The Vision Workbook:
A User's Guide to the NASA Vision Workbench v1.0

Matthew D. Hancher
Michael J. Broxton
Laurence J. Edwards

Intelligent Systems Division
NASA Ames Research Center

DRAFT

December 4, 2006

Acknowledgements

Many people have contributed to making this first Vision Workbench open-source release a reality. Thanks to Terry Fong, Kalmanje KrishnaKumar, and Dave Korsmeyer in the Intelligent Systems Division at NASA Ames for supporting us in this research and allowing us to pursue our crazy dreams. Thanks to Larry Barone, Martha Del Alto, Robin Orans, Diana Cox, Kim Chrestenson, and everyone else who helped us navigate the NASA open source release process. Thanks to Randy Sargent, Matt Deans, Liam Pedersen, and the rest of the Intelligent Robotics Group at Ames for lending their incredible expertise and being our guinea pigs time and again. Thanks to our interns—Kerri Cahoy, Ian Saxton, Patrick Mihelich, Joey Gannon, and Aaron Rolett—for bringing many exciting features of the Vision Workbench into being. Finally, thanks to all our users, past, present and future, for making software development enjoyable and worthwhile.

Portions of this work were funded by the Mars Critical Data Products Initiative and the Exploration Technology Development Program.

Contents

1 Introduction **7**

2 Getting Started **11**
2.1 Obtaining the Vision Workbench . . . 11
2.2 Building the Vision Workbench . . . 12
2.3 A Trivial Example Program . . . 13
2.4 Configuring the Build System . . . 14

3 Working with Images **19**
3.1 The `ImageView` Class . . . 19
3.1.1 The Basics . . . 19
3.1.2 The Standard Pixel Types . . . 20
3.1.3 Copying `ImageViews` . . . 22
3.1.4 `ImageView` as a STL-Compatible Container . . . 22
3.1.5 Image Planes . . . 23
3.2 Image File I/O . . . 23
3.2.1 Reading and Writing Image Files . . . 23
3.2.2 More Sophisticated File I/O . . . 24
3.3 Manipulating Images . . . 25
3.3.1 Simple Image Manipulation . . . 25
3.3.2 Image Algorithms . . . 27

4 Image Processing **31**
4.1 Image Filtering . . . 31
4.1.1 The Special-Purpose Filters . . . 31
4.1.2 Edge Extension Modes . . . 32
4.1.3 General Convolution Filtering . . . 33
4.2 Doing Math with Images . . . 34
4.2.1 Image Operators . . . 34
4.2.2 Mathematical Functions . . . 35
4.3 Vectors and Matrices . . . 36
4.3.1 Vectors and Vector Operations . . . 36
4.3.2 Matrices and Matrix Operations . . . 39
4.4 Transforming or Warping Images . . . 42
4.4.1 Transform Basics . . . 42
4.4.2 Creating a New Transform . . . 44
4.4.3 Advanced Techniques . . . 46

5 The Core Vision Workbench Type System **49**
5.1 The Scalar Types . . . 49
5.2 Type Deduction . . . 50
5.3 The Pixel Types . . . 51

6 The Camera Module **55**
6.1 The Pinhole Camera Model . . . 55
6.1.1 Perspective Projection . . . 55
6.2 The Camera Model Base Class . . . 57
6.3 Built-in Camera Models . . . 58
6.3.1 Pinhole Cameras . . . 58
6.3.2 Linescan Cameras . . . 59
6.4 Tools for Working With Camera Images . . . 60
6.4.1 Inverse Bayer Pattern Filtering . . . 60
6.4.2 Exif Exposure Data . . . 60

7 The `Mosaic` Module **65**
7.1 `ImageComposite` and Multi-Band Blending . . . 65
7.2 `ImageQuadTreeGenerator` . . . 67

8 High Dynamic Range Imaging **71**
8.1 Merging Bracketed Exposures . . . 71
8.1.1 Converting LDR Images to an HDR Image . . . 73
8.1.2 The Camera Response Curves . . . 73
8.2 Tone Mapping . . . 74
8.2.1 Global Operators . . . 74
8.2.2 Local Operators . . . 76
8.3 Command Line Tools . . . 76
8.4 Other Resources . . . 76

9 The Cartography Module **81**
9.1 Software Dependencies . . . 81
9.2 The `GeoReference` Class . . . 82
9.2.1 The Datum . . . 82
9.2.2 The Affine Transform . . . 82
9.2.3 Putting Things Together . . . 83
9.3 Geospatial Image Processing . . . 83
9.3.1 The `GeoTransform` Functor . . . 83
9.4 Georeferenced File I/O . . . 83
9.4.1 `DiskImageResourceGDAL` . . . 83

10 Advanced Topics **85**
10.1 Working with Shallow Views . . . 85
10.2 Efficient Algorithms and `pixel_accessor` . . . 85
10.3 Rasterization, Efficiency, and Tiled Computation . . . 85
10.4 Generic Image Buffers . . . 85
10.5 The File I/O System . . . 85
10.6 Frequency-Domain Image Processing . . . 85

Chapter 1

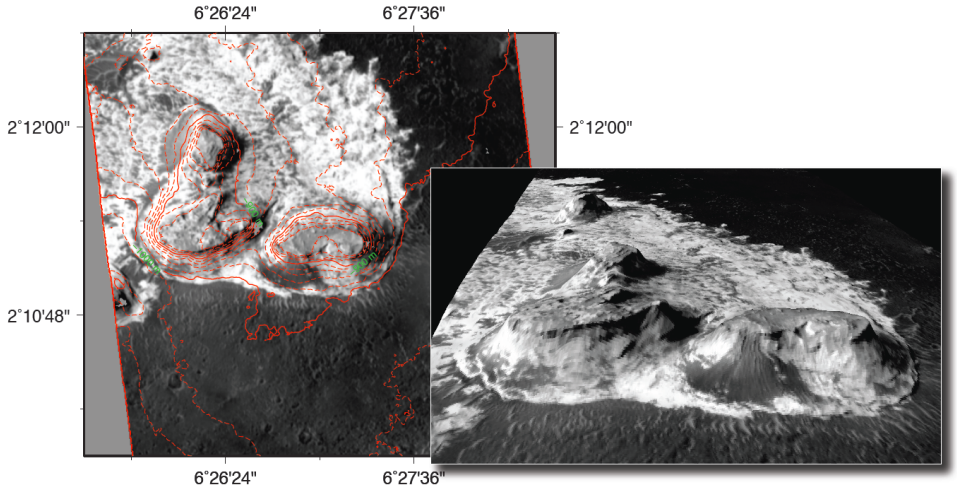
Introduction

This document is designed to be a gentle introduction to programming with the NASA Vision Workbench, a C++ image processing and machine vision library. The Vision Workbench was developed through a joint effort of the Intelligent Robotics Group (IRG) and the Adaptive Control and Evolvable Systems Group (ACES) within the Intelligent Systems Division at the NASA Ames Research Center in Moffett Field, California. It is distributed under the NASA Open Source Agreement (NOSA) version 1.3, which has been certified by the Open Source Initiative (OSI). A copy of this agreement is included with every distribution of the Vision Workbench in a file called `COPYING`.

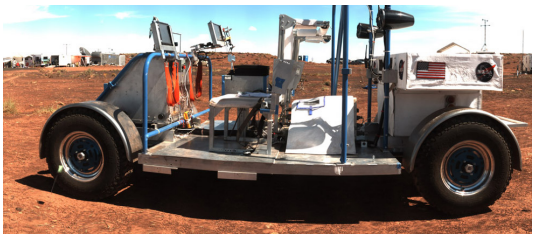
You can think of the Vision Workbench as a "second-generation" C/C++ image processing library. It draws on the authors' experiences over the past decade working with a number of "first-generation" libraries, such as OpenCV and VXL, as well as direct implementations of image processing algorithms in C. We have tried to select and improve upon the best features of each of these approaches to image processing, always with an eye toward our particular range of NASA research applications. The Vision Workbench has been used within NASA for a wide range of image processing tasks, including alignment and stitching of panoramic images, high-dynamic-range imaging, texture analysis and recognition, lunar and planetary map generation, and the production of 3D models from stereo image pairs. A few examples of image data that has been processed with the Vision Workbench are show in Figure 1.1.

The Vision Workbench was designed from the ground up to make it quick and easy to produce efficient implementations of a wide range of image processing algorithms. In many cases code written using the Vision Workbench is significantly smaller and more readable than code written using more traditional approaches. At its core is a rich set of template-based image processing data types representing pixels, images, and operations on those images, as well as mathematical entities like vectors and geometric transformations and image file I/O. On top of this core it also provides a number of higher-level image processing and machine vision modules, providing features including camera geometry modeling, high-dynamic-range imaging, interest point detection and matching, image mosaicing and blending, and geospatial data management.

That said, the Vision Workbench is not for everyone, and in particular it is not intended as a drop-in replacement for any existing image processing toolkit. It is specifically designed for image processing in the context of machine vision, so it lacks support for things like indexed color palettes that are more common in other areas. It also lacks a number of common features that the authors have simply not yet had a need for, such as morphological operations. If you encounter one of these holes while using the Vision Workbench please let us know: if it is an easy hole to fill we may be able to do so quickly. Finally, there are many application-level algorithms, such as face recognition, that



(a)



(b)

(c)

(d)

Figure 1.1: Examples of image data processed with the help of the Vision Workbench. (a) A Martian terrain map generated from stereo satellite imagery. (b,c) Original and high-dynamic-range image mosaics from a NASA field test. (d) A lunar base map generated from the Clementine data set.

have been implemented using other computer vision systems and are not currently provided by the Vision Workbench. If one of these meets your needs there is no compelling reason to re-implement it using the Vision Workbench instead. On the other hand, if no existing high-level tool solves your problem then you may well find that the Vision Workbench provides the most productive platform for developing something new.

Since this is the first public release of the Vision Workbench, we thought we should also provide you with some sense of the direction the project is headed. It is being actively developed by a small but growing team at the NASA Ames Research Center. A number of features are currently being developed internally and may released in the future, including improved mathematical optimization capabilities, a set of Python bindings, and stereo image processing tools. Due to peculiarities of the NASA open-source process we cannot provide snapshots of code that is under development and not yet approved for public release. If you have a specific use for features that are under development, or in general if you have suggestions or feature requests, please let us know. Knowing our users' needs will help us prioritize our development and, in particular, our open-source release schedule.

We hope that you enjoy using the Vision Workbench as much as we have enjoyed developing it! If you have any questions, suggestions, compliments or concerns, please let us know. Contact information is available at the bottom of the `README` file included with your distribution.

Chapter 2

Getting Started

This chapter describes how to set up and start using the Vision Workbench. It explains how to obtain the Vision Workbench and its prerequisite libraries, how to build and install it, and how to build a simple example program. This chapter does *not* discuss how to program using the Vision Workbench. If that's what you're looking for then skip ahead to Chapter 3.

2.1 Obtaining the Vision Workbench

Most likely if you are reading this document then you already know where to obtain a copy of the Vision Workbench sources if you haven't obtained them already. However, if not, a link to the most up-to-date distribution will always be available from the NASA Ames open-source software website, at `opensource.arc.nasa.gov`.

In addition to obtaining the Vision Workbench, you will also need to obtain and install whatever pre-requisite libraries you will need. The only strict requirement is the Boost C++ Libraries, a set of extensions to the standard C++ libraries that is available from `www.boost.org`. Many modern Linux systems come with some version of Boost already installed, generally in the directory `/usr/include/boost`. The Vision Workbench has been tested with Boost versions 1.32 and later.

Other libraries are required only if you want to use particular features of the Vision Workbench. A summary of the various libraries that the Vision Workbench will detect and use if present is given in Table 2.1. It lists the particular Vision Workbench module that uses the library, whether it is required or optional for that module, and where the library can be obtained. Details of each of the modules and the features that are enabled by each dependency are given in the corresponding sections of this book. If you are just starting out with the Vision Workbench, it is generally fine to begin only with Boost. You can always go back and rebuild the Vision Workbench with support for additional features later if you discover that you need them.

One dependency that is worth discussing briefly is LAPACK, which provides Vision Workbench with a computational linear algebra back end. LAPACK is a comprehensive and widely used linear algebra support library in the public domain. LAPACK also require the Basic Linear Algebra Subroutines (BLAS) library, which is usually bundled with LAPACK.

The basic matrix and vector algebra in the Math module does not depend on LAPACK and BLAS, however the routines in `<vw/Math/LinearAlgebra.h>` will only be built if LAPACK is detected by the build system. For your convenience, we provide a stand-alone LAPACK and BLAS distribution on the Vision Workbench web page. This distribution has been tested with the Vision Workbench, so we recommend its use if you are installing LAPACK for the first time. However, other versions of LAPACK and BLAS that come pre-installed on your system will probably work

Name	Used By	Source
Boost	All	`http://www.boost.org/`
Boost	Portions of Math, HDR	See note in Section 2.1
PNG	FileIO (opt.)	`http://www.libpng.org/`
JPEG	FileIO (opt.)	`http://www.ijg.org/`
TIFF	FileIO (opt.)	`http://www.libtiff.org/`
OpenEXR	FileIO (opt.)	`http://www.openexr.com/`
PROJ.4	Cartography (req.)	`http://www.remotesensing.org/proj/`
GDAL	Cartography (opt.)	`http://www.remotesensing.org/gdal/`

Table 2.1: A summary of Vision Workbench dependencies.

just as well. In particular, Mac OS X users *do not* need to install LAPACK; machine optimized linear algebra support is provided by Apple's `veclib` framework on Mac OS X. Remember to add the `-framework veclib` flag when linking your application against the Vision Workbench if you are using the functions in `<vw/Math/LinearAlgebra.h>` on the mac platform.

2.2 Building the Vision Workbench

If you are using a UNIX-like platform such as Linux or Mac OS it is generally straightforward to build the Vision Workbench once you have installed any necessary libraries. First unpack the distribution, go to the distribution's root directory, and configure the build system by running "`./configure`". This script will examine your machine to determine what build tools to use and what libraries are installed as well as where they are located. Near the end of its output it will list whether or not it was able to find each library and which Vision Workbench modules it is going to build. You should examine this output to confirm that it was able to find all the libraries that you had expected it to. If not then you may need to configure the build system to search in the right places, as discussed in Section 2.4.

Assuming the output of the `configure` script looks good, you can now proceed to build the Vision Workbench itself by running "`make`". Most of the Vision Workbench is header-only, so "building" the Vision Workbench should be relatively quick. Once the build is complete, confirm that things are working properly by building and running the unit tests by typing "`make check`". If there are no errors, the final step is to install the Vision Workbench headers, library, and sample programs using "`make install`". By default the installation location is the directory `/usr/local`, so you will need to obtain the necessary privileges to write to this directory using a command such as `su` or `sudo`. If you do not have administrator privileges on you computer then see Section 2.4 for information on how to specify an alternative installation directory.

Building the Vision Workbench under Windows is possible, but it is not currently automatically supported. The easiest thing to do is to include the `.cc` files from the Vision Workbench modules that you want to use directly in your own project file. You will of course still need to install the Boost libraries as well as any other libraries you want to use. Pre-built Windows versions of a number of libraries, such as the JPEG, PNG, and TIFF libraries, are available online from the GnuWin32 project at `gnuwin32.sourceforge.net`. You will need to configure your project's include file and library search paths appropriately. Also be sure to configure your project to define the preprocessor symbol `NOMINMAX` to disable the non-portable Windows definitions of `min()` and `max()` macros, which interfere with the standard C++ library functions of the same names.

```
1   #include <iostream>
2   #include <vw/Image.h>
3   #include <vw/FileIO.h>
4
5   int main( int argc, char *argv[] ) {
6     try {
7       VW_ASSERT( argc==3, vw::ArgumentErr() << "Invalid command-line args." );
8       vw::ImageView<float> image;
9       read_image( image, argv[1] );
10      write_image( argv[2], image );
11    }
12    catch( vw::Exception& e ) {
13      std::cerr << "Error: " << e.what() << std::endl;
14      std::cerr << "Usage: vwconvert <source> <destination>" << std::endl;
15      return 1;
16    }
17    return 0;
18  }
```

Listing 1: [`vwconvert.cc`] A simple demonstration program that can copy image files and convert them from one file format to another.

2.3 A Trivial Example Program

Now that you've built and installed the Vision Workbench let's start off with a simple but fully-functional example program to test things out. The full source code is shown in Listing 1. You should be able to obtain an electronic copy of this source file (as well as all the others listed in this book) from wherever you obtained this document. For now don't worry about how this program works, though we hope it is fairly self-explanatory. Instead, just make sure that you can build and run it successfully. This will ensure that you have installed the Vision Workbench properly on your computer and that you have correctly configured your programming environment to use it.

The program reads in an image from a source file on disk and writes it back out to a destination file, possibly using a different file format. When reading and writing images, the Vision Workbench infers the file format from the file extension of the filename. This example program takes the source and destination filenames as two command-line arguments. For example, to convert a JPEG image called `image.jpg` in the current directory into a PNG image you might say:

```
vwconvert image.jpg image.png
```

Note that exactly what image file formats are support will depend on what file format libraries you have installed on your system.

In order to to build this program you will need to configure your compiler to find the Vision Workbench headers and then configure your linker to find not only the Vision Workbench libraries but also all of the libraries that the Vision Workbench in turn requires.

Some Vision Workbench header files include boost headers, and the compiler needs to be able to find these files when you build your application. No additional configuration is necessary if boost is installed in a stardard system directory, however for non-standard installations, you will need to direct the compiler (usually using the `-I` flag) to the right directory. Note that the Vision

Workbench's dependency on boost is unique in this regard; you do not normally need to configure the compiler to find header files for Vision Workbench third party library dependencies.

Keeping track of nested library dependencies like this can be difficult. The Vision Workbench addresses this problem using the GNU `libtool` utility, and we suggest that you use it too. All Vision Workbench libraries are built with an accompanying `libvw<module_name>.la` file that encodes dependency information that `libtool` later uses to pull in all required library dependencies automatically. It's easy to use, and it lets you take advantage of the work that the Vision Workbench build system does to locate your libraries and sort out their dependencies.

Listing 2 shows a sample `Makefile` that demonstrates how to build a Vision Workbench application using `libtool`, among other things. If you already have your own `Makefile` or other build system, the important section to look at is the section titled "Linking rule". It demonstrates how to invoke `libtool` to build a program: invoke the compiler as you usually would, but prefix the command with "`libtool --mode=link`". This will make `libtool` interpret the command line it has been given as a linking command, filling in all the specifics about library dependencies. In this case it will recognize the `-lvw` option, and will expand it to include references to all the libraries upon which the Vision Workbench depends.

