data

```
// cast the buffers to be of type wType
float* in = (float* )inBuf;
float* out = (float* )outBuf;

// iterate through all channels
for (int ci = start.c; ci < end.c; ci++) {
    int cSrcIndex = ci*inStr.c;
    int cDstIndex = ci*outStr.c;

// iterate through z dimension
for (int zi = start.z; zi < end.z; zi++) {
    int zSrcIndex = zi*inStr.z + cSrcIndex;
    int zDstIndex = zi*outStr.z + cDstIndex;

// iterate through y dimension
for (int yi = start.y; yi < end.y; yi++) {
    int srcIndex = start.x*inStr.x + yi*inStr.y + zSrcIndex;
    int dstIndex = start.x*outStr.x + yi*outStr.y + zDstIndex;

    // iterate through x dimension
    for (int xi = start.x; xi < end.x;
        xi++, srcIndex += inStr.x, dstIndex += outStr.x) {
        float sum = bias;// bias is inherited from ilOpImg
        // cast kernVal to a float
        float* kr = (float* )kernVal;

        //iterate through nonzero kernel values
        for (int k = 0 ; k < kernSz ; k++) {
               sum += in[srcIndex+kernOff[k]] * kr[k];
        }

  // note use of kernOff to access the correct input value
        out[dstIndex] = sum;
    }
  }
 }
}
```

Deriving from ilConvImg or ilSepConvImg

The ilConvImg class performs general convolution on an image, and the ilSepConvImg class performs separable convolution. You might want to derive from these classes if kernel values aren't available at the time the operator is constructed because they depend on certain input parameters. In this case, you'd define a **resetOp()** function in the derived class that computes the *x* and *y* kernel values from input parameters. Then you could use the inherited functions **setXKernel()**, **setYKernel()**, and **setKernelSize()** to specify the kernel and its size, after which you'd need to explicitly call ilConvImg's or ilSepConvImg's version of **resetOp()**. Remember that the kernel for ilConvImg should be a two-dimensional matrix, while that for ilSepConvImg should be two separate vectors. You should also set the edge mode and bias value.

Deriving from ilWarpImg or ilPolyWarpImg

The ilWarpImg class provides basic support for warping an image; it defines **getPage()** so that derived classes need to implement only the virtual function **addrGen()**, which should contain the warping algorithm. ilPolyWarpImg, which inherits from ilWarpImg, provides basic support for warping an image using up to seventh-order polynomials; it implements **addrGen()** but requires that the user explicitly define the coefficients of the polynomials. Often, users know the kind of warp effect they want to achieve, but they don't know how to specify coefficients to achieve this effect. The two operators that derive from ilPolyWarpImg—ilRotZoomImg and ilTieWarpImg—provide the user with an indirect way of specifying the coefficients. For example, ilRotZoomImg lets you specify an angle of rotation, and then it performs the work necessary to compute the coefficients needed to achieve the rotation.

There are two principal motivations for deriving your own warp operator:

- If you need a warping algorithm that uses higher-order polynomials (eighth-order and above), you should derive a new operator directly from ilWarpImg.
- If you want to define a new way of specifying the warping coefficients, you can derive from ilPolyWarpImg as long as seventh-order polynomials are sufficient for your needs. If they're not sufficient, you'll need to derive from ilWarpImg.

If you derive from ilWarpImg, you need to implement the pure virtual function **addrGen()**, which transforms output image coordinates to input image coordinates. This function is called by **getPage()** as needed to generate coordinates. The calling sequence of **addrGen()** is shown below:

```
virtual void addrGen(float upos, float vpos, float ustep,
               ilXYSfloat *addrs, int count) = 0;
```

The first two arguments, *upos* and *vpos*, define the starting coordinate in the output image, and *count* defines the number of addresses that must be transformed. The *ustep* value indicates how to step through the output image data; add it to the current *u* coordinate to determine the next coordinate that needs to be transformed. The *v* coordinate is always constant for each call to **addrGen()**. Return the computed input image coordinates in the *addrs* array.

The virtual function **checkTile()** determines the bounds of the input tile required for a given output tile. ilWarpImg's version of this function accomplishes this by computing the minimum and maximum values in both the *x* and *y* dimensions of the transformed address on an 8x8 subsampled grid covering the output tile. If you can provide a more efficient way to compute the bounds, you can override this function. If you do, you might want to use **maxInBuf()**, which returns the size of the input buffer in bytes. The ilWarpImg reference page provides more information about how to define your own version of **checkTile()**.

If you derive from ilPolyWarpImg, you need to call only **setCoeff()**. Your derived class should translate the user-specified arguments into coefficients and pass them to **setCoeff()**. See the ilPolyWarpImg reference page for more information.

Deriving from ilFMonadicImg or ilFDyadicImg

The ilFMonadicImg and ilFDyadicImg classes provide the basic support for operators that perform pixelwise computations on images that have been converted to the frequency domain. To implement a frequency domain filter, derive from ilFFiltImg, as explained in “Deriving from ilFFiltImg” on page 299 (or use ilFMultImg). Both ilFMonadicImg and ilFDyadicImg expect the input image(s) to be in the format produced by ilRFFTfImg (or by ilFFTOp's **ilRfftf()** function). As their names suggest, ilFMonadicImg expects a single input image, and ilFDyadicImg expects two input images. Their subclasses are shown below.

ilFMonadicImg's Subclasses	ilFDyadicImg's Subclasses
ilFConjImg	ilFCrCorrImg
ilFRaisePwrImg	ilFDivImg
	ilFMultImg

Both classes implement **getPage()** for you so that you have to implement your algorithm only in **cmplxVectorCalc()**. This function processes a vector of complex values; **getPage()** calls it as needed to process an entire page of data. The calling sequence for ilFMonadicImg's **cmplxVectorCalc()** is shown below:

```
virtual void cmplxVectorCalc(float* vect,
              int rr, int ri, int size);
```

The first argument, *vect*, is a pointer to a vector of *size* number of complex values. On input, *vect* holds the data to be processed, and on output it holds the processed data. Use *rr* and *ri* to step through this vector: *rr* is the stride between the real parts of two consecutive complex numbers in *vect*, and *ri* is the stride between the real and imaginary part of a complex number in *vect*.

An example of a class derived from ilFMonadicImg would be an operator that converts rectangular coordinates to polar coordinates. Such an operator would need to declare only two member functions:

```
class ilFPolarImg : public ilFMonadicImg {
private:
    void cmplxVectorCalc(float* vect, int rr, int ri,
               int size);
public:
    ilFPolarImg(ilImage* src);
}
```

In this example, **cmplxVectorCalc()** is declared private since it's assumed that ilFPolarImg won't have subclasses. Example 6-18 shows how the constructor and **cmplxVectorCalc()** functions might be implemented.

Example 6-18 Constructor and Member Functions of a Class Derived from ilFMonadicImg to Convert Coordinates

```
ilFPolarImg::ilFPolarImg(ilImage* src1)
{
    setValidType(ilFloat);
    addValidOrder(ilSeparate);
    setNumInputs(1);
    addInput(src1);
}

void
ilFPolarImg::cmplxVectorCalc(float* vect, int rr, int ri,
             int size)
{
 int i, k;
 for (i = k = 0; k < size; i += rr, k++) {
    float real = vect[i];
    float imag = vect[i + ri];
    vect[i] = fsqrt (real*real + imag*imag);
    vect[i+ri] = fatan2 (imag, real);
 }
}
```

For classes derived from ilFDyadicImg, **cmplxVectorCalc()** takes more arguments since there are two input vectors that need processing:

```
virtual void cmplxVectorCalc(float* vect1, int rr1, int ri1,
               float* vect2, int rr2, int ri2,
               int size, int ch) = 0;
```

In this case, *vect1* and *vect2* are pointers to the input vectors, which are of the same *size*. On input, they hold data to be processed, and on output, *vect1* holds the output data and *vect2* is unchanged. You can use *rr1*, *ri1*, *rr2*, and *ri2* to step through these vectors. The final argument, *ch*, indicates which channel is currently being processed. This argument is ignored in most cases, but you can use it when the computation being performed depends on the channel. For example, when a cross-correlation is computed, each channel's output is normalized by the average value of that channel.

Here's what the declaration of ilFMultImg (which multiplies two Fourier images) might look like:

```
class ilFMultImg : ilFDyadicImg {
private:
    void cmplxVectorCalc(float* vect1, int rr1, int ri1,
               float* vect2, int rr2, int ri2,
               int size, int ch);
public:
    ilFMultImg(ilImage* src1, ilImage* src2);
}
```

A possible implementation of this class is shown in Example 6-19.

Example 6-19 A Class Derived from ilFDyadicImg to Multiply Two Fourier Images

```
ilFMultImg::ilFMultImg(ilImage* src1, ilImage* src2)
{
 setValidType(ilFloat);
 addValidOrder(ilSeparate);
 setNumInputs(2);
 addInput(src1);
 addInput(src2);
}

void
ilFMultImg::cmplxVectorCalc(float* vect1, int rr1, int ri1,
 float* vect2, int rr2, int ri2, int size, int )
{
 int i, j, k;
 for (i = j = k = 0; k < size; i += rr1, j += rr2, k++) {
     float real1 = vect1[i];
     float imag1 = vect1[i + ri1];
     float real2 = vect2[j];
     float imag2 = vect1[j + ri2];
     vect[i] = real1*real2 + imag1*imag2;
     vect[i+ri1] = real2*imag1 - imag2*real1;
 }
}
```

Deriving from ilFFiltImg

The ilFFiltImg class provides basic support for operators that perform frequency filtering, such as ilFExpFiltImg and ilFGaussFiltImg. This class is particularly useful when the filter can be described as a real-valued analytic function. The input image must be in the format produced by ilRFFTfImg or by ilFFTOp's **ilRfftf()** function.

Since ilFFiltImg implements **getPage()**, all you have to do to derive from this class is provide your algorithm in the **freqFilt()** function:

```
virtual float freqFilt(int u, int v) = 0;
```

This function returns the filter value at the frequency coordinates *u* and *v*, which are the coordinates in the *x* and *y* directions, respectively. If *nx* and *ny* are the *x* and *y* dimensions of the original spatial-domain image, then:

$$0 \le u < \frac{nx}{2} + 1 \text{ and } -\frac{ny-1}{2} \le v \le \frac{ny}{2}$$

The following example shows a low-pass frequency filter implementation:

```
class ilFLowPassImg : public ilFFiltImg {
private:
 float cutoff;
 float freqFilt(int u, int v)
 {return fexp(-(u**2 + v**2)/cutoff**2);}
public:
 ilFLowPassImg(ilImage* src, float cutoff);
 void setCutOff(float val) {cutoff = val; setAltered();}
}
```

```
ilFLowPassImg::ilFLowPassImg(ilImage* src, float cutoff)
{
 setValidType(ilFloat);
 addValidOrder(ilSeparate);
 setNumInputs(1);
 addInput(src);
 setCutoff(cutoff);
}
```

The constructor for this class takes an input source image and a cutoff level as arguments. The **freqFilt()** function is implemented as shown below:

$$e^{-\left(\frac{u^2+v^2}{cutoff^2}\right)}$$

Deriving from ilRoi

If you derive your own ilRoi class, you need to know how ilRoi expects valid and invalid regions to be defined. Valid and invalid regions are described in terms of their *run length*, which is the number of pixels between the current valid pixel, for example, and the first invalid pixel, traveling as specified by the coordinate space. To derive from ilRoi, you need to implement **getNextRle()** and **getNextIRle()**, which are pure virtual functions, so that they compute the next valid and invalid run lengths, respectively:

```
virtual int getNextRle(ilRoiMap &map) = 0;
virtual int getNextIRle(ilRoiMap &map) = 0;
```

These functions should start at the specified point (*x*, *y*) and search for the desired run; they should return TRUE or FALSE, depending on whether a run of the desired type is found. The ilRoiMap class encapsulates all the information needed and returned by the **getNextRle()**, **getNextIRle()**, **getBoundBox()**, and ilRoiIter's functions. The ilRoiMap class also makes it easy to attach the same ROI to different images at different offsets. The constructor for this class is shown below:

```
ilRoiMap(ilImage *img, int xoffset=0, int yoffset=0,
             ilTile *clipBox=NULL, int skip=FALSE, int x=0,
             int y=0, ilCoordSpace spc=ilCoordSpace(0));
```

**img*	Specifies the image to be associated with the ROI.
xoffset, yoffset	Specify the offsets into the image at which the ROI is to be placed.
clipBox	If not NULL, restricts the search for valid or invalid run lengths to the rectangle specified. (The ilTile class is declared in the header file *il/ilTile.h*; an ilTile object defines a rectangle, using six arguments—an origin (*x*,*y*,*z*) and dimensions (*nx*,*ny*,*nz*).)
skip	Indicates whether to skip the run length that encompasses the starting point given by (*x*,*y*); generally, it should be FALSE to begin with so that the first run length at the specified starting point isn't skipped.

spc	Specifies the desired coordinate space for the starting coordinates of the run length and bounding box (see below); the clipping box and the offsets are assumed to be in this coordinate space. With the default, NULL, the coordinate space of the image *img* is used.

Both **getNextRle()** and **getNextIRle()** should update the *x*, *y*, and *len* members of ilRoiMap that are passed to them. In order to do this, declare your ROI class as a friend of ilRoiMap. You can do this by adding the line "friend myRoiClass" to the class definition of ilRoiMap in the include file */usr/include/il/ilRoiMap.h.*

Use the ilRoiIter class to cycle through the valid or invalid run lengths in an ROI. As shown below, the constructor to ilRoiIter takes a pointer to an ROI object and to an ilRoiMap object:

```
ilRoiIter (ilRoi *roi, ilRoiMap *map);
```

Use the ilRoiIter class functions **moreRle()**, **moreIRle()**, **getX()**, **getY()**, and **getLen()** to obtain the starting locations and length of the valid or invalid run. Table 6-6 illustrates the values that these five functions would compute given the two starting points shown for the row of image data presented in Figure 6-3. (Assume the O's represent valid data and the X's represent invalid data; the numbers are shown as counters for convenience.)

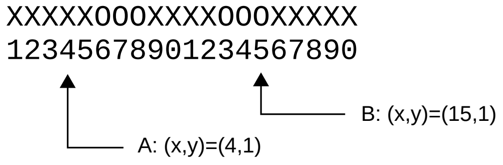


Figure 6-3 Valid and Invalid Data in a Row of Pixels

Table 6-6 Valid and Invalid Run Lengths

	moreRle()		**moreIRle()**	
	Point A	**Point B**	**Point A**	**Point B**
getX()	6	4	9	16
getY()	1	1	1	1
getLen()	3	[a]	4	5
return value	TRUE	FALSE	TRUE	TRUE

a. When FALSE is returned by moreRle() or moreIRle(), values returned by getX(), getY(), and getLen() are unchanged, so the value returned by getLen() might be invalid.

These ilRoiIter functions are normally used in a loop to cycle through the run lengths, as shown below:

```
ilRoi *myRoi;
ilTile myClipBox(1, 1, 0, 10, 10, 1);
ilRoiMap myMap(myImg, 10, 20, &myClipBox, FALSE, 0, 0,
               ilLowerLeftOrigin);
ilRoiIter myIter(myRoi, &myMap);

while ( myIter.moreRle() ) {
    int startx = myIter.getX();
    int starty = myIter.getY();
    int len = myIter.getLen();
    // process the run data here
}
```

This loop allows you to step through all the valid runs of data and perform whatever processing is necessary. Now suppose you want to use the same ROI on another image to obtain the invalid run lengths. Your code might look like this:

```
myIter.init(anotherImg);

while ( myIter.moreIRle() ) {
    int startx = myIter.getX();
    int starty = myIter.getY();
    int len = myIter.getLen();
    // process the run data here
}
```

Note the use of the **init()** function to reset *myIter*'s image. There are several such initializing functions that allow you to begin the search for valid or invalid run lengths with the same ilRoiIter object. All these functions set the *skip* flag to FALSE and initialize the starting point to be the origin of the clipping box (if one is specified) or the image (if there is no clipping box). You can also reset other parameters by choosing the appropriate **init()** function:

```
void init(ilImage *img);
void init(ilRoi *roi);
void init(ilTile *clipBox);
void init(ilCoordSpace spc);

void init(ilImage *img, int xoffset, int yoffset);
void init(ilImage *img, int xoffset, int yoffset,
               ilTile *clipBox, ilCoordSpace spc);
```

Also note that you can use ilRoi's **getRoiType()** function to find out whether a particular ROI is an ilBitMapRoi or an ilRectRoi. The corresponding values returned by this function are ilBitMap and ilRect, as listed in the definition of the ilRoiType enumerated type in the header file *il/ilType.h*. You can add your own ROI types to this enumerated type if you define other types of ROIs.

Finally, when deriving from ilRoi, you also need to implement the pure virtual function **getBoundBox()**:

```
void getBoundBox(ilRoiMap &map, ilTile &box)=0;
```

This function returns by reference the bounding box (*box*), which is the smallest rectangle that contains the entire valid region within the clipping box specified by *map*. If a clipping box is not specified, the entire image is searched. The coordinate space of *box* matches that specified by *map*; if *map*'s coordinate space is NULL, the coordinate space of the image is used.

Chapter 7

Optimizing Your Application

This chapter explains how you can improve the performance of your application by optimizing memory usage, taking complete advantage of hardware acceleration features, and controlling multi-threading.

Chapter 7

Optimizing Your Application

This chapter is intended for programmers who are somewhat familiar with the IL and who want to optimize their applications. This chapter has three major sections:

- "Managing Memory Usage" on page 308 describes how to optimize the memory usage of your application.
- "Using Hardware Acceleration" on page 315 describes what operations can be accelerated on different graphics hardware.
- "Controlling Multi-threading" on page 336 describes how to optimize the performance of your application by adjusting the multi-threading facility of the IL.

Managing Memory Usage

You can optimize the performance of your application by making knowledgable decisions about the use of memory resources. Three areas in which you can optimize use of memory are:

- use of cache
- page size
- buffer size

The following sections describe these three areas in greater detail.

Optimizing Use of Cache

You can optimize the use of cache in your application in a number of ways. You can change the size of the cache, control the automatic growth of cache that can occur if multi-threading is turned on, set priority on an image in cache, and use tools to monitor the use of cache.

Before reading further, you might want to refer to other parts of this manual that describe caching. To learn about:

- caching and paging, read “The Cache” on page 46.
- changing cache size using the functions **ilSetMaxCacheSize()** and **ilSetMaxCacheFraction()**, read “Managing Cache” on page 49.
- using the **ilCompactCache()** and **ilFlushCache()** functions to compact global cache memory, read “Managing Cache” on page 49

Cache Size

This section describes how to determine the cache size that is most appropriate to your application. Every class descended from ilCacheImg (including all the image operators) needs memory for a cache, which holds pages of image data. By default, the IL cache size is 30% of the total user memory on the system. In some applications this is too large, in others it’s too small.

The optimum cache size for any particular IL program depends on the size of the images that the program manipulates and on the type of operations it performs on the data.

If your application:

- operates on small images, you can set the size of the cache to be the size of the image, minimizing both memory and total processing needs.
- operates on large images, you will need a larger cache. A program with a large image cache improves performance because it saves the processing overhead required to move data in and out of memory. However, if the cache is too large and uses up main memory, you could potentially be swapping pages in and out of virtual memory on your system, which degrades performance.
- displays image data, its cache should be large enough to hold the displayed window of data.
- just produces a reduced resolution version of an image in another image file, you can get by with a smaller cache.

Typically, the cache won't be able to hold everything needed for an operation. For these cases, set the cache at least large enough to hold both:

- one page of output data
- the number of pages of input data required to produce that page

For example, suppose that you're copying an image with pages that are 128 pixels square (these are the default page dimensions for FIT images) to an image that sets the page width to match the width of the image (this is true for SGI RGB images). Further, suppose that both images are 2K pixels wide and that the SGI image sets its page height to 64 pixels. Figure 7-1 shows the two images and the pages contained in them. (This figure isn't drawn to scale.)

