OpenImageIO 0.1

Programmer Documentation

(in progress)

Editor: Larry Gritz

Date: 28 August, 2008

I kinda like "Oy-e-oh" with a bit of a groaning Yiddish accent, as in "OIIO, did you really write yet another file I/O library?"

Dan Wexler

Contents

1 Introduction

Welcome to OpenImageIO!

1.1 Overview

OpenImageIO provides simple but powerful `ImageInput` and `ImageOutput` APIs that abstract the reading and writing of 2D image file formats. They don't support every possible way of encoding images in memory, but for a reasonable and common set of desired functionality, they provide an exceptionally easy way for an application using the APIs support a wide — and extensible — selection of image formats without knowing the details of any of these formats.

Concrete instances of these APIs, each of which implements the ability to read and/or write a different image file format, are stored as plugins (i.e., dynamic libraries, DLL's, or DSO's) that are loaded at runtime. The OpenImageIO distribution contains such plugins for several popular formats. Any user may create conforming plugins that implement reading and writing capabilities for other image formats, and any application that uses OpenImageIO would be able to use those plugins.

The OpenImageIO distribution contains several utility programs that operate on images, each of which is built atop `ImageInput` and `ImageOutput`, and therefore may read or write any image file type for which an appropriate plugin is found at runtime. Paramount among these utilities is `iv`, a really fantastic and powerful image viewing application.

The library also implements the helper class `ImageBuf`, which is a handy way to store and manipulate images in memory. `ImageBuf` itself uses `ImageInput` and `ImageOutput` for its file I/O, and therefore is also agnostic as to image file formats.

Finally, the library implements a `TextureSystem` class that can be used for filtered lookups of multiresolution (MIP-map) textures, environment maps, and shadow maps. The texture library not only uses `ImageInput` to be format-agnostic, but it also transparently manages a cache of image data so that it can access truly vast amounts of texture (thousands of texture files totaling tens of GB) very efficiently using only a tiny amount (tens of megabytes at most) of runtime memory.

All of this is released as "open source" software using the very permissive BSD license. So you should feel free to use any or all of OpenImageIO in your own software, whether it is private or public, open source or proprietary, free or commercial. You may also modify it on your own. You are also encouraged to contribute to the continued development of OpenImageIO and to share any improvements that you make on your own, though you are by no means required to do so.

1.2 Simplifying Assumptions

OpenImageIO is not the only image library in the world. Certainly there are many fine libraries that implement a single image format (including the excellent `libtiff`, `jpeg-6b`, and `OpenEXR` that OpenImageIO itself relies on). Many libraries attempt to present a uniform API for reading and writing multiple image file formats. Most of these support a fixed set of image formats, though a few of these also attempt to provide an extensible set by using the plugin approach.

But in our experience, these libraries are all flawed in one or more ways: (1) They either support only a few formats, or many formats but with the majority of them somehow incomplete or incorrect. (2) Their APIs are not sufficiently expressive as to handle all the image features we need (such as tiled images, which is critical for our texture library). (3) Their APIs are *too complete*, trying to handle every possible permutation of image format features, and as a result are horribly complicated.

The third sin is the most severe, and is almost always the main problem at the end of the day. Even among the many open source image libraries that rely on extensible plugins, we have not found one that is both sufficiently flexible and has APIs anywhere near as simple to understand and use as those of OpenImageIO.

Good design is usually a matter of deciding what *not* to do, and OpenImageIO is no exception. We achieve power and simplicity only by making simplifying assumptions. Among them:

- OpenImageIO only deals with ordinary 2D images, and to a limited extent 3D volumes, and image files that contain multiple (but finite) independent images within them. OpenImageIO **does not deal with motion picture files.** At least, not currently.
- Pixel data are 8- 16- or 32-bit int (signed or unsigned), 16- 32- or 16-bit float. NOTHING ELSE. No < 8 bit images, or pixels boundaries that aren't byte boundaries. Files with < 8 bits will appear to the client as 8-bit unsigned grayscale images.
- Only fully elaborated, non-compressed data are are accepted and returned by the API. Compression or special encodings are handled entirely within an OpenImageIO plugin.
- Color space is grayscale or RGB. Non-spectral data, such as XYZ, CMYK, or YUV, are converted to RGB upon reading.
- All color channels have the same data format. Upon read, an `ImageInput` ought to convert all channels to the one with the highest precision in the file.
- All image channels in a subimage are sampled at the same resolution. For file formats that allow some channels to be subsampled, they will be automatically up-sampled to the highest resolution channel in the subimage.
- Color information is always in the order R, G, B, and the alpha channel, if any, always follows RGB, and z channel (if any) always follows alpha. So if a file actually stores ABGR, the plugin is expected to rearrange it as RGBA.

It's important to remember that these restrictions apply to data passed through the APIs, not to the files themselves. It's perfectly fine to have an OpenImageIO plugin that supports YUV

data, or 4 bits per channel, or any other exotic feature. You could even write a movie-reading `ImageInput` (despite OpenImageIO's claims of not supporting movies) and make it look to the client like it's just a series of images within the file. It's just that all the nonconforming details are handled entirely within the OpenImageIO plugin and are not exposed through the main OpenImageIO APIs.

Historical Origins

OpenImageIO is the evolution of concepts and tools I've been working on for two decades.

In the 1980's, every program I wrote that output images would have a simple, custom format and viewer. I soon graduated to using a standard image file format (TIFF) with my own library implementation. Then I switched to Sam Leffler's stable and complete `libtiff`.

In the mid-to-late-1990's, I worked at Pixar as one of the main implementors of PhotoRealistic RenderMan, which had *display drivers* that consisted of an API for opening files and outputting pixels, and a set of DSO/DLL plugins that each implemented image output for each of a dozen or so different file format. The plugins all responded to the same API, so the renderer itself did not need to know how to the details of the image file formats, and users could (in theory, but rarely in practice) extend the set of output image formats the renderer could use by writing their own plugins.

This was the seed of a good idea, but PRMan's display driver plugin API was abstruse and hard to use. So when I started Exluna in 2000, Matt Pharr, Craig Kolb, and I designed a new API for image output for our own renderer, Entropy. This API, called "ExDisplay," was C++, and much simpler, clearer, and easier to use than PRMan's display drivers.

NVIDIA's Gelato (circa 2002), whose early work was done by myself, Dan Wexler, Jonathan Rice, and Eric Enderton, had an API called "ImageIO." ImageIO was *much* more powerful and descriptive than ExDisplay, and had an API for *reading* as well as writing images. Gelato was not only "format agnostic" for its image output, but also for its image input (textures, image viewer, and other image utilities). We released the API specification and headers (though not the library implementation) using the BSD open source license, firmly repudiating any notion that the API should be specific to NVIDIA or Gelato.

For Gelato 3.0 (circa 2007), we refined ImageIO again (by this time, Philip Nemec was also a major influence, in addition to Dan, Eric, and myself[1]). This revision was not a major overhaul but more of a fine tuning. Our ideas were clearly approaching stability. But, alas, the Gelato project was canceled before Gelato 3.0 was released, and despite our prodding, NVIDIA executives would not open source the full ImageIO code and related tools.