You can test this by creating an empty directory and copying the files `vwconvert.cc` and `Makefile.example` into it, renaming the latter as simply `Makefile`. (Both of these files are included in the Vision Workbench source distribution in the directory `docs/workbook`.) You should then be able to build the program by running "`make`". This assumes that you have `libtool` installed on your computer. If not, don't worry: the Vision Workbench includes a copy of the `libtool` script in the base directory of the source distribution. If you see an error message suggesting that `libtool` cannot be found you may need to modify your `Makefile` so that the `LIBTOOL` variable explicitly points to this file.

If you choose not to use `libtool` then you will need to manually ensure that all the necessary dependencies are linked in to your program. The easiest way to be sure that you aren't missing any is to look inside the same files that `libtool` would use to generate the list, the `.la` files. For example, the `vw` library that is included by the `-lvw` option points to the file `lib/libvw.la` underneath whatever directory you installed the Vision Workbench in. This is a human-readable file that lists this library's dependencies, among other things. If any of these dependency libraries are themselves `.la` files then you will need to examine them in turn to find all the recursive dependencies. As you can imagine, this is a cumbersome process, and we suspect that in then end you'll be much happier using `libtool` directly instead.

Using `libtool` on Mac OS X

Users of Mac OS X should be aware that the `libtool` command available in this environment is different than the GNU `libtool` we are discussing here. On these systems, you will need to use the `glibtool` command or use the `libtool` script in the root of the Vision Workbench source distribution directory.

2.4 Configuring the Build System

The Vision Workbench build system offers a variety of configuration options that you provide as command-line flags to the `configure` script. We'll discuss a few of the most important options here, but for a complete list you can run "`./configure --help`". As an alternative to

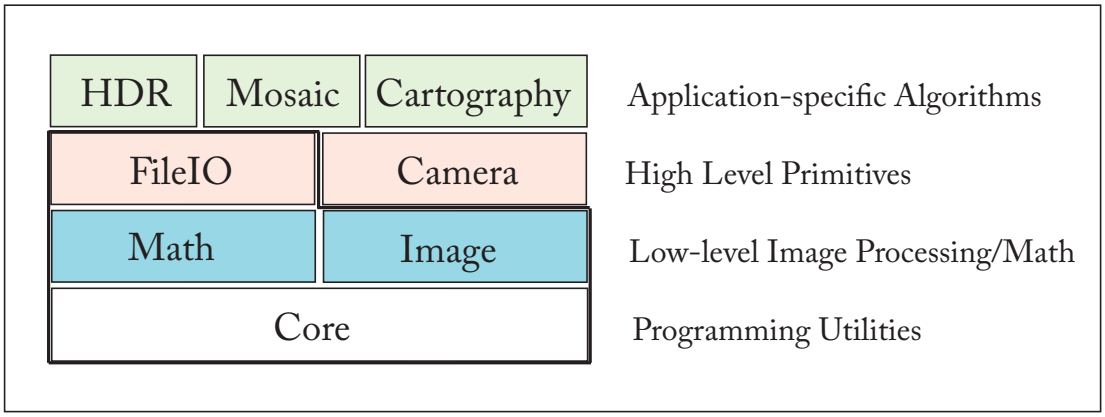


Figure 2.1: *Vision Workbench inter-module dependencies.* Module in this figure depend on those beneath them. These dependencies split the modules into four general classes of increasing complexity and sophistication. The modules surrounded by the bold outline are considered the "foundation" modules that are part of the most basic Vision Workbench distribution.

specifying command-line flags every time, you may instead create a file called `config.options` with your preferences in the base directory of the Vision Workbench repository. A file called `config.options.example` is provided that you can copy and edit to your liking. Note that none of this has any impact on Visual Studio users, who must instead configure their projects by hand.

The single most important option is the `--with-paths=PATHS` flag, where you replace `PATHS` with a whitespace-separated list of paths that the build system should search when looking for installed libraries. For example if you specify the option `--with-paths=/foo/bar` then it will search for header files in `/foo/bar/include`, library files in `/foo/bar/lib`, and so on. The default search path includes a number of common locations for user-installed libraries, such as `/usr/local`, `$(HOME)/local`, and `/sw`. The `PKG_PATHS` configuration file variable has the same effect as this option.

The next most important options have the form `--enable-module-foo[=no]`, where `foo` is replaced by the lower-case name of a module such as `mosaic` or `hdr`. This allows you to control whether or not certain modules are built. Disabling modules that you do not use can speed up compilation and testing time, which is especially useful if you are making changes to the Vision Workbench source and need to recompile often. The corresponding configuration file variables have the form `ENABLE_MODULE_FOO`, in all-caps, and are set to either `yes` or `no`.

It is worth mentioning that the Vision Workbench has several inter-module dependencies that you should take into account when enabling and disabling modules. These are shown in Figure 2.4.

Two handy options, `--enable-optimize` and `--enable-debug`, determine the compiler options used when building the few library files. You can again specify an optional argument of the form `=no` to disable the corresponding feature, and you can also specify a particular optimization level in the same manner. For example, if you want to make it as easy as possible to debug Vision Workbench code using a debugger you might use `--enable-optimize=no --enable-debug` to disable all optimizations and include debugging symbols. The corresponding configuration file variables are `ENABLE_OPTIMIZE` and `ENABLE_DEUBG`. Keep in mind that since most Vision Workbench code is header-only you should remember to configure your own project similarly or you may not notice any difference. For normal non-debugging use, we strongly recommend that you enable moderate

compiler optimization; much of the heavily templatized and generic Vision Workbench code requires basic optimizations such as function inlining to achieve a reasonable level of performance.

Finally, to specify that the build system should install the Vision Workbench someplace other than `/usr/local`, specify the path using the `--prefix=PATH` option. The corresponding configuration file variable is, of course, called `PREFIX`.

```
# The Vision Workbench installation prefix (/usr/local by default)
VWPREFIX = /usr/local

# If you don't have libtool installed, you can specify the full
# path to the libtool script in the base directory of your Vision
# Workbench source tree, e.g. $(HOME)/VisionWorkbench-1.0/libtool
LIBTOOL = libtool

# Compilation flags:
# -O3   turns on optimization, which you should almost always do
# -g    enables debugging support
# -Wall turns on all compiler warnings
CXXFLAGS = -I$(VWPREFIX)/include -O3 -g -Wall

# Boost:
# The Vision Workbench header files require the boost headers. If
# boost is installed in a non-standord location, you may need
# to uncomment this line and insert the path to the boost headers.
# CXXFLAGS += -I<path to boost include dir>

# Linking flags:
# -lvw includes the Vision Workbench core libraries
LDFLAGS = -L$(VWPREFIX)/lib -lvw

# Object files:
# List the object files needed to build your program here.
SOURCES = vwconvert.o

# Linking rule:
# Duplicate and modify this rule to build multiple programs.
vwconvert: $(OBJECTS)
        $(LIBTOOL) --mode=link $(CXX) $(LDFLAGS) -o $@ $^

# Clean-up rule:
clean:
        rm -f *.o *~ \#*
```

Listing 2: [`Makefile.example`] An example `Makefile` that shows how to build a Vision Workbench program using `libtool`.

Chapter 3

Working with Images

This chapter is designed to be a first introduction to programming using the Vision Workbench. It describes images, pixels, color spaces, image file I/O, and basic image manipulation, setting the stage for the fundamental image processing operations described in Chapter 4.

3.1 The `ImageView` Class

The `ImageView` class is the centerpiece of the Vision Workbench in most applications. Simply put, it represents an image in memory. The class is similar to related classes that appear in other C++ computer vision libraries, including VXL, GIL, and VIGRA, so if you are already familiar with one of those libraries you should find nothing too foreign here.

3.1.1 The Basics

An `ImageView` represents a two-dimensional rectangular array of data, such as an image of pixels. It is actually a class *template*, and when you declare an `ImageView` object you specify the particular kind of data that it should contain. For example, you can make an `ImageView` of RGB (red/green/blue) pixels to represent a full-color image or an `ImageView` of vectors to represent a vector field. You specify the pixel type as a template parameter to the `ImageView` class like this:

```
ImageView<PixelRGB<float32> > my_image;
```

In this case we've made a full-color RGB image. Notice that `PixelRGB` is itself a template: here we've specified that we want each channel of each RGB pixel to be stored as a 32-bit floating-point number. All of the core pixel types in the Vision Workbench are themselves templates like this.

The `ImageView` class is defined in the C++ header file `<vw/Image/ImageView.h>`, and the standard pixel types are defined in the header `<vw/Image/PixelTypes.h>`. Thus, for the above line of code to compile you must include those two headers at the top of your program. (Alternatively, all of the header files relating to basic image manipulation are collected together in the convenience header `<vw/Image.h>`.) Furthermore, all of the core classes and functions of the Vision Workbench are defined in the C++ namespace `vw`. One way to use them is to be fully specific:

```
vw::ImageView<vw::PixelRGB<vw::float32> > my_image;
```

The other way, which may be simpler for new users, is to bring the entire `vw` namespace into the global scope by saying

```
using namespace vw;
```

at the top of your program after you've included the necessary headers. For brevity, in the examples in this book we will often assume that you have included the necessary headers and we will omit explicit references to namespace `vw`. The exception to this is the complete programs, such as `vwconvert.cc` (Listing 1, above), which are intended to be fully self-contained.

By default the dimensions of an `ImageView` are zero, which may not be what you want. One option is to specify an image's dimensions when we construct it:

```
ImageView<PixelRGB<float> > my_image( 320, 240 );
```

This creates an image with 320 columns and 240 rows. If we ever want to set or change the size of an image later on in the code we can use the `set_size()` method:

```
my_image.set_size( 640, 480 );
```

You can also find out how many columns or rows an image has using the `cols()` and `rows()` methods, respectively:

```
int width = my_image.cols();
int height = my_image.rows();
```

Note that when you call `set_size()` with new image dimensions the Vision Workbench allocates a new chunk of memory of the appropriate size. This is a destructive operation: any old data is not copied into the new buffer, and the old buffer will be automatically deallocated if no other objects are using it.

Once you've made an `ImageView`, the simplest way to access a particular pixel is by indexing directly into it:

```
PixelRGB<float> some_pixel = my_image( x, y );
```

In this example we've assumed that `x` and `y` are integer variables with the desired pixel's coordinates. For a less trivial example, one way to fill our image with the color red would be to loop over all the rows and columns, setting each pixel at a time:

```
PixelRGB<float> red(1.0, 0.0, 0.0);
for ( int y=0; y<my_image.rows(); ++y )
  for ( int x=0; x<my_image.cols(); ++x )
    my_image(x,y) = red;
```

This is not the fastest way to access the pixels of an image, but it is arguably the most flexible. (Later we will learn about much simpler ways to fill an image with a single color.)

3.1.2 The Standard Pixel Types

The Vision Workbench provides a number of standard pixel types that you can use to manipulate the most common sorts of images. We've already encountered `PixelRGB`, the standard RGB pixel type. As we mentioned earlier, this is a template class whose template parameter specifies the underlying numeric data type used to store each channel of the pixel. This is called the pixel's *channel type*. The Vision Workbench defines convenient platform-independent names for the standard channel types, so that you never have to worry about whether `int` or `short` is 16 bits wide on your platform.

Type	Description	Notes
`int8`	Signed 8-bit integer	
`uint8`	Unsigned 8-bit integer	Common for low-dynamic-range imaging
`int16`	Signed 16-bit integer	
`uint16`	Unsigned 16-bit integer	
`int32`	Signed 32-bit integer	
`uint32`	Unsigned 32-bit integer	
`int64`	Signed 64-bit integer	
`uint64`	Unsigned 64-bit integer	
`float32`	32-bit floating point	Common for high-dynamic-range imaging
`float64`	64-bit floating point	

Table 3.1: The standard Vision Workbench channel types.

Type	Description	Channels
`PixelGray<T>`	Grayscale	Grayscale value (`v`)
`PixelGrayA<T>`	Grayscale w/ alpha	Grayscale value (`v`), alpha (`a`)
`PixelRGB<T>`	RGB	Red (`r`), green (`g`), blue (`b`)
`PixelRGBA<T>`	RGB w/ alpha	Red (`r`), green (`g`), blue (`b`), alpha (`a`)
`PixelHSV<T>`	HSV	Hue (`h`), saturation (`s`), value (`v`)
`PixelXYZ<T>`	XYZ	CIE 1931 X (`x`), Y (`y`), and Z (`z`) channels
`Vector<T,N>`	An `N`-dimensional vector	`N` vector components
`T`	A unitless scalar	N/A

Table 3.2: The standard Vision Workbench pixel types. The channel type `T` should generally be one of the types from Table 3.1.

These Vision Workbench channel types are listed in Table 3.1. These are the only channel types with which the Vision Workbench has been tested, so it is best to stick to these unless you have a compelling reason not to.

The standard pixel types are listed in Table 3.2. The first four, used for grayscale and RGB images with and without alpha channels, are the most common. (For those of you who are unfamiliar with the term, an *alpha* channel is used to represent the opacity of a pixel. For the rest of you, note that the Vision Workbench generally stores alpha pixels in pre-multiplied form.)

Each of the channels in a pixel can be accessed by indexing into it directly, as in `my_pixel(i)` or `my_pixel[i]`. The order of the channels is the same as the order in which they appear in the name of the type. If you know a particular pixel's type you can also access it's channels by name, so for example `my_rgb_pixel.r()` access an RGB pixel's red channel. (Note that grayscale values are accessed via `v()`, for "value".)

When you are writing Vision Workbench programs you may often find yourself working with only one pixel type at a time. In this case it can be convenient to place a `typedef` near the top of your file defining a convenient shorthand:

```
typedef vw::ImageView<vw::PixelRGB<float32> > Image;
```

This way you can refer to your RGB image type by the much shorter identifier `Image`. In the remainder of this book when we say `Image` you may assume that you may substitute the `ImageView` class type that is most appropriate for your application.

Standard conversions are provided among all the RGB and grayscale pixel types, and also between PixelRGB and the special color types PixelHSV and PixelXYZ. The ImageView class can take advantage of these pixel conversions to perform color space conversion on entire images. For example, images are generally stored on disk in an RGB color space but it is sometimes helpful to convert them to HSV for processing. This is easy with the Vision Workbench:

```
ImageView<PixelRGB<float> > rgb_image;
read_image( rgb_image, filename );
// Convert the RGB image to HSV:
ImageView<PixelHSV<float> > hsv_image = rgb_image;
```

(We'll have more to say about read_image() shortly, but it does what you'd expect.) Later you could assign the HSV image back to an RGB image prior to saving it to disk.

3.1.3 Copying ImageViews

In the Vision Workbench, ImageView objects have *shallow copy semantics*. That is, when you copy an ImageView you're making a new ImageView that points to the *same* data, rather than a new copy of the data. This is a relatively inexpensive operation, which makes it perfectly reasonable to do things like construct a std::vector of ImageViews. The underlying image data is reference-counted, and when the last ImageView stops using a block of image data it is deallocated.

Though this behavior can be quite powerful, it may not always be what you want. If you ever need to make a duplicate of an ImageView, so that you can modify one without affecting the other, you should use the copy() function found in <vw/Image/Algorithms.h>.

```
// This makes a shallow copy, pointing to the same:
Image new_image_1 = my_image;
// This makes a deep copy, pointing to new, identical data:
Image new_image_2 = copy( my_image );
```

It is important to understand that this shallow copy behavior only applies when the source and destination image types—and in particular the source and destination pixel types—are *identical*. If the pixel types are different then you are not actually making a copy in the C++ sense of the word but are instead assigning one image view to another. In the above example involving RGB and HSV images, even though the source and destination objects are both ImageViews they in fact have different types because they have different template parameters. Therefore the data is copied deeply while being converted to the new pixel type. This holds even if the source and destination pixel types differ only in their underlying channel type.

3.1.4 ImageView as a STL-Compatible Container

An ImageView can be thought of as a container of pixels, and in fact you can use it as a standard C++ container class. The iterator type is, as expected, called ImageView<T>::iterator, and it allows you to access each of the pixels of an image one at a time. The begin() and end() methods return iterators pointing to the first and one-past-the-last pixels, respectively. The first pixel is located at position $(0, 0)$, and incrementing the iterator advances to the next column. After it passes through the last column, the iterator wraps around to the beginning of the next row.

This C++ Standard Template Library (STL) compliant iterator exists mainly to allow you to take advantage of the many *algorithms* provided by the STL that operate on containers. For example, you can use sort() to sort all of the pixel values in an image.

```
std::sort( my_image.begin(), my_image.end() );
```

That particular example may be more cute than it is useful, but others occur more frequently. For instance, you can use `std::count()` to count the number of pixels with a particular value, or `std::replace()` to replace all pixels that have one value with another.

3.1.5 Image Planes

The `ImageView` class also supports another feature found in many other image processing libraries: image planes. Like duct tape, planes are the wrong solution to almost every problem, and we discourage their use. Basically, planes allow you to store some number of two-dimensional pixel arrays of the same size ("planes") together in a single object. Planes are different from channels in that the number and meaning the planes is not specified at compile time. This means that the Vision Workbench can not take advantage of that information as readily: for example, it has no way to know whether a three-plane image is RGB, HSV, or something altogether different, and it cannot optimize operations by unrolling inner loops as it is able to with channels. (It may not be readily apparent, but the sample program shown in Listing 1 demonstrates one of the very few possibly-legitimate uses of planes; this will be discussed more in the following section on File I/O.)

To create a multi-plane image, pass the desired number of planes as a third argument to the `ImageView` constructor or to the `set_size()` method. You can query the number of planes in an image with the `planes()` method. To access a pixel in particular plane of an image, pass the plane as a third argument when indexing into the image.

```
Image my_image(320,240,3);      // A 3-plane image
my_image.set_size(320,240,3);   // Same here
int planes = my_image.planes(); // Now planes == 3
Pixel pix = my_image(x,y,p);    // Access a pixel
```

Once again, if you are thinking about using planes we encourage you to first consider these alternatives. If you want a way to store a collection of related images, consider using a `std::vector` of `ImageViews` instead. If you just want to store a bunch of numbers at each pixel location, consider using `Vector<T,N>` as a pixel type.

3.2 Image File I/O

The most common way to get image data into and out of the Vision Workbench is by loading and saving images using file I/O. There are several mechanisms for doing this, varying in complexity, flexibility and (for the time being) completeness of implementation.

3.2.1 Reading and Writing Image Files

The simplest method for file I/O is to use the `read_image()` and `write_image()` functions, passing them an `ImageView` and the filename of the image file on disk that you would like to read from or write to.

```
read_image( image, filename );
write_image( filename, image );
```

Name	Extension(s)	Description
PNG	`.png`	Standard for loss-less compression
JFIF/JPEG	`.jpg`, `.jpeg`	Standard for lossy compression, no alpha
TIFF	`.tif`, `.tiff`	Highly flexible, complicated
OpenEXR	`.exr`	High dynamic range
PDS	`.img`	Planetary Data System images

Table 3.3: The standard Vision Workbench image file formats. Which formats your installation supports depends on what supporting libraries you have installed. Adding support for additional file formats is discussed in Chapter 10.