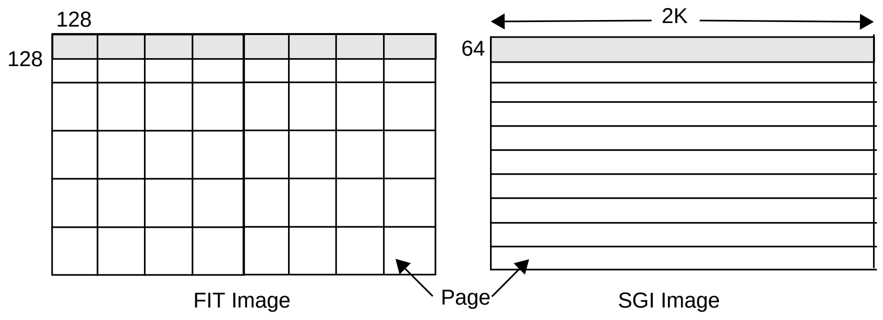


Figure 7-1 Varying Page Dimensions

To write a single 2 KB x 64 SGI page, you need data from all the FIT pages that span the width of the image. Thus, in this example, set the cache size to (2 KB x 64 + 2 KB x 128) x 3 bytes (assuming that there are 3 channels and that the data type is ilChar). Add about 10% to this figure to allow for the size of page descriptors and other overhead. This allows all needed pages to be held in the cache. If the cache is smaller than this, the data can still be processed, but FIT pages are bumped out of the cache and then read back in as successive SGI pages are written.

Effect of Multi-threading on Cache

The use of multi-threading can affect the size of cache in an application (see "Multi-threading" on page 71). With multi-threading enabled, the cache can grow larger than its preset limit if all the pages contained within it are locked down and another page must be brought into the cache. This growth of cache prevents deadlock, but can cause the application to use more memory than you wish. To prevent this behavior, do one of the following:

- reduce the number of threads (so that there are never more threads than pages in the cache)
- reduce the size of each page (so that there are enough pages in the cache for all the threads)
- increase the size of the cache (so that there is one page for each thread)

For example, if there is room in the cache for only two of the operator's pages but there are four threads, the cache may be grown so that it contains four pages. If this is unacceptable, either reduce the number of threads to two or reduce the size of a page by half (so that the cache can contain twice as many, or four, pages). Multi-threaded applications always need more memory to run efficiently; the best solution is to add more memory to your system. If this is not possible, the next best solution is to reduce the page size.

Cache Priority

As explained in “Priority” on page 50, the pages of an image that are brought into cache as the result of an operation on the image are kept there until the cache becomes full. When the cache is full, decisions must be made about which pages are kept in cache and which are discarded and replaced by new pages.

The IL attempts to optimize the use of cache. You can also affect the caching process by using the **setPriority()** and **lockPage()** methods. It is helpful, when you are optimizing your use of cache, to understand actions the IL is also taking to accomplish this. The IL considers these factors as it manages the contents of cache:

- time since the last reference to a page. Pages most recently referenced are least likely to be overwritten.
- number of references made to a page. Pages that are frequently referenced are least likely to be overwritten.
- the destination of a page. The IL automatically raises the priority of a page request for data that is directly displayed. This has the effect of caching data at the end of a displayed chain.

Sometimes it makes sense to cache data at points other than at the end of a chain. The reference counting used in the page replacement algorithm can help to accomplish this caching, but in cases where explicit knowledge of the application is required, you can use the **setPriority()** method of ilImage to set the priority of the image containing the specified page. For instance, you may want to raise the priority of the file input to a long chain to avoid rereading the input if the chain is expected to be altered.

You may also want to raise the priority of the input to an operator that is having its parameters interactively modified, although again the reference counting built into IL will tend to automatically increase the priority for you.

Monitoring the Cache

You can monitor image data cache usage in two ways:

- by using the image tool ilMonitor. This provides an interactive means for you to monitor the use of the cache. See “Image Tools” on page 351 for more information about ilMonitor.
- by setting the environment variable IL_MONITOR_CACHE to a value of 1. This causes the IL to print a message for each page loaded into the cache or deleted from the cache. The message identifies the page location in its associated image and the class and address of that image.

It is often important to know about the operator images (such as color converters) that are automatically inserted by IL. You can use **ilDumpChain()** to print out a simple description of an IL chain.

An example using this environment variable is shown below:

```
% setenv IL_MONITOR_CACHE 1
% imgview /usr/demos/data/images/weather.fit
Page (0,0,0,0) loading in Color(0x10034ec8)
Page (0,0,0,0) loading in FIT(0x1001d010)
```

This example shows that a color converter operator image has been used to cache the data from the FIT image in frame-buffer format. It also shows the background view with ilConstantImg as input that is automatically created by ilDisplay. You can use this technique to identify cache thrashing if you suspect it’s occurring. You can eliminate such problems by one of the techniques described in the preceding sections.

For more challenging situations, you may want to use the **setPagingCallback()** method in ilCacheImg. Refer to the ilCacheImg reference page for more details.

Note: Do not attempt to use **setPagingCallback()** and ilMonitor at the same time since ilMonitor uses the **setPagingCallback()** mechanism.

Page Size

Image data is always cached in pages. A file image's page dimensions match those used to store the image on disk. By default, an operator's page size is defined by its input images. Certain operators override this default size, which can affect the caching of images. Some images also let you set the size of the pages in the cache and the data type and ordering of the cached data. The data type and ordering affect how data is cached, so if you change these attributes, you might also want to change the size of the cache.

Optimum Page Size

Operators are usually the only images that allow you to set the page size. The ideal page size depends on the particular application, but in general you want an image's page size to be as close as possible to that of whichever image it's being copied to or read from. If the application involves roaming on a large image, however, the page size should be relatively square. The functions that change page size are defined by ilImage and are explained in "Page Size" on page 52.

Large pages use up more memory, which is a problem when the cache grows beyond its limit and starts allocating extra pages to get around deadlock. See the previous section for suggested solutions. Making pages too small, however, forces too much processing overhead. A page shouldn't be smaller than 32 x 32 bytes, and in general the total number of bytes in a page should be between 16KB and 64KB. This range typically works out to be 128 x 128 to 256 x 256 when measured in pixels. Some operators, such as the frequency domain ones, are more efficient when the page size is a power of 2.

Maximizing Efficiency When Copying Pages

The **copyTile()** function is an efficient way to copy a tile of data from one ilImage to another:

```
ilStatus copyTile(int x, int y, int nx, int ny,
               ilImage* other, int ox, int oy,
               int* chanList=NULL, int from=1);
```

By default, the tile is copied to the calling image from the image pointed to by *other*. The *x* and *y* arguments specify the origin of the tile in the destination image, and *nx* and *ny* specify the size of the tile. The tile that's to be copied is located at (*ox,oy*) in the *other* image. (If the tile is at the same location in both the source and destination images, then *x=ox* and *y=oy.*) If the source and destination images have different coordinate spaces, the data is transposed automatically as necessary. The last argument, *from*, allows you to reverse the direction of the copy; if it's 0, the tile is copied to *other* from the calling image.

The default direction (from *other* to the calling image) is the most efficient direction when you're copying to a file image (that is, one that inherits from ilFileImg) because the page dimensions of the calling image define the units of processing. With the default direction, the calling image is the destination image. As the copy proceeds, each page of the calling image is "locked down" in the cache and filled in completely from the *other* image. If you use the reverse direction for copying, the IL may have to shuffle the destination image's pages in and out of the cache; since the destination is really a file, this results in unnecessary file input / output operations, which in turn adversely affects the performance of the copying procedure.

Buffer Space

You may sometimes need a temporary buffer to work on image data. Using **copyTile()** instead of **getTile()** or **setTile()** to transfer data between images eliminates the need for temporary buffers, saving you memory. **copyTile()** is explained in "Accessing Image Data" on page 54.

In addition to temporary buffers you may allocate to hold data, the IL allocates buffers to operate on data internally. The amount of buffer space that the IL can allocate at any one time depends on the number of threads running concurrently. If three threads are performing image processing operations on three tiles, in general, three buffers of the necessary sizes must be used. However, extra buffer space is not used if the operator in question is locking down pages, transferring data from input cache to output cache, and operating on the data "in-place." Certain operators derived from ilMonadicImg do this. If you derive a new operator from ilMonadicImg or any of its descendants, you might want to ensure that your derived class operates on its data in-place by setting its **inPlace** member variable in the constructor.

Using Hardware Acceleration

The IL provides a transparent mechanism to accelerate some sequences of operations in an operator chain. It uses specialized graphics devices to implement this hardware acceleration. Depending on your hardware platform, you can improve the performance of your application by understanding how the IL implements hardware acceleration and by choosing operators that take advantage of this feature. Read "Using Graphics Hardware for Acceleration" on page 73 to learn more about hardware acceleration.

Hardware acceleration takes place automatically and whenever possible, without explicit user intervention. This following sections describe these features of hardware acceleration:

- how to control it
- what triggers it
- what operators are accelerated (not all are)
- hardware acceleration on non-RealityEngine platforms
- hardware acceleration on RealityEngine platforms

Initiating Hardware Acceleration

Typically, hardware acceleration is triggered by entering the **copyTileCfg()** method of ilOpImg. Such a copyTile operation occurs when an ilView needs to be repainted. In this case, hardware acceleration helps improve interactive performance of the IL. Hardware acceleration can also be triggered through the **getSubTile3D()** method of ilOpImg. In this case, general performance is improved, including that of non-displayed operations such as the storage of processed image data directly to a file on disk.

Controlling Hardware Acceleration

Sometimes you may want to disable hardware acceleration."Turning off Hardware Acceleration" on page 63 tells you when you might want to do this.

The IL provides functions that allow you select the type of hardware acceleration appropriate to your application. These functions use an enable parameter that specifies the objects for which hardware acceleration is to apply, The enable parameter is an ANDed combination of any of the values shown below:

ilHwNone	All hardware acceleration is disabled.
ilHwAll	All hardware acceleration is enabled (default).
ilHwCopyTile	Hardware acceleration is initiated when a **CopyTile()** operation to the display is issued.
ilHwGetTile	Hardware acceleration is initiated when a **GetTile()** operation is issued.
ilHwAlways	Hardware acceleration is performed even though it is globally disabled.

Disabling Hardware Acceleration Globally

To disable hardware acceleration completely, use the:

- environment variable IL_HW_ACCELERATE
- global convenience function **ilHwAccelerate()** and pass in the enable parameter. The function **ilHwIsEnabled()** returns the enable value set by **ilHwAccelerate()**.

```
ilHwAccelerate(int enable);
int ilHwIsEnabled();
```

The global function takes precedence over the IL_HW_ACCELERATE environment variable.

Disabling Hardware Acceleration for an Operator

To disable hardware acceleration for a specified operator, use the **hwAccelerate()** function that’s defined in ilOpImg and ilImgStat. It accepts one argument—TRUE to enable acceleration on the operator (the default) or FALSE to disable acceleration. For example, to ensure that the operators in the sample program from "A Sample Program in C++" on page 4 are not accelerated in hardware, you can modify the code as follows:

```
ilSharpenImg sharperImg(inImg, 0.5);
ilRotZoomImg rotatedImg(&sharperImg, 90.0);
sharperImg.hwAccelerate(FALSE);
rotatedImg.hwAccelerate(FALSE);
```

Disabling Hardware Acceleration for a Specified Objects

To enable and disable hardware acceleration of objects of a specified class, use the **ilHwAccelerateClass()** function. The **ilHwIsEnabledClass()** function returns the class-specific enable value as set by **ilHwAccelerateClass()**.

```
ilHwAccelerateClass(ilClassId id, int enable);
int ilHwIsEnabledClass(ilClassId id);
```

The following example enables hardware acceleration for all objects of the class ilRotZoomImg.

```
ilHwAccelerateClass(ilClassID(ilRotZoomImg), ilHwCopyTile)
```

Accelerated Operations

To take advantage of the hardware acceleration built into the IL, you must use the operators that are implemented in the hardware you are using. For example, if your program is running on a RealityEngine and you have the choice of using either a Fourier domain operator or an accelerated spatial domain operator to do the same or a similar image processing operation, use the accelerated operator in the chain when quick operation is a priority.

It is possible that some image operations performed by a given IL operator are accelerated while other operations performed by the same operator are not. For example, an ilConvImg can be accelerated on a RealityEngine for a kernel size of 3 x 3, 5 x 5, or 7 x 7, but not for any other kernel size. Table 7-1 indicates which operations are accelerated on different graphics platforms.

Table 7-1 Accelerated Operations by Platform

Image Operator	PI, Starter, Elan	GT, GTX	VGX	Reality-Engine
ilRotZoomImg (integer zoom only, no rotation)[a]	✘	✘	✘	✘
ilRotZoomImg (continuous zoom only, no rotation)[a]			✘	✘
ilBlendImg (constant alpha only)		✘	✘	✘
ilInvertImg, ilAndImg, ilOrImg, and ilXorImg	✘		✘	✘
ilPolyWarpImg (and its descendants)[b]				✘
ilConvImg and ilSepConvImg (and their descendants)[cd]				✘
ilImgStat				✘
ilScaleImg and ilHistScaleImg[d]				✘
ilLutImg (and its descendants)[e]				✘
Color model conversions[f]				✘
Data type conversions[g]				✘

a. Nearest neighbor resampling only.

b. Nearest neighbor, bilinear, or bicubic resampling only.

c. 3 x 3, 5 x 5, or 7 x 7 kernels only. ilPadSrc, ilPadDst, and ilNoPad modes only.

d. Bias value accepted.

e. At most, 4096 entries and 4 components.

f. To / from ilABGR, ilRGB, ilRGBA, ilBGR, ilMinWhite, and ilMinBlack (there is no conversion from color to grayscale).

g. All types except ilDouble.

Hardware Acceleration on Non-RealityEngine Platforms

The IL provides only limited support for image processing operations not running on the VTX or the RealityEngine and the RealityEngine^2. On the PI, Starter, Elan, Extreme, GT, GTX, VGX, and Indy platforms, only the last one or two operations in an image chain can be accelerated in the graphics hardware. These operations include simple zooming using **rectzoom()**, blending using **blendfunction()**, and ALU-based operations using **logicop()** (see Table 7-1).

None of these graphics platforms has the auxiliary buffer required for hardware acceleration. Thus, it is not possible to accelerate sequences of operations that require more than one pixel transfer operation. Moreover, data type and color model conversion are not supported by these graphics platforms. Only the ilABGR color model and ilUChar data type are accelerated.

Hardware Acceleration on the RealityEngine

The RealityEngine is the most sophisticated graphics device supported by the IL for hardware acceleration. The operations implemented in the RealityEngine are shown in Table 7-1. An IL operator that is accelerated on the RealityEngine can be of any IL type except ilDouble, but it must have a color model that is ilABGR, ilRGB, ilRGBA, ilBGR, ilMinWhite, or ilMinBlack. Moreover, pixel format conversion among these color models is supported, except that conversion from three- or four-component pixels to single-component pixels is not supported.

The remainder of this section tells you more about how hardware acceleration on the RealityEngine works with the IL. It describes:

- RealityEngine architecture
- pixel transfer rate on the RealityEngine
- functional path of image processing operations
- overview of hardware acceleration
- the hardware pass
- auxiliary buffers

- auxiliary buffer management on multiple pipes
- texture
- static update

“Suggestions for Further Reading” on page xxi gives a reference for additional reading about the RealityEngine.

RealityEngine Architecture

The VTX, RealityEngine, and RealityEngine^2 graphics systems have essentially the same architecture. This consists of a Geometry Engine board (GE), one to four Raster Manager (RM) boards, and a DisplayGenerator (DG) board. From the perspective of the programmer, only the features of the GE and RM boards are interesting.

A GE board holds a bus interface chip, the Command Processor chip, and six, eight, or twelve programmable geometry processors. In addition to performing normal 3-D graphics operations, these processors are also used as pixel processors for GL pixel operations like **convolve()** and for GL texture loading operations like **subtexload()**.

The Raster Manager boards take descriptions of geometry in screen space from the GE board and rasterize the geometry, possibly with image or texture mapping applied. There are from one to four Raster Boards, which contain the framebuffer and texture memory.

Pixel Transfer Rate on the RealityEngine

Any image processing operation performed by the RealityEngine is limited by the pixel transfer rate of the relevant RealityEngine data path (see Figure 7-2). Most paths have a transfer rate in the range of 5-35 million pixels per second (Mpix/sec). The exception is texture rendering, which moves data from texture memory to the framebuffer at rates of:

- 40 Mpix/sec for 1 RM board and ABGR bicubic resampling
- 80 Mpix/sec for 2 RM boards and luminance bicubic resampling
- 320 Mpix/sec for 4 RM boards with luminance bilinear resampling

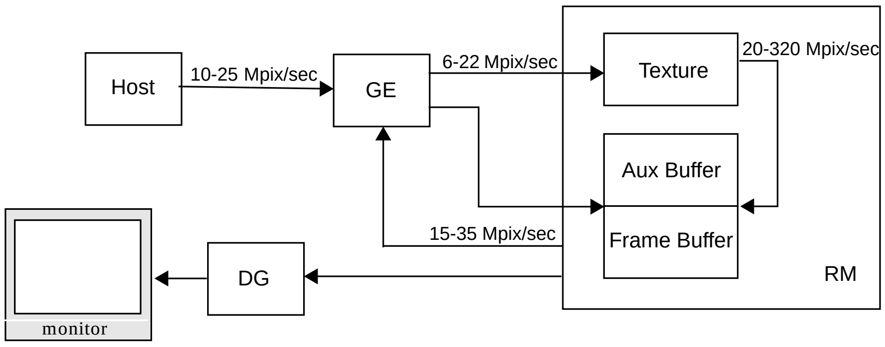


Figure 7-2 RealityEngine Pixel Transfer Paths (transfer rates are in millions of pixels per second)

In order to evaluate the transfer rate of a sequence of image processing operations that are accelerated by the RealityEngine graphics hardware, you should know the paths over which the data travels and the rates for those paths. Figure 7-2 shows the different paths between functional blocks and the variability in data transfer rates. The RealityEngine GL subroutines used by the IL and the paths they access are listed in Table 7-2. Consult the reference pages for these subroutines for additional information.

Note: Unfortunately, it can be difficult to accurately estimate the overall performance of the system for a particular image processing sequence because so many parameters significantly affect these rates, including the number of components per pixel, efficiency at various page sizes, and so forth. If you need very accurate performance estimates, it is best to benchmark the GL operations on the target machine.

Table 7-2 GL Subroutines used by the IL and the Affected Data Path

GL Subroutine	Host to GE	GE to FB/Aux	GE to Texture	Texture to FB/Aux	FB/Aux to GE
bgntmesh, endtmesh, istexloaded				✘	
blendfunction, logicop				✘	
convolve		✘			✘
fbsubtexload			✘		✘
ilbuffer, ildraw, wmpack		✘		✘	
lrectwrite	✘	✘			
pixelmap		✘			
pixeltransfer, pixmode		✘	✘		
readcomponent, readsource					✘
rectcopy		✘			✘
subtexload	✘		✘		
tevbind, texbind	✘		✘		
tevdef, texdef2d	✘				

Functional Path of RealityEngine Image Processing Operations

The RealityEngine has been programmed to perform several important image processing operations as part of its pixel transfer path. That is, when a block of pixels is transferred, either by an **lrectwrite()**, a **rectcopy()**, or a **subtexload()** operation, the sequences of image processing operations shown in Figure 7-3 can be in effect. These sequences of operations constitute what can be computed in a single pass. If the desired sequence of operations cannot be accomplished in a single hardware pass (for example, convolve with a warp as input), the sequence can be divided into two or more subsequences, each of which can be computed in a single pass (see “The Hardware Pass in Detail” on page 324). The RealityEngine auxiliary buffers (see the section, “Auxiliary Buffers” on page 327) provide storage for intermediate results created during a multi-pass operation.

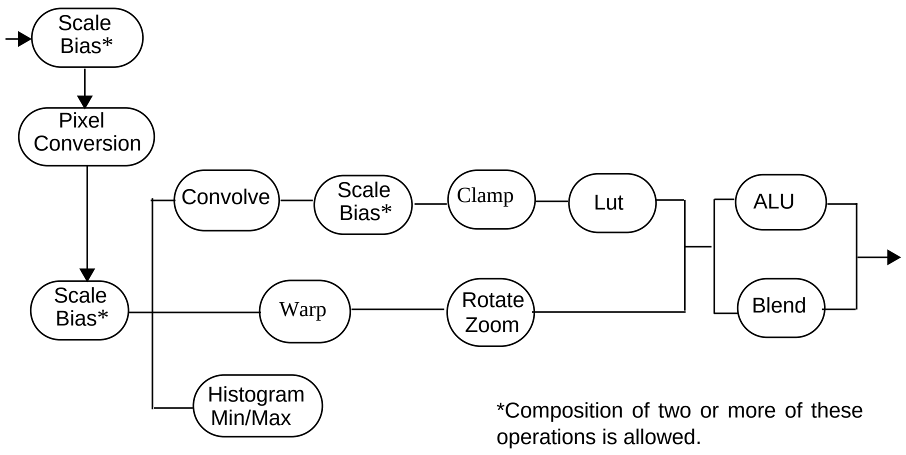


Figure 7-3 Image Processing Operations Accomplished in a Single Pixel Transfer on the RealityEngine

Overview of Hardware Acceleration on the RealityEngine

The goal of hardware acceleration is to efficiently map a sequence of one or more image processing operations onto specialized graphics devices. The IL looks at operations in an image chain to determine where acceleration applies. If the last operator in an image can be accelerated in hardware, it is. The IL then accelerates any operators preceding the last one that can be accelerated along with the last one in the chain.