After I left NVIDIA, I was determined to recreate this work once again – and ONLY once more – and release it as open source from the start. Thus, OpenImageIO was born. I started with the existing Gelato ImageIO specification and headers (which were BSD licensed all along), and made some more refinements since I had to rewrite the entire implementation from scratch anyway. I think the additional changes are all improvements. This is the software you have in your hands today.

[1] Gelato as a whole had many other contributors; those I've named here are the ones I recall contributing to the design or implementation of the ImageIO APIs

Acknowledgments

Part I

The ImageIO Library

2 Image I/O API

2.1 Data Type Descriptions: `TypeDesc`

There are two kinds of data that are important to OpenImageIO:

- *Internal data* is in the memory of the computer, used by an application program.
- *Native file data* is what is stored in an image file itself (i.e., on the "other side" of the abstraction layer that OpenImageIO provides).

Both internal and file data is stored in a particular *data format* that describes the numerical encoding of the values. OpenImageIO understands several types of data encodings, and there is a special type, `TypeDesc`, that allows their enumeration. A `TypeDesc` describes a base data format type, aggregation into simple vector and matrix types, and an array length (if it's an array).

`TypeDesc` supports the following base data format types, given by the enumerated type `BASETYPE`:

`UINT8`	8-bit integer values ranging from 0..255, corresponding to the C/C++ `unsigned char`.
`INT8`	8-bit integer values ranging from -128..127, corresponding to the C/C++ `char`.
`UINT16`	16-bit integer values ranging from 0..65535, corresponding to the C/C++ `unsigned short`.
`INT16`	16-bit integer values ranging from -32768..32767, corresponding to the C/C++ `short`.
`UINT`	32-bit integer values, corresponding to the C/C++ `unsigned int`.
`INT`	signed 32-bit integer values, corresponding to the C/C++ `int`.
`FLOAT`	32-bit IEEE floating point values, corresponding to the C/C++ `float`.
`DOUBLE`	64-bit IEEE floating point values, corresponding to the C/C++ `double`.
`HALF`	16-bit floating point values in the format supported by OpenEXR and OpenGL.

A `TypeDesc` can be constructed using just this information, either as a single scalar value, or an array of scalar values:

`TypeDesc` `(BASETYPE btype)`
`TypeDesc` `(BASETYPE btype, int arraylength)`

Construct a type description of a single scalar value of the given base type, or an array of such scalars if an array length is supplied. For example, `TypeDesc(UINT8)` describes

an unsigned 8-bit integer, and `TypeDesc(FLOAT,7)` describes an array of 7 32-bit float values. Note also that a non-array `TypeDesc` may be implicitly constructed from just the `BASETYPE`, so it's okay to pass a `BASETYPE` to any function parameter that takes a full `TypeDesc`.

In addition, `TypeDesc` supports certain aggregate types, described by the enumerated type `AGGREGATE`:

`SCALAR`	a single scalar value (such as a raw `int` or `float` in C). This is the default.
`VEC2`	two values representing a 2D vector.
`VEC3`	three values representing a 3D vector.
`VEC4`	four values representing a 4D vector.
`MATRIX44`	two values representing a 4×4 matrix.

And optionally, several vector transformation semantics, described by the enumerated type `VECSEMANTICS`:

`NOXFORM`, `COLOR`	synonyms that indicate that the item is not a spatial quantity that undergoes any particular transformation.
`POINT`	indicates that the item represents a spatial position and should be transformed by a 4×4 matrix as if it had a 4th component of 1.
`VECTOR`	indicates that the item represents a spatial direction and should be transformed by a 4×4 matrix as if it had a 4th component of 0.
`NORMAL`	indicates that the item represents a surface normal and should be transformed like a vector, but using the inverse-transpose of a 4×4 matrix.

These can be combined to fully describe a complex type:

```
TypeDesc (BASETYPE btype, AGGREGATE agg, VECSEMANTICS xform=NOXFORM)
TypeDesc (BASETYPE btype, AGGREGATE agg, int arraylen)
TypeDesc (BASETYPE btype, AGGREGATE agg, VECSEMANTICS xform, int arraylen)
```

Construct a type description of an aggregate (or array of aggregates), with optional vector transformation semantics. For example, `TypeDesc(HALF,COLOR)` describes an aggregate of 3 16-bit floats comprising a color, and `TypeDesc(FLOAT,VEC3,POINT)` describes an aggregate of 3 32-bit floats comprising a 3D position.

Note that aggregates and arrays are different. A `TypeDesc(FLOAT,3)` is an array of three floats, a `TypeDesc(FLOAT,COLOR)` is a single 3-channel color comprised of floats, and `TypeDesc(FLOAT,3,COLOR)` is an array of 3 color values, each of which is comprised of 3 floats.

Of these, the only ones commonly used to store pixel values in image files are scalars of `UINT8`, `UINT16`, `FLOAT`, and `HALF` (the last only used by OpenEXR, to the best of our knowledge).

Note that the `TypeDesc` (which is also used for applications other than images) can describe many types not used by OpenImageIO. Please ignore this extra complexity; only the above

simple types are understood by OpenImageIO as pixel storage data types, though a few others, including STRING and MATRIX44 aggregates, are occasionally used for *metadata* for certain image file formats (see Sections 3.2.5, 4.2.4, and the documentation of individual ImageIO plugins for details).

2.2 Image Specification: ImageSpec

An ImageSpec is a structure that describes the complete format specification of a single image. It contains:

- The image resolution (number of pixels).
- The origin, if its upper left corner is not located beginning at pixel (0,0).
- The full size and offset of an abstract "full" or "display" image, useful for describing cropping or overscan.
- Whether the image is organized into *tiles*, and if so, the tile size.
- The *native data format* of the pixel values (e.g., float, 8-bit integer, etc.).
- The number of color channels in the image (e.g., 3 for RGB images), names of the channels, and whether any particular channels represent *alpha* and *depth*.
- Any presumed gamma correction or hints about color space of the pixel values.
- Quantization parameters describing how floating point values should be converted to integers (if cases where users pass real values but integer values are stored in the file). This is used only when writing images, not when reading them.
- A user-extensible (and format-extensible) list of any other arbitrarily-named and -typed data that may help describe the image or its disk representation.

2.2.1 ImageSpec Data Members

The ImageSpec contains data fields for the values that are required to describe nearly any image, and an extensible list of arbitrary attributes that can hold metadata that may be user-defined or specific to individual file formats. Here are the hard-coded data fields:

```
int width, height, depth
int x, y, z
```

width, height, depth are the size of the data of this image, i.e., the number of pixels in each dimension. A depth greater than 1 indicates a 3D "volumetric" image.

x, y, z indicate the *origin* of the pixel data of the image. These default to (0,0,0), but setting them differently may indicate that this image is offset from the usual origin.

Therefore the pixel data are defined over pixel coordinates [x ... x+width-1] horizontally, [y ... y+height-1] vertically, and [z ... z+depth-1] in depth.