Notice that the order of arguments to these two functions is reversed: in both cases the destination is first and the source second.

Both functions determine the image file type by looking at the extension of the filename that you provide them. The exact set of file formats that are supported depends on which file format libraries the Vision Workbench found on your system when you build it. For example JPEG support depends on `libjpeg`, and so forth. The file formats that the Vision Workbench is designed to support are listed in Table 3.3. Note that the file extensions are case-insensitive.

Image data on disk is generally stored with one of the four standard pixel types: grayscale or RGB with or without alpha. The image reading and writing routines will freely convert between these formats. You should generally create an `ImageView` with the pixel type that you would like to work with and let the file I/O system take care of the rest.

```
ImageView<PixelGrayA<float> > image;
read_image( image, "some_file.jpg" );
```

In this example we loaded in a JPEG image file (which has an RGB pixel format) and then converted the data grayscale and padded it with a constant alpha value of 1.0, corresponding to fully opaque. Attempting to save this image back as a JPEG file would reverse the conversion. (Any transparency is composited on to a black background whenever the alpha channel is removed.)

3.2.2 More Sophisticated File I/O

We will only provide an overview of the more advanced file I/O techniques here. Many of them are partially (in some cases barely) implemented. If you want to use any of these features you can learn more about them in Chapter 10.

Images on disk are handled via an abstract image resource class, called `DiskImageResource` and defined in `<vw/FileIO/DiskImageResource.h>`. You can create one directly using the same file-extension-based file type deduction mechanism discussed above.

```
DiskImageResource *dir1 = DiskImageResource::open( filename );
DiskImageResource *dir2 = DiskImageResource::create( filename, format );
```

In the first case we are opening an existing file, and in the second case we are creating a new file. Creating a new file resource requires providing some hints about the underlying image format, such as its dimensions and pixel type, which are supplied by a `GenericImageFormat` object.

Once you have a resource you can query it for information about its dimensions, pixel format and channel type. For example, you can choose to process different pixel formats differently.

```
switch( dir1->pixel_format() ) {
  case VW_PIXEL_GRAY: /* process grayscale file */ break;
  case VW_PIXEL_RGB:  /* process RGB file */       break;
  /* ... */
}
```

You can use the `DiskImageResource`'s `read()` and `write()` methods to read the data into or write the data out of an `ImageView`, respectively.

If you wish to force a particular file format, you can create a resource object of the appropriate type directly.

```
DiskImageResourcePNG *dirp1 = new DiskImageResourcePNG( filename );
DiskImageResourcePNG *dirp2 = new DiskImageResourcePNG( filename, format );
```

In this case we show how to create PNG image resources. If you do this then you can take advantage of any special services provided by the particular file format's resource type, such as the ability to read or write special file header information.

Finally, you can make a read-only `ImageView`-like object that corresponds to an image on disk. This is called a `DiskImageView` and is defined in the header of the same name. This can be used to process images that are too large to be loaded into memory all at once.

3.3 Manipulating Images

We have seen how images are represented via the `ImageView` class, how to save and load them to and from disk, and how to manipulate their pixels individually. Now it is time to begin discussing how to perform slightly higher-level operations on images.

3.3.1 Simple Image Manipulation

We begin with the simple image manipulation functions listed in Table 3.4 and defined in the header file `<vw/Image/Manipulation.h>`. Many of these should be self-explanatory. The results of applying several of these transforms to an image are shown in Figures 3.1(b)–3.1(i). The 90-degree rotation functions are one of the few places where the Vision Workbench makes any kind of assumption about the interpretation of the x, y coordinate system. When it is necessary to make a distinction we assume that the origin $(0, 0)$ is the top-left corner of the image. If you have been interpreting the origin as the top-right or bottom-left you will need to invert your notion of clockwise vs. counter-clockwise when calling these two functions.

None of these functions, by themselves, modify image data or produce new images. Instead, each function returns a special *view* on to the same image data. In most cases you will assign the result to another `ImageView`, causing the data to be processed and the resulting image to be stored in the new buffer:

```
image2 = flip_vertical( image1 );
```

It's worth taking a moment to study exactly what goes on behind the scenes when you perform an operation like this. First the Vision Workbench resizes the destination image (`image2` in the above example) if necessary so that its dimensions are the same as those of the source image (a flipped version of `image1`). Second it computes the result of the operation, storing the result in the destination image as it goes. The important point is that if the destination image already has the

Function	Description
`rotate_180(im)`	Rotate the image 180 degrees
`rotate_90_cw(im)`	Rotate the image 90 degrees clockwise
`rotate_90_ccw(im)`	Rotate the image 90 degrees counter-clockwise
`flip_vertical(im)`	Flip the image vertically
`flip_horizontal(im)`	Flip the image horizontally
`transpose(im)`	Transpose the `x` and `y` coordinates of the image
`crop(im,x,y,c,r)`	Crop the image, specifying (x, y) and $(cols, rows)$
`crop(im,bbox)`	Crop the image, specifying a bounding box
`subsample(im,factor)`	Subsample the image by an integer factor
`subsample(im,xfac,yfac)`	Subsample the image by integer factors in x and y
`select_col(im,col)`	Refers to an individual column of an image
`select_row(im,row)`	Refers to an individual row of an image
`select_plane(im,plane)`	Refers to an individual plane of an image
`select_channel(im,channel)`	Refers to an individual channel of an image
`channels_to_planes(im)`	Interprets a multi-channel image as a multi-plane image
`pixel_cast<PixelT>(im)`	Casts an image to a new pixel type
`pixel_cast_rescale<PixelT>(im)`	Casts an image to a new pixel type, with rescaling
`channel_cast<ChanT>(im)`	Casts an image to a new channel type
`channel_cast_rescale<ChanT>(im)`	Casts an image to a new channel type, with rescaling
`planes_to_channels<PixelT>(im)`	Interprets a multi-plane image as a multi-channel image
`weighted_rgb_to_gray(im)`	Converts RGB to grayscale with default weights
`weighted_rgb_to_gray(im,r,g,b)`	Converts RGB grayscale with the given weights

Table 3.4: The simple image manipulation functions, defined in the header file `<vw/Image/Manipulation.h>`. The functions in the top section return writable views.

same dimensions as the source image then it is *not* resized or reallocated. This avoids unnecessary memory allocations in common situations, such as when you are processing many identically-sized images in a loop. However, it also means that you must be careful when processing an image and assigning it back to itself:

```
image = flip_vertical( image );   // Bad idea: self-assignment
```

In this example, the destination image clearly has the same dimensions as the source (since they are the same image) and so no new image buffer is allocated. As a result the `flip_vertical` operation will clobber the source image with partial results, producing garbage. One solution to this problem is to force the creation of a temporary buffer using the `copy` function:

```
image = copy( flip_vertical( image ) );   // Much better
```

The functions listed in the upper section of Table 3.4 all provide new ways of accessing the same data without doing any additional processing. As a result, these functions are all able to return *writable* views of their image argument. That is, you can use them to modify an image by placing them on the left side of an equals sign. For example, suppose you want to add a small inset to a larger image, by copying a small image into the larger one at a particular position. One easy way is to specify the destination region using the `crop()` function:

Function	Description
`copy(im)`	Produce a deep copy of an image
`fill(im,value)`	Fill an image with a pixel value *in-place*
`clamp(im,[low],[high])`	Clamp values to the given range
`normalize(im,[low],[high])`	Normalize values to the given range
`threshold(im,[thresh],[low],[high])`	Threshold an image to two values
`grassfire(im)`	Compute the grassfire image of an image
`bounding_box(im)`	Return the bounding box of an image
`nonzero_data_bounding_box(im)`	Compute the bounding box of nonzero data
`image_blocks(im,width,height)`	Tile an image with bounding boxes

Table 3.5: The simple image algorithms defined in the header file `<vw/Image/Algorithms.h>`.

```
int cols = small_image.cols(), rows = small_image.rows();
crop( large_image, xpos, ypos, cols, rows ) = small_image;
```

Here we've cropped a region of the large image and used it for writing instead of reading. Note that the assignment proceeds just as before: first the destination image dimensions are checked, and then the data is copied. However in this case the Vision Workbench will throw an exception if the dimensions differ, since it is not meaningful to "resize" a cropped region in the same sense that you can freely resize an `ImageView`. This approach can also be used, for example, to replace one channel of a multi-channel image using `select_channel()`.

The functions listed in the lower section of Table 3.4, on the other hand, all do a small amount of processing of pixel values. The `pixel_cast()` function converts all the pixels in an image to the given new pixel type. The `pixel_cast_rescale()` variants rescale the values if the channel type has changed, e.g. mapping the 0–255 range of `uint8` on to the 0.0–1.0 nominal range of `float32`. The `channel_*` variants cast the pixels to have the given new channel type, leaving the overall pixel format unchanged. The `pixels_to_channels()` function takes a multi-plane image and reinterprets it as a multi-channel image with the given pixel type. Finally, `weighted_rgb_to_gray` converts RGB pixels to the corresponding grayscale pixel type using an arbitrary weighting of the red, green, and blue channels. The default weights are based on a human perceptual model that weights green most strongly, followed by red and then blue.

3.3.2 Image Algorithms

We will now introduce a number of additional simple image operations that are defined in the header file `<vw/Image/Algorithms.h>`. You have already seen one of them, `copy()`, which forces the creation of a deep copy of an image in a new buffer. The rest are listed in Table 3.5. The result of two of these functions can be seen in Figures 3.1(j) and 3.1(k). We hope to implement a number of additional image algorithms, mirroring the STL container algorithms but optimized for images, at some point in the future.

The `fill()` function is noteworthy because it is currently the only core Vision Workbench function that modifies image data *in-place*. It is especially useful for filling a single channel of an image. For example, you can use it to make an RGBA image fully opaque.

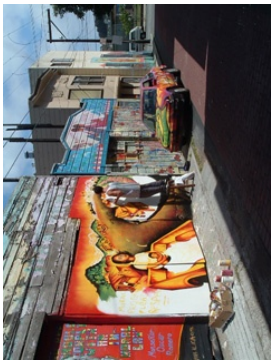
```
fill( select_channel( rgba_image, 3 ), 1.0 );
```

(Note that 1.0 represents fully-opaque if the image has a floating-point channel type.)

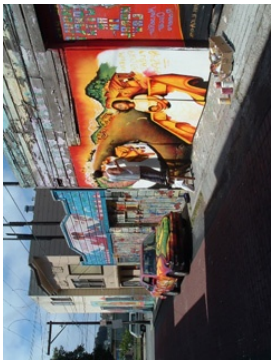
(a) `mural` (Original)

(b) `rotate_180(mural)`

(c) `rotate_90_cw(mural)`



(d) `rotate_90_ccw(mural)`



(e) `transpose(mural)`

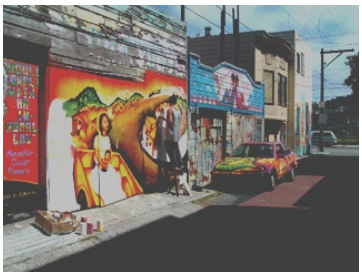
(f) `flip_vertical(mural)`

(g) `flip_horizontal(mural)`

(h) `crop(mural,80,60,160,120)`

(i) `subsample(mural,2)`

(j) `threshold(mural,0.5)`



(k) `clamp(mural,0.25,0.75)`

Figure 3.1: Sample output from the simple image operations discussed in this section.

The `clamp()`, `normalize()`, and `threshold()` functions return modified versions of their image arguments. You can assign the result back to the original image, or you can save it in a different image instead and keep the original. The `clamp()` function clamps the values in the image to the given range. The `normalize` function scales and shifts the values of an image so that the values span the specified range. The default range is from zero to the nominal maximum value for the channel type, e.g. 1.0 for floating-point images. This is particularly useful for saving intermediate results of your algorithms to disk for debugging. Finally, the `threshold` function returns a two-valued image based on whether the pixels in the source image is greater than or less than the given threshold value. The default high and low output values are the same as for `norm`, and the default threshold is zero. For example, this line will convert a floating-point grayscale image to pure black-and-white:

```
image = threshold( image, 0.5 );
```

The `grassfire()` algorithm, named for the algorithm that it implements, is more specialized. It takes an image and efficiently computes how far each pixel is from from a pixel whose value is zero, assuming that pixels outside the image boundaries all have zero value. It measures distance in the four-connected Manhattan sense, i.e. as the sum of the horizontal and vertical distances. This algorithm is used in a variety of applications, such as avoiding obstacles and unknown terrain in path planning.

Chapter 4

Image Processing

Now that we've covered all the basics of how to manipulate images, it's time to move on to some more interesting image processing tasks. We begin with an introduction to image filtering, followed by a discussion of image math. We then take a brief detour to introduce the Vision Workbench's `Vector` and `Matrix` classes before describing image transformation and warping.

By the end of this chapter you will have encountered all of the core building blocks that comprise the heart of the Vision Workbench. There are a number of directions that you can go from here, depending on what you are hoping to accomplish. We conclude this chapter with an overview of the many more specialized features of the Vision Workbench and a discussion of where to look (in this book and elsewhere) in order to learn more about them.

4.1 Image Filtering

Image filtering has traditionally been the bread and butter of image processing software packages. The Vision Workbench includes a number of functions to perform the most common filtering operations. We will first describe the special-purpose filters, and then we will discuss the more general convolution-based linear filtering functions. All of the filter functions discussed in this section are defined in the header file `<vw/Image/Filter.h>`. We will not discuss frequency-domain filtering in this chapter; that is covered later in Section 10.6.

4.1.1 The Special-Purpose Filters

At the moment only three special-purpose filters are fully supported. The first is a Gaussian smoothing or blurring filter, which convolves the image with a discrete Gaussian kernel that has a user-specified standard deviation (a.k.a. "sigma") and user-specified size in each axis. In order for the filter to accurately approximate a Gaussian, the size of the kernel should be at least a few times the standard deviation. However, unnecessary computation is performed if the size is much larger than that. You can omit the size arguments, in which case the function will pick a kernel size based on your standard deviation that is reasonable for most applications. In the most common case the two standard deviations are equal, in which case you need only specify a single value for sigma.

```
result = gaussian_filter( image, sigma );
result = gaussian_filter( image, xsigma, ysigma );
result = gaussian_filter( image, xsigma, ysigma, xsize, ysize );
```

Function	Description
`gaussian_filter(im,...)`	Apply a Gaussian smoothing filter to an image
`derivative_filter(im,...)`	Apply a discrete differentiation filter to an image
`laplacian_filter(im,...)`	Apply a discrete Laplacian filter to an image
`convolution_filter(im,...)`	Apply a general 2D convolution filter to an image
`separable_convolution_filter(im,...)`	Apply a separable convolution filter to an image

Table 4.1: The Vision Workbench image filtering functions, defined in `<vw/Image/Filter.h>`.

Type	Description
`ConstantEdgeExtend`	Assume constant (i.e. nearest-neighbor) edge extension
`ZeroEdgeExtend`	Assume zero-value edge extension

Table 4.2: The edge extension modes.

In these examples, the `sigma` arguments are generally floating-point whereas the `size` variables are integers.

The next filter is the derivative filter, which performs a discrete spatial differentiation of your image. Here again, you can specify the order of differentiation in the two axes as well as the filter kernel size.

```
result = derivative_filter( image, xderiv, yderiv );
result = derivative_filter( image, xderiv, yderiv, xsize, ysize );
```

There is a minimum filter size below which it is not possible compute any given derivative, and these functions will throw an exception if you try. For the most part it is a good idea to just let the Vision Workbench pick the kernel size.

The final special-purpose filter is the Laplacian filter, which performs a discrete approximation to the Laplacian operation $\nabla^2 = \frac{d^2}{dx^2} + \frac{d^2}{dy^2}$.

```
result = laplacian_filter( image );
```

This filter does not take any special parameters. Note that if you are accustomed to using a "larger" derivative or Laplacian filter to reduce the effect of noise, you are probably better off applying a smoothing operation (e.g. via `gaussian_filter()`) first.

4.1.2 Edge Extension Modes

To filter the regions near the edges of an image properly, filters like these need to make some sort of assumption about the contents of the source image *beyond* the image boundaries. This is generally referred to as "edge extension". The default assumption made by the filters discussed in this section is that in each direction the image is extended with a constant value equal to the value of the nearest edge pixel. However, you can specify an alternative edge extension mode if you wish, by passing an extra argument to the filters. The C++ type of the argument determines the edge extension mode used.

```
result = gaussian_filter( image, 3.0, ConstantEdgeExtend() );
result = gaussian_filter( image, 3.0, ZeroEdgeExtend() );
```

Both of these examples filter the source image using a standard deviation of three pixels and an automatically-chosen kernel size. However, the first explicitly requests the default edge extension behavior, while the second requests that the source image be assumed to be zero outside the image boundaries.

Notice the "extra" set of parentheses after the names of the edge extension modes. Remember that those names are C++ *types*, and you can only pass an *object* as an argument to a function. Those parentheses invoke the edge extension type's constructor, returning a dummy object that you pass as the final argument to the filtering function. If you find this confusing, don't worry too much about it right now. Just keep in mind that when you're using a type as an argument to a function to change it's behavior you need the extra parentheses. The types that are currently supported as edge extension modes are listed in Table 4.2.

4.1.3 General Convolution Filtering

Most of the filters used in image processing are convolution filters, which express each output pixel as a fixed weighted sum of neighboring input pixels. An image convolution filter is usually described by a rectangular array of weights called the *kernel*. The easiest way to think about an image kernel is as the result that you would desire from the filter if the input image had the value 1 at the origin and zero everywhere else. (This is also known as the "impulse response" of the filter.) For example, a first-order derivative filter in the x direction might have the kernel $[\ 1 \quad 0 \quad -1\ ]$. In this case we also need to know that the middle number of the kernel (the zero in this case) is the kernel's origin.

In the Vision Workbench, convolution kernels—which as we've said are nothing more than rectangular arrays of numbers—are represented by images. The pixel type for a kernel should generally be a scalar type such as `float`. Once you've put the kernel that you'd like into an image it is straightforward to use it to filter another image.

```
ImageView<float> kernel;
/* set up your kernel here */
result = convolution_filter( image, kernel );
```

In this case the Vision Workbench assumes that the center pixel of the kernel is the kernel's origin. If this is not what you want then you can specify the coordinates of the kernel's origin explicitly instead.

```
result = convolution_filter( image, kernel, ox, oy );
```

In either case you can also optionally specify an edge extension mode, just like you could for the special-purpose filters.

Convolution filtering can be computationally expensive if the kernel is large. Fortunately, many useful kernels have a special form that makes it possible to improve the performance considerably. These are called *separable* kernels, and are themselves the result of convolving a single-column image with a single-row image. In other words, the kernel K must satisfy $K(x, y) = K_x(x)K_y(y)$ for some functions K_x and K_y. The Gaussian and derivative filters are both of this form, for example, though the Laplacian filter is not.