The IL creates a new object, ilHwPass, to perform an accelerated operation. If two operations can be accelerated together, a single hardware pass performs those two operations. If two adjacent operations the end of a chain can both be accelerated but not combined, two hardware passes are created, one for each operation.

If a pass can be created for an operator, a method is called on that pass to complete the requested operation. The pass will lock the input pages in a manner similar to ilCacheImg and then perform primitive transfer operations such as **rectcopy()** and **lrectwrite()**.

The Hardware Pass in Detail

This section describes how the IL builds a hardware pass that performs one or more IL operations using graphics hardware. The next section, "Functional Path of RealityEngine Image Processing Operations," describes which IL operations can be combined into a single hardware operation and the order in which the operations must occur in the image chain for this combined operation to happen.

The ilHwPass object represents a primitive pixel transfer operation such as an **lrectwrite()**, a **rectcopy()**, or a texture rendering operation that may also perform one or more image processing operations.

The IL hardware pass is designed to achieve hardware acceleration while coping with the following issues:

- Hardware acceleration can occur at any point in an image chain, since a display operation, which initiates the hardware acceleration, can be applied to any operator in the chain.
- A single hardware pass can perform more than one IL operation, since a graphics device can execute more than one image processing operation. An example of this is convolution followed by a look-up table operation. Thus, the mapping of IL operations to the primitive pixel transfer operations performed by the graphics device is many-to-one (that is, many IL operations to one pixel transfer).
- The IL must defer most decisions regarding hardware acceleration until the **copyTileCfg()** call is issued, since the hardware acceleration configuration depends on destination parameters (such as the ilConfig structure) that are not known until then.

When a **copyTileCfg()** is issued on an ilOpImg and the destination is an ilGLDisplayImg, then control is passed from the ilOpImg to the associated ilHwPass to attempt to complete the requested operation. If the ilHwPass is unable to configure itself for the requested operation, then control is returned to **copyTileCfg()** and normal, CPU-based processing ensues.

Initially, an ilOpImg has no hardware pass associated with it and thus it must construct one. The manner by which the pass is constructed is instrumental in achieving the many-to-one mapping of image processing operations to primitive pixel transfer operations mentioned above.

1. The ilOpImg first requests the hardware pass of its input. It then attempts to compose its operation with the ilHwPass of its input to form a new composite pass, which is composed of this ilOpImg and one or more images preceding it in the image chain. For example, in Figure 7-4, the ilLutImg is able to compose its operation with the preceding ilSharpenImg operation to create a composite operation to perform both IL operations. In this figure, ilHwPass2 performs the combined ilSharpenImg and ilLutImg operations. ilHwPass1 performs just the ilSharpenImg operation. This is used only if the result of the ilSharpenImg operation is to be displayed.

.

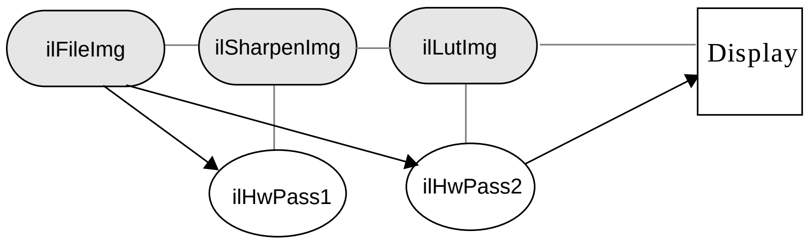


Figure 7-4 Composite Operation

2. If the attempt to create a composite operation fails because the composite operation is not supported by a single pixel transfer, then a new pass is constructed representing the single image processing operation of the ilOpImg in question, and its input is set to be the output of the input pass. The IL must create temporary storage to cache the data between passes. It does this in an auxiliary buffer, which is described in “Auxiliary Buffers” on page 327. Figure 7-5 illustrates this multi-pass operation.

.

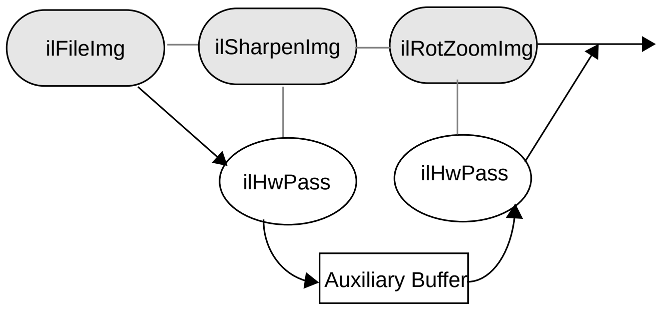


Figure 7-5 Multi-pass Operation

3. If the attempt to create a new pass also fails, most likely because the graphics device does not support the auxiliary framebuffer storage necessary to chain the output of one pass to the input of another, then the input of the pass is set to be the input of the ilOpImg, that is, the result of the unaccelerated input computation.
4. Once constructed, a pass is retained until the ilOpImg is reset, or until the ilOpImg is deleted.

Consider now hardware acceleration performed on an image chain similar to that of the sample program from Chapter 1, “Writing an ImageVision Library Program.” Figure 7-6 contains a diagram of the image chain. For simplicity, this chain does not write its result back to disk.

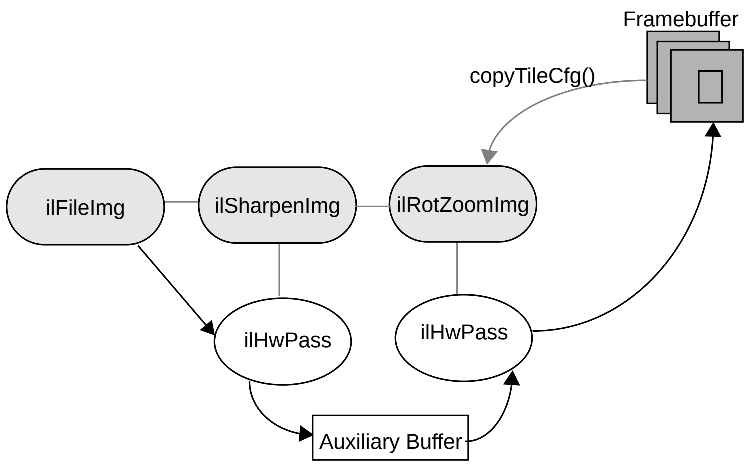


Figure 7-6 Hardware Acceleration on an Image Chain

Auxiliary Buffers

The RealityEngine includes one or more auxiliary buffers that the IL uses to cache image data between hardware passes. An auxiliary buffer is reserved space in the framebuffer. All IL processes share auxiliary buffer space, which restricts other rendering functionality since the auxiliary buffer cannot be moved or resized once it is reserved. If a non-IL application reserves the space needed for the auxiliary buffer first (for example, to do z-buffering or multi-sampling), any IL application that tries to reserve it is locked out. This restricts hardware acceleration to single pass operations. An analogous situation occurs if an IL application reserves the space in the framebuffer first. This lockout ensures that the data in the auxiliary buffer is not overwritten by any concurrently running applications.

Figure 7-7 shows the use of an auxiliary buffer to implement the creation of a composite pass that performs the ilSharpenImg and ilLutImg operations. In this diagram, dashed lines show linkages in the image chain, the black lines show data flow into the hardware passes, heavy gray lines represent input to images.

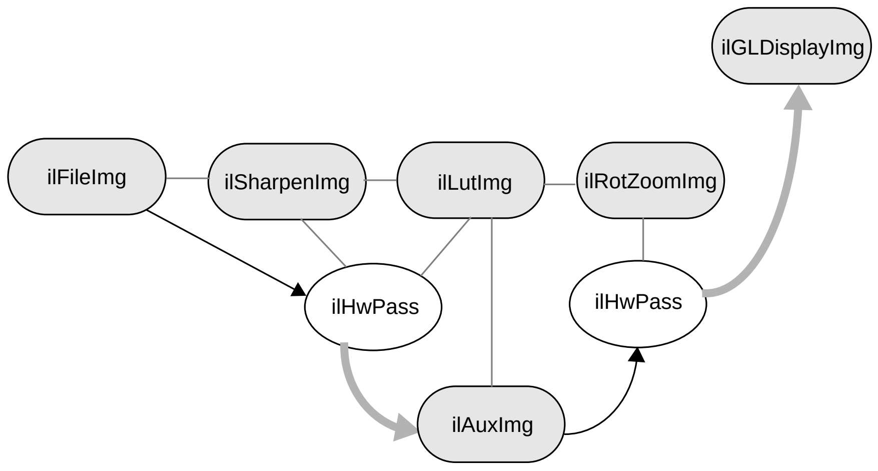


Figure 7-7 New Chain Constructed with Hardware Passes

Data is stored in the auxiliary buffer as 12 bits per component and up to 4 components per pixel. This image data is held in an ilAuxImg object, which is an IL image that is used internally and not exposed for external use. An ilAuxImg object resides in the auxiliary buffers and holds intermediate results for multi-pass operations. It has a page size of 128 x 128 pixels (except in static update, see “Static Update” on page 331). An ilAuxImg is similar to an ilMemCacheImg except that the total number of pages is much more limited and the pages reside in the framebuffer. As a rule, pages of an ilAuxImg remain unlocked except during a computation. However, pages of an ilAuxImg that have been loaded into a texture must remain locked since IRIS GL requires them to be available for reloading at any time. When a page allocation fails because all of the pages in the auxiliary buffers are locked, the operation is aborted and the equivalent non-accelerated operation is attempted instead.

The auxiliary buffers are managed as a shared global resource among all IL-based processes through a global arena. The total number of auxiliary buffers available to the IL depends on the number of Raster Managers (RMs) installed in the RealityEngine, the current video format, and whether other non-IL applications have already reserved some of the auxiliary buffers. The first IL process that attempts to allocate an ilAuxImg automatically initializes the global arena used for mediating auxiliary buffer access and reserves some of the auxiliary buffers for exclusive IL use. The environment variable IL_AUX_BUFFERS limits the number of auxiliary buffers reserved for use by the IL. Subsequent IL processes simply join the existing global arena.

The global arena exists as long as at least one IL process is attached to it. When the last IL process attached to the global arena terminates, the arena is destroyed. If an IL process terminates prematurely, it is possible that auxiliary buffer pages locked by that process will remain locked indefinitely. The only way to correct this situation is to destroy the arena by terminating all IL processes attached to the arena. The shared global arena is located in */var/tmp/.ilAuxBufferArena*. This file can safely be deleted if no IL processes are executing.

Auxiliary Buffer Management on Multiple Pipes

Systems that have more than one RealityEngine subsystem also have more than one disjoint set of auxiliary buffers. Each pass that writes its output to an ilAuxImg has a copy of the image in each of the sets of auxiliary buffers. The auxiliary buffer cache is not shared across subsystems because it is impractical to copy image data between them.

Texture and Its Limitations

Polynomial warp operations are accelerated on the RealityEngine by exploiting its texture rendering facility. Pages of the input image are loaded into texture memory using the GL call **subtexload()** or **fbsubtexload()** if the input is an ilAuxImg. The desired geometric transformation is applied to the texture and the resulting transformed image is rendered using **v2f()** and **v2i()**. The texture is limited in size to 1024 x 1024 pixels for bilinear and nearest neighbor interpolation, and to 512 x 512 for bicubic interpolation. If the input image is larger than this size,[a] only a subportion of the image can be held in texture memory at any given time. If the image is smaller than this

[a] Actually a one page border of fill value must also be loaded into texture, so the actual limit is the texture size minus one page width.

critical size, then it can be loaded in its entirety and rendered rapidly. However, larger images must be paged and are therefore limited by the **subtexload()** speed, which is substantially less than the rendering speed.

To optimize the texture's efficiency, the IL constrains warps and rotates / zooms in the RealityEngine to a page size of 64 x 64.

Image data is stored internally in texture in a variety of formats, depending on the operator color model and resampling method. See Table 7-3 for the detail about internal formats.

Table 7-3 Texture Image Internal Formats

	Resampling	Type	
Color Model	**ilBicubic**	**ilBilinear/ilNewNb**	
ilRGB	TX-RGB-12	TX_RGB-12	MinBits>8
ilBGR		TX_RGB_8	5<MinBits<=8
		TX_RGB_5	otherwise
ilABGR	TX_RGBA_12	TX_RGBA_12	MinBits>8
ilRGBA		TX_RGBA_8	4<MinBits<=8
		TX_RGBA_4	otherwise
ilMinWhite	TX_I_12A_4	TX_I_12A_4	

See the *texdef* reference pages for more information about internal formats. Table 7-3 shows the texture sizes associated with the various internal formats.

Table 7-4 Texture Size for Internal Formats

Internal Format	Texture Size[a]
TX_I_12A_4	1024 x 1024
TX_RGB_5	1024 x 1024
TX_RGBA_4	1024 x 1024
TX_RGBA_8	1024 x 512 or 512 x 1024
TX_RGB-12	512 x 512
TX_RGBA_12	512 x 512

a. Double these values for RM5.

You can use the **setMinComponentBits()** method of ilWarpImg to set the minimum number of bits per component in the internal texture format. In this way, you can influence the choice of internal format and consequent size and rendering of the texture.

Static Update

An ilAuxImg is made up of pages so that as you roam over a displayed image, the requisite intermediate ilAuxImg pages can be loaded into the auxiliary buffers and then later reclaimed. However, when the ilAuxImg is reset, any pages that it has cached in the auxiliary buffers are discarded. Therefore, if an operator chain is being repeatedly reset, perhaps because an operator parameter is being interactively varied, then the ilAuxImg pages are repeatedly being discarded and recomputed, incurring a fair amount of overhead. In order to obtain peak throughput for non-roaming iterative reset, a non-paged processing mode called “Static Update” is automatically activated.

An ilView can distinguish between normal roaming and static update by checking if a reset has just occurred. If so, the ilView passes a static update “hint” to **copyTileCfg()** and ultimately to the hardware acceleration logic. When in static update mode, the page size and origin of the ilAuxImg are changed to coincide with the size and position of the requested tile. The static update hint is also propagated to the input of the pass and therefore applies to all passes in a multi-pass operation. However, static update is not propagated to the input of an accelerated warp operation because of the constraints to texture input.

It is possible to disable and also to force static update using the **setStaticUpdate()** and **setAutoStaticUpdate()** methods of ilView. **isStaticUpdate()** checks the status of the static update mode.

```
isStaticUpdate();
setStaticUpdate(int enable);
setAutoStaticUpdate(int enable);
```

Using a Dedicated GL Rendering Thread

Your IL application can use a dedicated thread for GL access to maximize interactive performance on the RealityEngine. In the multi-threaded IL environment, only one thread at a time can be used for GL rendering. This limitation exists because the GL is not mp-safe. The IL normally uses spinlocks to insure that only one thread at a time accesses the GL. This method switches GL rendering from one thread to another, which incurs a high overhead. Having a dedicated thread for the GL avoids this overhead.

Figure 7-8 shows how the dedicated GL thread works. Client threads (any thread except the dedicated GL thread) place requests for GL rendering in the input queue. The dedicated thread removes an entry from this queue, performs the rendering operation, and places the request in an output queue. Client threads check the output queue for completed requests while they are waiting for "wait" criterion to be satisfied and delete any completed requests they find.

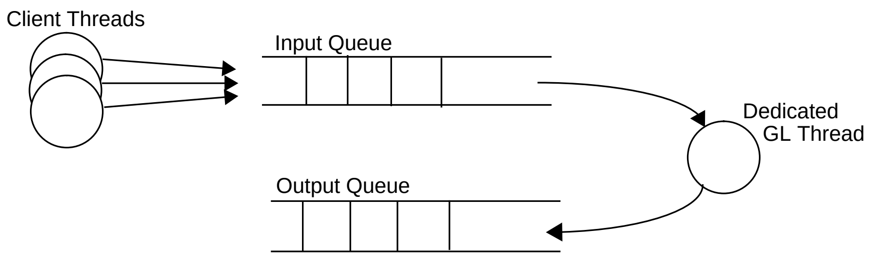


Figure 7-8 Dedicated Il Rendering Thread

Controlling the Rendering Thread

There are two ways to control the dedicated IL rendering thread.

1. Use the global call **ilHwThreadEnable()** to enable and disable the dedicated rendering thread. TRUE enables the thread, FALSE disables it. The GL rendering thread is disabled by default.

   ```
   ilHwThreadEnable(int enable);
   ```

2. Use the global calls **ilHwThreadSuspend()** and **ilHwThreadResume()** to temporarily suspend and resume a dedicated IL rendering thread. **ilHwThreadSuspend()** suspends the thread if it is active and does nothing if there is no active dedicated thread. Calls to suspend and resume can be nested. The last matching resume reactivates the thread if it was active prior to the first suspend. Do not suspend the dedicated IL thread while it is in the process of GL rendering.

It is imperative that the rendering thread not be forced to sleep due to mp contention or insufficient number of processors. (The minimum sleep interval is 10ms or ~2/3 of a 60Hz frame.) This implies that the default number of IL threads (number of processors +1) is probably too many if the rendering thread is active. It is recommended that you set the total number of IL threads to the number of processors - 1 when using the rendering thread.

Using the Rendering Thread

The ilHwRequest class allows you to create requests for the GL rendering thread. The constructor for the class is:

```
ilHwRequest(ilDisplay* Display);
```

Objects of class ilHwRequest are queued for execution by the rendering thread through the static member function ilHwRequest::**dispatch()**. Rendering requests that are posted in this manner are processed on a first-come first-served basis.

```
void dispatch(ilHwRequest* req, int (*wait)() = NULL);
```

The client threads wait inside the **dispatch()** function until the “wait” criterion (that is passed as a callback function) is satisfied. The default wait action is to return immediately. However, you can use this wait criterion to regulate the flow of requests pending to be executed. For instance, you may want to limit the number of pending window refreshes to be less than some constant number of frames so that there is not a significant lag between user actions and display response. In this case, you can dispatch a request once per frame that increments a counter, and then decrements it when the request is destroyed. The callback can test this counter against the limiting value and return TRUE if the number of pending display refreshes exceeds the threshold.

The ilHwRequest class defines the static function **flush()** that inserts a dummy request in the queue and waits for it to be executed. This effectively flushes all requests dispatched prior to the **flush()**.

```
static void flush();
```

The requests are deleted after they are executed; so they should be allocated using new and then forgotten after being dispatched. They are deleted by the threads that are forced to wait for the queue to drain during **dispatch()**, rather than by the rendering thread. This keeps mp contention of the rendering thread to a minimum.

Mixing Application Rendering with the IL Rendering Thread

If the application makes GL calls directly (i.e. not through the IL), it must synchronize with the rendering thread since GL is not mp safe. There are several ways to do this.

1. Temporarily suspend the rendering thread by calling **ilHwThreadSuspend()**.
2. Queue application rendering that is to be executed by the rendering thread in one of two ways:
    - Derive a new subclass from the base class ilHwRequest. Override the virtual **exec()**, which gets called by the rendering thread when the request is taken off the queue.
    - Queue a callback using the IL global **ilHwCallback()**. The queued callback function is called by the rendering thread when it is taken off the queue.

Restrictions on the exec() virtual

The hardware acceleration logic avoids redundant calls of costly GL routines by filtering these calls through a global GL state object. If an **exec()** method makes direct GL calls, it may confuse the IL's picture of the current GL state. (If you suspend the rendering thread using **ilHwThreadSuspend(),** then this restriction does not apply because the GL state is forgotten when the thread is resumed.)

To get a pointer to the global GL state object, call ilHwState::**getGLState()**. Table 7-5 provides a provisional list of the protected GL calls, and how to call them, if it is required.