`int full_width, full_height, full_depth`
`int full_x, full_y, full_z`

These fields define a "full" or "display" image window over the region [`full_x` ... `full_x+full_width-1`] horizontally, [`full_y` ... `full_y+full_height-1`] vertically, and [`full_z` ... `full_z+full_depth-1`] in depth.

Having the full display window different from the pixel data window can be helpful in cases where you want to indicate that your image is a *crop window* of a larger image (if the pixel data window is a subset of the full display window), or that the pixels include *overscan* (if the pixel data is a superset of the full display window), or may simply indicate how different non-overlapping images piece together.

`int tile_width, tile_height, tile_depth`

If nonzero, indicates that the image is stored on disk organized into rectangular *tiles* of the given dimension. The default of (0,0,0) indicates that the image is stored in scanline order, rather than as tiles.

`TypeDesc format`

Indicates the native format of the pixel data values themselves, as a `TypeDesc` (see 2.1). Typical values would be `TypeDesc::UINT8` for 8-bit unsigned values, `TypeDesc::FLOAT` for 32-bit floating-point values, etc.

NOTE: Currently, the implementation of OpenImageIO requires all channels to have the same data format.

`int nchannels`

The number of *channels* (color values) present in each pixel of the image. For example, an RGB image has 3 channels.

`std:vector<std::string> channelnames`

The names of each channel, in order. Typically this will be `"R"`, `"G"`,`"B"`, `"A"` (alpha), `"Z"` (depth), or other arbitrary names.

`int alpha_channel`

The index of the channel that respresents *alpha* (pixel coverage and/or transparency). It defaults to -1 if no alpha channel is present, or if it is not know which channel represents alpha.

`int z_channel`

The index of the channel that respresents *z* or *depth* (from the camera). It defaults to -1 if no depth channel is present, or if it is not know which channel represents depth.

`LinearitySpec linearity`

Describes the mapping of pixel values to real-world units. `LinearitySpec` is an enumerated type that may take on the following values:

- `Linear` (the default) indicates that pixel values map linearly.
- `GammaCorrected` indicates that the color pixel values have already been gamma corrected, using the exponent given by the `gamma` field. (It is still assumed that non-color values, such as alpha and depth, are linear.)
- `sRGB` indicates that color values are encoded using the sRGB mapping. (It is still assumed that non-color values are linear.)

`float gamma`

The gamma exponent, if the pixel values in the image have already been gamma corrected (indicated by `nonlinear` having a value of `GammaCorrected`). The default of 1.0 indicates that no gamma correction has been applied.

```
int quant_black, quant_white, quant_min, quant_max;
float quant_dither
```

Describes the *quantization*, or mapping between real (floating-point) values and the stored integer values. Please refer to Section 3.2.6 for a more complete explanation of each of these parameters.

`ParamValueList extra_attribs`

A list of arbitrarily-named and arbitrarily-typed additional attributes of the image, for any metadata not described by the hard-coded fields described above. This list may be manipulated with the `attribute()` and `find_attribute()` methods.

2.2.2 `ImageSpec` member functions

`ImageSpec` contains the following methods that manipulate format specs or compute useful information about images given their format spec:

`ImageSpec` `(int xres, int yres, int nchans, TypeDesc fmt = UINT8)`

Constructs an `ImageSpec` with the given x and y resolution, number of channels, and pixel data format.

All other fields are set to the obvious defaults – the image is an ordinary 2D image (not a volume), the image is not offset or a crop of a bigger image, the image is scanline-oriented (not tiled), channel names are "R", "G", "B," and "A" (up to and including 4 channels, beyond that they are named "channel *n*"), the fourth channel (if it exists) is assumed to be alpha, values are assumed to be linear, and quantization (if *fmt* describes an integer type) is done in such a way that the maximum positive integer range maps to (0.0, 1.0).

`void` **`set_format`** `(TypeDesc fmt)`

Sets the format as described, and also sets all quantization parameters to the default for that data type (as explained in Section 3.2.6).

`void` **`default_channel_names`** `()`

Sets the `channelnames` to reasonable defaults for the number of channels. Specifically, channel names are set to "R", "G", "B," and "A" (up to and including 4 channels, beyond that they are named "channel*n*".

```
static TypeDesc
format_from_quantize (int quant_black, int quant_white,
                      int quant_min, int quant_max)
```

Utility function that, given quantization parameters, returns a data type that may be used without unacceptable loss of significant bits.

`size_t` **`channel_bytes`** `()`

Returns the number of bytes comprising each channel of each pixel (i.e., the size of a single value of the type described by the `format` field).

`size_t` **`pixel_bytes`** `()`

Returns the number of bytes comprising each pixel (i.e. the number of channels multiplied by the channel size).

`size_t` **`scanline_bytes`** `()`

Returns the number of bytes comprising each scanline (i.e. `width` pixels).

`size_t` **`tile_bytes`** `()`

Returns the number of bytes comprising an image tile (if it's a tiled image).

`size_t` **`image_bytes`** `()`

Returns the number of bytes comprising an image of these dimensions.

```
void attribute (const std::string &name, TypeDesc type,
                int nvalues, const void *value)
```

Add a metadata attribute to `extra_attribs`, with the given name, data type, and number of values. The `value` pointer specifies the address of the data to be copied.

```
void attribute (const std::string &name, unsigned int value)
void attribute (const std::string &name, int value)
void attribute (const std::string &name, float value)
void attribute (const std::string &name, const char *value)
void attribute (const std::string &name, const std::string &value)
```

Shortcuts for passing attributes comprised of a single integer, floating-point value, or string.

```
ParamValue * find_attribute (const std::string &name,
                             TypeDesc searchtype=UNKNOWN,
                             bool casesensitive=false)
```

Searches `extra_attribs` for an attribute matching `name`, returning a pointer to the attribute record, or NULL if there was no match. If `searchtype` is `TypeDesc::UNKNOWN`, the search will be made regardless of the data type, whereas other values of `searchtype` will reject a matching name if the data type does not also match. The name comparison will be exact if `casesensitive` is true, otherwise in a case-insensitive manner if `caseinsensitive` is false.

3 ImageOutput: Writing Images

3.1 Image Output Made Simple

Here is the simplest sequence required to write the pixels of a 2D image to a file:

```
#include "imageio.h"
using namespace OpenImageIO;
...

const char *filename = "foo.jpg";
const int xres = 640, yres = 480;
const int channels = 3;  // RGB
unsigned char pixels[xres*yres*channels];

ImageOutput *out = ImageOutput::create (filename);
ImageSpec spec (xres, yres, channels, TypeDesc::UINT8);
out->open (filename, spec);
out->write_image (TypeDesc::UINT8, pixels);
out->close ();
delete out;
```

This little bit of code does a surprising amount of useful work:

- Search for an ImageIO plugin that is capable of writing the file (`"foo.jpg"`), deducing the format from the file extension. When it finds such a plugin, it creates a subclass instance of `ImageOutput` that writes the right kind of file format.

  ```
  ImageOutput *out = ImageOutput::create (filename);
  ```

- Open the file, write the correct headers, and in all other important ways prepare a file with the given dimensions (640×480), number of color channels (3), and data format (unsigned 8-bit integer).

  ```
  ImageSpec spec (xres, yres, channels, TypeDesc::UINT8);
  out->open (filename, spec);
  ```

- Write the entire image, hiding all details of the encoding of image data in the file, whether the file is scanline- or tile-based, or what is the native format of data in the file (in this

case, our in-memory data is unsigned 8-bit and we've requested the same format for disk storage, but if they had been different, `write_image()` would do all the conversions for us).

```
out->write_image (TypeDesc::UINT8, &pixels);
```

- Close the file, destroy and free the `ImageOutput` we had created, and perform all other cleanup and release of any resources needed by the plugin.

```
out->close ();
delete out;
```

3.2 Advanced Image Output

Let's walk through many of the most common things you might want to do, but that are more complex than the simple example above.