The Vision Workbench provides special support for efficient convolution filtering with separable kernels. You must supply the *separated* kernel, i.e. two one-dimensional kernels.

```
result = separable_convolution_filter( image, xkernel, ykernel );
result = separable_convolution_filter( image, xkernel, ykernel, ox, oy );
```

Per-pixel Sum	Per-pixel Difference	Per-pixel Product	Per-pixel Quotient
`image + image`	`image - image`	`image * image`	`image / image`
`image += image`	`image -= image`	`image *= image`	`image /= image`
`image + value`	`image - value`	`image * value`	`image / value`
`image += value`	`image -= value`	`image *= value`	`image /= value`
`value + image`	`value - image`	`value * image`	`value / image`

Table 4.3: The Vision Workbench image operators are included automatically when you include `<vw/Image/ImageMath.h>`).

As in the general 2D convolution case, the origin of the kernel is assumed to be in the middle if you do not specify otherwise and in either case you can add an optional argument specifying the edge extension mode. You can still supply the one-dimensional kernels as images, just as you did in the general 2D convolution case, but here you can also provide them in another STL-compliant container, such as a `std::vector` or (as we shall introduce later this chapter) a `vw::Vector`. If you do chose to represent the kernels as images, remember that each should have one of the dimensions set to 1.

4.2 Doing Math with Images

In image processing it is often desirable to perform some mathematical operation on every pixel of an image, or to corresponding pixels from several images. For example gamma correction involves applying a mathematical function to each pixel, and background subtraction involves subtracting the corresponding pixels from two images. In the Vision Workbench, these operations and others like them fall under the rubric of "image math", and the functions to support them are defined in the header `<vw/Image/ImageMath.h>`.

4.2.1 Image Operators

In most cases writing code to perform image math is trivial. The mathematical expressions that you would normally write for individual pixels work just as well for whole images of pixels. For example, consider the background subtraction problem mentioned above.

```
result_image = input_image - background_image;
```

That's all there is to it. Setting up an IIR low-pass filter to estimate the background image is just as easy.

```
background_image = alpha*input_image + (1-alpha)*background_image;
```

(Here we're assuming that `alpha` is a small positive floating-point number.) The important point is that there is no need for you to write a loop that performs an operation like this on each pixel. Just write the mathematical expression, replacing pixels with images, and you're all set.

This works, of course, because the Vision Workbench has overloaded the standard C++ mathematical operators to work on images. These operators are listed in Table 4.3. Operation with scalars is treated identically to per-pixel operation with constant-value images. In order to simplify division with large images, the image division operators have been designed so that division by zero returns zero instead of throwing an exception.

Function	Description	Function	Description
`sin`	Sine, $\sin x$	`asin`	Inverse sine, $\sin^{-1} x$
`cos`	Cosine, $\cos x$	`acos`	Inverse cosine, $\cos^{-1} x$
`tan`	Tangent, $\tan x$	`atan`	Inverse tangent, $\tan^{-1} x$
`atan2`	Two-argument form of inverse tangent, $\tan^{-1} x/y$		
`sinh`	Hyperbolic sine, $\sinh x$	`cosh`	Hyperbolic cosine, $\cosh x$
`tanh`	Hyperbolic tangent, $\tanh x$	`exp`	Exponential, e^x
`log`	Natural logarithm, $\ln x$	`log10`	Base-10 logarithm, $\log_{10} x$
`ceil`	Ceiling function, $\lceil x \rceil$	`floor`	Floor function, $\lfloor x \rfloor$
`sqrt`	Square root, $\sqrt{x}$	`pow`	Power function, x^y
`asinh`	Inverse hyperbolic sine, $\sinh^{-1} x$	`acosh`	Inverse hyperbolic cosine, $\cosh^{-1} x$
`atanh`	Inverse hyberbolic tangent, $\tanh^{-1} x$	`cbrt`	Cube root, $\sqrt[3]{x}$
`exp2`	Base-2 exponential, 2^x	`expm1`	Exponential minus 1, $e^x - 1$
`log2`	Base-2 logarithm, $\log_2 x$	`log1p`	Lograithm of one-plus, $\ln(1 + x)$
`tgamma`	Gamma function, $\Gamma(x)$	`lgamma`	Log of Gamma function, $\ln \lvert\Gamma(x)\rvert$
`hypot`	Hypotenuse, $\sqrt{x^2 + y^2}$	`copysign`	Sign-copying function
`round`	Rounding function	`trunc`	Floating-point truncation
`fdim`	Positive difference, $\max(x - y, 0)$		

Table 4.4: The Vision Workbench image math functions, as defined in `<vw/Image/ImageMath.h>`. The functions in the bottom section are not available under the Windows operating system.

There is one important issue to bear in mind when using image operators: the underlying per-pixel operations must themselves be meaningful. For example, multiplying an image whose pixel type is `PixelGray` by an image whose pixel type is `PixelRGB` is not well-defined, and attempting to do so will result in a compiler error. The Vision Workbench will not automatically "promote" the grayscale image to RGB.

This raises the question of what happens when you multiply two images both of whose pixel type is, for example, `PixelRGB`. What does it mean to multiply two RGB colors? Multiplication is defined for numbers, not colors. The answer is that in this situation the Vision Workbench will actually perform the mathematical operation on a per-*channel* basis rather than just a per-pixel basis.

A good rule of thumb when working with image operators is to restrict yourself to operating on images of the same type, or combinations of images of one type and images of scalars. As long as you obey this rule you should find that the image operators always do what you expect.

4.2.2 Mathematical Functions

Of course, C++ provides a range of mathematical functions, too, such as exponentials and logarithms, trigonometric functions, and so forth. The Vision Workbench extends these functions to operate on images as well. The supported functions are listed in Table 4.4. Note that these image functions are built on top of the standard C++ functions that operate on regular numbers. Therefore, the Vision Workbench only supports those functions that are provided by your platform. In

particular, the bottom half of Table 4.4 lists functions that are *not* currently available under the Microsoft Windows operating system.

You can use these functions just like you use the mathematical operators: write the same expression that you would write for individual pixels, but substitute whole images instead.

```
float gamma = 1.8;
result_image = pow( input_image, gamma );
```

This example demonstrates how to use the `pow()` function to gamma-correct an image. Here the variable `gamma` is a floating-point number representing the desired gamma correction factor for the entire image. However, if instead we wanted to apply a variable gamma correction factor on a per-pixel basis, the following code would do the trick.

```
ImageView<float> gamma_image; // Initialize with different gamma values
result_image = pow( input_image, gamma_image );
```

This example demonstrates that the arguments of a two-argument mathematical function can be either scalar or image values. Just as with the operators, scalar arguments are treated the just like a constant-value image.

Note that unlike the normal mathematical functions that C++ inherited from C, it is not necessary (or correct) to use a different function name when you are working with `float` image data than you would use to work with `double` image data. The function names listed in Table 4.4 are correct for image math in all cases. Those in turn use the proper underlying mathematical functions as appropriate—for example, `sin()` invokes `sinf()` on each pixel if it is applied to a `float`-based image.

4.3 Vectors and Matrices

Before introducing the next image processing topic, image transformation and warping, we must first take a brief detour to introduce the Vision Workbench vector and matrix classes. We will assume in this chapter that you have a good familiarity with the underlying mathematical entities that these classes represent. Note that our mathematical usage of the word "vector" here is somewhat different from the C++ standard library's use of the word to mean a dynamically-resizable array.

4.3.1 Vectors and Vector Operations

The Vision workbench vector class is called, appropriately enough, `Vector`. Like `ImageView`, `Vector` is a template class whose first template parameter is required and specifies the underlying numeric type. However, while the dimensions of an image are always specified at run-time via the image's constructor or the `set_size()` method, `Vector` comes in two variants. The first form behaves in just the same way, but the second form has a fixed size that is specified at compile time. This eliminates the need for frequent dynamic allocation when working with vectors in the common case when the vector dimension is known.

Declaring either type of vector is straightforward:

```
Vector<float> vector1(3);
Vector<float,3> vector2;
```

Both of those statements declare three-dimensional vectors of floating-point numbers. In the first case the vector is allocated dynamically on the heap and the size could have been chosen at run-time. In the second case the vector is allocated statically on the stack, but the dimension can *not* vary at run time. The first form is generally useful when, say, reading a large vector of data in from a file, while the second form is more useful when performing geometric computations.

The second, fixed-dimension form also has special constructors that you can use to initialize the vector contents:

```
Vector<float,3> vector2(1,2,3);
```

These constructors are available with up to four arguments. Alternatively, you can construct both fixed-size and dynamically-sized vector with data copied from a block of memory that you point them to:

```
float *some_data;
Vector<float> vector1(3, some_data);
Vector<float,3> vector2(some_data);
```

Remember that this copies the data, so it can be inefficient; see the discussion of `VectorProxy` below for an alternative. Three of the most commonly used vector types have special aliases, for convenience:

```
typedef Vector<double,2> Vector2;
typedef Vector<double,3> Vector3;
typedef Vector<double,4> Vector4;
```

These types are used throughout the Vision Workbench as the standard geometric vector types.

You can query a vector about its size (i.e. dimension or length) with the `size()` method, and you can index into a vector to access individual elements:

```
for( unsigned i=0; i<vector1.size(); ++i ) vector1(i) = 0;
```

This example loops over all the elements of a vector, setting them to zero. You can also into a vector with square brackets instead of parentheses if you prefer. For fixed-length vectors there is one more way to access up to the first three elements, via methods called `x()`, `y()`, and `z()`.

```
vector2.x() = 0;  // Set the first element to zero
```

These methods are only available if the vector has sufficient length. For example, attempting to use the `z()` method of a vector of type `Vector<float,2>` will result in a compile-time error. Remember, these methods are only available for fixed-size vectors, *not* dynamically-sized ones. Dynamically-sized vectors, however, can be resized:

```
vector1.set_size(10);
```

The `set_size()` function takes an optional second argument that specifies whether or not the vector contents should be preserved. This argument defaults to `false`, so in the above example the old contents (if any) are lost.

The `Vector` classes support the standard mathematical operations of vector addition and subtraction and scalar multiplication and division via the usual C++ operators. They also support the a range of elementwise mathematical operations, such as adding a scalar to each element or multiplying the corresponding elements of two vectors, via functions of the form `elem_*`. There are

Function	Description
`- vector`	Vector negation
`vector + vector`	Vector sum
`vector - vector`	Vector difference
`vector * scalar`	Scalar product
`scalar * vector`	Scalar product
`vector / scalar`	Scalar quotient
`vector += vector`	Vector sum assignment
`vector -= vector`	Vector difference assignment
`vector *= scalar`	Scalar product assignment
`vector /= scalar`	Scalar quotient assignment
`elem_sum(vector,vector)`	Elementwise vector sum (same as `+` operator)
`elem_sum(vector,scalar)`	Elementwise sum of a vector and a scalar
`elem_sum(scalar,vector)`	Elementwise sum of a scalar and a vector
`elem_diff(vector,vector)`	Elementwise vector difference (same as `-` operator)
`elem_diff(vector,scalar)`	Elementwise difference of a vector and a scalar
`elem_diff(scalar,vector)`	Elementwise difference of a scalar and a vector
`elem_prod(vector,vector)`	Elementwise product of two vectors
`elem_prod(vector,scalar)`	Elementwise vector product (same as `*` operator)
`elem_prod(scalar,vector)`	Elementwise vector product (same as `*` operator)
`elem_quot(vector,vector)`	Elementwise quotient of two vectors
`elem_quot(vector,scalar)`	Elementwise quotient (same as `/` operator)
`elem_quot(scalar,vector)`	Elementwise quotient of a scalar and a vector
`norm_1(vector)`	1-norm of a vector, i.e. $\sum \lvert v_i \rvert$
`norm_2(vector)`	Euclidean 2-norm of a vector, i.e. $\sqrt{\sum v_i^2}$
`norm_2_sqr(vector)`	Squared 2-norm of a vector, i.e. $\sum v_i^2$
`norm_inf(vector)`	Infinity-norm of a vector, i.e. $\max \lvert v_i \rvert$
`sum(vector)`	Sum of elements, i.e. $\sum v_i$
`prod(vector)`	Product of elements, i.e. $\prod v_i$
`normalize(vector)`	The normalized form of a vector, i.e. $v/\lvert v \rvert$
`dot_prod(vector,vector)`	Vector dot product, i.e. $u \cdot v$
`cross_prod(vector,vector)`	Vector dot product, i.e. $u \times v$

Table 4.5: The vector math functions defined in `<vw/Math/Vector.h>`.

a number of vector norms and related functions, as well as a vector dot product and cross product. (The cross product is, of course, only valid for three-dimensional vectors.) The complete list of vector math functions defined in `<vw/Math/Vector.h>` is given in Table 4.5.

A `Vector` object is also a *container* in the C++ Standard Template Library sense of the word. There is a `Vector<...>::iterator` type that serves as the vector's iterator, and there are `begin()` and `end()` methods that return iterators to the first and one-past-the-last elements, as usual. This can be an extremely convenient way to load data into and out of `Vector`s.

You can extract a portion of a vector using the `subvector()` function, which takes three arguments: the original vector, the position of the first element to extract, and the number of elements in the resulting vector:

```
Vector<float,3> vector2 = subvector(vector1,5,3);
```

This example copies the fifth, sixth, and seventh elements of `vector1` into a new three-element vector.

The streaming operator `<<` is also defined for writing vectors to C++ output streams, which you can use to dump vector contents for debugging:

```
Vector<float,3> vector2(1,2,3);
std::cout << vector2 << std::endl;
// The output is: [3](1,2,3)
```

Note that the size of the vector is printed first, followed by the vector's contents.

Sometimes it can be useful to work with data that is already stored in memory as though it were stored in a `Vector` object. As long as the data is stored in the usual packed format this is easy to do using the special `VectorProxy` type, which also comes in fixed-size and dynamically-sized variants:

```
float some_data[10] = {0,1,2,3,4,5,6,7,8,9};
VectorProxy<float> proxy1(10, some_data);
VectorProxy<float,10> proxy2(some_data);
```

The constructor arguments are the same as are used in `Vector` to initialize a vector with data from a block of memory, except the data is not copied. You can now treat these proxy objects just like the were regular `Vector`s, except the contents will be stored in the region of memory that you pointed them to. In some situations this can be considerably more efficient than copying the data unnecessarily. (It is of course not possible to resize a `VectorProxy`, since the proxy does not have any control over the memory that it is using.)

4.3.2 Matrices and Matrix Operations

The Vision Workbench `Matrix` class is the matrix counterpart to the `Vector` class, and behaves quite similarly. Once again, there are fixed-dimension and dynamically-sized versions:

```
Matrix<float> matrix1(3,3);
Matrix<float,3,3> matrix2;
```

Note that the arguments to matrix-related functions such as these constructors are given in i, j order, i.e. row followed by column. This is *different* from images, where arguments are given in x, y order, i.e. column followed by row. You may find this confusing at first if you are moving to the Vision Workbench from an environment like Matlab where there is no distinction between images and matrices. However, it is in keeping with the standard index ordering seen in the bulk of the image processing and mathematics literatures, respectively.

You can initialize the matrix with data already stored in memory, as long as the data is stored in a packed row-major format:

```
float some_data[4] = {1,2,3,4};
Matrix<float> matrix1(2,2,some_data);
Matrix<float,2,2> matrix2(some_data);
```

As in the case of `Vector`, the initialization data is *copied* into the matrix in this case, but there is also a proxy form that allows you treat in-memory data like an ordinary matrix:

```
float some_data[4] = {1,2,3,4};
MatrixProxy<float> matrix1(2,2,some_data);
MatrixProxy<float,2,2> matrix2(some_data);
```

The three most common matrix types have been given convenient aliases:

```
typedef Matrix<double,2,2> Matrix2x2;
typedef Matrix<double,3,3> Matrix3x3;
typedef Matrix<double,4,4> Matrix4x4;
```

These types are again the standard types used throughout the Vision Workbench in geometric applications.

You can query a matrix's dimensions using the `rows()` and `cols()` methods, and can index into the matrix to access individual elements. There are two ways to do this:

```
matrix(row,col) = 1;    // "New"-style indexing
matrix[row][col] = 1;   // "Old"-style indexing
```

A dynamically-sized matrix can be resized using the `set_size()` method:

```
matrix.set_size(rows,cols);
```

As in the case of resizing vectors, the default behavior is that any old data is not saved. The `set_size()` method takes an optional third boolean parameter that can be set to `true` to request that it preserve the overlapping entries.

Once you've made one or more matrices you can use a wide range of mathematical operator and functions to manipulate them. The standard C++ operators, elementwise math functions, and a number of other functions similar to those for vectors are supported. A list of the matrix math functions is given in Table 4.6. Notice that some of these functions also operate with vectors: all vector functions that involve matrices are defined in `<vw/Math/Matrix.h>` instead of `<vw/Math/Vector.h>`.

There is a special method, `set_identity()`, that can be used to set a square matrix to the identity matrix of that size.

```
Matrix<float> id(3,3);
id.set_identity();
```

If you want to treat a single row or column of a matrix as though it were a vector, you can do so using the `select_row()` and `select_col()` function:

```
Vector<float> first_row = select_row(matrix,1);
select_column(matrix,2) = Vector3(1,2,3);
```

The second of these examples illustrates that you can use the `select_*` functions to write into matrix rows and columns as well as read them out. Finally, you can treat a block of a matrix as a smaller matrix in its own right using the `submatrix()` function:

```
Matrix<float> block = submatrix(matrix,row,col,rows,cols);
```

Function	Description
`- matrix`	Matrix negation
`matrix + matrix`	Matrix sum
`matrix - matrix`	Matrix difference
`matrix * scalar`	Scalar product
`scalar * matrix`	Scalar product
`matrix / scalar`	Scalar quotient
`matrix += matrix`	Matrix sum assignment
`matrix -= matrix`	Matrix difference assignment
`matrix *= scalar`	Scalar product assignment
`matrix /= scalar`	Scalar quotient assignment
`matrix * matrix`	Matrix product
`matrix * vector`	Matrix-vector product
`vector * matrix`	Vector-matrix product
`elem_sum(matrix,matrix)`	Elementwise matrix sum (same as `+` operator)
`elem_sum(matrix,scalar)`	Elementwise sum of a matrix and a scalar
`elem_sum(scalar,matrix)`	Elementwise sum of a scalar and a matrix
`elem_diff(matrix,matrix)`	Elementwise matrix difference (same as `-` operator)
`elem_diff(matrix,scalar)`	Elementwise difference of a matrix and a scalar
`elem_diff(scalar,matrix)`	Elementwise difference of a scalar and a matrix
`elem_prod(matrix,matrix)`	Elementwise product of two matrices
`elem_prod(matrix,scalar)`	Elementwise matrix product (same as `*` operator)
`elem_prod(scalar,matrix)`	Elementwise matrix product (same as `*` operator)
`elem_quot(matrix,matrix)`	Elementwise quotient of two matrixs
`elem_quot(matrix,scalar)`	Elementwise quotient (same as `/` operator)
`elem_quot(scalar,matrix)`	Elementwise quotient of a scalar and a matrix
`norm_1(matrix)`	Matrix 1-norm
`norm_2(matrix)`	Matrix 2-norm
`norm_frobenius(matrix)`	Matrix Frobenius norm
`sum(matrix)`	Sum of elements, i.e. $\sum v_i$
`prod(matrix)`	Product of elements, i.e. $\prod v_i$
`trace(matrix)`	Matrix trace, i.e. $\sum M_{ii}$
`transpose(matrix)`	Matrix transpose, i.e. M^T
`inverse(matrix)`	Matrix inverse, i.e. M^{-1}

Table 4.6: The matrix math functions defined in `<vw/Math/Matrix.h>`.