Table 7-5 ilHwState Member Functions Used to Make GL Call

GL Call	ilHwState Member Function
winset	ilHwState::setDest()
scrmask, ortho, viewport, mmode, etc	bracket with GL pushviewport and popviewport calls
pixmode	ilHwState::setup(), ilHwState::doStride(), ilHwState::setFlip()
cpack	ilHwState::doCpack()
wmpack	ilHwState::doWmpack()
readsource, readcomponent	ilHwState::auxRead()

Also, certain IL calls during the **exec()** method can trigger deadlock. In particular, get methods on ilLink-derived objects usually call **resetCheck()** and possibly **reset()**. In general, it is a good practice to avoid **reset()** during the **exec()** virtual. In particular, it is an error to call ilGLDisplayImg::**reset()** during the **exec()** virtual.

On a related note, an IL object can go through **reset()** one or more times between the time that the request is posted and the time that it is executed. Therefore, you cannot rely on the parameters of the object to remain unchanged. Moreover, cache pages will be invalid, unless they are locked.

A well-designed ilHwRequest object caches all requisite parameters in its constructor (or at least prior to dispatch) and locks any input pages and other ancillary objects. The **exec()** virtual itself should do very little, other than the necessary GL calls. In the destructor, unlock the input pages, etc.

Controlling Multi-threading

Typically, the IL automatically uses multi-threading to optimize common functions (like operating on pages of data). This is explained in "Multi-threading" on page 71. In addition, you can explicitly control or query some of the classes used in the IL's multi-threading facility:

ilSemaphore	a several-thread "lock" that limits the number of process threads that may simultaneously access some shared data structure or resource.
ilSpinLock	a single-thread lock that limits simultaneous access to some shared data structure or resource to one thread at a time.
ilArena	an area of CPU memory shared by multiple processes (threads); the IL allocates semaphores and spinlocks from an arena.

These classes are described in more detail in the following sections. In addition, several other classes work with those above to compose the multi-threading facility, but you can't typically control them to your advantage. They're briefly described below for your convenience:

ilThread	an execution thread. A dispatcher assigns requests from the queue to threads. A thread blocks if it is waiting for a request to complete or if there are no requests in the queue.
ilDispatcher	an abstract class (derived to work with particular operators) that dispatches requests to a global request queue and spawns threads as needed to service the queue.
ilRequest	a request for an I/O operation or computation that is dispatched to the request queue and which is eventually completed by a thread.

Controlling Threads

Internally, the IL keeps track of how many threads can be used for CPU computations. This is usually equal to the number of processors on the machine, but you can control or limit the number of processors the IL uses by setting the environment variable IL_COMPUTE_THREADS to the maximum number of threads you want to apply. (IL_COMPUTE_THREADS should never be more than the number of processors or the performance of the IL application will suffer.) The IL also keeps track of how many additional “spare” threads it has for use in I/O operations. This is usually equal to zero, but you can change it by setting the environment variable IL_SPARE_THREADS. (The application that you are running counts as an implicit “spare” thread.) Alternatively, you can set both internal values at once with the global function **ilMpSetMaxProcs()**:

```
ilSetMaxProcs(int compute, int spare);
```

You must call this function before issuing any request that uses the multi-processing capabilities; in other words, you cannot retroactively change the number of allowed threads. Making the number of CPU and spare threads both equal to zero disables multi-threading. You will run out of space in the arena if you allow too many threads. By default, the maximum limit set by the IL is 40 threads. You can increase this limit by setting the environment variable, IL_ARENA_MAXUSERS, to a larger value.

It’s also possible to control the use of multi-threading on individual operators using the **enableMP()** method on ilCacheImg:

```
void enableMP(int on=TRUE)
```

Calling this method with *on* set to FALSE will prevent the object from issuing concurrent **getPage()** requests to satisfy its tile requests.

Semaphores and Locks

To limit access by concurrently running threads to a shared data structure or resource, use an ilSemaphore or ilSpinLock. The ilSpinLock class is preferred for short duration locking; ilSemaphore is recommended for operations that may take some time. For example, the IL uses a semaphore to limit the number of concurrent executions of the computational part of an operator to the number of available processors. On the other hand, IL uses an ilSpinLock to prevent concurrent access to the caching data structures.

Controlling Arenas

An ilArena is an area of shared memory from which semaphores, locks, and threads are created. Only a limited number of threads can share an arena, and this number is set when the ilArena is created:

```
ilArena(int maxUsers = 24);
```

ilArena’s other member function, **getHandle()**, takes no arguments and returns a **void*** that represents the pointer to the arena. This value is NULL if multi-threading is not supported. (You can use this function to determine if the IL can use its multi-threading facilities on your system.)

Note: Linking an IL application with other libraries that create arenas can cause an address space collision. To prevent such collisions, set the environment variable IL_ARENA_ADDRESS to a value that will not overlap the address space created by the arena of another library.

Interaction with Multi-threaded Applications

You can use the IL with multi-threaded applications provided your application follows these guidelines:

- If you create threads, don’t destroy them unless you disable IL’s use of the prctl(PR_SETEXITSIG, SIGHUP) call. IL uses this call to force all threads to be terminated when an IL-based program encounters an error or upon normal termination. If you override the default behavior of the IL with respect to the child threads, take care to clean up these threads on process termination. If you use ilThread to create your threads, its destructor will safely terminate just that thread.
- Don’t issue calls to alter the chain in any way while a **getTile()**, **setTile()**, **copyTile()**, or **fillTile()** request is in progress in another thread. Unpredictable results will occur if you do so, as the alteration methods are not safe for multi-threading.

- Issuing multiple tile requests concurrently is safe for multi-threading.
- You will probably want to use **ilMpSetMaxProcs()** to limit IL's use of threads, if you create your own threads, to avoid excessive thrashing of CPU resources. You can also use the **enableMP()** method on ilCacheImg to turn off multi-threading on individual operators and file images.

Chapter 8

The Programming Environment

This chapter describes the programming environment in which your IL application runs and tools you can use to write, compile, and debug your program.

Chapter 8

The Programming Environment

This chapter provides information on the programming environment available on Silicon Graphics workstations. Special tools that may help you in writing, compiling, and debugging your IL program are discussed.

This chapter contains the following major sections:

- "Compiling and Linking an IL Program" on page 344 describes what you need to do to compile an IL program written in C++, C, or Fortran.
- "Reading the Reference Pages" on page 348 explains how to read the class reference pages. These reference pages don't follow the standard UNIX reference page format.
- "Debugging an IL C++ Program" on page 349 briefly mentions how to debug your program.
- "Image Tools" on page 351 describes some image tools that were developed using the IL.
- "Online Source Code" on page 352 describes the IL-related code that's available online.

Compiling and Linking an IL Program

The following sections show you how to compile and link IL programs written in C++, C, or Fortran.

Programs Written in C++

To compile an IL program written in C++, use the following command line:

```
CC -g sample.c++ -o sample -lil
```

Libraries that you must link to include the IL library itself. (See the *CC* reference page for more information about the C++ compiler.)

By default, the **.so* libraries are used to link your programs. In general, you should not use the static, or **.a*, libraries unless you want to keep your application in one complete binary. If you do choose to use the static libraries, your command must be as follows:

```
CC -g sample.c++ -o sample /usr/lib/libil.a -lgl -ldl -lm
```

If you are using the static libraries, the libraries you must link to include the IL library itself, the GL shared library, the X Window library (if you are creating a mixed-model X and GL application), the math library, the C++ library, the multiprocessing library, and the dynamic linking library.

A Sample Makefile

Here's a sample Makefile for compiling IL programs:

```
# Makefile for IL test programs

SHELL = /bin/sh
# If you want to debug,turn on the “-g” option.
FLAGS = -g

MAINS= sample.c++
OBJS = ${FILES:.c++=.o}
PROGS = ${MAINS:.c++=}

LIBS = -lil

.c++:
        CC $(FLAGS) $< -o $@ $(LIBS)
```

```
.c++.o:
        CC $(FLAGS) -c $<

clean:
        rm -rf $(OBJS) $(PROGS)
        rm -rf core
```

Image File Format Libraries

The image file formats are in their own libraries, stored in */usr/lib/ImageVision/filefmt*. If your program explicitly creates an object of type ilTIFFImg, ilPCDImg, ilFITImg, or ilSGIImg (as opposed to using **ilCreateImgFile()**), you must link with the corresponding library.

In the compile line, you must add:

```
-L/usr/lib/ImageVision/filefmt -lilFMT
```

where *FMT* is TIFF, FIT, PCD, or SGI.

You also need to set LD-LIBRARY_PATH at runtime.

Linking with Libraries in Other Languages

If you program in C++, you'll probably want to link with object files and libraries written in languages other than C++, especially C. In order to do so, you must include in your program declarations for the functions you wish to call. In most cases, you can do this by including appropriate header files with the #include directive. For the standard C header files supplied by Silicon Graphics, using #include is all you need to do. For example, if you are going to use C standard I/O and the Graphics Library, write:

```
#include <stdio.h>
#include <gl/gl.h>
```

If you want to call C functions from within a C++ program, either directly or by file inclusion, make sure that the C++ program contains correctly prototyped declarations for the functions. Also, the function declarations need to be recognizable by the C++ translator as declaring functions whose definitions are in C.

These steps are necessary because C++ normally encodes function names to support overloading. For example, the real name of a function declared in a C++ program as:

```
 void printf(char*, ...)           is         printf__FPce.
```

The **printf()** function in *libc.so*, however, is called **printf**. To allow a C++ program to call functions written in C, C++ provides linkage specifications. To use the standard **printf()** function, for example, write:

```
extern "C" {
   void printf(char *, ...);
}
```

within the C++ source file that calls **printf()**, or within a header file that is included by the source file. The *extern C* statement tells the translator that the function linkage should be done according to the conventions used by the C programming language.

If you want to adapt an existing C header file or create a header file of your own containing C function declarations, and you want to be able to include it in either C or C++ programs, you can use the symbol *__cplusplus* (with two underscores preceding it). *__cplusplus* is always defined for C++ compilations and is otherwise undefined. Thus, you can enclose C function declarations with:

```
#ifdef __cplusplus
extern "C" {
#endif
```

and

```
#ifdef __cplusplus
}
#endif
```

This scheme is used to create the C and Fortran interfaces to the IL.

Programs Written in C or Fortran

Link your C object files to the *libcil.so* library, the C version of the IL. For example, to compile a C program called *ctest.c*, use this line:

```
cc -g ctest.c -o carprot -lcil
```

The IL is compatible with ANSI C. To use the older, pre-ANSI dialect, add **–cckr** to the command line. Ignore any warnings generated during compilation.

Link your Fortran programs to the Fortran version of the IL—*libfil.so*. The Fortran compilation line looks like:

```
f77 -g ftest.f -o ftest -lfil
```

See the *C* and *f77* reference pages for more information about the C and Fortran compilers.

A Sample Makefile

Here is how you might write a short *Makefile* to compile IL programs:

```
# A very simple Makefile for IL test programs

SHELL = /bin/sh
FLAGS = -g

CMAINS = csample.c
COBJS = ${CMAINS:.c=.o}
CPROGS = ${CMAINS:.c=}
CLIBS = -lcil

FMAINS = fsample.f
FOBJS = ${FMAINS:.f=.o}
FROGS = ${FMAINS=.f=}
FLIBS = -lfil

.f:
    f77 $(FLAGS) $< -o $@ $(FLIBS)

.c:
    cc $(FLAGS) $< -o $@ $(CLIBS)

clean:
    rm -rf $(COBJS) $(FOBJS)
    rm -rf core

clobber: clean
    rm $(FROGS) $(CPROGS)
```

Reading the Reference Pages

The IL reference pages don’t look like typical reference pages, since they’re *class* reference pages. They’re available online by typing *man ilClassName* in a shell window. (A printed version of the reference pages is available as an option; see the Introduction for ordering information.)

The C++, C, and Fortran versions of the class reference pages share a similar format; the main sections of each reference page are described below:

Name	The class name and a one-line description of the class.
Inherits From	A colon-separated list of superclasses, beginning with the base class.
Header File	The class’s header file.
Class Description	Describes how the class fits into the IL and how to use it. This section briefly mentions the most important functions associated with the class. The C++ version also contains information about deriving from the class, if appropriate.
Class Member Function Summary	Lists the prototypes of the functions associated with the class. They’re grouped functionally with headings that indicate the general task they perform. Functions that are protected are identified as such. This section should be a synopsis of the class.
Function Descriptions	Describes what each function does and what its arguments mean; sometimes code examples are included. This section is arranged alphabetically so that you can easily find the description of a particular function of interest.
Inherited Member Functions	Alphabetical list of the functions inherited from superclasses.
See Also	Other reference pages of interest.
Notes (optional)	Special information about the class.

Debugging an IL C++ Program

This section gives an overview of how you can debug your IL program by using *dbx*. It may be easier to debug your program if you use environment variables to turn off the multi-threading facility of the IL before compiling a program for debugging. You may also need to turn off the hardware acceleration facility. See “Multi-threading” on page 71 and “Using Graphics Hardware for Acceleration” on page 73 for information about how to turn off these facilities.

Compiling for Debugging

You can debug IL C++ programs with *dbx*. Compile and link with the **–g** option for best results; **–g** must be specified to allow the examination of local variables. If you don’t compile with **–g**, you can still set breakpoints, and function names will be recognized, but variable names won’t. Compiling with **–g** also disables most optimizations.

See the *dbx Reference Manual* for a detailed description of the debugger.

Referring to Function Names

The most important thing you need to know when debugging C++ programs with *dbx* is how to refer to functions and data members:

- Member functions. Refer to these as *classname::functionname*. For example, to set a breakpoint in class C’s member function **f()**, type:

  ```
  stop in C::f
  ```

  If there is more than one member function named **f()**, this command will set a breakpoint in every such function. (However, you can’t set a breakpoint in an in-line function.)
- Global C++ functions. Refer to these as *::functionname*. For example, to set a breakpoint in the global function **f()**, type:

  ```
  stop in ::f
  ```
- Non-C++ functions. Refer to these as *functionname*. For example, to set a breakpoint in **printf()**, type:

```
stop in printf
```

- Data members. You cannot refer to a data member by its name alone, even if the program is stopped in a member function. To refer to data member *m*, use *this–>m*.

The following example illustrates various possibilities:

```
#include <stdio.h>

class foo {
   int n;
public:
   foo() {n = 0;}// this is an inline function
   foo(int x);
   int bar();
   int bar(int);
};

int foo:: bar()
{
   return n;
}

int foo:: bar(int x)
{
   return n + x;
}

foo::foo(int x)
{
   n = x;
}

int square(int x) // this is a global function
{
   return x * x;
}

main()
{
   foo a;
   foo b = 11;
   int x = a.bar();
   int y = b.bar(x) + square(x);
   printf("y = %d\n", y);
}
```

If you type:

```
stop in foo::foo
```

execution will stop in the constructor for the variable *b* but not in the constructor for the variable *a* because you cannot set a breakpoint by name in an in-line function.

If you type:

```
stop in foo::bar
```

execution will stop both when *a.bar* is called and when *b.bar* is called because the debugger is unable to distinguish between the overloaded functions.

To stop in *square*, type:

```
stop in ::square
```

To stop in *printf* (a C function), type:

```
stop in printf
```

Image Tools

Several useful utilities are provided for displaying, copying, and manipulating images. These image tools are based on the IL and therefore support TIFF, SGI, PCD (Photo CD), PCDO, GIF, and FIT file formats. They are installed in */usr/sbin*, and most of them are documented in the *IRIS Utilities User's Guide*. (They also have reference pages.)

imgcopy	*Image Copy*. Copies a specified region of an input image file to an output image file. It can also be used to convert between IL-supported file formats. See the *imgcopy* reference page.
imginfo	*Image Info*. Reports image information such as size, data type, color model, and file format for any IL-supported file format. See the *imginfo* reference page.

imgview	*Image View*. Allows any combination of IL-supported image files to be displayed and manipulated. The images may be roamed, dragged, cropped, or wiped separately or simultaneously. See the *imgview* reference page.
imgworks	*Image Works*. Provides a graphical user interface for manipulating images. Images can be brightened, darkened, histogram-equalized, thresholded, zoomed, rotated, flipped, sharpened, and blurred. See the *imgworks* reference page.
imgformats	*Image Format*s. Lists all the IL-compatible formats currently installed.
ilmonitor	*IL monitor*. A graphical tool that monitors image objects and the image data cache and dynamically alters global cache and some image attributes.

Online Source Code

To provide you with source code examples, the IL installs several directories in */usr/people/4Dgifts/examples/ImageVision*, as described below:

- *ilguide* contains the whole-program examples presented in this guide. They're provided so that you can compile and run them as you read the relevant discussion in the guide.
- *ilapps* contains sample IL applications such as *imgcopy* and *imgview* (which are described above) as well as several others. These applications serve as examples of how to program with the IL and serve as possible templates for developing new applications.
- *ilsrc* contains IL source code that may be of use if you extend the IL by deriving your own classes. It includes the source for the FIT file format, the ilViewer class, and several operators. You might want to examine the corresponding header files, which are in */usr/include/il*.
- *iltutorial* contains a series of programs that build on one another. The first in the series (*ex0.c++*) simply opens and displays an IL image file. The other programs use various operators and display techniques.

You can examine the *README* files in the various directories for more information on each of the code examples. Also, each of the directories containing whole programs has an appropriate *Makefile*; to compile any of the programs, simply type:

```
make <program name>
```

where <*program name*> is the name of the file minus its *.c++* suffix.

Appendix A

Introduction to C++

This chapter introduces the basic concepts of programming in C++. It briefly covers the principal concepts that differentiate C++ from non-object-oriented languages. Rather than providing a definitive overview, it gives C and Fortran programmers a basic grasp of the C++ concepts and phrases that are occasionally used in this guide. If it has the side benefit of piquing the interest of C and Fortran programmers enough to give C++ a try, so much the better. One primary benefit of programming in C++ is that you can extend the IL as you wish—for example, to include support for your file format or for an image processing algorithm.

Objects and Classes

If you know that C++ is an object-oriented language, you correctly assume that objects play a major role in a C++ program. An *object* is an instance of a C++ *class*. A class is a fancy data type that defines not only data elements as in a data structure but functions that manipulate those data elements. These data elements are called the class's *data members*, since they *belong* to the class; similarly, the functions that manipulate the data members are called *member functions*.

One key member function is the *constructor*, which contains instructions about how to create a class object. Typically, the constructor initializes the values of the data members. The class *destructor* deallocates the class object. In C++, you can have the compiler automatically create objects for you:

```
goodClass myGoodClass(anArg);
```

This statement defines the variable *myGoodClass* as being an instance of the class goodClass; it invokes the goodClass constructor to create *myGoodClass*, passing in the variable *anArg* as an argument to the constructor. Since storage is allocated for *myGoodClass*, you can now invoke any of its member functions:

```
myGoodClass.doItNow(someArg, anotherArg);
```

This statement invokes the **doItNow()** function, explicitly passing in two arguments and implicitly passing the data elements of *myGoodClass*. Note the use of the *dot operator* (“.”) to access the **doItNow()** member function of the goodClass. You can use this operator to access either a data member or a member function of a class object. Since the *myGoodClass* object is created automatically, it is also deleted (its storage freed) automatically.

You can explicitly create an object as shown below:

```
goodClass* myGoodClassP = new goodClass(anArg);
```

Here, the goodClass constructor is explicitly called with *anArg* as the argument; note that the constructor has the same name as the class and that it returns a pointer to the class object. So, instead of a class object, you now have a pointer to a class. In this case, to access one of its members, you have to use the *arrow operator* (“->”):

```
myGoodClass->doItNow(someArg, anotherArg);
```

Since you’ve explicitly created the *myGoodClassP* object, it won’t be automatically deleted. You have to do this yourself:

```
delete myGoodClassP;
```

This statement calls the goodClass destructor to delete the object.

Inheritance

Classes can *inherit* selected data members and member functions from other classes; inherited members are available for use by a class just as though they were defined in the class itself. A class that inherits members from another is said to *derive from* that class. Thus, classes exist in an *inheritance hierarchy.* As shown in the inheritance hierarchy in Figure A-1, bestClass inherits from betterClass, which itself inherits from goodClass.

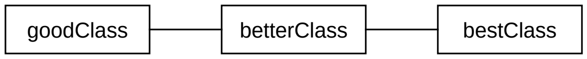


Figure A-1 Sample Inheritance Hierarchy

In this example, betterClass is “better” since it inherits members from goodClass and also defines its own; similarly, bestClass inherits members from goodClass and betterClass, and it defines its own. The root of a hierarchy is called the *base class*—in this example, the base class is goodClass. Typically, the base class has several *subclasses* that derive from it; it defines general capabilities common to every class in the hierarchy. A subclass then adds definitions of whatever members it needs to implement in order to provide its specific functionality.