3.2.1 Writing individual scanlines, tiles, and rectangles

The simple example of Section 3.1 wrote an entire image with one call. But sometimes you are generating output a little at a time and do not wish to retain the entire image in memory until it is time to write the file. OpenImageIO allows you to write images one scanline at a time, one tile at a time, or by individual rectangles.

Writing individual scanlines

Individual scanlines may be written using the `write_scanline()` API call:

```
...
unsigned char scanline[xres*channels];
out->open (filename, spec);
int z = 0;   // Always zero for 2D images
for (int y = 0;  y < yres;  ++y) {
    ... generate data in scanline[0..xres*channels-1] ...
    out->write_scanline (y, z, TypeDesc::UINT8, scanline);
}
out->close ();
...
```

The first two arguments to `write_scanline()` specify which scanline is being written by its vertical (y) scanline number (beginning with 0) and, for volume images, its slice (z) number (the slice number should be 0 for 2D non-volume images). This is followed by a `TypeDesc` describing the data you are supplying, and a pointer to the pixel data itself. Additional optional arguments describe the data stride, which can be ignored for contiguous data (use of strides is explained in Section 3.2.3).

All `ImageOutput` implemenations will accept scanlines in strict order (starting with scanline 0, then 1, up to `yres-1`, without skipping any). See Section 3.2.7 for details on out-of-order or repeated scanlines.

The full description of the `write_scanline()` function may be found in Section 3.3.

Writing individual tiles

Not all image formats (and therefore not all `ImageOutput` implementations) support tiled images. If the format does not support tiles, then `write_tile()` will fail. An application using OpenImageIO should gracefully handle the case that tiled output is not available for the chosen format.

Once you `create()` an `ImageOutput`, you can ask if it is capable of writing a tiled image by using the `supports("tiles")` query:

```
...
ImageOutput *out = ImageOutput::create (filename);
if (! out->supports ("tiles")) {
    // Tiles are not supported
}
```

Assuming that the `ImageOutput` supports tiled images, you need to specifically request a tiled image when you `open()` the file. This is done by setting the tile size in the `ImageSpec` passed to `open()`. If the tile dimensions are not set, they will default to zero, which indicates that scanline output should be used rather than tiled output.

```
int tilesize = 64;
ImageSpec spec (xres, yres, channels, TypeDesc::UINT8);
spec.tile_width = tilesize;
spec.tile_height = tilesize;
out->open (filename, spec);
...
```

In this example, we have used square tiles (the same number of pixels horizontally and vertically), but this is not a requirement of OpenImageIO. However, it is possible that some image formats may only support square tiles, or only certain tile sizes (such as restricting tile sizes to powers of two). Such restrictions should be documented by each individual plugin.

```
unsigned char tile[tilesize*tilesize*channels];
int z = 0;   // Always zero for 2D images
for (int y = 0;  y < yres;  y += tilesize) {
    for (int x = 0;  x < xres;  x += tilesize) {
        ... generate data in tile[] ..
        out->write_tile (x, y, z, TypeDesc::UINT8, tile);
    }
}
out->close ();
...
```

The first three arguments to `write_tile()` specify which tile is being written by the pixel coordinates of any pixel contained in the tile: x (column), y (scanline), and z (slice, which should always be 0 for 2D non-volume images). This is followed by a `TypeDesc` describing the data you are supplying, and a pointer to the tile's pixel data itself, which should be ordered by increasing slice, increasing scanline within each slice, and increasing column within each scanline. Additional optional arguments describe the data stride, which can be ignored for contiguous data (use of strides is explained in Section 3.2.3).

All `ImageOutput` implemenations that support tiles will accept tiles in strict order of increasing y rows, and within each row, increasing x column, without missing any tiles. See Section 3.2.7 for details on out-of-order or repeated tiles.

The full description of the `write_tile()` function may be found in Section 3.3.

Writing arbitrary rectangles

Some `ImageOutput` implementations — such as those implementing an interactive image display, but probably not any that are outputting directly to a file — may allow you to send arbitrary rectangular pixel regions. Once you `create()` an `ImageOutput`, you can ask if it is capable of accepting arbitrary rectangles by using the `supports("rectangles")` query:

```
...
ImageOutput *out = ImageOutput::create (filename);
if (! out->supports ("rectangles")) {
    // Rectangles are not supported
}
```

If rectangular regions are supported, they may be sent using the `write_rectangle()` API call:

```
unsigned int rect[...];
... generate data in rect[] ..
out->write_rectangle (xmin, xmax, ymin, ymax, zmin, zmax, TypeDesc::UINT8, rect);
...
```

The first six arguments to `write_rectangle()` specify the region of pixels that is being transmitted by supplying the minimum and maximum pixel indices in x (column), y (scanline), and z (slice, always 0 for 2D non-volume images). The total number of pixels being transmitted is therefore:

```
(xmax-xmin+1) * (ymax-ymin+1) * (zmax-zmin+1)
```

This is followed by a `TypeDesc` describing the data you are supplying, and a pointer to the rectangle's pixel data itself, which should be ordered by increasing slice, increasing scanline within each slice, and increasing column within each scanline. Additional optional arguments describe the data stride, which can be ignored for contiguous data (use of strides is explained in Section 3.2.3).

3.2.2 Converting formats

The code examples of the previous sections all assumed that your internal pixel data is stored as unsigned 8-bit integers (i.e., 0-255 range). But OpenImageIO is significantly more flexible.

You may request that the output image be stored in any of several formats. This is done by setting the `format` field of the `ImageSpec` prior to calling `open`. You can do this upon construction of the `ImageSpec`, as in the following example that requests a spec that stores data as 16-bit unsigned integers:

```
ImageSpec spec (xres, yres, channels, TypeDesc::UINT16);
```

Or, for an `ImageSpec` that has already been constructed, you may reset its format using the `set_format()` method (which also resets the various quantization fields of the spec to the defaults for the data format you have specified).

```
ImageSpec spec (...);
spec.set_format (TypeDesc::UINT16);
```

Note that resetting the format must be done *before* passing the spec to `open()`, or it will have no effect on the file.