You can also use this function to write into a region of a matrix, much as in the previous example using `select_col()`.

Like `Vector`, `Matrix` is a C++ STL-compatible container class. The `Matrix<...>::iterator` iterates over the elements of a matrix in the same order that the `ImageView`'s iterator does: across each row, moving down the matrix from each row to the next. This is again a good method for loading or extracting matrix data from other containers. To extract the matrix data to a stream for debugging output you can use the `<<` stream output operator:

```
double data[4] = {1,2,3,4};
```

```
Matrix2x2 matrix(data);
std::cout << matrix << std::endl;
// The output is: [2,2]((1,2)(3,4))
```

Again, the output includes the matrix dimensions (rows followed by cols), followed by the matrix data.

4.4 Transforming or Warping Images

We return now to our discussion of image processing by introducing a new concept: image transformation. Most of the image processing operations we have dealt with so far (with the exception of the simple transforms in Section 3.3.1) have operated on pixel values. Image transformation, or warping, is a common image processing operation that operates instead on a pixels *location*.

4.4.1 Transform Basics

Let's start with a basic example of image transformation. First, include the `<vw/Image/Transform.h>` header file. Now, imagine you would like to translate all of the pixels in the image 100 pixel positions to to right. This operation does nothing to the pixel values except to relocate them in the image. The Vision Workbench provides a convenient method for performing this operation.

```
double u_translation = 100;
double v_translation = 0;
result_image = translate(input_image, u_translation, v_translation);
```

This simple example already raises some interesting questions. How big is the output image? What happens to the pixels that are translated off the right edge of the image? What value is used to fill in pixels where the original image has no data?

The answer to the first question is straight-forward. By default, the transformed image will have the same dimensions as the input image. However, you can easily override this behavior by selecting a different region from the output image using the `crop()` function. For example, you could grow the right side of the output image to include the shifted pixels.

```
result_image = crop(translate(input_image, u_translation, v_translation),
                    0, 0, input_image.cos() + x_translation, input_image.rows());
```

If `input_image` was 320x240, `result_image` will be 420x240 pixels and it will have a 100x240 black band on its left side.

This is a good time to stop to consider what is really happening here, because the ability to arbitrarily crop the output of a transformed image is extremely useful. Under the hood, our call to translate is returning an object that behaves like an image (so it can be cropped), but it is actually presenting an image-like interface to some processed, edge-extended data. Thus, you can use `crop()` to select a region of pixels anywhere in the pixel space that contains the resulting image. It is not until you *assign* the cropped image to `result_image` that this data is once again rasterized and stored as a contiguous block in memory.

Note that the Vision Workbench adopts a consistent coordinate system when working with pixels in the transformed image space. The origin is at the upper left hand corner of the original image, with the u coordinate increases as you move down the rows of the image. Figure 4.1 shows

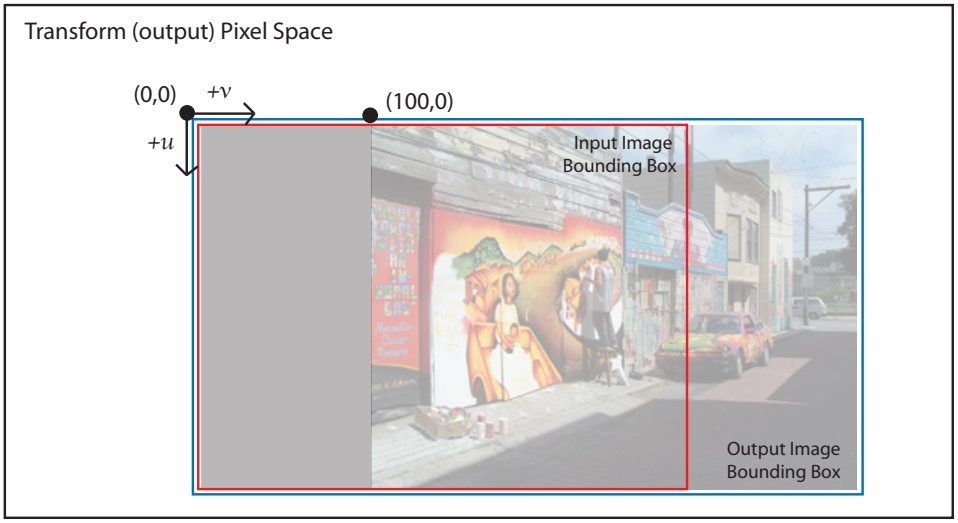


Figure 4.1: Using the `crop()` function, you can select any region of the transformed (in this case, translated) image that you need.

this coordinate system and the input and output bounding boxes in the case of the the cropped, translated image example we have been working with.

Using this intuition, we can now answer the second question posed above. When the pixels are translated off of the right edge of the image, they disappear unless they are explicitly selected using `crop()`. The only other reasonable behavior might have been to have the pixels wrap around and enter on the left side of the image. This is not supported using the Vision Workbench `translate()` function, however, as you will learn in the next section, such transformations are still possible using the general transform framework.

Finally, we arrive at the third question: what pixel value is used to fill area where the original image has no data? To answer this, think back to the discussion of edge extension modes for the filter routines in section 4.1.2. Edge extension behavior in the transform routines of the Vision Workbench are specified in an identical fashion.

```
result_image = translate(input_image,
                         u_translation, v_translation,
                         ConstantEdgeExtend());
```

In this example, the left 100x240 block of `result_image` will contain the "smeared out" pixels from the left side of the input image. Of course, this is probably not what you wanted, so the default behavior edge extension behavior for `translate()` is set to `ZeroEdgeExtend()`.

One final point before we move on to talking about image transformations more generally. Consider this block of code:

```
double u_transformations = 100.5;
double v_transformation = 30.7;
result_image = translate(input_image, u_translation, v_translation,
                         ConstantEdgeExtend(), BicubicInterpolation());
```

Here, the image is translated by a non-integer number of pixels. This is a totally reasonable thing to do, but it raises the question of how one accesses a non-integer pixel location in the

Type	Description
`NearestNeighborInterpolation`	Use the nearest integer valued pixel location
`Bilinear Interpolation`	Linearly interpolation based on the four nearest pixel values
`Bicubic Interpolation`	Quadritic interpolation based on the nine nearest pixel values

Table 4.7: The Vision Workbench Interpolation Modes.

source image. The answer: interpolation. As with edge extension, you can specify the interpolation mode by passing in a dummy argument to the `translate()` function. Table 4.7 shows the built-in interpolation types.

4.4.2 Creating a New Transform

Having now addressed some of the fundamental issues that arise when transforming images, we now turn our discussion to how one might formulate and implement new a image transformation algorithm.

In the most general sense, a transform is computed by performing the following two steps for every pixel in the output image.

- Given the coordinates X_{out} of a pixel in the output image, apply a transformation that yields the coordinates X_{in} of a source pixel in the input image.
- Use some edge extension and interpolation scheme to determine the pixel value of the input image at X_{in} (it may fall in between integer pixels coordinates or outside of the input image entirely) and set the value of the output image at X_{out} to this value.

When formulating a new image transformation algorithm, the first step where all of the interesting work happens. The code for interpolation and edge extension is important, but usually incidental to the transformation under development. Ideally, one would focus exclusively on writing code to perform the geometric calculations in step one. To help us with this task, we will introduce a new programming idiom that appears commonly in the Vision Workbench: the *functor*.

Technically, a functor is a C++ class that has implemented the `operator()` method. Once created, such a class can be called and passed around in place of a normal C++ function. It behaves identically except that, as a C++ object, the functor can maintain its own state (possibly initialized when the functor is constructed). In the Vision Workbench, we use this definition more loosely to mean any small function object that adheres to a small, pre-determined interface. But, rather than linger over semantic details, let's jump straight to an example so that you can see what we mean.

Let's look at the definition for the functor that describes image translation, shown in Listing 3. You'll notice that this class has defined three methods: a constructor and two methods called `forward()` and `reverse()`. The class also inherits from `TransformBase<>`, but that's not something to dwell on here. For now just be aware that `TransformBase<>` provides default implementations that throw `vw::UnimplErr()` exceptions in case the subclass does not implement both methods.

The constructor is used to initialize the state of this functor; in this case, an offset in x and y. The `reverse()` method is the most important. It performs step one in our list at the beginning of this section. Pretty simple, right? Although the transformation in this example is nothing special, the `reverse()` method could potentially be long and complicated. So long as it returns a pixel position in the input image in the end, we're happy.

```
class TranslateTransform : public TransformBase<TranslateTransform> {
  double m_xtrans, m_ytrans;
public:
  TranslateTransform(double x_translation, double y_translation) :
    m_xtrans( x_translation ) , m_ytrans( y_translation ) {}

  // Given a pixel coordinate in the ouput image, return
  // a pixel coordinate in the input image.
  inline Vector2 reverse(const Vector2 &p) const {
    return Vector2( p(0) - m_xtrans, p(1) - m_ytrans );
  }

  // Given a pixel coordinate in the input image, return
  // a pixel coordinate in the output image.
  inline Vector2 forward(const Vector2 &p) const {
    return Vector2( p(0) + m_xtrans, p(1) + m_ytrans );
  }
};
```

Listing 3: [`transform-functor.h`] An example transform functor that performs image translation.

The `forward()` method performs the inverse operation of the `reverse()` method. This method is not always necessary. We'll discuss it more in Section 4.4.3.

The beauty of the `TranslateTransform` class, or any other class that defines a set of `forward()` and `reverse()` methods is that it can be passed as an argument to the `transform()` function.

```
result_image = transform(input_image,
                         TranslateTransform(u_translation, v_translation));
```

This block of code performs the very same transformations as our call to `translate()` in the previous section. (In fact, `translate()` is just a thin wrapper around `transform()` provided for convenience.) As with the previous example, the edge extend and interpolation modes can be supplied as dummy arguments to `transform()`.

As you can probably now see, the possibilities are endless! For example, we could also have used `HomographyTransform` (another built-in transform functor) to describe the same translation. The linear homogeneous transform that encodes a 100 pixel shift to the right is:

$$H = \begin{pmatrix} 1 & 0 & 100 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{4.1}$$

Dropping the `HomographyTransform` into `transform()` yields the same `result_image` once again.

```
vw::Matrix<double> H = ... // defined as above
result_image = transform(input_image, HomographyTransform(H));
```

Any transform functor that adheres to this simple interface, including one of your own devising, can be passed into `transform`. To summarize, you can create and use your own transformation functor `foo` by following these steps.

- Inherit from `public TransformBase<foo>`
- Define a constructor that stores any state information that you need
- Define a `reverse()` method
- Define a `forward()` method (optional, see Section 4.4.3)

Of course, some of the most common transform functors are provided for you as part of the Vision Workbench. refer to Table 4.8 for a list of built-in classes available in `<vw/Image/Transform.h>`

Type	Function	Description
`ResampleTransform`	`resample()`	Scale an image, resizing the output image as needed
`TranslateTransform`	`translate()`	Translate an image
`HomographyTransform`		Apply a linear homogeneous transformation (3x3 Matrix)
`PointLookupTransform`		Apply a transformation based on a lookup table image
`PointOffsetTransform`		Apply a transformation based on an offset table image

Table 4.8: Built-in transform functors and (if available) their function interface.

4.4.3 Advanced Techniques

To wrap up our discussion of the transform methods, here are some advanced techniques that you may find useful when working with image transformations.

It is not uncommon to ask, for a given transform functor, what bounding box in the transformed image space contains the complete set of transformed pixels. The `compute_transformed_bbox()` routine answers this question by performing the `forward()` transformation for each pixel location in the input image and growing a bounding box to contain all of the forward transformed pixels. This bounding box can be passed directly as the second argument to `crop()`.

```
// Output image is cropped to contain all transformed pixels
BBox2f result_bbox = compute_transformed_bbox(input_image,
                                              MyTransformFunctor());
result_image = crop(transform(input_image, MyTransformFunctor()),
                    result_bbox);
```

For performance limited applications, you may find `compute_transformed_bbox_fast()` more appropriate. It computes the bounding box by applying `forward()` to the perimeter pixels of the input image only. This should produce identical results to the "slow" version so long as the perimeter pixels of the input image form the perimeter of the output image under the transformation in question.

Finally, we would like to point out the existence of the `RadialTransformAdaptor` class. This class is useful when it is easier or more natural to describe a transformation in terms of polar coordinates $[r, \theta]$ instead of the usual cartesian coordinates $[u, v]$.

To use `RadialTransformAdaptor`, you write your transform functor as usual, but you interpret the components in the input and output `Vector2` to be $[r, \theta]$, in that order. Assuming you have created a class `MyRadialTransform` in this manner, you can apply it as follows:

```
result_image = transform(input_image,
                         RadialTransformAdaptor(MyRadialTransform(),
                                                input_image));
```

The `RadialTransformAdaptor` creates a polar coordinate system wherein the center of the image is the origin, and a value of $r = 1.0$ is equal to the distance from the center to the left edge of the image.

Chapter 5

The Core Vision Workbench Type System

The Vision Workbench is an example of what is often called a "multi-paradigm" C++ library. That is, different components of the library adopt different C++ programming models, such as the generic programming model or the object-oriented programming model, often in combination. At the core, however, is a set of data types and related tools that fall largely within the template-based generic programming paradigm. The purpose of this chapter is to describe this core type system in some detail. If your intention is simply to *use* the Vision Workbench for image processing tasks you can probably afford to skim or even skip this material. The primary intended audience is programmers who wish to extend the Vision Workbench's core capabilities in one way or another.

Data types in the Vision Workbench can be broadly divided into three categories: the fundamental data types, including the simple numeric types; the compound types, such as RGB pixel types and vectors; and the container types, such as images. We shall discuss each of these in turn. (Other C++ types, such as functors, do make an appearance in the Vision Workbench, but these are not *data* types *per se*.)

5.1 The Scalar Types

The most fundamental data types of all are the built-in C++ integral and floating-point numeric types. In scientific programming contexts like image processing and machine vision it is generally important to be able to specify the exact nature of the numeric data types that you are working with. Unfortunately the C++ language makes few promises about the sizes of data types such as `int` or even `char`. To work around this limitation, the Vision Workbench provides a number of portable `typedef`s that you are encouraged to use instead. These are listed in Table 5.1. (In fact most of these data types are simple wrappers around similar types provided by the Boost `cstdint` library.)

The Vision Workbench uses standard C++ complex numbers, as defined in the standard header file `<complex>`. The `std::complex<>` class takes a single template parameter, the underlying numeric type to be used for the real and imaginary components, which should usually be one of the scalar types listed in Table 5.1. In particular, it is almost always best to use one of the floating-point types for complex numbers. For example, `std::complex<float64>` is a good choice for use in frequency-domain image processing (discussed in Section 10.6).

There is a special type trait template, `IsScalar<>`, that you can use in template code to determine whether or not a type is a simple numeric type like we have described in this section. It

Type	Description	Notes
`int8`	Signed 8-bit integer	
`uint8`	Unsigned 8-bit integer	Most common for low-dynamic-range imaging
`int16`	Signed 16-bit integer	
`uint16`	Unsigned 16-bit integer	
`int32`	Signed 32-bit integer	
`uint32`	Unsigned 32-bit integer	
`int64`	Signed 64-bit integer	
`uint64`	Unsigned 64-bit integer	
`float32`	32-bit floating point	Most common for high-dynamic-range imaging
`float64`	64-bit floating point	

Table 5.1: The core Vision Workbench scalar `typedef`s, defined in `<vw/FundamentalTypes.h>`.

Class	Description
`SumType<T1,T2>`	Result type of a sum operation
`DifferenceType<T1,T2>`	Result type of a difference operation
`ProductType<T1,T2>`	Result type of a product operation
`QuotientType<T1,T2>`	Result type of a quotient operation

Table 5.2: The Vision Workbench type deduction classes, defined in `<vw/TypeDeduction.h>`.

inherits from either `boost::true_type` or `boost::false_type` accordingly. Its primary use is to prevent template functions such as scalar multiplication from being too general:

```
template <class ScalarT>
typename boost::enable_if< IsScalar<ScalarT>, MyClass >::type
operator*( MyClass const& m, ScalarT s ) {
  /* compute result */
}
```

In this example we use the Boost `enable_if` library to restrict the definition of the `*` operator to cases where the second argument really is a scalar. Without this restriction this would have been an overly-general function definition and would likely have caused problems if we had attempted to defined any other product for the `MyClass` class later on. If you decide to extend the Vision Workbench to support additional scalar types, such as bigints, you should specialize `IsScalar<>` accordingly to ensure proper behavior.

5.2 Type Deduction

The C++ language has many intricate rules for type promotion and deduction in complex mathematical expressions. Unfortunately it provides no built-in mechanism to extend this automatic type deduction system or query its behavior. Consider adding two images with compatible but different pixel types: what should the resulting pixel type be? The Vision Workbench provides a standard set of type deduction traits classes, defined in `<vw/TypeDeduction.h>` and listed in Table 5.2, that allow you to both query and specialize the type deduction behavior of Vision Workbench types. Like all Vision Workbench type computation classes, they "return" their result types in a member type named `type`.

Type	Description	Channels
PixelGray<T>	Grayscale	Grayscale value (v)
PixelGrayA<T>	Grayscale w/ alpha	Grayscale value (v), alpha (a)
PixelRGB<T>	RGB	Red (r), green (g), blue (b)
PixelRGBA<T>	RGB w/ alpha	Red (r), green (g), blue (b), alpha (a)
PixelHSV<T>	HSV	Hue (h), saturation (s), value (v)

Table 5.3: The Vision Workbench color-space pixel types, defined in <vw/PixelTypes.h>.