A *superclass* can declare a member function as *virtual*, giving a subclass the opportunity to provide its own definition of that function. In some cases, virtual functions are simply declared but not implemented at all in a superclass. These are called *pure virtual* functions, and they must be overridden by a subclass’s own version. You can’t create an object of a class that contains pure virtual functions; such a class is called an *abstract* class.

Note: If a superclass declares a member function to be virtual, any of its subclasses may define its own definition of that function, including any subclass that is indirectly descended from the superclass through another subclass that has provided its own definition of that function.

Public versus Protected versus Private

A class can’t use all of its superclass’s members. Some of a class’s members are declared *private*, and they’re available for use only by the member functions of that class. Other members are declared *protected*, and these are available for use by derived classes. Yet other members are declared *public*, and they’re accessible anywhere in the program.

Passing by Reference

The C++ language allows variables to be passed *by reference* (as Fortran does). For example, here's the declaration of a query function **getAttribute()**, which returns an attribute's value by reference:

```
void getAttribute(int& val);
```

Here's how you use this function:

```
int x;
myGoodClass.getAttribute(x);
```

It looks like **getAttribute()** is taking the variable itself, but behind the scenes, C++ actually passes a pointer to *x*.

Default Values

Another handy thing C++ allows you to do is to specify default values for a function's arguments. You do this when you declare the function:

```
void thisFunction(int arg1, int arg2 = 5);
```

Subsequently, you can call **thisFunction()** without explicitly specifying the second argument:

```
myGoodClass.thisFunction(3);
```

This statement invokes the function, passing in 3 as the first argument and 5 as the second. Additionally, you can specify whatever value you wish for the second argument instead of relying on the default, as shown below:

```
myGoodClass.thisFunction(3, 7);
```

Summary of All Classes

This appendix lists all the classes that make up the IL. Each of these classes has its own reference page. Convenience functions that don't belong to any particular class are also listed here. These functions have reference pages as well.

Class or Function	Description
ilABGRImg	Converts to the ABGR color model
ilAbsImg	Computes the pixelwise absolute value of an image
ilAddImg	Computes the pixelwise addition of two images
ilAndImg	Computes the pixelwise logical AND of two images
ilArena	Defines an area of CPU memory shared by multiple threads
ilArithLutImg	Performs a generalized arithmetic operation using a look-up table
ilBGRImg	Converts to the BGR color model
ilBitArray	Provides a limited subscriptable bit array
ilBitMapRoi	Defines a bitmap-based region of interest (ROI)
ilBlendImg	Blends images
ilBlurImg	Blurs an image
ilBuffer	Provides a four-dimensional resizable buffer
ilCacheImg	Implements image data caching
ilCMYKImg	Converts to the CMYK color model
ilColorImg	Converts to the ABGR color model

Class or Function	Description
ilCombineImg	Combines two images controlled by an ROI
ilCompactCache	Supports a compact pool of pages from global cache
ilCompassImg	Performs a directional gradient transform of an image
ilConfig	Defines configuration of pixel data
ilConstImg	Defines a constant-valued image
ilConvImg	Performs general image convolution
ilCreateImgFile	Creates an image file
ilDataIsSigned(), ilDataMin(), ilDataMax(), ilDataSize(), ilDataType()	Functions that manipulate IL data types
ilDictionary	Implements a dictionary of named elements
ilDilateImg	Performs morphological dilation on an image
ilDisplay	Manages the display of images in an X window
ilDisplayImg	Defines an image that exists in the frame buffer
ilDivImg	Computes pixelwise division of two images
ilDyadicImg	Provides basic support for dual-input operators
ilEnablePrefetch	Enables and disables the prefetching of pages
ilEnviron	Provides support for environment variables
ilErodeImg	Performs a morphological erosion on an image
ilExpImg	Performs pixelwise exponentiation of an image
ilFalseColorImg	Performs false coloring of multispectral images
ilFConjImg	Computes the conjugate of a Fourier image and normalizes the complex value by a real factor
ilFCrCorrImg	Computes the cross-correlation of two Fourier images
ilFDivImg	Divides two Fourier images

Class or Function	Description
ilFDyadicImg	Provides basic support for dual-input Fourier operators
ilFExpFiltImg	Applies an exponential Fourier filter to a Fourier image
ilFFiltImg	Provides basic support for Fourier filter operators
ilFFTOp	Performs a forward, inverse, or average fast Fourier transform of an image
ilFGaussFiltImg	Applies a Gaussian Fourier filter to a Fourier image
ilFileFormat	Registers supported image file formats
ilFileImg	Provides basic support for image files
ilFITImg	Creates an image file in the FIT format
ilFlushCache	Frees memory in global cache
ilFMagImg	Computes the magnitude values of a Fourier image
ilFMergeImg	Merges magnitude and phase images into a Fourier image
ilFMonadicImg	Provides basic support for single-input Fourier operators
ilFMultImg	Multiplies two Fourier images
ilFPhaseImg	Computes the phase values of a Fourier image
ilFRaisePwrImg	Raises the magnitude values of a Fourier image by a power
ilFSpectImg	Computes the spectrum of a Fourier image
ilGBlurImg	Performs a two-dimensional Gaussian blur of an image
ilGetClassPropSet	Accesses class property set by name
ilGetCompactFraction	Queries the fragmentation threshold

Class or Function	Description
ilGetCurCacheSize(), ilGetMaxCacheSize(), ilGetMaxCacheFraction()	Convenience functions that query the state of the global page data cache; see the ilHwAccelerate reference page for details
ilGIFImg	Accesses an image file in the GIF format
ilGLDisplayImg	Defines an image that exists in the frame buffer (GL rendering version)
ilGlobalName	Defines a global name, accesses global name space
ilGLViewer	Handles operations on ilDisplay triggered by GL events in a standard, elegant way
ilGLXConfig	Configures and creates X windows
ilGrayImg	Converts to the gray-scale color model
ilHashTable	Base class from which hash table implementations can be derived
ilHistEqImg	Performs histogram equalization of an image
ilHistLutImg	Base class for operators that compute a lookup table based on a histogram
ilHistNormImg	Performs histogram normalization of an image
ilHistScaleImg	Performs histogram scaling of an image
ilHSVImg	Converts to the HSV color model
ilHwAccelerate(),	Convenience function for enabling and disabling hardware acceleration
ilHwAccelerateClass	Enables and disables hardware acceleration of objects of a specified class
ilHwIsEnabled	Convenience function for querying the state of hardware acceleration
ilHwRequest	Provides support for hardware acceleration
ilHwState	Manages the GL state
ilHwThreadEnable	Enables and disables the use of a dedicated thread for hardware-accelerated rendering

Class or Function	Description
ilHwThreadResume	Resumes a dedicated rendering thread
ilHwThreadSuspend	Temporarily suspends the dedicated rendering thread
ilImage	Provides basic support for images
ilImgStat	Computes the histogram, minimum, maximum, mean, and standard deviation of an image
ilIndexableList	Provides an indexable linked list
ilIndexableStack	Manages an indexable list as a stack
ilInvertImg	Performs one's complement of an image
ilKernel, ilDoubleKernel, ilFloatKernel, ilLongKernel, ilShortKernel	Classes that define kernels
ilLaplaceImg	Performs edge detection using Laplacian kernels
ilLink	Provides for chaining and setting attributes
ilLinkItem	Base class that creates elements of a doubly linked list
ilLinkIter	Iterator for ilLink
ilList	Base class for a simple doubly-linked list
ilListIter, ilListIterRev	Iterators for ilList and ilIndexableList
ilLogImg	Computes the pixelwise logarithm of an image
ilLut	Defines a lookup table
ilLutImg	Translates an image using a lookup table
ilMatrix, ilFloatMatrix, ilLongMatrix	Classes that define matrices
ilMaxFltImg	Performs max filtering of an image
ilMaxImg	Computes the pixelwise maximum of two images
ilMedFltImg	Performs median filtering of an image

Class or Function	Description
ilMemCacheImg	Implements data caching in main memory
ilMemoryImg	Defines an image array resident in memory
ilMergeImg	Merges several images into one
ilMinFltImg	Performs minimum filtering of an image
ilMinImg	Computes the pixelwise minimum of two images
ilMinMax(), ilMin(), ilMax()	Functions for performing minimum and maximum comparisons
ilMonadicImg	Provides basic support for single-input operators
ilMpSetMaxProcs()	Convenience function to set multi-processing parameters; see the ilHwAccelerate reference page for more details
ilMultiplyImg	Computes the pixelwise multiplication of two images
ilName	Creates named elements for an ilDictionary
ilNegImg	Performs two's complement of an image
ilNopImg	Provides for caching on non-cached images
ilOpenImgFile()	Function that opens an image file
ilOpImg	Provides basic support for operators
ilOrImg	Computes pixelwise logical OR of two images
ilPage	Defines a page of image data in a cache
ilPageRequest	Defines a page of data as a request
ilPCDImg	Provides support for the Kodak Photo CD image pack file format class
ilPCDOImg	Accesses the Kodak PCD overview image file
ilPiecewiseImg	Performs linear mapping of lookup table images
ilPixel	Defines a pixel
ilPolyWarpImg	Performs a two-dimensional seventh-order warp

Class or Function	Description
ilPowerImg	Raises image data by a specified power
ilPreFetcher	Manages prefetch paging
ilPrefetchIsEnabled	Checks the status of prefetching
ilPriorityItem	Creates a priority item
ilPriorityList	List of items sorted by priority
ilProperty	Creates a name / value pair
ilPropList	Manages a property set as an indexable list
ilPropSet	Creates a collection of properties
ilPropSetIter	Controls iteration through a property set
ilPropTable	Manages a property set as a hash table
ilRankFltImg	Performs two-dimensional rank filtering on an image
ilRasterIter	Iterates through the scanlines that define a convex polygon
ilRectRoi	Defines a rectangular ROI
ilRFFTfImg	Performs a real forward fast Fourier transform
ilRFFTiImg	Performs a real inverse fast Fourier transform
ilRGBImg	Converts to the RGB color model
ilRobertsImg	Performs edge detection using Roberts kernels
ilRoi	Defines an ROI
ilRoiImg	Associates an ROI with an image
ilRoiIter	Cycles through run lengths in an ROI
ilRoiMap	Declares ROI attributes
ilRotZoomImg	Rotates, zooms, and flips an image
ilSaturateImg	Performs color saturation of an image

Class or Function	Description
ilScaleImg	Performs a linear scaling of an image
ilSemaphore	Limits access to a shared data structure to some maximum number of process threads
ilSepConvImg	Performs an image convolution using a separable kernel
ilSetCompactFraction	Sets the fragmentation threshold
ilSetMaxCacheSize(), ilSetMaxCacheFraction()	Convenience functions that set the state of the global page data cache; see the ilHwAccelerate reference page for details.
ilSGICmapLut(), ilSGIFileLut()	Functions that create or access color maps
ilSGIImg	Creates an image file in the SGI format
ilSGIPaletteImg	Converts to the RGBPalette color model
ilSharpenImg	Sharpens an image
ilSize	Defines the size of an image
ilSmallBitArray	Defines a bit array
ilSobelImg	Performs edge detection using Sobel kernels
ilSpatialImg	Provides basic support for spatial operators
ilSpinLock	Manages spinlock services
ilSqRootImg	Computes the pixelwise square root of an image
ilSquareImg	Computes the pixelwise square of an image
ilStackBuffer	Provides a four-dimensional resizable buffer with better performance than an ilBuffer
ilStereoView	Associates a stereo view (two images) with a region in an ilDisplayImg
ilSubImg	Defines a rectangular portion of an image as an independent image
ilSubtractImg	Computes the pixelwise subtraction of two images

Class or Function	Description
ilSwitchImg	Implements a switch construct in an image operator chain
ilThread	Manages a shared group of processes
ilThreshImg	Applies a threshold to an image
ilTieWarpImg	Warps an image by specifying tie points
ilTIFFImg	Creates an image file in the TIFF format
ilTile, ilTileFloat	Defines a three-dimensional rectangle of image data
ilTileImgIter, ilTileIter	Cycles through the pages spanning a tile
ilView	Associates an image with a region in an ilDisplayImg
ilViewer	Handles operations on ilDisplay triggered by X events in a standard, elegant way
ilViewIter	Iterates through ilDisplay's view stack
ilWarpImg	Provides basic support for warping an image
ilXDisplayImg	Defines an image that exists in the frame buffer (X rendering version)
ilXImage	Translates between an XImage and an ilImage
ilXorImg	Computes the pixelwise exclusive-OR of two images

Appendix C

Data Types, Data Orderings, and Color Models

This appendix provides a summary of the default data type, data ordering, and color model attributes of IL images. It tells you how to determine the:

- default color model used by the IL for an ilOpImg object in a chain of image operators
- data types and data orderings and working data types of objects derived from ilOpImg

Determining Color Model

If an application or derived class does not use the **setDataType()** function to explicitly set the color model of an ilOpImg object, the color model defaults to the lowest common ancestor of the input images in the following diagram:

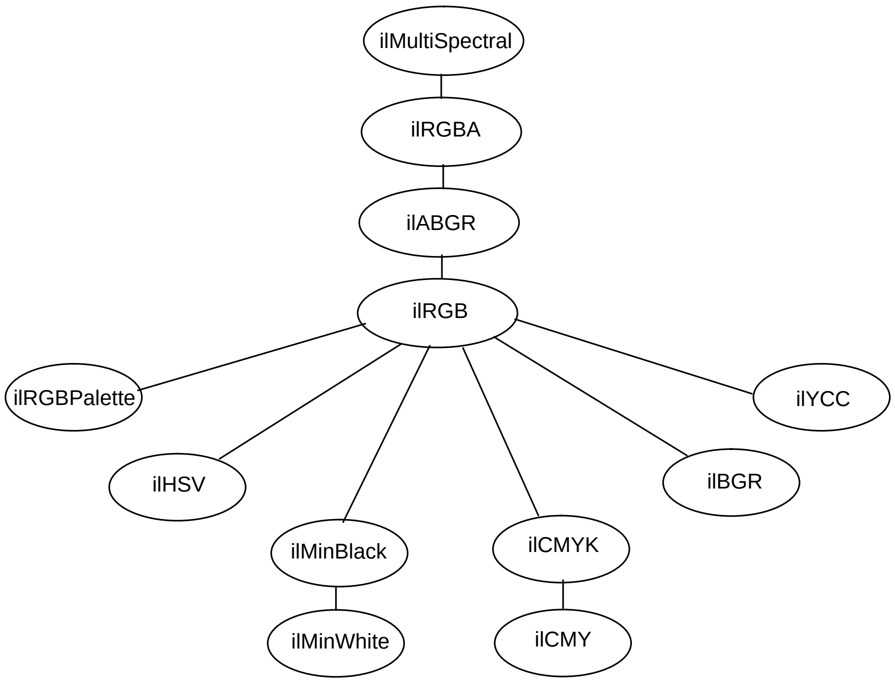


Figure C-1 Determining Color Model

Determining Operator Data Types and Orderings

The tables on the following pages list the output data types, data orderings, and working data types for classes derived from ilOpImg.

Output Data Types and Orderings

The data type or data order can be set explicitly to one of the valid types or orderings by calling the ilImage member function **setDataType()** or **setOrder()**, respectively. If the data type or order is not set explicitly in this manner, they default to the “smallest” of the valid types or orderings that is at least as “great” as each input type or order. Here “small” and “great” refer to the numeric values of the types and orderings, as defined in *il/ilTypes.h*.

Working Types

An ilOpImg object has a “working type”, which is the data type used for calculations. If a class has no working types specified in the table below, the working type is the same as the output data type.

For those classes with working types listed in the table, the working type used is the smallest listed type that is able to express every value of the output data type. If none of the listed types can do this, then the largest of the listed types is used (for example, an ilColorImg with output type ilDouble will use ilFloat as its working type).

Color Conversion

Table C-1 Data Types and Orderings for Color Conversion Operators

Operator	Output Data Types	Output Data Orderings	Working Types
ilColorImg	ilUChar	ilInterleaved	
ilRGBImg	ilUShort		
ilBGRImg	ilFloat		
ilCMYKImg			
ilHSVImg			
ilGrayImg			
illABGRImg			
ilSGIPaletteImg			
ilFalseColorImg	ilUChar	ilInterleaved	
	ilFloat		
ilSaturateImg	ilUChar	ilInterleaved	
	ilUShort		
	ilShort		
	ilFloat		

Arithmetic and Logical Transformations

Table C-2 Data Types and Ordering for Arithmetic and Logical Transformations

Operator	Output Data Types	Output Data Orderings	Working Types
ilAbsImg	ilUChar ilUShort ilULong ilFloat ilDouble	any	
ilAddImg	any except ilBit	any	
ilAndImg	ilUChar ilChar ilUShort ilShort ilULong ilLong	any	
ilDivImg	any except ilBit	any	
ilArithLutImg ilExpImg ilLogImg ilPowerImg ilSqRootImg ilSquareImg	any except ilBit	any	integer types[a]

Table C-2 (continued) Data Types and Ordering for Arithmetic and Logical Transformations

Operator	Output Data Types	Output Data Orderings	Working Types
ilInvertImg	ilBit ilChar ilUChar ilShort ilUShort ilLong ilULong	any	
ilMaxImg	any except ilBit	any	
ilMinImg	any except ilBit	any	
ilMultiplyImg	any except ilBit	any	
ilNegImg	ilChar ilShort ilLong ilFloat ilDouble	any	
ilOrImg	ilUChar ilChar ilUShort ilShort ilULong ilLong	any	

Table C-2 (continued) Data Types and Ordering for Arithmetic and Logical Transformations

Operator	**Output Data Types**	**Output Data Orderings**	**Working Types**
ilSubtractImg	any except ilBit	any	
ilXorImg	ilUChar ilChar ilUShort ilShort ilULong ilLong	any	

a. The working type for an ilLutImg or ilArithLut object is derived from the domain of the LUT rather than the output type. It is the smallest integer type in which the minimum and maximum indices of the lookup table can be expressed.

Geometric Transformations

Table C-3 Data Types and Orderings for Geometric Transformations

Operator	**Output Data Types**	**Output Data Orderings**	**Working Types**
ilWarpImg ilRotZoomImg ilPolyWarpImg ilTieWarpImg	ilUChar ilUShort ilShort ilFloat	any	ilFloat[a]

a. The working type for an ilWarpImg object will be ilFloat unless the resampling type is ilNearNb, in which case the working type will be the same as the output type.

Spatial Domain Transformations and Edge Detection

Table C-4 Data Types and Orderings for Spatial Domain Transformations and Edge Detection

Operator	Output Data Types	Output Data Orderings	Working Types
ilConvImg ilBlurImg ilSharpenImg ilCompassImg	any except ilBit	any	ilFloat ilDouble
ilDilateImg	ilUChar ilUShort ilULong	any	
ilErodeImg	ilUChar ilUShort ilULong	any	
ilRankFiltImg ilMinFltImg ilMedFltImg ilMaxFltImg	any except ilBit	any	
ilSepConvImg ilGBlurImg	any except ilBit	any	ilFloat ilDouble
ilLaplaceImg	ilUChar ilShort ilLong ilFloat ilDouble	ilInterleaved	

Table C-4 (continued) Data Types and Orderings for Spatial Domain Transformations and Edge Detection

Operator	Output Data Types	Output Data Orderings	Working Types
ilRobertsImg	ilUChar ilUShort ilULong ilFloat ilDouble	any	
ilSobelImg	ilUChar ilUShort ilULong ilFloat ilDouble	any	

Frequency Domain Transformations

Table C-5 Data Types and Orderings for Frequency Domain Transformations

Operator	Output Data Types	Output Data Orderings	Working Types
ilRFFTfImg	ilFloat	ilSeparate	
ilRFFTiImg	ilFloat	ilSeparate	
ilFConjImg	ilFloat	ilSeparate	
ilFCrCorrImg	ilFloat	ilSeparate	
ilFDivImg	ilFloat	ilSeparate	
ilFExpFiltImg	ilFloat	ilSeparate	
ilFGaussFiltImg	ilFloat	ilSeparate	
ilFMagImg	ilFloat	ilSeparate	
ilFMergeImg	ilFloat	ilSeparate	
ilFMultImg	ilFloat	ilSeparate	
ilFPhaseImg	ilFloat	ilSeparate	
ilFRaisePwrImg	ilFloat	ilSeparate	
ilFSpectImg	ilFloat	ilSeparate	
ilFMonadicImg	ilFloat	ilSeparate	
ilFDyadicImg	ilFloat	ilSeparate	
ilFFiltImg	ilFloat	ilSeparate	

Radiometric Transformations

Table C-6 Data Types and Orderings for Radiometric Transformations

Operator	Output Data Types	Output Data Orderings	Working Types
ilScaleImg ilHistScaleImg	ilUChar ilUShort ilShort ilFloat ilDouble	any	
ilThreshImg	any	any	
ilLutImg ilHistLutImg ilHistEqImg ilHistNormImg ilPiecewiseImg	any except ilBit	any	integer types [a]

a. The working type for an ilLutImg or ilArithLut object is derived from the domain of the LUT rather than the output type. It is the smallest integer type in which the minimum and maximum indices of the lookup table can be expressed.