Individual file formats, and therefore `ImageOutput` implementations, may only support a subset of the formats understood by the OpenImageIO library. Each `ImageOutput` plugin implementation should document which data formats it supports. An individual `ImageOutput` implementation may choose to simply fail to `open()`, though the recommended behavior is for `open()` to succeed but in fact choose a data format supported by the file format that best preserves the precision and range of the originally-requested data format.

It is not required that the pixel data passed to `write_image()`, `write_scanline()`, `write_tile()`, or `write_rectangle()` actually be in the same data format as that requested as the native format of the file. You can fully mix and match data you pass to the various `write` routines and OpenImageIO will automatically convert from the internal format to the native file format. For example. the following code will open a TIFF file that stores pixel data as 16-bit unsigned integers (values ranging from 0 to 65535), compute internal pixel values as floating-point values, with `write_image()` performing the conversion automatically:

```
ImageOutput *out = ImageOutput::create ("myfile.tif");
ImageSpec spec (xres, yres, channels, TypeDesc::UINT16);
out->open (filename, spec);
...
float pixels [xres*yres*channels];
...
out->write_image (TypeDesc::FLOAT, pixels);
```

Note that `write_scanline()`, `write_tile()`, and `write_rectangle` have a parameter that works in a corresponding manner.

Please refer to Section 3.2.6 for more information on how values are translated among the supported data formats by default, and how to change the formulas by specifying quantization in the `ImageSpec`.

3.2.3 Data Strides

In the preceeding examples, we have assumed that the block of data being passed to the `write` functions are *contiguous*, that is:

- each pixel in memory consists of a number of data values equal to the declared number of channels that are being written to the file;
- successive column pixels within a row directly follow each other in memory, with the first channel of pixel x of immediately following last channel of pixel $x-1$ of the same row;
- for whole images, tiles or rectangles, the data for each row immediately follows the previous one in memory (the first pixel of row y immediately follows the last column of row $y-1$);
- for 3D volumetric images, the first pixel of slice z immediately follows the last pixel of of slice $z-1$.

Please note that this implies that data passed to `write_tile()` be contiguous in the shape of a single tile (not just an offset into a whole image worth of pixels), and that data passed to `write_rectangle()` be contiguous in the dimensions of the rectangle.

The `write_scanline()` function takes an optional `xstride` argument, and the `write_image()`, `write_tile()`, and `write_rectangle` functions take optional `xstride`, `ystride`, and `zstride` values that describe the distance, in *bytes*, between successive pixel columns, rows, and slices, respectively, of the data you are passing. For any of these values that are not supplied, or are given as the special constant `AutoStride`, contiguity will be assumed.

By passing different stride values, you can achieving some surprisingly flexible functionality. A few representative examples follow:

- Flip an image vertically upon writing, by using *negative y* stride:

```
unsigned char pixels[xres*yres*channels];
int scanlinesize = xres * channels * sizeof(pixels[0]);
...
out->write_image (TypeDesc::UINT8,
                  (char *)pixels + (yres-1)*scanlinesize,  // offset to last
                  AutoStride,                      // default x stride
                  -scanlinesize,                   // special y stride
                  AutoStride);                     // default z stride
```

- Write a tile that is embedded within a whole image of pixel data, rather than having a one-tile-only memory layout:

```
unsigned char pixels[xres*yres*channels];
int pixelsize = channels * sizeof(pixels[0]);
int scanlinesize = xres * pixelsize;
...
out->write_tile (x, y, 0, TypeDesc::UINT8,
```

```
                    (char *)pixels + y*scanlinesize + x*pixelsize,
                    pixelsize,
                    scanlinesize);
```

- Write only a subset of channels to disk. In this example, our internal data layout consists of 4 channels, but we write just channel 3 to disk as a one-channel image:

```
// In-memory representation is 4 channel
const int xres = 640, yres = 480;
const int channels = 4;  // RGBA
const int channelsize = sizeof(unsigned char);
unsigned char pixels[xres*yres*channels];

// File representation is 1 channel
ImageOutput *out = ImageOutput::create (filename);
ImageSpec spec (xres, yres, 1, TypeDesc::UINT8);
out->open (filename, spec);

// Use strides to write out a one-channel "slice" of the image
out->write_image (TypeDesc::UINT8,
                  (char *)pixels + 3*channelsize,  // offset to chan 3
                  channels*channelsize,            // 4 channel x stride
                  AutoStride,                      // default y stride
                  AutoStride);                     // default z stride
...
```

Please consult Section 3.3 for detailed descriptions of the stride parameters to each `write` function.

3.2.4 Writing a crop window or overscan region

The `ImageSpec` fields `width`, `height`, and `depth` describe the dimensions of the actual pixel data.

At times, it may be useful to also describe an abstract *full* or *display* image window, whose position and size may not correspond exactly to the data pixels. For example, a pixel data window that is a subset of the full display window might indicate a *crop window*; a pixel data window that is a superset of the full display window might indicate *overscan* regions (pixels defined outside the eventual viewport).

The `ImageSpec` fields `full_width`, `full_height`, and `full_depth` describe the dimensions of the full display window, and `full_x`, `full_y`, `full_z` describe its origin (upper left corner). The fields `x`, `y`, `z` describe the origin (upper left corner) of the pixel data.

These fields collectively describe an abstract full display image ranging from [`full_x` ... `full_x+full_width-1`] horizontally, [`full_y` ... `full_y+full_height-1`] vertically, and [`full_z` ... `full_z+full_depth-1`] in depth (if it is a 3D volume), and actual pixel data over the pixel coordinate range [`x` ... `x+width-1`] horizontally, [`y` ... `y+height-1`] vertically, and [`z` ... `z+depth-1`] in depth (if it is a volume).

Not all image file formats have a way to describe display windows. An `ImageOutput` implementation that cannot express display windows will always write out the `width` × `height` pixel data, may upon writing lose information about offsets or crop windows.

Here is a code example that opens an image file that will contain a 32×32 pixel crop window within an abstract 640×480 full size image. Notice that the pixel indices (column, scanline, slice) passed to the `write` functions are the coordinates relative to the full image, not relative to the crop widow, but the data pointer passed to the `write` functions should point to the beginning of the actual pixel data being passed (not the the hypothetical start of the full data, if it was all present).

```
int fullwidth = 640, fulllength = 480; // Full display image size
int cropwidth = 16, croplength = 16;  // Crop window size
int xorigin = 32, yorigin = 128;      // Crop window position
unsigned char pixels [cropwidth * croplength * channels]; // Crop size!
...
ImageOutput *out = ImageOutput::create (filename);
ImageSpec spec (cropwidth, croplength, channels, TypeDesc::UINT8);
spec.full_x = 0;
spec.full_y = 0;
spec.full_width = fullwidth;
spec.full_length = fulllength;
spec.x = xorigin;
spec.y = yorigin;
out->open (filename, spec);
...
int z = 0;   // Always zero for 2D images
for (int y = yorigin;  y < yorigin+croplength;  ++y) {
    out->write_scanline (y, z, TypeDesc::UINT8, (y-yorigin)*cropwidth*channels);
}
out->close ();
```

3.2.5 Writing metadata

The `ImageSpec` passed to `open()` can specify all the common required properties that describe an image: data format, dimensions, number of channels, tiling. However, there may be a variety of additional *metadata*[1] that should be carried along with the image or saved in the file.