Syntax	Description
PixelChannelType<PixT>::type	The pixel type's underlying channel type
PixelNumChannels<PixT>::value	The number of channels in the pixel type
PixelChannelCast<PixT,ChT>::type	The same pixel type with a new channel type
PixelIsCompound<PixT>	Is the pixel type a compound type?
PixelMakeReal<Pixt>::type	Converts to the corresponding real channel type
PixelMakeComplex<Pixt>::type	Converts to the corresponding complex channel type

Table 5.4: The pixel traits and pixel type computation classes.

As a trivial example, imagine writing a template function that simply computes the sum of its two arguments. What should its return type be? We can use SumType<> to compute it:

```
template <class T1, class T2>
inline typename SumType<T1,T2>::type sum( T1 a, T2 b ) {
  return a + b;
}
```

Remember that the C++ language does not allow this to be fully automatic. If you define a new type with an unusual addition operator, you will need to manually specialize SumType<> at the same time. However, the most common default behaviors are provided. For example, any built-in type is assumed to be promoted to any user-defined type, and any user-defined type operating with itself is assumed to return itself. These type deduction classes do also replicate the standard C++ promotion behavior when used with the built-in numeric types.

5.3 The Pixel Types

It is possible to use any of the fundamental scalar types described in the previous section as an ImageView's pixel type. However in most circumstances a *compound* pixel type, consisting of one or more channels with associated semantics, is more appropriate. The Vision Workbench provides several compound pixel types corresponding to the most common color spaces used in image processing. These are listed in Table 5.3. Each is a template class taking one template parameter, the scalar type used to store the channels. For example, the native pixel type of a standard JPEG image is represented by PixelRGB<uint8>.

Several type trait classes are provided for use in writing generic pixel manipulation code and are listed in Table 5.4. The first section of the table lists the classes that you must specialize when you write a new compound pixel type. The second section of Table 5.4 lists convenience types that are defined in terms of the other, specialized types. For example, here is how the types are specialized for the RGB pixel type:

```
template <class ChannelT>
struct PixelChannelType<PixelRGB<ChannelT> > {
  typedef ChannelT type;
};

template <class ChannelT>
struct PixelNumChannels<PixelRGB<ChannelT> > {
  static const unsigned value = 3;
};

template <class OldChT, class NewChT>
struct PixelChannelCast<PixelRGB<OldChT>, NewChT> {
  typedef PixelRGB<NewChT> type;
};
```

When you define a new pixel type, you will usually want to define a provide a similar set of template specializations. To simplify the process, a macro is provided that you can use to automatically specialize the templates in the usual manner:

```
VW_DECLARE_PIXEL_TYPE( PixelRGB, 3 );
```

This macro expands to the same set of template specializations shown above, describing a pixel type named `PixelRGB` with three channels.

Note that it is *not* necessary to declare a type using this macro, or even to provide specializations for the traits templates described above, just to use that type as the pixel type for an image. These specializations are only necessary if you want to declare a type with multi-channel semantics. For any other type, the default Vision Workbench behavior is to treat the type as a single-channel pixel type whose channel type is equal to the type itself.

Several convenience functions are also provided to simplify working with pixels in generic template functions. The first is the `pixel_channel_cast<>()` function, which casts a pixel to a pixel of the corresponding pixel type but with the specified channel type. The syntax mirrors the built-in C++ casting functions, except the template parameter is the new channel type instead of the new type as a whole. In the following example we explicitly down-cast the channel type of a pixel from `float64` to `float32` in order to pass it to a function that happens to take a `PixelRGB<float32>` argument:

```
PixelRGB<float64> pixel;
some_function( pixel_channel_cast<float32>( pixel ) );
```

Note that this is not needed in the more common case that the function that you wish to call is itself generic and can accept any channel type.

Sometimes it is desirable to apply a function to each channel of a pixel, or to corresponding channels from two pixels. Re-scaling a pixel or adding two pixels on a per-channel basis can be cast into this form, for example. You can use the generic function `apply_per_pixel_channel()` to do this. Here is a trivial example that demonstrates re-scaling:

```
float32 triple(float32 v) { return 3*v; }
// Later, in some other function...
PixelRGB<float> pixel(.1,.2,.3);
PixelRGB<float> result = apply_per_pixel_channel(&triple,pixel);
```

In this case it would have been simpler to multiply the pixel by 3 directly, but the point is that we could have performed any arbitrarily complex operation on each channel instead. The binary form simply takes an extra pixel argument:

```
float32 sum(float32 a, float32 b) { return a+b; }
// Later...
PixelRGB<float> pixel1(.1,.2,.3), pixel2(.2,.1,.4);
PixelRGB<float> result = apply_per_pixel_channel(&sum,pixel1,pixel2);
```

Chapter 6

The Camera Module

Cameras are the interface between images and the real world, and as such, their importance in computer vision cannot be understated. In fact, some would say that computer vision algorithms are distinguished by the fact that they endeavor to associate the processed pixel data with objects in the real world for the purposed of measurement, tracking, or display. This is achieved by modeling the geometric and physical properties of the device that was used to capture the original image. This is the purpose of the camera module.

The camera module includes built-in models for generic pinhole and line-scan imagers and a set of generic functions for linearizing (removing lens distortion) and epipolar rectifying (e.g. for stereo) when these operations are relevant. These classes and functions can be imported into your code by including `<vw/Camera.h>`.

Because you will likely encounter new camera geometries not supported by the built-in classes, the camera module is designed to be extensible. You can provide your own camera model by inheriting from and adopting the interface of the `CameraModel` abstract base class, which is defined in the header file `<vw/Camera/CameraModel.h>`.

Finally, the camera module provides a basic set of tools for working with images from real-world camera systems: bayer pattern filtering and EXIF parsing. We will cover all of these features in more detail, but we begin this chapter by establishing some terminology while exploring the most common camera geometry in use today: the pinhole camera model.

6.1 The Pinhole Camera Model

The pinhole camera model describes the geometry found in nearly all commercial digital camera systems today. It is characterized by a lens assembly that focuses light onto a two dimensional array of pixels (usually a sensor with light sensitive circuits such as a CCD or CMOS device). We will use the pinhole model to establish some terminology that will be used throughout the rest of this chapter. Be warned that the model we are about to develop is simplistic; many of the non-ideal characteristics of a real-world optical system (e.g. lens distortion) are not modeled in this simple example. Refer to the CAHVOR model in Section 6.3 if you require a pinhole camera model that more accurately models lens distortion.

6.1.1 Perspective Projection

Figure 6.1 shows the geometry of a basic pinhole camera. The light gray area represents the 2D array of pixels, and it is referred to as the *image plane*. The origin of the 3D coordinate system is

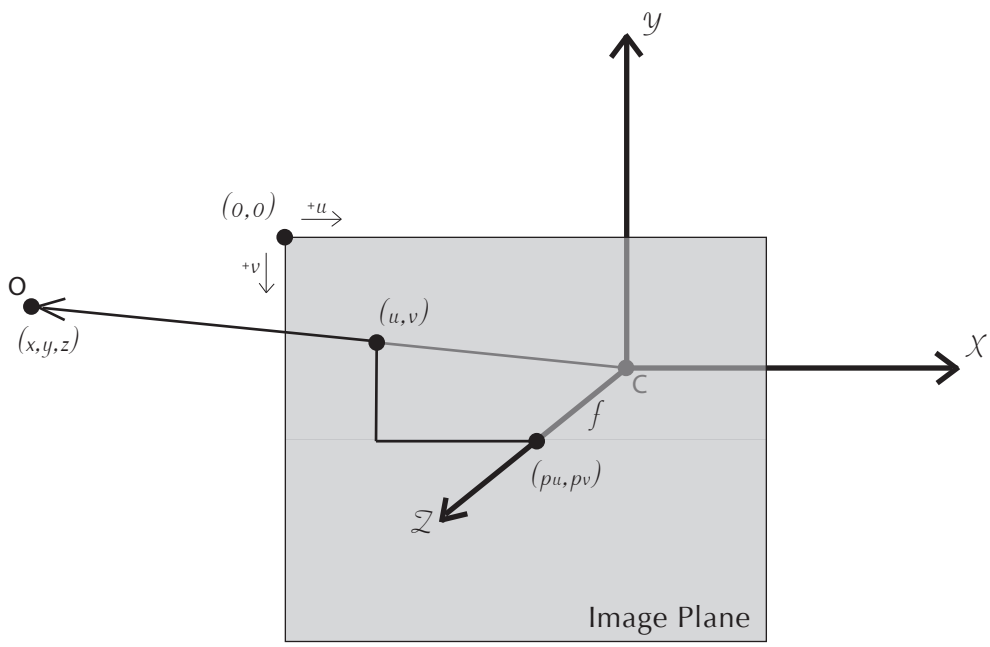


Figure 6.1: The basic pinhole camera model.

the point C, which is the center of projection or *camera center* of the imager. When a 3D point O is imaged by the camera, it appears at the pixel located where segment $\overline{OC}$ intersects the image plane at point (u, v). A line segment $\overline{OC}$ that is perpendicular to the image plane intersects this plane at the *principal point*, (p_u, p_v).

All of the points imaged by the camera appear on a line that passes through C. If the coordinates of $\mathbf{O}$ are (x, y, z), then the position of the point on the imager can be determined by projecting it onto the plane $z = +f$:

$$u = \frac{f}{\sigma}\left(\frac{x}{z}\right) - p_u \tag{6.1}$$

$$v = \frac{f}{\sigma}\left(\frac{-y}{z}\right) - p_v \tag{6.2}$$

Here, f is the focal length of the imager in meters, σ is the size of a pixel in *m/pixel*, and (p_u, p_v) are the offset in pixels of the principal point (this offset moves the origin of the image from the principal point to the upper left hand corner, which is the "origin" usually adopted when indexing images).

Equations 6.1 and 6.2 constitute the *forward projection* portion of the camera model; this is analogous to the process of "capturing" an image with a real camera. Our model should tell us exactly what what pixel location to look at if you wanted to see the point O in the image.

Notice how some information is lost during forward projection. Any point along $\overline{OC}$ will be imaged to the same point (u, v) on the image plane, so if we were to start with a point $P = (u, v)$ on the image plane, and we wanted to find the original 3D point O, the best we could do would be to say that it appears somewhere along the ray $\overrightarrow{CP}$. The origin and direction of the ray can be computed as follows:

$$\overrightarrow{CP}_{origin} = C \tag{6.3}$$

$$\overrightarrow{CP}_{direction} = \frac{(u+p_u, -(v+p_v), f)}{||(u+p_u, -(v+p_v), f)||_2} \tag{6.4}$$

This operation, which we call *back projection*, can still provide useful information that can be used in a full 3D reconstruction despite the ambiguity in the actual position of O. Imagine that you have two cameras that have imaged the same point O from two different viewpoints at pixel locations P_1 and P_2 in their respective image planes. Using simple geometry, you can reconstruct the position of O by computing the intersection of the two rays emanating from each camera center through P_1 and P_2. This is the technique commonly referred to as stereo reconstruction, and it is one of the many ways that you can make use of the information provided by back projection.

6.2 The Camera Model Base Class

As we have seen, a camera model provides a means for forward projection ("imaging" 3D points onto a 2D array of pixels) and back projection (finding the ray along which a 3D points must lie given a 2D pixel where it was imaged). All camera models in the Vision Workbench derived from the CameraModel abstract base class, which enforces this basic interface.

Forward projection of a 3D point is handled by the `point_to_pixel()` method.

```
CameraModel* camera_model = new MyDerivedCameraModelClass;
Vector3 world_coordinates;
Vector2 pixel_coordinates = camera_model.point_to_pixel(world_coordinates);
```

Remember that in C++, you will need to maintain a pointer to the derived camera model class in order to ensure that the virtual inheritance mechanism calls the method from the derived class rather than the base class.

The back projection operation is split into two separate API calls. The `camera_center()` method returns the origin of the ray, and the `pixel_to_vector()` method returns its direction. Remember that any of the points that lie along this ray would have been imaged at (u, v), so this pixel-to-ray operation leaves some ambiguity about the true location of the point O.

```
CameraModel* camera_model;
Vector3 ray_origin = camera_model.camera_center(pixel_coordinates);
Vector3 ray_direction = camera_model.pixel_to_vector(pixel_coordinates);
```

Camera Coordinate Systems

Vision Workbench camera model classes take and return coordinates that are *not* homogeneous. That is, coordinates do not need to augmented with an additional homogeneous scaling element before being passed to camera module routines (e.g. a 2D vector $(325, 206)$ in cartesian coordinates is often represented as $(325, 206, 1)$ in homogeneous coordinates). Homogeneous coordinates have certain advantages in projective geometry (e.g. they allow a translation of the coordinates to be encoded as a matrix multiplication), however we have chosen not to adopt this convention.

Camera Model	*Header File*	*Imager Type*	*Details*
CAHV	`CAHVModel.h`	Pinhole	Basic pinhole camera model
CAHVOR	`CAHVORModel.h`	Pinhole	Models lens distortion
Linescane	`LinescanCameraModel.h`	Linescan	Generic Linescan Model
Linear Pushbroom	`LinearPushbroomModel.h`	Linescan	Assumes linear flight path
Orbiting Pushbroom	`OrbitingPushbroomModel.h`	Linescan	Models curvature of orbit

Table 6.1: Built-in camera models can be found in `vw/Camera/`

6.3 Built-in Camera Models

The Vision Workbench comes with several "built-in" camera models. These classes satisfy the needs of most common applications, and they can also serve as a design reference for your own camera model classes. Each class models a specific geometry and, to varying extents, the non-ideal characteristics of the camera system such as lens distortion.

The list of built-in models are summarized in Table 6.3. The following sections describes the two basic classes of built-in camera model: those that model pinhole cameras (where the imager is a 2D array of pixels), and those that model linescan cameras (where the imager is a 1D line of pixels).

6.3.1 Pinhole Cameras

The *CAHV camera model* has been widely used in NASA planetary mission for rover navigation and scientific camera systems [4]. It is a basic pinhole camera model that does not model lens distortion or other optical aberrations. The CAHV model is so named because the camera intrinsic and extrinsic parameters are jointly parameterized by four 3-dimensional vectors: C,A,H, and V. This compact representation leads to a very efficient forward projection operation, which is the strength of the CAHV model. Forward projection of a real world point O can be computed using

$$u = \frac{(O-C)\cdot H}{(O-C)\cdot A} \tag{6.5}$$

$$v = \frac{(O-C)\cdot V}{(O-C)\cdot A} \tag{6.6}$$

The user has two choices when initializing a CAHV camera model. First, they can construct the object by directly supplying four 3-vectors to the constructor.

```
Vector3 C,A,H,V;
CameraModel* cam = new CAHVModel(C,A,H,V);
```

Alternatively, users seeking to use the CAHV class as a general purpose pinhole camera model may find it easier to use the more verbose constructor wherein the camera extrinsics and intrinsics are explicitly supplied.

```
double focal_length;
Vector2 pixel_size;
double principal_point_h, principal_point_v;
```

```
Vector3 camera_center, pointing_vector;
Vector3 horizontal_vector, vertical_vector; // Defines image plane orientation

CameraModel* cam = new CAHVModel(focal_length, pixel_size, principal_point_h,
                                 principal_point_v, camera_center,
                                 pointing_vector, horizontal_vector,
                                 vertical_vector);
```

The *CAHVOR model* is an expanded camera model with two additional 3-vectors (O and R) that describe lens distortion introduced by the camera lens.

Refer the Doxygen-generated API documentation for more information about constructing CAHV and CAHVOR camera models.

6.3.2 Linescan Cameras

Linescan imagers capture images using a sensor containing a 1-dimensional array of pixels. The image is formed by capturing successive scan-lines as the camera platform is rotated or moved. For example, a flat-bed scanner or photocopier is a familiar example of such a system. The sensor is swept in the so-called *along-track* direction along the document, composing the final image by concatenating several thousand adjacent 1-dimensional images taken at evenly spaced positions.

Linescan sensors are fairly uncommon in commercial camera systems, but they appear frequently on satellites that capture photographs of terrain from orbit. The orbital motion of the satellite is used in much the same way as the motion of the sensor in the photocopier; the *across-track* dimension of the image corresponds to the projection of 3D points through the lens onto the sensor, and the along-track dimension of the image corresponds to successive scanlines taken at different times as the satellite moves.

The geometry of the linescan imager is subtly different from the geometry of the pinhole camera. See Figure 6.2. Although there is still a center of projection created by the camera's optics (and points are still imaged using a perspective projection in the across-track direction), this point moves as the camera moves, and as a result, the along-track position of a pixel in a linescan image is purely a function of the position and orientation of the camera; both a function of time. The orientation of the image (in the sense that u indexes the columns of an image and v indexes its rows) must be chosen consistent with Figure 6.2 when working with `LinescanModel` and its relatives.

In the special case where the motion of the linescan sensor is linear and its orientation is fixed, the projection of the points onto the image in the along-track direction is orthographic. These assumptions are the basis for the *Linear Pushbroom Model*, which can be found in the header file `<vw/Camera/LinearPushbroomModel.h>`. If you are interested in understanding this model in detail, we recommend you read the excellent paper by Gupta and Hartly [2].

If you must relax the assumption about a linear flight path somewhat to allow sensor pose to vary and the camera motion to lie along a curve (as is common with orbiting camera systems), the *Orbiting Pushbroom Model* is an appropriate choice. In the Orbiting Pushbroom model, the user supplies a series of evenly spaced samples of position and orientation and specifies the time interval (in seconds) between samples. A sparse set of samples is sufficient for this model: interpolation occurs for points in between the supplied positions and orientations.

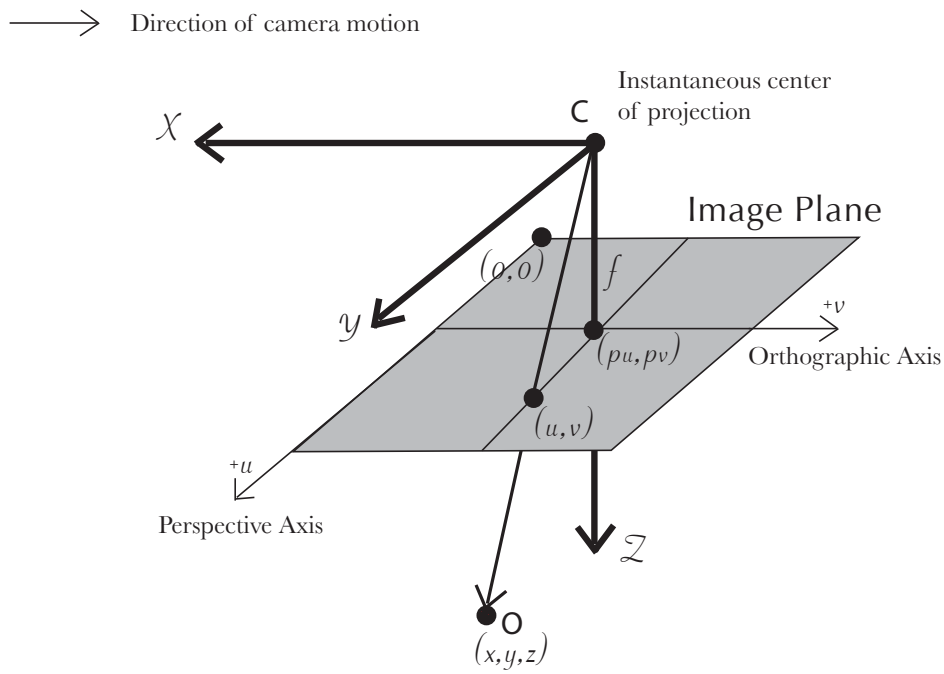


Figure 6.2: Geometry of the Linescan Camera Model.