Combining Images

Table C-7 Data Types and Orderings for Operators that Combine Images

Operator	Output Data Types	Output Data Orderings	Working Types
ilBlendImg	any except ilBit	any	
ilCombineImg	any except ilBit	any	

Null Operation

Table C-8 Data Type and Ordering for the NULL Operator

Operator	Output Data Types	Output Data Orderings	Working Types
ilNopImg	any	any	

Results of Operators

This appendix presents examples in the following sections of all the operators that give visible results:

For more information on using these operators and what effect they have on image data, see Chapter 4, “Operating on an Image.” More specific information on how to apply each operator is located in its header file and in its reference page.

Original, unprocessed images are presented where necessary. Some images combine two original images. These images are either reversed copies of the same image or two extremely similar images.

Color Conversion

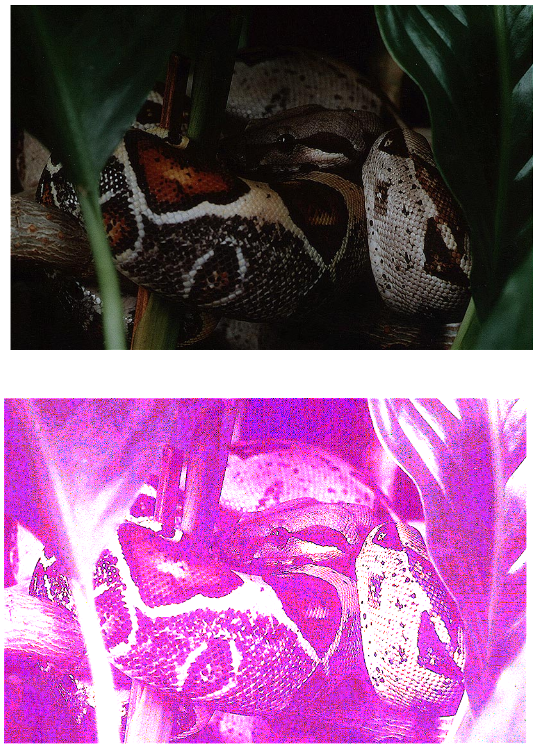


Figure D-1 ilFalseColorImg

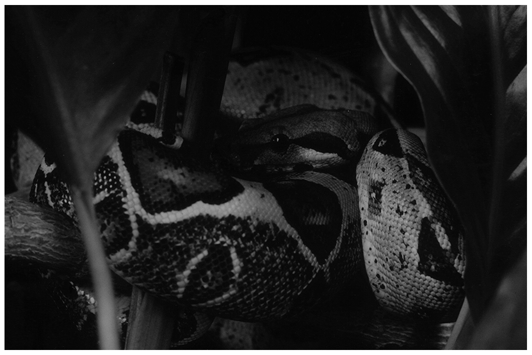


Figure D-2 ilGrayImg

Arithmetic and Logical Transformations

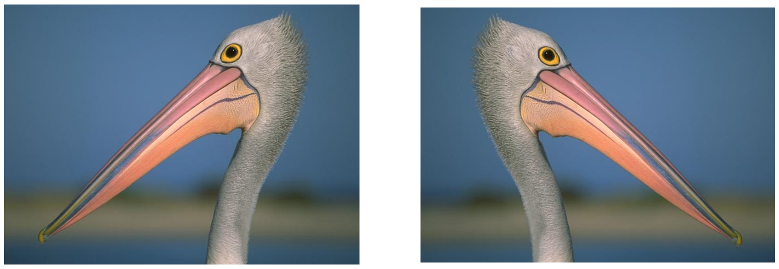


Figure D-3 Original Image and Flipped Image

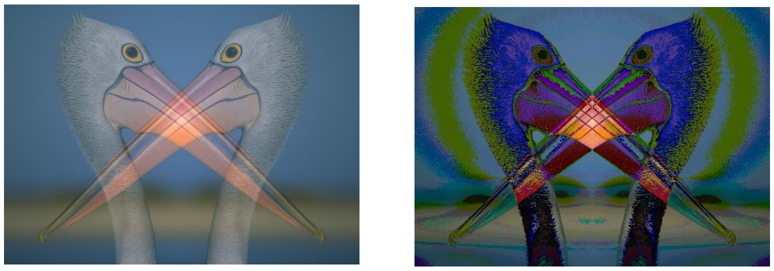


Figure D-4 ilAddImg and ilAndImg

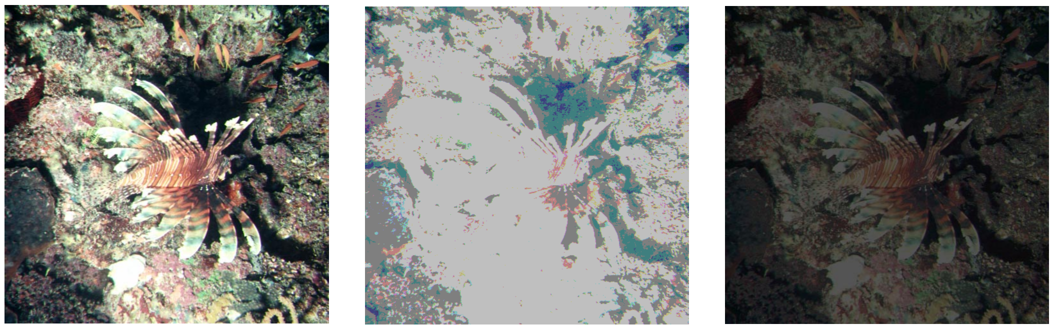


Figure D-5 ilDivImg

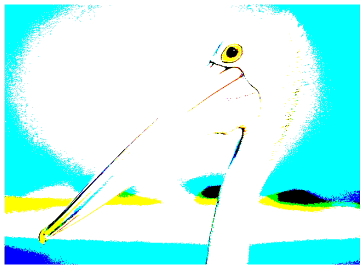
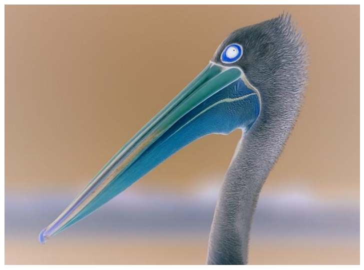



Figure D-6 ilExpImg and ilInvertImg

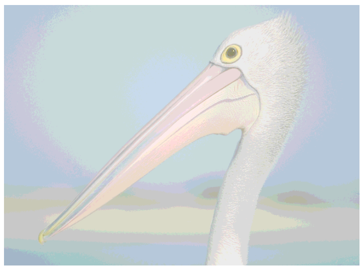
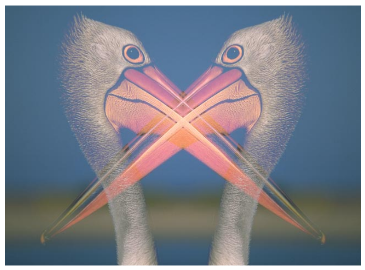



Figure D-7 ilLogImg and ilMaxImg

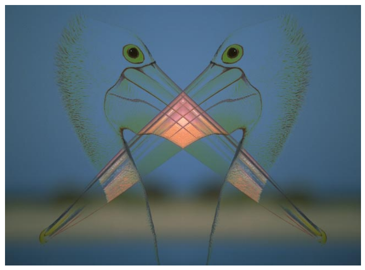
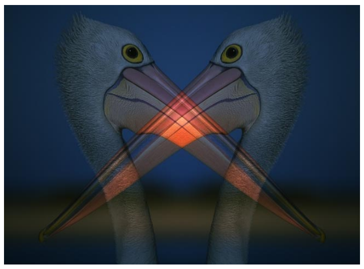



Figure D-8 ilMinImg and ilMultiplyImg

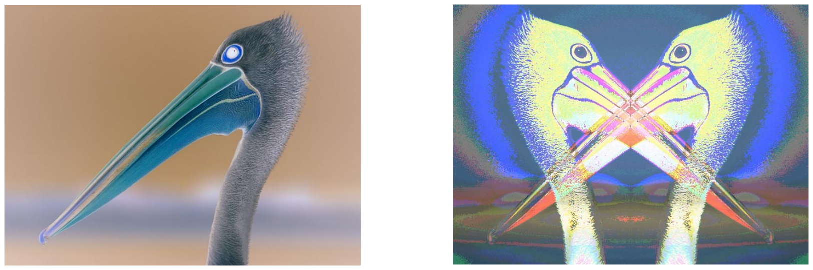


Figure D-9 ilNegImg and ilOrImg

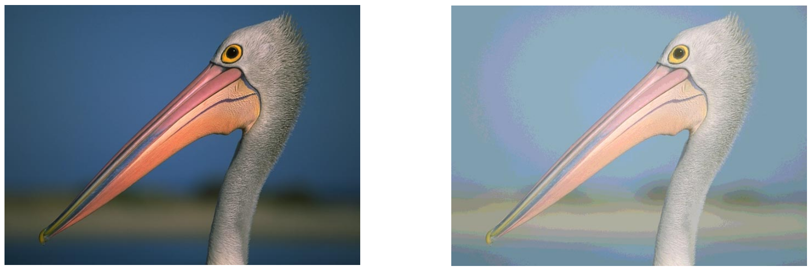


Figure D-10 ilPowerImg and ilSqRootImg

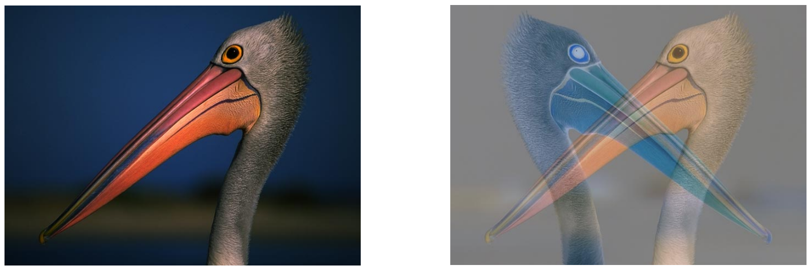


Figure D-11 ilSquareImg and ilSubtractImg

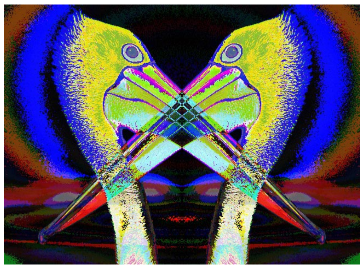


Figure D-12 ilXorImg

Geometric Transformations

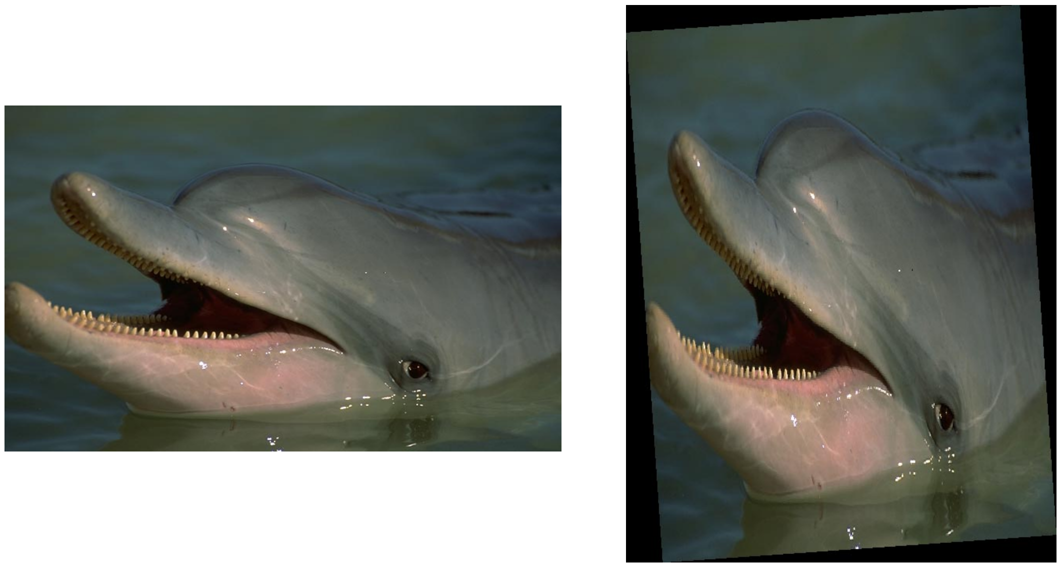


Figure D-13 Original and ilRotZoomImg

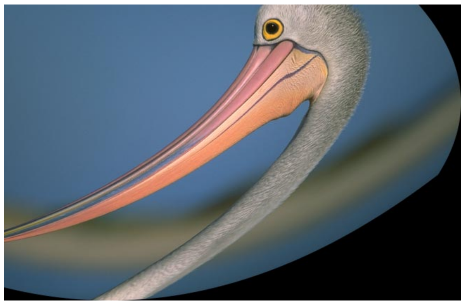


Figure D-14 ilWarpImg

Spatial Domain Transformations

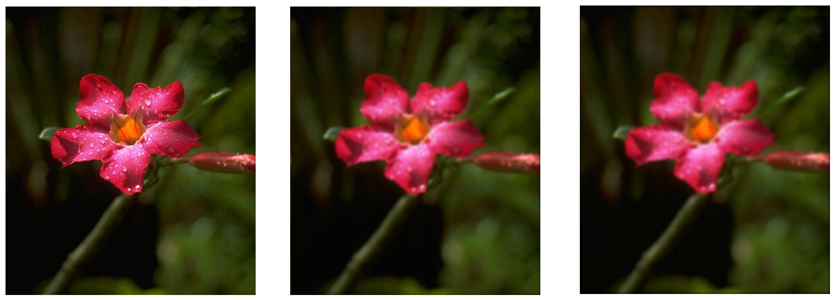


Figure D-15 Original, ilBlurImg and ilGBlurImg

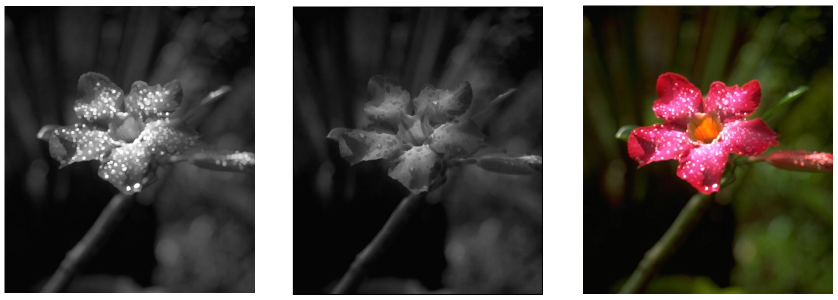


Figure D-16 ilDilateImg, ilErodeImg, and ilMaxFltImg

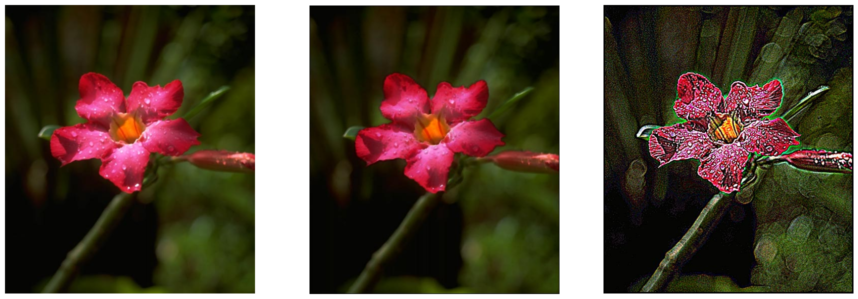


Figure D-17 ilMedFltImg, ilMinFltImg, and ilSharpenImg

Edge Detection

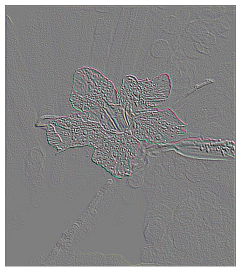


Figure D-18 ilCompassImg

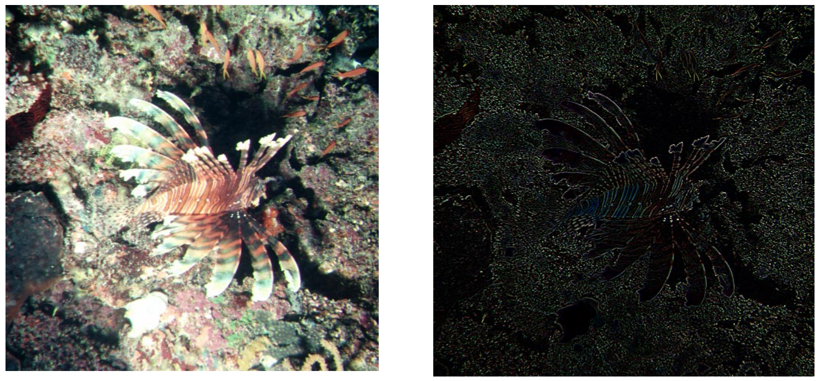


Figure D-19 ilLaplaceImg (original and filtered image)

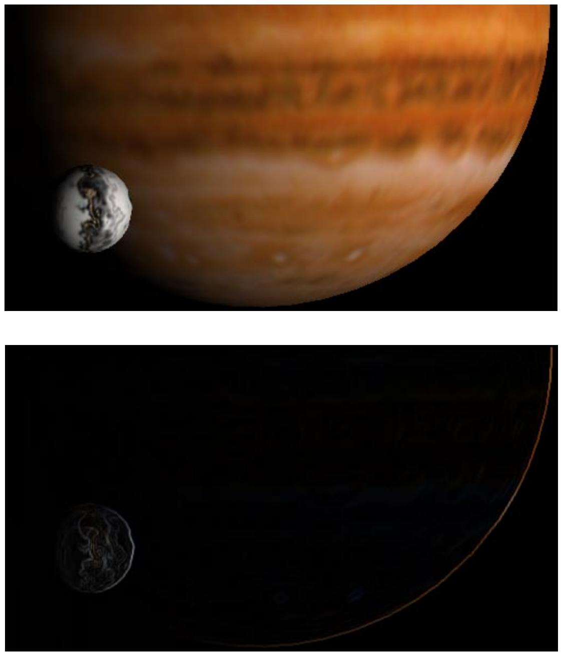


Figure D-20 ilRobertsImg (original and filtered image)

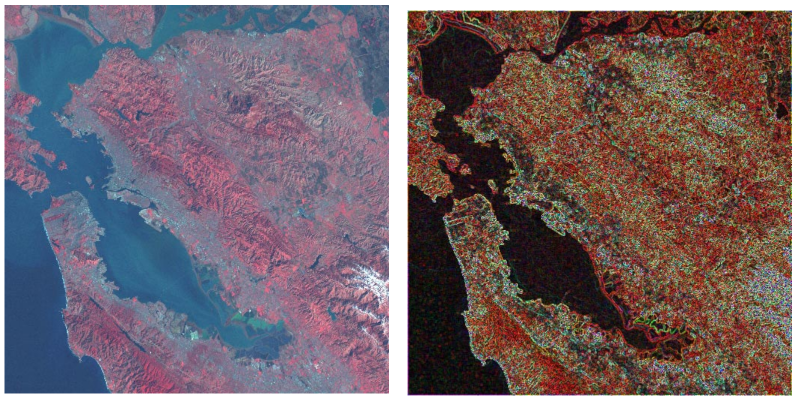


Figure D-21 ilSobelImg (original and filtered image)

Frequency Domain Transformations

The frequency domain transformations are of limited interest as illustrations. For the purposes of this appendix, one example is shown. In the example, an original image is presented along with its appearance in the frequency, or Fourier domain, and the filtered resultant image is shown in both the spatial and frequency domains.