The remainder of this section explains how to store additional metadata in the `ImageSpec`. It is up to the `ImageOutput` to store these in the file, in indeed the file format is able to accept the data. Individual `ImageOutput` implementations should document which metadata they respect.

Channel names

In addition to specifying the number of color channels, it is also possible to name those channels. Only a few `ImageOutput` implementations have a way of saving this in the file, but some do, so you may as well do it if you have information about what the channels represent.

[1] *Metadata* refers to data about data, in this case, data about the image that goes beyond the pixel values and description thereof.

By convention, channel names for red, green, blue, and alpha (or a main image) should be named "R", "G", "B", and "A", respectively. Beyond this guideline, however, you can use any names you want.

The ImageSpec has a vector of strings called channelnames. Upon construction, it starts out with reasonable default values. If you use it at all, you should make sure that it contains the same number of strings as the number of color channels in your image. Here is an example:

```
int channels = 4;
ImageSpec spec (width, length, channels, TypeDesc::UINT8);
spec.channelnames.clear ();
spec.channelnames.push_back ("R");
spec.channelnames.push_back ("G");
spec.channelnames.push_back ("B");
spec.channelnames.push_back ("A");
```

Here is another example in which custom channel names are used to label the channels in an 8-channel image containing beauty pass RGB, per-channel opacity, and texture s,t coordinates for each pixel.

```
int channels = 8;
ImageSpec spec (width, length, channels, TypeDesc::UINT8);
spec.channelnames.clear ();
spec.channelnames.push_back ("R");
spec.channelnames.push_back ("G");
spec.channelnames.push_back ("B");
spec.channelnames.push_back ("opacityR");
spec.channelnames.push_back ("opacityG");
spec.channelnames.push_back ("opacityB");
spec.channelnames.push_back ("texture_s");
spec.channelnames.push_back ("texture_t");
```

The main advantage to naming color channels is that if you are saving to a file format that supports channel names, then any application that uses OpenImageIO to read the image back has the option to retain those names and use them for helpful purposes. For example, the iv image viewer will display the channel names when viewing individual channels or displaying numeric pixel values in "pixel view" mode.

Specially-designated channels

The ImageSpec contains two fields, alpha_channel and z_channel, which can be used to designate which channel indices are used for alpha and z depth, if any. Upon construction, these are both set to -1, indicating that it is not known which channels are alpha or depth. Here is an example of setting up a 5-channel output that represents RGBAZ:

```
int channels = 5;
ImageSpec spec (width, length, channels, format);
spec.channelnames.push_back ("R");
spec.channelnames.push_back ("G");
spec.channelnames.push_back ("B");
```

```
spec.channelnames.push_back ("A");
spec.channelnames.push_back ("Z");
spec.alpha_channel = 3;
spec.z_channel = 4;
```

There are two advantages to designating the alpha and depth channels in this manner:

- Some file formats may require that these channels be stored in a particular order, with a particular precision, or the `ImageOutput` may in some other way need to know about these special channels.

- Certain operations that make sense for colors should not apply to alpha or z. For example, if your call to `write` reduces precision (e.g., converts from `float` to integer pixels) it will will typically add random *dither* to eliminate banding artifacts in the quantization. But for a variety of reasons, you want to add dither only to color channels and not to alpha. So setting `alpha_channel` will cause `write` to not dither that channel.

Linearity hints

We certainly hope that you are using only modern file formats that support high precision and extended range pixels (such as OpenEXR) and keeping all your images in a linear color space. But you may have to work with file formats that dictate the use of nonlinear color values. This is prevalent in formats that store pixels only as 8-bit values, since 256 values are not enough to linearly represent colors without banding artifacts in the dim values.

Since this can (and probably will) happen, the `ImageSpec` has fields that allow you to explain what color space your image pixels are in. Each individual `ImageOutput` should document how it uses this (or not).

The `ImageSpec` field `linearity` can take on any of the following values:

`ImageSpec::UnknownLinearity` the default, indicates that you have made no claim about the color space of your pixel data.

`ImageSpec::Linear` indicates that the pixel values you are pssing repesent linear values.

`ImageSpec::GammaCorrected` indicates that the color pixel values (but not alpha or z) that you are passing have already been gamma corrected (raised to the power $1/\gamma$), and that the gamma exponent may be found in the `gamma` field of the `ImageSpec`.

`ImageSpec::sRGB` indicates that the color pixel values that you are passing are already in sRGB color space.

Here is a simple example of setting up the `ImageSpec` when you know that the pixel values you are writing are linear:

```
ImageSpec spec (width, length, channels, format);
spec.linearity = ImageSpec::Linear;
...
```

If a particular `ImageOutput` implementation is required (by the rules of the file format it writes) to have pixels in a particular color space, then it will convert the color values of your image to the right color space if it is not already in that space. For example, JPEG images must be in sRGB space, so if you declare your pixels to be `Linear`, the JPEG `ImageOutput` will convert to sRGB.

If you leave the linearity set to the default of `UnknownLinearity`, the values will not be transformed, since the plugin can't be sure that it's not in the correct space to begin with.

The linearity only describes color channels. An `ImageOutput` plugin will assume that alpha or depth (z) channels (designated by the `alpha_channel` and `z_channel` fields, respectively) always represent linear values and should never be transformed.

Arbitrary metadata

For all other metadata that you wish to save in the file, you can attach the data to the `ImageSpec` using the `attribute()` methods. These come in polymorphic varieties that allow you to attach an attribute name and a value consisting of a single `int`, `unsigned int`, `float`, `char*`, or `std::string`, as shown in the following examples:

```
ImageIOFormatString spec (...);
...

unsigned int u = 1;
spec.attribute ("Orientation", u);

float x = 72.0;
spec.attribute ("dotsize", f);

std::string s = "Fabulous image writer 1.0";
spec.attribute ("Software", s);
```

These are convenience routines for metadata that consist of a single value of one of these common types. For other data types, or more complex arrangements, you can use the more general form of `attribute()`, which takes arguments giving the name, type (as a `TypeDesc`), number of values (1 for a single value, > 1 for an array), and then a pointer to the data values. For example,

```
ImageIOFormatString spec (...);

// Attach a 4x4 matrix to describe the camera coordinates
float mymatrix[16] = { ... };
spec.attribute ("worldtocamera", TypeDesc(FLOAT,MATRIX), 1, &mymatrix);

// Attach an array of two floats giving the CIE neutral color
float neutral[2] = { ... };
spec.attribute ("adoptedNeutral", TypeDesc::FLOAT, 2, &neutral);
```

In general, most image file formats (and therefore most `ImageOutput` implementations) are aware of only a small number of name/value pairs that they predefine and will recognize. Some

file formats (OpenEXR, notably) do accept arbitrary user data and save it in the image file. If an `ImageOutput` does not recognize your metadata and does not support arbitrary metadata, that metadatum will be silently ignored and will not be saved with the file.