6.4 Tools for Working With Camera Images

This section describes several tools that simplify the process of working with image captured by real cameras.

6.4.1 Inverse Bayer Pattern Filtering

Most imaging sensors are inherently grayscale capture devices. In order to capture color, some imagers have a hardware color filter placed in front of the pixels on the CCD. This is called a *Bayer filter*. The Vision Workbench provides the `inverse_bayer_filter()` function (found in the header file `<vw/Camera/BayerFilter.h>`) which interprets the raw, grayscale pixel values from the sensor and produces a color image by interpreting the Bayer filter effect.

6.4.2 Exif Exposure Data

Digital cameras store data about the settings used to take a picture in the image file according to the EXIF standard [1]. EXIF data is stored using the Image File Directory system described in the TIFF 6.0 standard. EXIF tags such as `FNumber` and `ExposureTime` can be useful for radiometrically calibrating your camera images. Unfortunately the standard is redundant and often poorly supported by camera manufacturers (for example, many hide the ISO setting in the maker note instead of storing it in the ISOSpeedRatings tag), so we cannot guarantee support for every camera.

The Camera module includes the `ExifView` class (defined in `<vw/Camera/Exif.h>`) for extracting this data from images. To create this class, you supply the filename of an image on disk.

Currently, JPEG and TIFF images are supported. ExifData and ExifView were based on `jhead`, an EXIF JPEG header and thumbnail manipulator program in the public domain [3].

```
// Reliably get F number.
ExifView view;
if (view.load_exif(``img.jpg'')) {
  double f = view.get_f_number();
  // ...
}
```

Bibliography

[1] "Exchangeable image file format for digital still cameras: Exif Version 2.2", (Japan Electronics and Information Technology Industries Association, 2002), http://www.exif.org/specifications.html.

[2] Gupta, Rajiv and Hartley, Richard. "Linear Pushbroom Cameras". IEEE Transactions on Pattern Analysis and Machine Intelligence. Vol.19 No. 9. September 1997

[3] Wandel, Matthias, "Exif Jpeg header and thumbnail manipulator program," 2006, http://www.sentex.net/ mwandel/jhead/.

[4] Yakimovsky, Y. and Cunningham R., "A System for Extracting Three-Dimensional Measurements from a Stereo Pair of TV Cameras " Computer Graphics and Image Processing 7, pp. 195-210. (1978)

Chapter 7

The Mosaic Module

The Mosaic module provides a number of tools for assembling and generating image mosaics, i.e. large images that are composed of a number of smaller images. The applications of this module include panoramic imaging and aerial/satellite image processing. There are three major facilities provided at this time: compositing many images together, using multi-band blending to seamlessly merge overlapping images, and generating on-disk image quad-trees to efficiently store very large images.

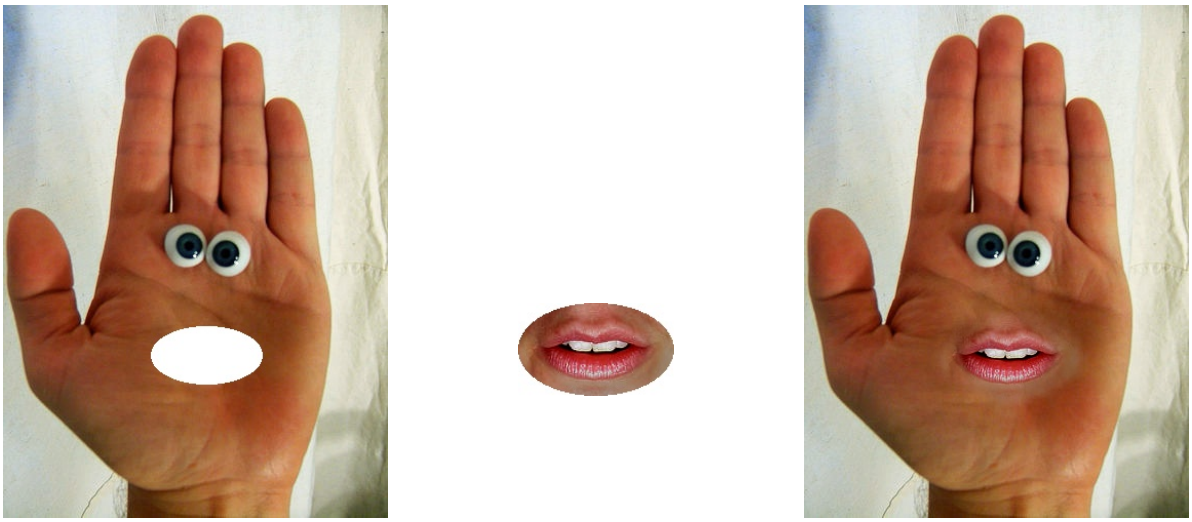
Note that the facilities described in this chapter are currently under active development, and there may be some API changes in future releases as new capabilities are added.

7.1 ImageComposite and Multi-Band Blending

The ImageComposite template class provides the ability to composite any number of source images together at arbitrary pixel offsets. It was originally designed for assembling tiled panoramas or aerial/satellite images that have been transformed into a common coordinate system, though it can be used for many other things as well.

The interface is fairly simple. Just like ordinary Vision Workbench images, an ImageComposite is templatized on its pixel type. In most cases you will want to use a pixel type that has an alpha channel, and if you want to perform image blending then the pixel type must be floating-point, so the most common pixel type is PixelRGBA<float32>. You can then configure whether you would like to use multi-band blending to merge your images or if you would simply like them overlayed by using the set_draft_mode() method. It takes a single argument which should be true if you simply want to overlay the images and fales if you want to use the blender. Blending is a significantly more expensive operation. If your images are simply tiles and do not overlap then draft mode is probably what you want. In blending mode you also have the option of asking the blender to attempt to fill in any missing (i.e. transparent) data in the composite using information from the neighboring pixels. You can enable or disable this behavior by calling the the set_fill_holes() method.

Once you have created the composite object, you add source images to it using the insert() method, which takes three arguments: the image you are adding, and the x and y pixel offset of that image within the composite. The ImageComposite does not store a copy of your image. Instead, it only stores a reference to it in the form of an ImageViewRef object. This means that you can easily do things like create a composite of images that could not all fit in memory simultaneously, e.g. by passing in DiskImageView objects. Note that only integer pixel offsets are supported: if you want to shift an image by a fractional amount you will first need to transform it accordingly. In most



(a) First source (b) Second source (c) Blended result

Figure 7.1: Example input and output images from the `ImageComposite` multi-band blender.[1]

cases you will need to pre-transform your source images anyway, so this applies no extra cost.

Once you have added all your images, be sure to call the `ImageComposite`'s `prepare()` method. This method takes no arguments, but it does two things. First, it computes the overall bounding box of the images that you have supplied, and shifts the coordinate system so that the minimum pixel location is $(0, 0)$ as usual. (For advanced users, if you prefer to use a different overall bounding box you may compute it yourself and pass it as an optional `BBox2i` argument to the `prepare()` method.) Second, if multi-band blending is enabled, it generates a series of mask images that are used by the blender. Currently these are saved as files in the current working directory. This is admittedly inconvenient behavior and will be changed in a future release.

Now that you've assembled and prepared your composite you can use it just like an ordinary image, except that per-pixel access is not supported. If the image is reasonably small then you can rasterize the entire image by assigning it to an `ImageView`. Alternatively, if the composite is huge the usual next step is to pass it as the source image to the quad-tree generator, discussed in the next section. You can also use `ImageComposite`'s special `generate_patch()` method to manually extract smaller regions one at a time. It takes a single `BBox2i` bounding-box, expressed in the re-centered coordinate frame, as its only argument.

Here's a simple example that illustrates how you might blend together a number of images on-disk. It assumes you already know the image filenames and their offsets within the composite, and that the total composite is small enough to sensible rasterize all at once.

```
ImageComposite<PixelRGBA<float> > composite;
for( int i=0; i<num_images; ++i ) {
  composite.insert( DiskImageView<PixelRGBA<float> >( image_filename[i] ),
                    image_offset[i].x(), image_offset[i].y() );
}
composite.prepare();
write_image( "composite.png", composite );
```

For a somewhat more fleshed-out example of how to blend images, see the example program `blend.cc` included with the `Mosaic` module sources.

7.2 ImageQuadTreeGenerator

The ability to assemble composites that are far larger than could be stored in memory all at once presents serious challenges. When viewing an image of that size, the ability to zoom in and out is of critical importance. Only a small fraction of the image data will ever be on-screen at a time at full resolution. However the entire data set may be visible at once at lower resolutions, and computing such reduced images on the fly can be prohibitively expensive. The usual solution to this problem is to pre-compute sub-sampled versions of the image at multiple levels of detail. The sub-sampling factors are often chosen to be successive powers of two, and the data at each resolution is typically chopped up into tiles for faster direct access.

The `ImageQuadTreeGenerator` class generates just such a representation of an image. You specify the tile size, the generator reduces the image by powers of two until it fits in a single tile. To generate each successive level of detail every tile is replaced by four tiles at twice the resolution. The resulting quad-tree of images is stored on disk in a hierarchical manner, making it easy to instantly access any tile at any resolution.

Like many things in the Vision Workbench, an `ImageQuadTreeGenerator` is templatized on its pixel type. The constructor takes two arguments, the pathname of the tree to be created on disk and the source image. You can then use several member functions to configure the quad-tree prior to generating it. The `set_bbox()` method, which takes a single `BBox2i` parameter, specifies that a particular region of the source image to be processed instead of the entire thing. The `set_output_image_file_type()` method sets the file type of the generated image files; the default is "`png`". The `set_patch_size()` function takes an integer argument specifying the patch size in pixels. Relatedly, the `set_patch_overlap()` function specifies how many pixels of the border of each patch overlap the neighboring patch. The default is 256-pixel patches with no overlap. Finally, the `set_crop_images()` method takes a boolean argument that controls whether or not the resulting images are cropped to the non-transparent region. Image cropping is enabled by default.

Once you have configured the `ImageQuadTreeGenerator` to your liking you can invoke its `generate()` method to generate the quad-tree on disk. It is stored in a directory with the name you provided as the first argument to the constructor with the extension "`.qtree`" appended. For example, if you specified the name of the quad-tree as `/home/mdh/example` then the result is stored in the directory `/home/mdh/example.qtree`. This directory typically contains three things. First is the lowest-resolution image of the tree, essentially a thumbnail, which is stored in an image with the same base name as the tree with the appropriate image file format extension appended. To continue the above example, if the file format type is "`png`" then the top-level image file's pathname will be `/home/mdh/example.qtree/example.png`. The next file in the top-level directory is a simple text file describing the bounding box of the top-level patch, with the same name but with the extension `.bbx` instead. The format of this file will be discussed below. Finally there is a subdirectory, which has the same name but no extension, that contains the next level of the tree.

Inside that subdirectory there are generally four image files, with names `0.png`, `1.png`, `2.png`, and `3.png`, containing to the four image patches at the second level of detail. The patches are numbered left-to-right and top-to-bottom, so `0` is the upper-left patch, `1` is the upper-right patch, and so on. There are also four corresponding `.bbx` files and four directories containing higher-resolution data for each patch. Each subdirectory likewise has four numbered images, bounding boxes, and further subdirectories. For example, the file `/foo/bar/myimage.qtree/myimage/0/1/3.png` would

[1]Original hand and face source images by `sheldonschwartz` and `vidrio`, respectively, and released under the Creative Commons license.

(a) Draft mode (simple overlay)

(b) Multi-band blending

Figure 7.2: A twelve-image mosaic composited using an `ImageMosaic` object, first (a) in draft mode and then (b) using multi-band blending.

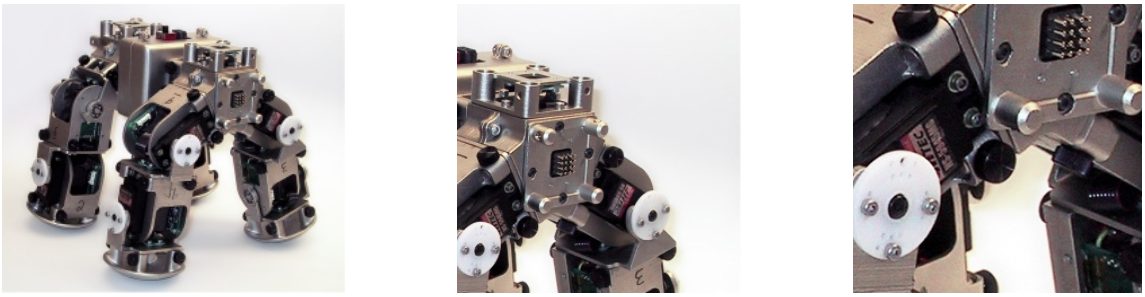


Figure 7.3: Patches at three successive resolutions generated by an `ImageQuadTreeGenerator` constructed with the name "`Walker`". The files are named `Walker.qtree/Walker.png`, `Walker.qtree/Walker/1.png`, and `Walker.qtree/Walker/1/2.png`, respectively.

be an image at the fourth level of detail. The subdirectories at the highest level of detail have no further subdirectories. Note that if cropping is enabled then it is possible that some directories will not have all four images; this occurs if any of the images is entirely empty.

Each `.bbx` file contains eleven numbers, represented as test strings, each on a line by itself. The first is the scale factor of that tile: it has the value 1 for the highest-resolution patches and values of the form 2^n for lower-resolution patches. You can equivalently think of this number as describing the size of each pixel at this resolution, as measured in full-resolution pixels. The next four numbers describe the bounding box of the patch within the full-resolution coordinate system. First are the x and y coordinates of the upper-left pixel, and then come the width and height of the image in pixels. To reiterate, these are measured in *full-resolution* pixels, and so these numbers will generally be multiples of the scale factor.

After this come a similar set of four numbers describing the bounding box of the unique image data in this patch. This is generally the same as the previous bounding box if there is no patch overlap. However, if the patch overlap has been set to a nonzero value then this second bounding box will describe the portion of this patch that does not overlap the corresponding regions of the neighboring patches. In other words, taken together these second bounding boxes perfectly tile the entire image without overlap. Finally, the last two values describe the width and height of the entire source image in pixels.

This file format is obviously quite arbitrary, and was designed to make it easy to write scripts to manipulate the quadtree data later. If you prefer, you can subclass the quadtree generator and overload the `write_meta_file()` function to generate metadata of a different sort.

Chapter 8

High Dynamic Range Imaging

Photographers have long understood that the range of brightness levels in a real-world scene is considerably larger than the range that can be captured by a camera's film or imaging sensor. The luminance of a outdoor scene can easily span five orders of magnitude, however typical digital cameras encode the brightness at a single pixel location using a modest 8-bits (2 orders of magnitude). Pixels that fall outside the *dynamic range* of the camera sensor will either be underexposed or overexposed; their values will be at the minimum or maximum pixel value of the camera, respectively.

Some digital cameras can save RAW images with higher dynamic range. These cameras can capture 12 bits per pixel, or 4096 brightness levels. This expanded dynamic range is often sufficient to capture scenes with a moderate amount of contrast. However, to capture scenes with very high dynamic range, you must generate a HDR image from a set of *bracketed* exposures: a group of low dynamic range images of the exact same scene taken with different exposure settings that vary from under-exposed to over-exposed. This technique is subject of section 8.1.

The resulting HDR image is generally stored with a channel type of `float` or `int16` to accommodate the expanded dynamic range. To store HDR images on disk we recommend using the OpenEXR or TIFF file formats, both of which support 32-bit floating point channel types.

Just as capturing HDR images can be challenging, visualizing them is also difficult. Print media and most display devices can only manage about 2 orders of magnitude of dynamic range, whereas an HDR image may have 5 or more orders of magnitude. Simply scaling the image to the display range will cause it to look overly dark or washed-out. Section 8.2 discusses a technique called *tone mapping* that reduces the dynamic range of an image while preserving as much as possible the visual contrasts of the original scene.

8.1 Merging Bracketed Exposures

As discussed in the introduction to this chapter, the limited dynamic range of the sensors on modern cameras necessitates a different strategy for building true HDR images: exposure bracketing. This frees us from hardware limitations and allows us to set the dynamic range of the output image arbitrarily by adjusting the number of exposures in the bracket. The more exposures in the bracket, the higher the dynamic range merged HDR image.