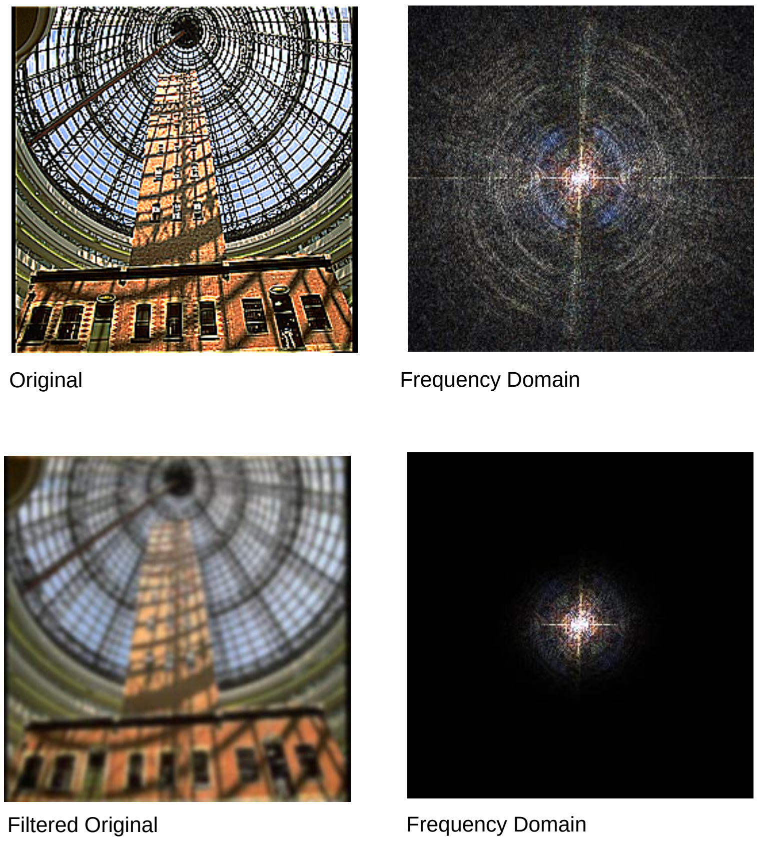


Figure D-22 ilFGaussFiltImg

Radiometric Transformations

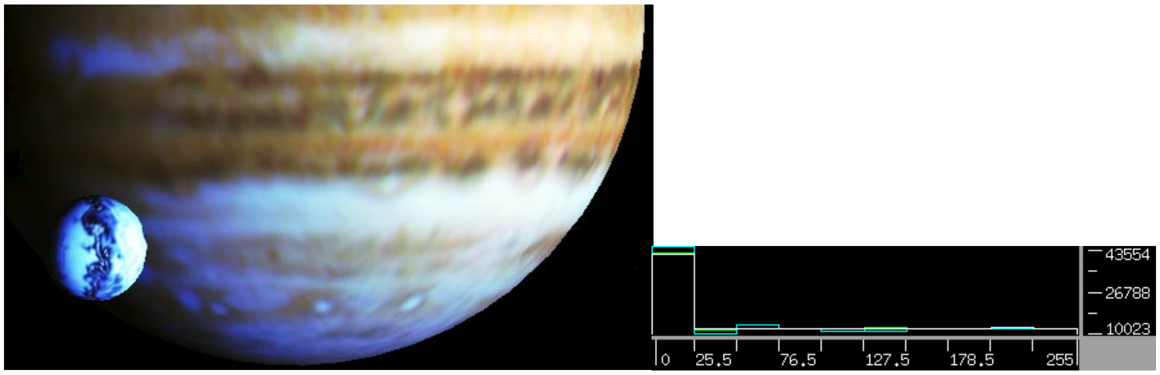


Figure D-23 ilHistEqImg (filtered image and histogram)

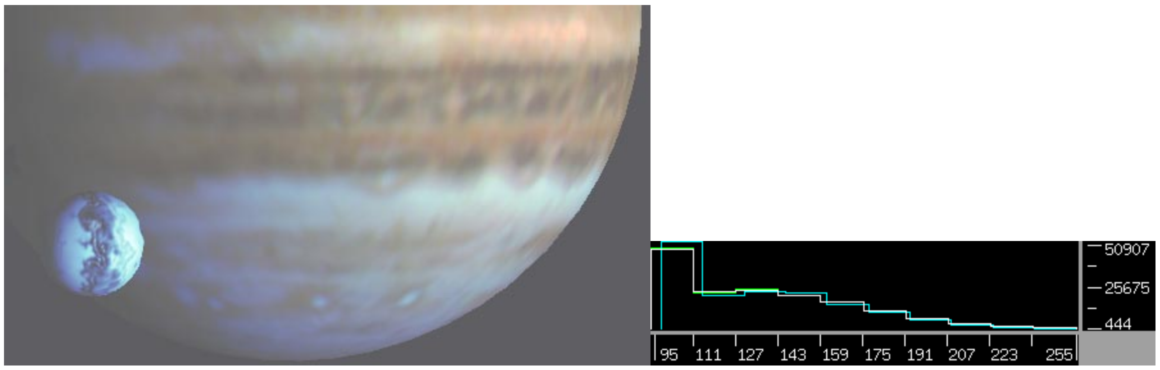


Figure D-24 ilHistNormImg (filtered image and histogram)

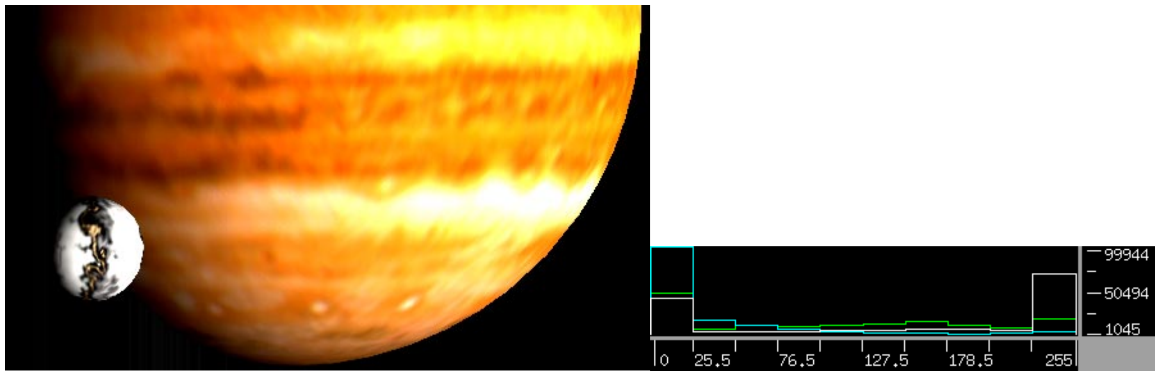


Figure D-25 ilHistScaleImg (filtered image and histogram)

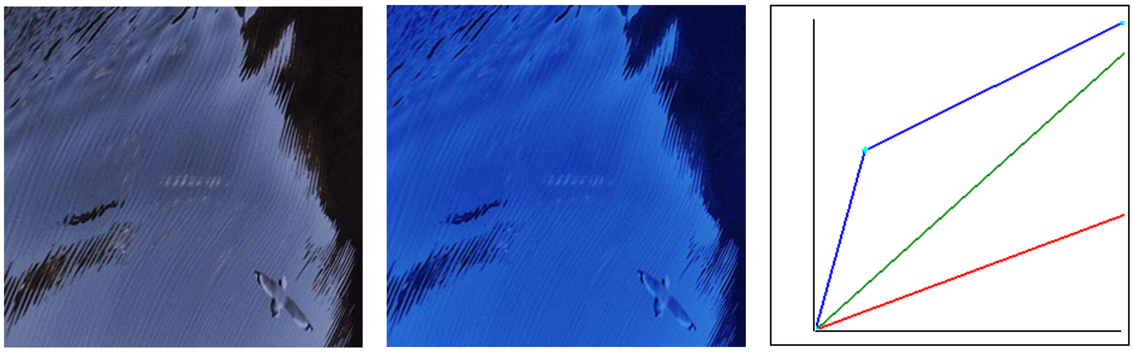


Figure D-26 ilLutImg (original, filtered image, and LUT editor)

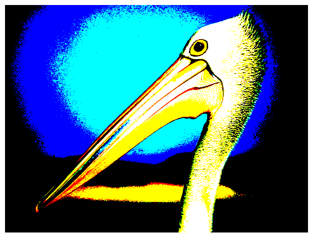
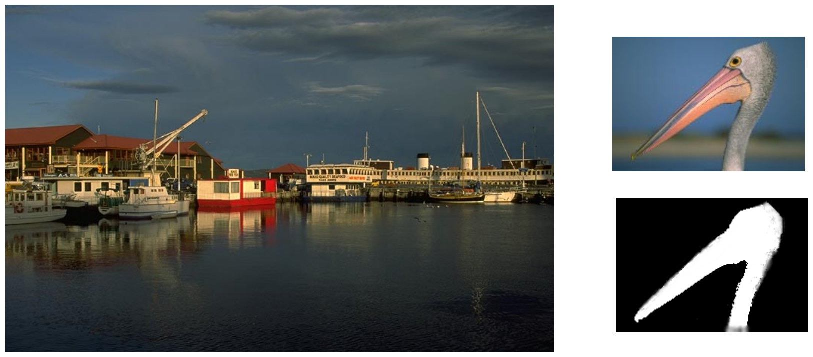


Figure D-27 ilThreshImg

Combining Images



Originals and Original Mask

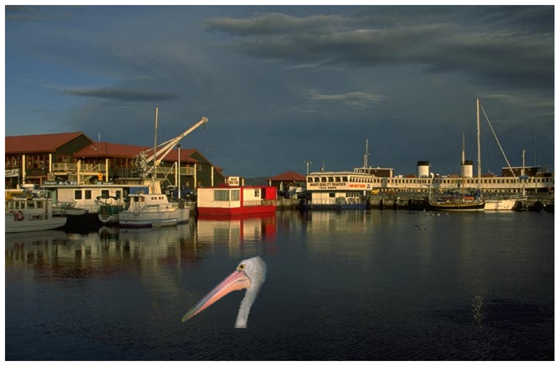


Figure D-28 ilBlendImg

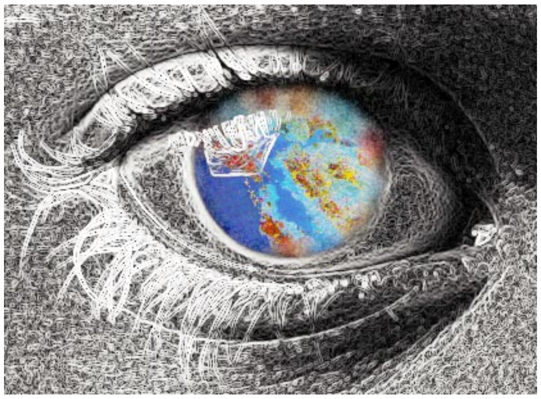


Figure D-29 ilCombineImg

Auxiliary Classes, Functions, and Definitions

This appendix describes IL classes not fully discussed elsewhere in this guide. It also lists all the error codes and enumerated types used by the IL. This appendix has the following major sections:

- "Auxiliary Classes" on page 396 briefly discusses the ilBitArray, ilBuffer and ilStackBuffer, ilConfig, ilKernel, ilLut, ilMatrix, ilPage, ilPixel, ilSize, and ilTile classes.
- "Useful Functions" on page 401 describes several functions that don't belong to any particular class. They're useful for such tasks as computing the size of IL data types and for performing minimum and maximum comparisons.
- "Convenient Structures" on page 403 lists the definitions of the ilCoord, ilSize, and various coefficient data structures.
- "Error Codes" on page 407 lists the error codes used by the IL.
- "Enumerated Types and Constants" on page 408 gives an annotated list of the enumerated types and constants defined in the IL.

Auxiliary Classes

All of the classes described in this section have their own reference pages; refer to them for more specific information about using these classes.

- The ilBitArray class implements a subscriptable bit array of limited functionality for conveniently operating on bit data.
- ilBuffer (allocated from the heap) and ilStackBuffer (allocated from the stack) are standalone objects that provide support for accessing a buffer in up to four dimensions. The call operator, (), is overloaded to operate on either type of buffer and returns a pointer to the specified element in the buffer. In addition, the an ilBuffer can be resized after being created. An ilStackBuffer is recommended for use in derived operators, since it tends to fragment the memory less than an ilBuffer does, thus resulting in better performance for an application. However, an ilStackBuffer cannot be resized.
- The ilConfig class is used in ilImage functions such as **getTile()** and **setTile()** to describe the configuration of pixel data. You can also use it when constructing an ilSubImage to map the configuration of the input image to that of the subimage. This class is described in more detail in "ilConfig" on page 397.
- ilKernel is the base class for deriving a strongly typed three-dimensional kernel. The kernel elements are stored in row-major form. An ilKernel is defined by *x*, *y*, and *z* dimensions, the kernel data, the kernel origin, and the data type of its elements. ilKernel also provides functions to access kernel attributes and data either by single elements or vectors. There are several typed kernels derived from ilKernel: ilShortKernel, ilLongKernel, ilFloatKernel, and ilDoubleKernel.
- The ilLut class is used to access and manipulate lookup tables. This class is described in more detail in "ilLut" on page 399.
- ilMatrix is identical in function to ilKernel except that it does not have a kernel origin. There are two typed kernels derived from ilMatrix: ilLongMatrix and ilFloatMatrix.
- The ilPage class is used to describe rectangular regions of an image in a cache (that is, pages). This class groups the eight values describing the origin (*x*,*y*,*z*,*c*) and size (*nx*,*ny*,*nz*,*nc*) of a page together in a convenient way.

- The ilPixel class abstracts the concept of a pixel of image data. It contains the data type, the number of channels, and a list of component values. Pixels are used as arguments to a number of ilImage functions and to some operator image constructors and functions.
- ilSize is used to describe the size of an IL image. This class groups the four values describing the size (*x*,*y*,*z*,*c*) of an image together in a convenient way.
- The ilTile class is used to describe arbitrary rectangular regions of an image (that is, tiles). This class groups the six values describing the origin (*x*,*y*,*z*) and size (*nx*,*ny*,*nz*) of a rectangle together in a convenient way.

ilConfig

The header file *il/ilConfig.h* defines the class ilConfig, which is used to describe the configuration of pixel data. Its fields describe the data type, pixel ordering, number of data channels, ordering of data channels, channel offset, coordinate space, and color model (the color model field is currently ignored by functions such as **getTile()** and **setTile()**). The code in Example E-1 shows the ilConfig constructors and fields.

Example E-1 ilConfig Constructors and Fields

```
class ilConfig {
public:
 ilType dtype;        // data type
 ilOrder order;       // pixel ordering
 ilCoordSpace space;  // coordinate space
 int nchans;          // number of channels
 int choff;           // channel offset
 int* channels;       // channel list

 ilConfig();
 // construct config to match parameters of image
 ilConfig(ilImage *img);
 ilConfig(ilType type, ilOrder ord, int nchan=0,
        int* chanList=NULL, int chanOff=0, ilCoordSpace
        spc=ilCoordSpace(0),
        ilColorModel color=ilColorModel(0));

 // extract inverted channel list
 void invert(int nc, int* chanList) const;
 //check if channel list can be inverted
 int isInvertable() const;

 //extract composed channel list
 void compose(int nc, int* in, int* out) const;

 //map channel number through channel list and offset
 int mapChan(int idx) const;
 int operator[](int idx) const;

 unsigned hints;   // ilHints *Internal Use Only*
};
```

The fields of an ilConfig are set with its constructor. The data type and pixel ordering arguments are required; the other arguments are optional. The channel list defines what channels of a source image are mapped into a destination image; the channel offset defines where to start counting the source channels as zero. For example, consider a source image with 11 channels (0...10) and suppose you wish to map channels 4, 5, and 6 to a destination image. You can do this by setting the number of channels to 3 and the channel offset to 4 (so that the first channel mapped is 4, and the next 3 channels in the source define all 3 channels). No channel list is necessary. Alternatively, you can set the number of channels to 3, the channel offset to 0, and the channel list to 3, 4, 5. The hints field is reserved for internal IL use only.

invert() is used to create a channel list of *nc* channels (written into *chanList*) that describes an inverse mapping between two images. For example, a source image defines three channels (0, 1, 2) and you have mapped 0 to 2, 1 to 0, and 2 to 1 in a destination image (the channel list to do this is 1, 2, 0). To map the destination to the source instead, use **invert()**. (This is useful to avoid creating a temporary buffer when copying from an ilDisplayImg to an ilOpImg, for example.) In the above example, the resulting channel list is 2, 0, 1. **isInvertable()** is used to determine whether the channel mapping has an inverse.

compose() is used to compose a channel list from a subset of another; you supply the number of channels, *nc*, and the subchannel list, *in*, and **compose()** writes its result to *out*. For example, a source image defines three channels (0, 1, 2) and you have mapped 0 to 2, 1 to 0, and 2 to 1 in a destination image (the channel list to do this is (1, 2, 0). However, the source image is actually an ilSubImg (a subimage of another ilImage) that contains no data itself. It specifies a subset of its parent image’s channels; they are 2, 4, and 6 (so it uses an ilConfig with channel list 2, 4, 6). To map directly from the source's parent image to the destination image, you need a composed channel list. The ilSubImg's channel list is specified as *in*, and the 3 values mapped to *out* are 4, 6, 2.

The member function **mapChan()** returns the contents of *channels* (the channel list) at the specified index added to the channel offset specified by *choff*. The index operator, [ ], is overloaded to perform the same function as **mapChan()**; both indicate what channel in the source maps to the specified channel in the destination. Both return -1 if the supplied index is less than 0 or greater than the number of channels.

ilLut

The header file *il/ilLut.h* defines a class, ilLut, used to describe lookup tables. The elements used to define an ilLut are the number of bits per channel, the number of channels, the data type, the table length, and the table data. The code in Example E-2 shows the constructors and member functions for ilLut.

Example E-2 ilLut Constructors and Member Functions

```
class ilLut {
public:
//Constructors
 ilLut();
 ilLut(void *table, int tabChannels, ilType tabType,
       int tabBits=-1);
 ilLut(int tabChannels, ilType tabType, int tabBits=-1);
 ilLut(int tabChannels, ilType tabType,
       double min, double max);

//Access methods
 void* getData() const;
 int getNumChans() const;
 ilType getType() const;
 int getLength() const;
 double getVal(int idx, int chan=0) const;
 void* getChan(int chan) const;
 void getDomain(double& min, double& max) const;
 void getRange(double& min, double& max) const;
 void* getOrigin(int chan) const;

//General
```

```
 void setData(void* dataPnt);
 ilStatus setVal(double val, int idx, int chan=0);
 ilStatus setDomain(double min, double max);

 void operator=(const ilLut& from); // copy from lut

};
```

The first constructor creates an ilLut with all of its member variables initialized to 0. The first two constructors assume that you allocate memory for the table and populate the table with data; the other constructors allocate memory but leave you to populate the table. Use **setVal()** to populate the table. The constructor that takes *min* and *max* arguments calculates the number of bits required to express that range. Typically, the size of the table data in bytes is defined as shown below:

table size = $2^{\textit{tabBits}}$ * *tabChannels*

The actual size of the table buffer (in bytes) depends on the data type:

```
table buffer size = table size * number of bytes required for
                    data type
```

Two convenience functions for creating an ilLut—**ilSGICmapLut()** and **ilSGIFileLut()**—are defined in the header files *il/ilSGICmapLut.h* and *il/ilSGIFileLut.h*. These functions both create a 3-channel ilLut class (typically corresponding to RGB values).

The **ilSGICmapLut()** function creates a 3-channel lookup table consisting of RGB color entries of data type ilChar:

```
ilLut* ilSGICmapLut(ilSGIBufferMode mode);
```

The size and contents of the returned ilLut depends on the ilSGIBufferMode specified: ilSgiDefault, ilSingleBuffer, or ilDoubleBuffer. The ilSgiDefault mode creates an 8-bit lookup table. The 256 entries of the lookup table are initialized to be those values defined by the default GL color map (refer to the **makemap()** reference page). The ilSingleBuffer and ilDoubleBuffer modes depend on the hardware and the display mode (single or double buffer) currently in use. In both modes, the size of the table is computed as:

table size = numChans * $2^{\text{number of bits}}$

The data is initialized to be the first *N* entries currently stored in the hardware's lookup table, where *N* = table size.

The **ilSGIFileLut()** function creates a lookup table from a file. The size and data are obtained from the file; the data is typically of data type ilShort. Currently, the only supported file format for restoring lookup tables is the ilSGIImg file format.