Each individual `ImageOutput` implementation should document the names, types, and meanings of all metadata attributes that they understand.

3.2.6 Controlling quantization

It is possible that your internal data format (that in which you compute pixel values that you pass to the `write` functions) is of greater precision or range than the native data format of the output file. This can occur either because you specified a lower-precision data format in the `ImageSpec` that you passed to `open()`, or else that the image file format dictates a particular data format that does not match your internal format. For example, you may compute `float` pixels and pass those to `write_image()`, but if you are writing a JPEG/JFIF file, the values must be stored in the file as 8-bit unsigned integers.

The conversion from floating-point formats to integer formats (or from higher to lower integer, which is done by first converting to float) is controlled by five fields within the `ImageSpec`: `quant_black`, `quant_white`, `quant_min`, `quant_max`, and `quant_dither`. Float 0.0 maps to the integer value given by `quant_black`, and float 1.0 maps to the integer value given by `quant_white`. Then, for color channels only (not alpha or depth), a random amount is added in the range (`-quant_dither..quant_dither`), in order to reduce banding artifats. The result is then clamped to lie within the range of `quant_min` and `quant_max`, inclusive. Finally, this result is truncated its integer value for final output. Here is the code that implements this transformation (`T` is the final output integer type):

```
float value = quant_black * (1 - input) + quant_white * input;
if (it's a color channel)
    value += quant_dither * (2 * random() - 1);
T output = (T) clamp ((int)(value + 0.5), quant_min, quant_max);
```

The values of the quantization parameters are set in one of three ways: (1) upon construction of the `ImageSpec`, they are set to the default quantization values for the given data format; (2) upon call to `ImageSpec::set_format()`, the quantization values are set to the defaults for the given data format; (3) or, after being first set up in this manner, you may manually change the quantization parameters in the `ImageSpec`, if you want something other than the default quantization.

Default quantization for each integer type is as follows:

Data Format	black	white	min	max	dither
UINT8	0	255	0	255	0.5
INT8	0	127	-128	127	0.5
UINT16	0	65535	0	65535	0.5
INT16	0	32767	-32768	32767	0.5
UINT	0	4294967295	0	4294967295	0.5
INT	0	2147483647	-2147483648	2147483647	0.5
FLOAT HALF DOUBLE	0	1	N/A	N/A	0

Note that the default is to use the entire positive range of each integer type to represent the floating-point (0..1) range. Floating-point types do not attempt to remap values, do not add dither, and do not clamp (except to their full floating-point range).

The default will almost always be what you want. But just as an example, here's how you would specify a quantization for a 16-bit file in which 1.0 maps to 16383 (14 bits of positive range) rather than filling the full 16 bit:

```
ImageSpec spec (width, length, channels, TypeDesc::UINT16);
spec.quant_black  = 0;
spec.quant_white  = 16383;
spec.quant_min    = 0;
spec.quant_max    = 16383;
spec.quant_dither = 0.5;
```

3.2.7 Random access and repeated transmission of pixels

All `ImageOutput` implemenations that support scanlines and tiles strict order of increasing z slice, increasing y scanlines/rows within each slice, and increasing x column within each row. It is generally not safe to skip scanlines or tiles, or transmit them out of order, unless the plugin specifically advertises that it supports random access or rewrites, which may be queried using:

```
ImageOutput *out = ImageOutput::create (filename);
if (out->supports ("random_access"))
    ...
```

Similarly, you should assume the plugin will not correctly handle repeated transmissions of a scanline or tile that has already been sent, unless it advertises that it supports rewrites, which may be queried using:

```
if (out->supports ("rewrite"))
    ...
```

3.2.8 Multi-image files and MIP-maps

Some image file formats support multiple discrete images to be stored in one file. Given a created `ImageOutput`, you can query whether multiple images may be stored in the file:

```
ImageOutput *out = ImageOutput::create (filename);
if (out->supports ("multiimage"))
    ...
```

If you are working with an `ImageOutput` that supports multiple images, it is easy to write these images. All you have to do is, after writing all the pixels of one image but before calling `close()`, call `open()` again for the next image and passing `true` as the optional third *append* argument. (See Section 3.3 for the full technical description of the arguments to `open()`.) The `close()` routine is called just once, after all subimages are completed.

Below is pseudocode for writing a MIP-map (a multi-resolution image used for texture mapping) that shows how to use multi-image:

```
const char *filename = "foo.tif";
const int xres = 512, yres = 512;
const int channels = 3;  // RGB
unsigned char *pixels = new unsigned char [xres*yres*channels];

// Create the ImageOutput
ImageOutput *out = ImageOutput::create (filename);

// Be sure we can support multi-res
if (! out->supports ("multiimage")) {
    std::cerr << "Cannot write a MIP-map\n";
    delete out;
    return;
}

// Set up spec for the highest resolution
ImageSpec spec (xres, yres, channels, TypeDesc::UINT8);

// Write images, halving every time, until we're down to
// 1 pixel in either dimension
while (spec.width >= 1 && spec.height >= 1) {
    out->open (filename, spec, true /* append mode */);
    out->write_image (TypeDesc::UINT8, pixels);
    // Assume halve() resamples the image to half resolution
    halve (pixels, spec.width, spec.height);
    // Don't forget to change spec for the next iteration
    spec.width /= 2;
    spec.height /= 2;
}
out->close ();
delete out;
```

In this example, we have used `write_image()`, but of course `write_scanline()`, `write_tile()`, and `write_rectangle()` work as you would expect, on the current subimage.

3.2.9 Custom search paths for plugins

When you call `ImageOutput::create()`, the OpenImageIO library will try to find a plugin that is able to write the format implied by your filename. These plugins are alternately known

as DLL's on Windows (with the `.dll` extension), DSO's on Linux (with the `.so` extension), and dynamic libraries on Mac OS X (with the `.dylib` extension).

OpenImageIO will look for matching plugins according to *searth paths*, which are strings giving a list of directories to search, with each directory separated by a colon (':'). Within a search path, any substrings of the form `${FOO}` will be replaced by the value of environment variable `FOO`. For example, the searchpath `"${HOME}/plugins:/shared/plugins"` will first check the directory `"/home/tom/plugins"` (assuming the user's home directory is `/home/tom`), and if not found there, will then check the directory `"/shared/plugins"`.

The first search path it will check is that stored in the environment variable `IMAGEIO_LIBRARY_PATH`. It will check each directory in turn, in the order that they are listed in the variable. If no adequate plugin is found in any of the directories listed in this environment variable, then it will check the custom searchpath passed as the optional second argument to `ImageOutput::create()`, searching in the order that the directories are listed. Here is an example:

```
char *mysearch = "/usr/myapp/lib:\${HOME}/plugins";
ImageOutput *out = ImageOutput::create (filename, mysearch);
...
```

3.2.10 Error checking

Nearly every `ImageOutput` API function returns a `bool` indicating whether the operation succeeded (`true`) or failed (`false`). In the case of a failure, the `ImageOutput` will have saved an error message describing in more detail what went wrong, and the latest error messages is accessible using the `ImageOutput` method `error_message()`, which returns the message as a `std::string`.