For convenience, the exposure ratio between consecutive images is usually a fixed value to guarantee a wide exposure range while maintaining even brightness overlap between adjacent images in the bracket. A factor of two is generally recommended. The shutter speed is generally the preferred method of varying the exposure in a bracket; changing the aperture or ISO setting can

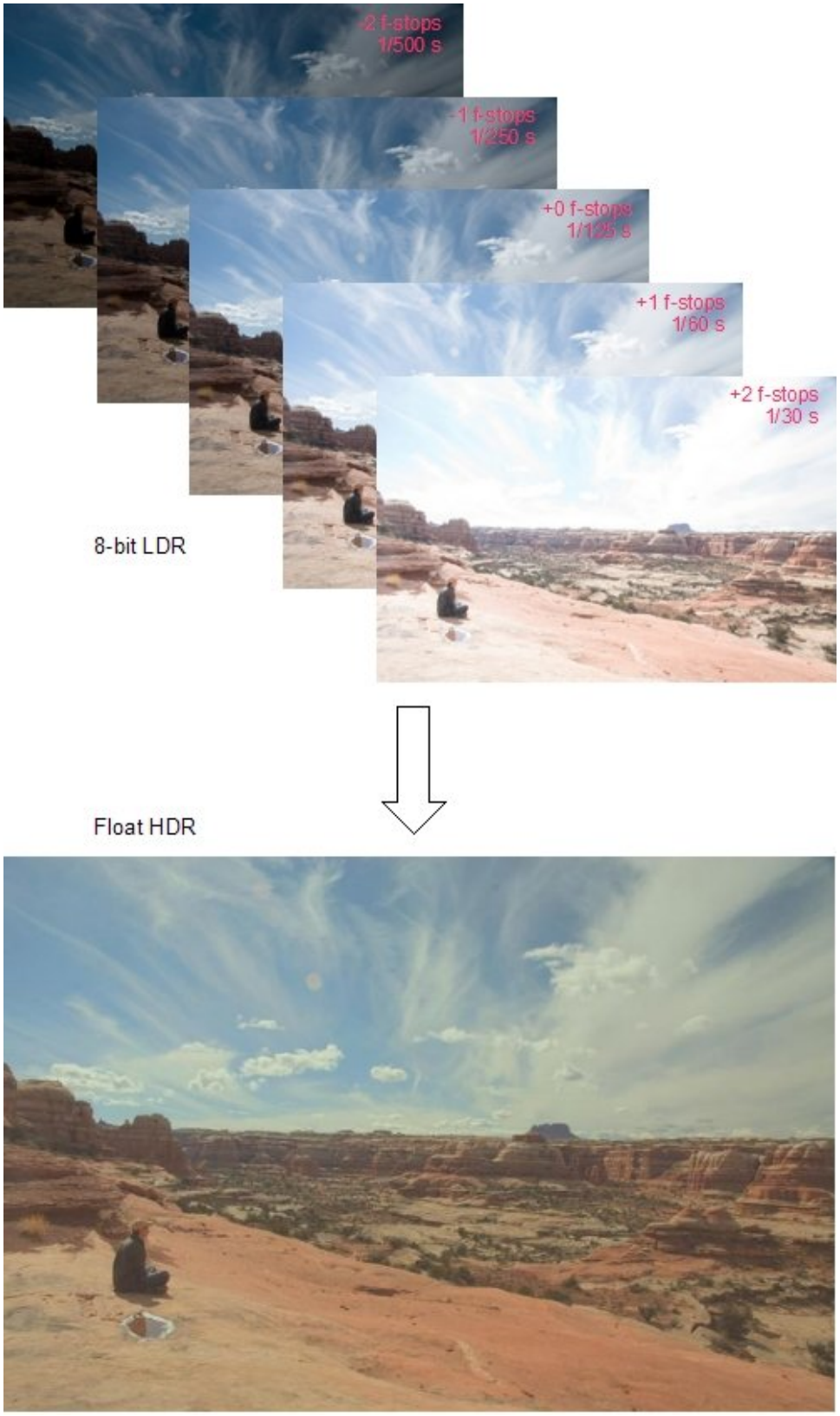


Figure 8.1: A stack of 8-bit-per-channel LDR images separated by one f-stop is merged a floating-point HDR image. The HDR image is tone-mapped for display using the Drago operator.

have the same effect on exposure but they may change the focus or increase noise.

8.1.1 Converting LDR Images to an HDR Image

The HDR module has a set of free functions that make stitching a stack of LDR images into an HDR image as simple as one function call. `process_ldr_images()` (in `<vw/HDR/LDRtoHDR.h>`) takes a `std::vector` of ImageViews (with grayscale or RGB pixels and a floating point channel type), sorted from darkest to brightest. This function assumes a constant exposure ratio between images; it can be specified or the default value of $\sqrt{2}$ can be used (corresponding to 1 f-stop, or a power of two in shutter speed). An overloaded version accepts a `std::vector<double>` of absolute brightness values for each image (as defined by the APEX system [6]) .

```
// Generate HDR image from HDR stack.
vector<ImageView<PixelRGB<double> > > images(num_images);

// ... Read input images ...

// Assum default exposure ratio
ImageView<PixelRGB<double> > hdr_image = process_ldr_images(images);
```

Most modern digital cameras store exposure information as EXIF metadata in the headers of images. The Vision Workbench supports reading this data from TIFF and JPEG files via the Camera Module, and the routine `process_ldr_images_exif()` capitalizes on this, generating an HDR image from an array of filenames of LDR images by computing brightness values from the files' EXIF data along the way.

8.1.2 The Camera Response Curves

The relationship between light entering a camera sensor and the pixel value that is recorded in the image is a non-linear. It depends on many factors, including the degree of gamma correction used in the image, the color balance across the range of brightness, and the camera's post processing settings (especially contrast). The function that encompasses all of these and maps the amount of light on the sensor (the luminance) to the pixel value is referred to as the camera response curve. See Figure 8.2 for an example.

To create a HDR image that truly represent the physical brightness levels in a scene, it is necessary to estimate the inverse of the camera response curves (we assume a separate curve for each channel) and apply it to the image data. That is, we would like to find a function that, given a pixel value in the image and its exposure information, returns the luminance of the scene at that point. `<vw/HDR/CameraCurve.h>` implements `estimate_inverse_camera_curve()`, a free function that estimates the inverse response curve as a polynomial. This routine takes a matrix of aligned pixel channel values from an HDR stack and their brightness ratios. This function is used internally by `LDRtoHDR.h`; most users will probably not need to call it directly.

The CameraCurve sources also supply `invert_curve()`, a free function that approximates the inverse of a polynomial curve, also as a polynomial. This is useful to recover the actual camera response curve (mapping a scaled luminance value to a pixel value) from the inverse response curve determined by `estimate_inverse_camera_curve()`. Re-applying the camera response curve to an HDR image after tone-mapping can improve the colors and overall visual appeal. See the code in Listing 4 for an example.

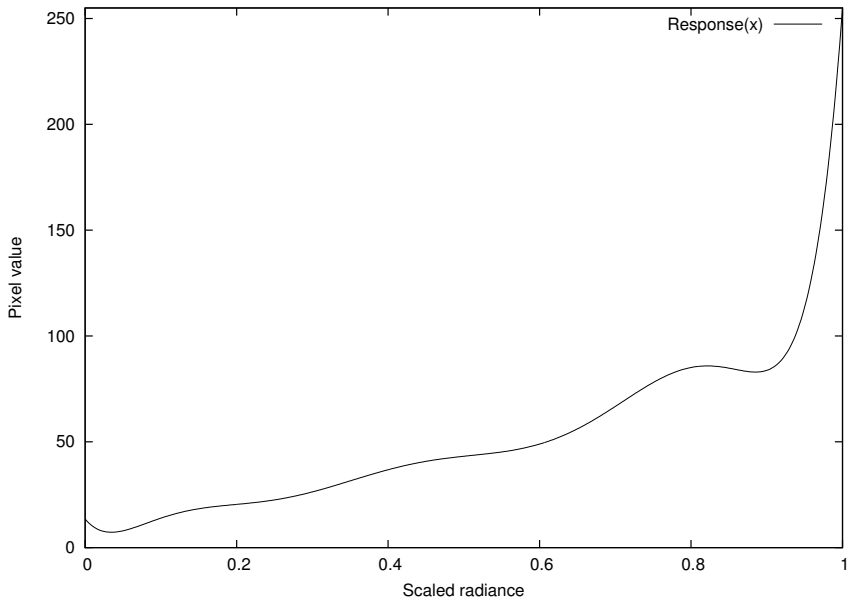


Figure 8.2: Camera response curve estimated from an HDR stack.

8.2 Tone Mapping

Since print media and most display technologies are inherently LDR, the dynamic range of a HDR image must be compressed before it can be displayed. Simply scaling the pixel luminances linearly yields poor results because the human visual system's response to luminance is approximately logarithmic rather than linear. A linear scaling tends to lose small details and local contrast, and the image as a whole will appear under or over-exposed.

A wide variety of tone-mapping operators have been proposed to compress the dynamic range of a HDR image while preserving details and local contrast as much as possible. Using an ideal tone-mapping operator, the observer of a tone-mapped LDR image would have a perceptual response matching that of the original HDR scene. Due to its greater realism, tone-mapping can vastly improves the appearance of the displayed image.

There are several broad classes of tone-mapping operators, including global operators, local operators, and operators that use the gradient or frequency domains. The HDR module currently includes one global operator and one local operator; they are described in the following sections.

8.2.1 Global Operators

Global tone-mapping operators apply the same compressive function to all pixels in the image. Such operators are implemented in `<vw/HDR/GlobalToneMap.h>`. Currently one such operator is implemented, the Drago Adaptive Logarithmic Mapping operator. For algorithm details see *High Dynamic Range Imaging* [9] or the original paper [3]. The Drago operator is currently the operator of choice in the HDR module. It is quite fast, produces good results for a wide range of images, and can usually be used with its parameters at default values. Optionally, the parameters bias

Figure 8.3: Top: an image taken in 12-bit RAW format. Middle: after tone-mapping with the Drago operator. Bottom: after tone-mapping with the Ashikhmin operator.

(controlling contrast, usually between 0.7 and 0.9), exposure factor (a simple multiplier to control the overall brightness of the image), and max display luminance (usually about 100) can be specified.

```
// Apply Drago tone-mapping operator
ImageView<PixelRGB<float> > tone_mapped = drago_tone_map(hdr_image);
```

8.2.2 Local Operators

Local tone-mapping operators compress a pixel's dynamic range in a way dependent on the neighborhood of pixels around it. These operators mimic the local adaptation of the human eye and are capable of more striking or artistic results than global operators, but they are also susceptible to artifacts such as excessive haloing and reverse gradients. `<vw/HDR/LocalToneMap.h>` currently implements the Ashikhmin local tone-mapping operator [1]. It is much slower than the Drago operator and more prone to artifacts, but may be useful for some images. Its only parameter is a threshold value (0.5 by default) which roughly controls the size of the neighborhood used for each pixel. A threshold value too large will result in haloing.

```
// Apply Ashikhmin tone-mapping operator
ImageView<PixelRGB<float> > tone_mapped = ashikhmin_tone_map(hdr_image);
```

8.3 Command Line Tools

The HDR module builds two small utilities for working with HDR images from the command terminal. If you simply type these command names with no arguments, you will see a list of acceptable arguments.

- `hdr_merge` : Merge LDR images into one HDR image
- `hdr_tonemap` : Tonemap an HDR image using the Drago operator

8.4 Other Resources

There are a number of freely available utilities which are useful for working with HDR images. The OpenEXR distribution [5] includes several utilities, including exrdisplay for displaying OpenEXR images. `exrtools` [2] provides utilities for converting between OpenEXR and other formats, performing basic operations on OpenEXR images, and a couple of tone-mapping utilities. `pfstools` [7] is a well-integrated set of utilities for reading, writing, manipulating and viewing HDR images. It has an associated pfstmo project that implements seven of the more prominent tone mapping operators.

```
#include <vw/vw.h>
#include <stdio.h>
#include <vector>
using namespace vw;
using namespace vw::HDR;

int main(int argc, char** argv) {
  std::vector<Vector<double> > curves;
  std::vector<string> files;
  for(int i = 1; i < argc; i++)
    files.push_back(argv[i]);
  // Process HDR stack using Exif tags
  ImageView<PixelRGB<double> > hdr_exif = process_ldr_images_exif(files,
                                          curves);
  write_image("hdr.exr", hdr_exif);

  // Apply Drago tone-mapping operator.
  ImageView<PixelRGB<double> > tone_mapped = drago_tone_map(hdr_exif);
  write_image("tm.jpg", tone_mapped);

  // Apply gamma correction and save.
  ImageView<PixelRGB<double> > gamma = pow(tone_mapped, 1.0/2.2);
  write_image("tm_gamma.jpg", gamma);

  // Re-apply camera response curves and save.
  // First must invert curves calculated earlier.
  std::vector<Vector<double> > inverse_curves(curves.size());
  for (int i = 0; i < curves.size(); i++) {
    invert_curve(curves[i], inverse_curves[i],
                 VW_HDR_RESPONSE_POLYNOMIAL_ORDER);
  }
  psi(tone_mapped, inverse_curves);
  write_image("tm_curved.jpg", tone_mapped);

  // Apply gamma correction after response curves.
  // Usually gives best results.
  ImageView<PixelRGB<double> > tm_c_g = pow(tone_mapped, 1.0/2.2);
  write_image("tm_c_g.jpg", tm_c_g);

  return 0;
}
```

Listing 4: [`ExifHDRExample.cc`] This is a simple test program that stitches an Exif-tagged HDR stack into an HDR image, performs tone-mapping, and saves several versions with different post-processing applied for comparison. Usually the best image is produced by re-applying the camera response curves and then gamma correcting.

Bibliography

[1] Ashikhmin, Michael, "A Tone Mapping Algorithm for High Contrast Images," *Eurographics Workshop on Rendering*, 2002: 1–11.

[2] Biggs, Billy, "exrtools: a collection of utilities for manipulating OpenEXR images,", 2004, http://scanline.ca/exrtools/.

[3] Drago et al., "Adaptive Logarithmic Mapping For Displaying High Contrast Scenes," *Eurographics*, **22**(3), 2003.

[4] Fattal, Raanan, et. al, "Gradient Domain High Dynamic Range Compression," *ACM Transactions on Graphics*, 2002.

[5] Industrial Light and Magic, "OpenEXR," (Lucasfilm Ltd., 2006), http://www.openexr.com.

[6] Kerr, Douglas, "APEX–The Additive System of Photographic Exposure," 2006, http://doug.kerr.home.att.net/pumpkin/APEX.pdf.

[7] Mantiuk, Rafal, and Grzegorz Krawczyk, "pfstools for HDR processing," 2006, http://www.mpi-inf.mpg.de/resources/pfstools/.

[8] Reinhard, Erik, et. al. "Photographic Tone Reproduction for Digital Images," *ACM Transactions on Graphics*, 2002.

[9] Reinhard, Erik, Greg Ward, Sumanta Pattanaik, and Paul Debevec, *High Dynamic Range Imaging*, (Boston: Elsevier, 2006).

[10] Ward, Greg, et al., "A Visibility Matching Tone Reproduction Operator for High Dynamic Range Scenes," *IEEE Transactions on Visualization and Computer Graphics*, 1997.

Chapter 9

The Cartography Module

[*Note: The documentation for this chapter is not yet complete...*]

The earliest robots in space were not planetary rovers – they were unmanned probes that studied the planetary bodies in our solar system from afar. Today there are roughly twenty extraterrestrial spacecraft in active communication with earth (and only two planetary rovers), so the bulk of the extraterrestrial data that we receive consists of imagery that originated on round(ish) surfaces. The natural thing to do with this data is to merge it together into a map, but when doing so we are faced with the same problem that has plagued cartographers for hundreds of years: how does one flatten the globe? The is the job of the Vision Workbench Cartography module: to make maps.

Before diving into an introduction on planetary cartography, we will point out another problem that is relatively new to Cartography. The amount of map data that we have collected about Earth and the other bodies in our solar system is *immense*. It is often impossible to store an entire mapping data set in memory all at once, so intelligent paging, caching, and storage strategies must be used in order to make working with this data tractable. For this reason, the Cartography module is particularly powerful when used in conjunction with Mosaic module (see Chapter 7), which is designed to efficiently process and combine extremely large data sets.

We will begin this chapter with a quick summary of the third party libraries that are needed to compile the Vision Workbench. We will then describe the `GeoRefence` class, which creates a relationship between pixel coordinates in an image and coordinates on a globe. Next, we will discuss the `GeoTransform` class, which provides a simple means of re-projecting map data. We finish this section with methods for reading and writing image files with embedded geospatial metadata.

9.1 Software Dependencies

The Cartography module is currently built on top of two third party libraries:

- GDAL [`http://www.remotesensing.org/gdal/`]
- Proj.4 [`http://proj.maptools.org/`]

In order to enable the Cartography module, you must have these libraries installed on your system before you configure and build the Vision Workbench. You may need to use the `PKG_PATHS` directive in the `config.options` file if you install them in a non-standard location as discussed in Chapter 2.

Once the library for the Cartography module has been built, the header files for GDAL and Proj.4 are no longer needed, so you can rely solely on linking in libraries when building your own application.

9.2 The `GeoReference` Class

When you point at a location on a map, you probably want to know where that location can be found in the real world. This relationship depends first and foremost on the familiar notion of a map's scale. However, this relationship is also affected by a subtle, but extremely important dependence on how the map is *projected*. That is, the image depicts a scene that sits on the surface of a spheroid. However, the image is flat, so at best it represents a very slightly distorted view of the surface.

One can image all sorts of different ways that the surface can be warped or projected onto a flat plane (or, at the very least, projecting onto a manifold that can be unfolded into a plane without distorting distances and areas – a sphere cannot be unfolded in this way). Generations of cartographers have struggled with this topological challenge, and as a result they have developed many different ways to "un-fold" the globe so that it can be represented as a flat image. Rather than attempt a description of these many techniques here, we suggest you look at this excellent web site describing all aspects of map projections.

`http://www.progonos.com/furuti/MapProj/CartIndex/cartIndex.html`

The Proj.4 manual is also recommended as a reference for the specific map projections supported by the Vision Workbench.

Now would be a good time to take a break from reading this section of the documentation to look over these references. When you return, we will dive into some code examples.

9.2.1 The Datum

A Vision Workbench `GeoReference` object is composed of three items:

- **The Projection**: As discussed above, this is the technique used to represent the round globe in a flat image.
- **The Affine Transform**: This is the geometric transformation between pixel coordinates in the image to coordinates in the map projection space.
- **The Datum**: Describes the approximate shape of the planetary body, as either a sphere or an ellipsoid.

9.2.2 The Affine Transform

Let's start by being explicit about the coordinate systems we will be working with. For images, we adopt the usual Vision Workbench coordinate system wherein the upper left corner of the image is the origin, the u coordinate increases as you move right along the columns of the image, and the v coordinate increases as you move down the rows.

For a planetary body, the coordinate of a point on the surface is typically measured in latitude, longitude, and radius (ϕ, θ, r). Lines of latitude are perpendicular to the axis of rotation and are

Method	Description
`set_sinusoidal()`	Sinusoidal Projection
`set_mercator()`	Mercator Projection
`set_orthographic()`	Orthographic Projection
`set_stereographic()`	Stereographic Projection
`set_UTM()`	Universal Transverse Mercator (UTM) Projection (Earth only)

Table 9.1: Currently supported `GeoReference` map projections.

measured from the center line, the *equator* (+/-90 degrees). Lines of Longitude are vertical, passing through both the North and South poles of the planet. It is measured from a vertical arc on the surface called the *meridian*. We will generally adopt an East positive frame of reference (latitude increases to the east of the meridian 0-360 degrees). Finally, the radius is measured from the point to the planet's center of mass. Note that this coordinate system is similar but not identical to spherical coordinates in a mathematical sense, where "latitude" would be measured from the North pole rather than the equator.

Under this set of assumptions, if we have a point $P_{img} = (u, v)$ in the image, and we want to relate it to some planetary coordinates

9.2.3 Putting Things Together

9.3 Geospatial Image Processing

9.3.1 The `GeoTransform` Functor

9.4 Georeferenced File I/O

9.4.1 `DiskImageResourceGDAL`

Chapter 10

Advanced Topics

Alas, this chapter has not yet been written. Contact the authors for assistance with any of these topics.

10.1 Working with Shallow Views

10.2 Efficient Algorithms and `pixel_accessor`

10.3 Rasterization, Efficiency, and Tiled Computation

10.4 Generic Image Buffers

10.5 The File I/O System

10.6 Frequency-Domain Image Processing