The data in an ilLut is stored in ilSeparate format—that is, the data for channel 0 is stored first, then the data for channel 1, and so on. Operators that use ilLuts assume this format. As an example of how data is stored and retrieved, consider an RGB color map of ilChar data. In this case, the table size is $3 \times 2^8 \times 1 = 768$ bytes. So, if the table is called *fooMap[]*, you'd use the following to obtain red, green, and blue values:

```
char redValue = fooMap[index];
char greenValue = fooMap[index+256];
char blueValue = fooMap[index+512];
```

Useful Functions

This section describes utility functions defined by the IL. These functions don't belong to any particular class, so they can be used anywhere in an IL program.

Computing the Size of Data Types

The IL defines constants that correspond to the data types it uses; it also defines a related set of functions for determining the sizes and possible values for these types. These constants are defined as the ilType enumerated type in the header file *il/ilTypes.h*:

ilBit	ilULong
ilUChar	ilLong
ilChar	ilFloat
ilUShort	ilDouble
ilShort	

The following two functions perform computations using the above data types; they're defined in the header file *il/ilDataSize.h* and described in the ilDataSize reference page:

```
int ilDataSize(ilType type, int count);
ilType ilDataType(double minVal, double maxVal);
```

The first function, **ilDataSize()**, returns the number of bytes needed to store *count* elements of data type *type*. By default, *count* is 1. Conversely, **ilDataType()** returns the first IL data type that's large enough to hold the range of values specified by *minVal* and *maxVal*.

The following two functions return the maximum and the minimum possible value, respectively, for the specified data type:

```
double ilDataMax(ilType type);
double ilDataMin(ilType type);
```

If you pass one of the ilTypes as an argument for **ilDataIsSigned()**, this function will return TRUE if the type is signed and FALSE (zero) otherwise. Remember that ilImage defines a similar function for an image, **isSigned()**, that returns TRUE if the image's data type is signed.

Minimum and Maximum Comparisons

The header file *il/ilMinMax.h* defines several in-line functions that determine the minimum and the maximum of two input values. There are separate functions for each of three data types—**int**, **float**, and **double**. The functions that use **float** data are shown below:

```
inline float ilMin(float a, float b);
inline float ilMax(float a, float b);
```

The **ilMin()** function returns the lesser of the two input values, and **ilMax()** returns the greater of the two values.

Converting to Color-index Mode

Use the following function to convert an RGB triplet to the closest corresponding value in the standard color map that's used in color-index mode. (This function is defined in the header file *il/ilColor.h*.)

```
int ilRGBtoSGIPalette(int r, int g, int b);
```

The function **ilColorModelChans(ilColorModel cm)**, also described in *il/ilColor.h*, can be used to determine the number of channels for a given color model *cm*.

Convenient Structures

This section lists the definitions of the ilCoord and various coefficient data structures.

Coordinate Data Structures

The structures listed in Table E-1 hold two- (*x*,*y*), three- (*x*,*y*,*z*), and four-dimensional (*x*,*y*,*z*,*c*) coordinates of various data types; they're defined in the *il/ilCoord.h* header file. ilXYS**, ilXYZS**, and ilXYZCS** are simple structures without any constructors, destructors, or convenience operators.

Table E-1 Coordinate Data Structures

Two-dimensional	Three-dimensional	Four-dimensional
ilXYchar, ilXYSchar	ilXYZchar, ilXYZSchar	ilXYZCchar, ilXYZCSchar
ilXYint, ilXYSint	ilXYZint, ilXYZSint	ilXYZCint, ilXYZCSint
ilXYfloat, ilXYSfloat	ilXYZfloat, ilXYZSfloat	ilXYZCfloat, ilXYZCSfloat
ilXYdouble, ilXYSdouble	ilXYZdouble, ilXYZSdouble	ilXYZCdouble, ilXYZCSdouble

Coefficients

The structures listed below, which are defined in the header file *il/ilPolyDef.h*, hold coefficients of one-dimensional and two-dimensional (first-, second-, third-, fourth-, fifth-, sixth-, and seventh-order) equations. They're useful with the operator classes that derive from ilWarpImg.

```
struct ilCoeff1 { float con, x; };
struct ilCoeff2 { float con, x, x2; };
struct ilCoeff3 { float con, x, x2, x3; };
struct ilCoeff4 { float con, x, x2, x3, x4; };
struct ilCoeff5 { float con, x, x2, x3, x4, x5; };
struct ilCoeff6 { float con, x, x2, x3, x4, x5, x6; };
struct ilCoeff7 { float con, x, x2, x3, x4, x5, x6, x7; };
```

```
struct ilCoeff1_2d {
 float con,
 y, x;
};
```

```
struct ilCoeff2_2d {
 float con,
 y, x,
 y2, xy, x2;
};
```

```
struct ilCoeff3_2d {
 float con,
 y, x,
 y2, xy, x2,
 y3, xy2, x2y, x3;
};
```

```
struct ilCoeff4_2d {
 float con,
 y, x,
 y2, xy, x2,
 y3, xy2, x2y, x3,
 y4, xy3, x2y2, x3y, x4;
};
```

```
struct ilCoeff5_2d {
 float con,
 y, x,
 y2, xy, x2,
 y3, xy2, x2y, x3,
 y4, xy3, x2y2, x3y, x4,
 y5, xy4, x2y3, x3y2, x4y, x5;
};

struct ilCoeff6_2d {
 float con,
 y, x,
 y2, xy, x2,
 y3, xy2, x2y, x3,
 y4, xy3, x2y2, x3y, x4,
 y5, xy4, x2y3, x3y2, x4y, x5,
 y6, xy5, x2y4, x3y3, x4y2, x5y, x6;
};

struct ilCoeff7_2d {
 float con,
 y, x,
 y2, xy, x2,
 y3, xy2, x2y, x3,
 y4, xy3, x2y2, x3y, x4,
 y5, xy4, x2y3, x3y2, x4y, x5,
 y6, xy5, x2y4, x3y3, x4y2, x5y, x6,
 y7, xy6, x2y5, x3y4, x4y3, x5y2, x6y, x7;
};

/* can be used for orders up to 7 -- one dimensional */
struct ilCoeff_1d {
 int order;
 struct ilCoeff7 c;
};

/* can be used for orders up to 7 -- two dimensional */
struct ilCoeff_2d {
 int order;
 struct ilCoeff7_2d c;
};
```

The ilCoeff_1d struct provides a convenience operator that evaluates a two dimensional polynomial at a given *y* point to return a one-dimensional polynomial:

```
ilCoeff_1d(ilCoeff_2d& cf, float y);
```

The ilCoeff_2d class provides a few convenience operators as well. The function call operator is overloaded to accept an (*x*,*y*) point and return the evaluated result of the polynomial at that point:

```
float operator()(float x, float y);
```

The above operator is also defined for ilCoeff1_2d. ilCoeff_2d also provides the assignment operator = to copy the coefficients of one polynomial to another, and the == operator to compare the coefficients of two polynomials.

ilCoeff1_2d also provides convenience functions **compose()** and **invert()**.

```
void compose (const ilCoeff1_2d& u, const ilCoeff1_2d& v);
static int invert(const ilCoeff1_2d& xc,
                  const ilCoeff1_2d& yc,
                  const ilCoeff1_2d& uc,
                  const ilCoeff1_2d& vc);
```

compose() can be used to collapse two one-dimensional transformations into one. The coefficients of this transformation are recomputed to account for the coordinate transformations defined by *u* and *v*. **invert()** can be used to reverse the transformation defined by *xc* and *yc*. The coefficients of the reverse transformation are returned in *uc, vc*.

You may use these convenience operators only if you are programming in C++.

Error Codes

This section lists the error codes defined in the header file *il/ilError.h* as the enumerated type ilStatus. The function **getStatus()** returns an ilImage's current status; many other functions also return the type ilStatus.

ilOKAY	Successful operation
ilBADFILEREAD	Error reading from file
ilBADFILEWRITE	Error writing to file
ilBADMALLOC	**malloc()** or **new** returned NULL
ilBADIMGFMT	Bad image file format
ilBADDIMS	Bad dimensions
ilBADOBJ	Bad object on construction
ilBADATTR	Bad attributes
ilFMTUNSUP	Unsupported file format
ilBADPIXTYPE	Bad pixel type
ilBADCONFIG	Unsupported configuration
ilNORANDOMSEEK	Can't do random seek
ilBADSEEK	Error seeking on file
ilBADDECODE	Failure on decompression
ilREADONLY	Object is not writable
ilBADFIELDSET	Failed to set field in file header
ilBADCOMPRESSION	Invalid image compression
ilNULLOBJ	NULL object passed as parameter
ilBADINPUT	Invalid input passed
ilBADCOLFMT	Bad color format
ilBADOP	Bad operation attempted
ilBADFILEOPEN	Error opening file
ilBADMAGIC	Invalid magic number in file
ilEMPTYFILE	File is empty

ilDATACLIPPED	Data has been clipped
ilOUTOFBOUND	Parameter(s) out of bounds
ilTOOMANYLOCKED	Too many pages locked in image cache
ilLUTSIZEMISMATCH	Incompatible number of channels in lut and image
ilZERODIVIDE	Attempted to divide by zero
ilUNSUPPORTED	Attempted operation is unsupported
ilUSEDOLDLIMITS	Used old limits for histogram calculation
ilBADPAGEDIMS	TIFF page dimensions must be multiples of 8
ilBADTIFFDIR	Could not index into TIFF directory
ilNOTRESIDENT	Page isn't resident in cache
ilHWACCELFAIL	Unable to complete hardware accelerated operation

Enumerated Types and Constants

The IL uses enumerated types and defined constants extensively; they're defined in header files such as *il/ilTypes.h* and *il/ilDisplayDefs.h*. This section lists these types and constants in the following functional groups, according to what they're used for: describing image attributes, controlling the effect of operators, and controlling the display facility. All of these types are described in more detail in the relevant chapters of this guide.

Also note that NULL, TRUE, and FALSE have been defined as follows in the header file *il/ilDefs.h*:

```
#ifndef NULL
#define NULL 0
#endif
#undef TRUE
#define TRUE 1
#undef FALSE
#define FALSE 0
```

Describing Image Attributes

```
/* Define supported image data types*/

enum ilType {
 ilBit        /* single-bit */
 ilUChar      /* unsigned character (byte)*/
 ilChar       /* Color (Red, Green, Blue triplets) */
 ilUShort     /* unsigned short integer (nominally 16 bits)*/
 ilShort      /* signed short integer*/
 ilULong      /* unsigned long integer*/
 ilLong       /* long integer*/
 ilFloat      /* floating point */
 ilDouble     /* double precision floating point */
};

/* Define supported color models */

enum ilColorModel {
 ilMinWhite      /* Grayscale with minimum value white */
 ilMinBlack      /* Grayscale with minimum value black */
 ilRGB           /* Color (Red, Green, Blue triplets) */
 ilRGBPalette    /* Color-mapped values */
 ilRGBA          /* Color with transparency(alpha channel)*/
 ilHSV           /* Hue, Saturation, Value */
 ilCMY           /* Cyan, Magenta, Yellow */
 ilCMYK          /* Cyan, Magenta, Yellow, black */
 ilBGR           /* Color (Blue, Green, Red triplets) */
 ilABGR          /* Color (Alpha, Blue, Green, Red) */
 ilMultiSpectral /* Arbitrary number of chans */
 ilYCC           /* PhotoCD color model (Luminance,*/
                 /* Chrominance) */
};

/* Define supported coordinate spaces */

enum ilCoordSpace {
 ilUpperLeftOrigin
 ilUpperRightOrigin
 ilLowerRightOrigin
 ilLowerLeftOrigin
 ilLeftUpperOrigin
 ilRightUpperOrigin
 ilRightLowerOrigin
 ilLeftLowerOrigin
};
```

```
/* Define supported compression schemes */

enum ilCompress {
    ilNoCompression
    ilCCITTFAX3        /* CCITT Group 3 fax encoding */
    ilCCITTFAX4        /* CCITT Group 4 fax encoding */
    ilLZW              /* Lempel-Ziv & Welch */
    ilPACKBITS         /* Macintosh RLE */
    ilSGIRLE           /* SGI’s RLE compression */
};

/* Define the pixel component ordering */

enum ilOrder {
 ilInterleaved      /* Store as RGBRGBRGBRGB... */
 ilSequential       /* Store as RRR..GGG..BBB.. per line */
 ilSeparate         /* Store channels in separate pages */
};

/* Define the supported image file formats */
/* The ilFormat type is defined for backward */
/* compatibility. It will be phased out in a future */
/* release. The defines for each format are also for */
/* backward compatibility. Just use the string value for */
/* new code. */

typedef char* ilFormat      /* For backwards compatibility */
                            /* Will be phased out*/
#define ilFIT_IMG “FIT”     /* FIT format (IL example */
                            /* format) */
#define ilTIFF_IMG “TIFF”   /* TIFF format (Tagged Image */
                            /* File Format) */
#define ilSGI_IMG “SGI”     /* Classic SGI format (.rgb or */
                            /* .bw files) */
#define ilPCD_IMG “PhotoCD” /* Kodak PhotoCD format */

/* Define the supported image object class types */

enum ilImageType {
 ilMEM_IMG      /* Memory resident image */
 ilFILE_IMG     /* File-based image */
 ilGLDISP_IMG   /* GL Frame-buffer resident image */
 ilXDISP_IMG    /* X frame-buffer image */
 ilOP_IMG       /* Operator image */
 ilSYNTH_IMG    /* Synthetic image (no memory used) */
 ilX_IMG        /* X image */
 ilTEX_IMG      /* Image loaded in texture memory */
 ilAUX_IMG      /* Image resident in aux buffer */
};
```

Controlling Operators

```
/* Define the supported ROI (region of interest) types */

enum ilRoiType {
 ilRect      /* ROI is a rectangle */
 ilBitmap    /* ROI is a char* bitmap */
};

/* Define the supported resampling methods for warp */

enum ilResampType {
 ilUserDef      /* User-defined resampling */
 ilNearNb       /* Nearest neighbor resampling */
 ilBiLinear     /* Bilinear resampling */
 ilBiCubic      /* Bicubic resampling */
 ilMinify       /* Minification resampling */
 ilAutoResamp   /* Resampling chosen automatically */
};

/* Define supported edge modes for spatial operators */

enum ilEdgeMode {
 ilNoPad      /* No padding - output shrinks */
 ilPadSrc     /* Pad source with fill value */
 ilPadDst     /* Pad destination with fill value */
 ilWrap       /* Wrap data from opposite edge */
 ilReflect    /* Reflect data at edge */
};

/* Define compass directions for ilCompassImg operator */

enum ilCompassDir {
 ilCompassN   =    0,   /* Compass points every 45 degrees */
 ilCompassNE  =   45,
 ilCompassE   =   90,
 ilCompassSE  =  135,
 ilCompassS   =  180,
 ilCompassSW  =  225,
 ilCompassW   =  270,
 ilCompassNW  =  315
};

/* Define flip modes for ilRotZoomImg operator */

enum ilFlip {
 ilNoFlip      /* No flip */
 ilXFlip       /* Flip about X axis of input */
 ilYFlip       /* Flip about Y axis of input */
};

/* Define supported composition modes for ilBlendImg */

enum ilCompose {
 ilImgA       /* Only A shows */
 ilImgB       /* Only B shows */
 ilAoverB     /* A overlaps B; both show */
 ilBoverA     /* B overlaps A; both show */
 ilAinB       /* Only A shows and only at intersection */
 ilBinA       /* Only B shows and only at intersection */
 ilAoutB      /* Only A shows and only out of intersection */
 ilBoutA      /* Only B shows and only out of intersection */
 ilAatopB     /* At intersection: only A. Outside: only B. */
 ilBatopA     /* At intersection: only B. Outside: only A. */
 ilAxorB      /* At intersection: nothing. Outside: both */
 ilAplusB     /* At intersection: both. Outside: both */
};

/* Define supported combinations for morphological ops
ilErodeImg and ilDilateImg */

enum ilMorphType {
 ilBinBin      /* Binary image morphed with binary SE */
 ilBinGray     /* Binary image morphed with g-s SE */
 ilGrayBin     /* Gray-scale image morphed with binary SE */
 ilGrayGraySet /* Gray-scale image morphed with g-s SE */
 ilGrayGrayFct /* Gray-scale image morphed with g-s SE; */
               /* morph is performed as a function op */
};

/* The scope enumerator specifies the search scope for a */
/* property set/get operation. When getting a property, */
/* the scope values may be OR’ed together to specify a */
/* compound search. When setting a property, the scope */
/* values are mutually exclusive */

enum ilScope {
 ilNullScope      /* search nothing */
 ilInstanceScope  /* search properties of this object */
 ilClassScope     /* search properties of this class */
 ilGlobalScope    /* search global properties */
};
```

Controlling the Display Facility

```
/* Render Modes (GL or X) */

enum ilRender {
 ilGLRender             /* Use GL rendering */
 ilXRender              /* Use X Windows rendering */
};

/* ilParamMode specifies how coordinates are interpreted */

enum ilParamMode {
 ilDelVal              /* Delta relative to current */
 ilAbsVal              /* Absolute value */
 ilRelVal              /* Relative to start, update */
 ilOldRel              /* Relative to start, no update */
 ilParamMask           /* For internal use only */
};

/* ilLocMode is used by getLoc() and setLoc() to find xy
location of a pixel in image and move image or view to
specified location */

enum ilLocMode {
 ilLocIn              /* Locate xy in image’s input space */
 ilLocOut             /* Locate xy in image’s output space */
 ilLocImg /           /* Locate by moving image */
 ilLocView            /* Locate by moving view */
 ilLocMask            /* For internal use only */
};

/* ilDispMode specifies various display modes such as those
for clipping, stopping, etc. */

enum ilDispMode {
 ilDefault           /* No clip or defer. do swap */
 ilClip              /* Clip to display or image */
 ilDefer             /* Defer painting */
 ilNoSwap            /* Don’t swap buffers */
 ilDop               /* Override Nop flag */
 ilDispMask          /* For internal use only*/
 ilDspCoord          /* ilDisplayImg coordinates passed */
 ilScrCoord          /* Screen coordinates passed */
 ilCoordMask         /* For internal use only */
 ilDefaultCmap       /* Use default colormap */
};
```

```
/* ilWipeMode specifies mode for display() and wipe(), and
is returned from findView() and findViewEdge() */

enum ilWipeMode {
 ilNoView         /* No view found by findView() */
 ilRightEdge      /* Wipe right edge, display from right*/
 ilLeftEdge       /* Wipe left edge, display from left */
 ilTopEdge        /* Wipe top edge, display from top */
 ilBottomEdge     /* Wipe bottom edge, display from bottom */
 ilAllEdge        /* Display at center, wipe as inset */
 ilNoEdge         /* No edge found by findViewEdge() */
 ilWipeMask       /* For internal use only */
};

/* ilAlignMode specifies the display() operator modes.
Combinations of ilWipeMode can alternatively be used for the
first 5 values */

enum ilAlignMode {
 ilBottomLeft         /* Lower-left corner */
 ilBottomRight        /* Lower-right corner */
 ilTopLeft            /* Upper-left corner */
 ilTopRight           /* Upper-right corner */
 ilCenter             /* Align to center */
 ilNoAlign            /* Do not realign (no change) */
 ilAlignMask          /* For internal use only */
};

/* ilSplitMode specifies the split() operator modes */

enum ilSplitMode {
 ilRelSplit          /* Split & pos image relative to view */
 ilAbsSplit          /* Split & pos image at origin */
 ilRowSplit          /* Split into rows */
 ilColSplit          /* Split into columns */
 ilPackSplit         /* Split views and pack together */
 ilSplitMask         /* For internal use only */
};

/* Miscellaneous display attributes */

enum ilDispMisc{
 ilLast             /* Add view to bottom of view stack */
 ilDefaultMargin    /* Default margin width for findEdge() */
 ilHighlight        /* Find view and highlight its borders */
 };
```

Index

We'd Like to Hear From You

As a user of Silicon Graphics documentation, your comments are important to us. They help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics to comment on:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please include the title and part number of the document you are commenting on. The part number for this document is 007-1387-030.

Thank you!

Three Ways to Reach Us

The **postcard** opposite this page has space for your comments. Write your comments on the postage-paid card for your country, then detach and mail it. If your country is not listed, either use the international card and apply the necessary postage or use electronic mail or FAX for your reply.

If **electronic mail** is available to you, write your comments in an e-mail message and mail it to either of these addresses:

- If you are on the Internet, use this address: techpubs@sgi.com
- For UUCP mail, use this address through any backbone site: *[your_site]*!sgi!techpubs

You can forward your comments (or annotated copies of manual pages) to Technical Publications at this **FAX** number:

415 965-0964