The exception to this rule is `ImageOutput::create`, which returns `NULL` if it could not create an appropriate `ImageOutput`. And in this case, since no `ImageOutput` exists for which you can call its `error_message()` function, there exists a global `error_message()` function (in the `OpenImageIO` namespace) that retrieves the latest error message resulting from a call to `create`.

Here is another version of the simple image writing code from Section 3.1, but this time it is fully elaborated with error checking and reporting:

```
#include "imageio.h"
using namespace OpenImageIO;
...

const char *filename = "foo.jpg";
const int xres = 640, yres = 480;
const int channels = 3;  // RGB
unsigned char pixels[xres*yres*channels];

ImageOutput *out = ImageOutput::create (filename);
if (! out) {
    std::cerr << "Could not create an ImageOutput for "
              << filename << ", error = "
              << OpenImageIO::error_message() << "\n";
```

```
        return;
    }
    ImageSpec spec (xres, yres, channels, TypeDesc::UINT8);

    if (! out->open (filename, spec)) {
        std::cerr << "Could not open " << filename
                  << ", error = " << out->error_message() << "\n";
        delete out;
        return;
    }

    if (! out->write_image (TypeDesc::UINT8, pixels)) {
        std::cerr << "Could write pixels to " << filename
                  << ", error = " << out->error_message() << "\n";
        delete out;
        return;
    }

    if (! out->close ()) {
        std::cerr << "Error closing " << filename
                  << ", error = " << out->error_message() << "\n";
        delete out;
        return;
    }

    delete out;
```

3.3 `ImageOutput` Class Reference

```
static ImageOutput * create (const std::string &filename,
                             const std::string &plugin_searchpath="")
```

Create an `ImageOutput` that can be used to write an image file. The type of image file (and hence, the particular subclass of `ImageOutput` returned, and the plugin that contains its methods) is inferred from the extension of the file name. The `plugin_searchpath` parameter is a colon-separated list of directories to search for ImageIO plugin DSO/DLL's.

```
const char * format_name ()
```

Returns the canonical name of the format that this `ImageOutput` instance is capable of writing.

```
bool supports (const std::string &feature)
```

Given the name of a *feature*, tells if this `ImageOutput` instance supports that feature. The following features are recognized by this query:

"tiles" Is this plugin able to write tiled images?

"rectangles" Can this plugin accept arbitrary rectangular pixel regions (via write_rectangle())? False indicates that pixels must be transmitted via write_scanline() (if scanline-oriented) or write_tile() (if tile-oriented, and only if supports("tiles") returns true).

"random_access" May tiles or scanlines be written in any order? False indicates that they must be in successive order.

"multiimage" Does this format support multiple images within a single file?

"volumes" Does this format support "3D" pixel arrays (a.k.a. volume images)?

"rewrite" Does this plugin allow the same scanline or tile to be sent more than once? Generally this is true for plugins that implement some sort of interactive display, rather than a saved image file.

"empty" Does this plugin support passing a NULL data pointer to the various write routines to indicate that the entire data block is composed of pixels with value zero. Plugins that support this achieve a speedup when passing blank scanlines or tiles (since no actual data needs to be transmitted or converted).

This list of queries may be extended in future releases. Since this can be done simply by recognizing new query strings, and does not require any new API entry points, addition of support for new queries does not break "link compatibility" with previously-compiled plugins.

```
bool open (const std::string &name, const ImageSpec &newspec, bool append=false)
```

Open the file with given name, with resolution, and other format data as given in newspec. This function returns true for success, false for failure. Note that it is legal to call open() multiple times on the same file without a call to close(), if it supports multi-image and the append flag is true – this is interpreted as appending images (such as for MIP-maps).

```
const ImageSpec & spec ()
```

Returns the spec internally associated with this currently open ImageOutput.

```
bool close ()
```

Closes the currently open file associated with this ImageOutput and frees any memory or resources associated with it.

```
bool write_scanline (int y, int z, TypeDesc format, const void *data,
            stride_t xstride=AutoStride)
```

Write a full scanline that includes pixels $(*, y, z)$. For 2D non-volume images, z is ignored. The `xstride` value gives the distance between successive pixels (in bytes). Strides set to the special value `AutoStride` imply contiguous data, i.e.,

```
xstride = spec.nchannels*format.size()
```

This method automatically converts the data from the specified `format` to the actual output format of the file. Return `true` for success, `false` for failure. It is a failure to call `write_scanline()` with an out-of-order scanline if this format driver does not support random access.

```
bool write_tile (int x, int y, int z, TypeDesc format, const void *data,
                 stride_t xstride=AutoStride, stride_t ystride=AutoStride,
                 stride_t zstride=AutoStride)
```

Write the tile with (x, y, z) as the upper left corner. For 2D non-volume images, z is ignored. The three stride values give the distance (in bytes) between successive pixels, scanlines, and volumetric slices, respectively. Strides set to the special value `AutoStride` imply contiguous data, i.e.,

```
xstride = spec.nchannels*format.size()
ystride = xstride*spec.tile_width
zstride = ystride*spec.tile_height
```

This method automatically converts the data from the specified `format` to the actual output format of the fil. Return `true` for success, `false` for failure. It is a failure to call `write_tile()` with an out-of-order tile if this format driver does not support random access.

```
bool write_rectangle (int xmin, int xmax, int ymin, int ymax, int zmin, int zmax,
                      TypeDesc format, const void *data,
                      stride_t xstride=AutoStride, stride_t ystride=AutoStride,
                      stride_t zstride=AutoStride)
```

Write pixels whose x coords range over `xmin`...`xmax` (inclusive), y coords over `ymin`...`ymax`, and z coords over `zmin`...`zmax`. The three stride values give the distance (in bytes) between successive pixels, scanlines, and volumetric slices, respectively. Strides set to the special value `AutoStride` imply contiguous data, i.e.,

```
xstride = spec.nchannels*format.size()
ystride = xstride*(xmax-xmin+1)
zstride = ystride*(ymax-ymin+1)
```

This method automatically converts the data from the specified `format` to the actual output format of the fil. Return `true` for success, `false` for failure. It is a failure to call `write_rectangle` for a format plugin that does not return true for `supports("rectangles")`.

```
bool write_image (TypeDesc format, const void *data,
                  stride_t xstride=AutoStride, stride_t ystride=AutoStride,
                  stride_t zstride=AutoStride,
                  ProgressCallback progress_callback=NULL,
                  void *progress_callback_data=NULL)
```

Write the entire image of `spec.width` × `spec.height` × `spec.depth` pixels, with the given strides and in the desired format. Strides set to the special value `AutoStride` imply contiguous data, i.e.,

```
xstride = spec.nchannels * format.size()
ystride = xstride * spec.width
zstride = ystride * spec.height
```

The function will internally either call `write_scanline()` or `write_tile()`, depending on whether the file is scanline- or tile-oriented.

Because this may be an expensive operation, a progress callback may be passed. Periodically, it will be called as follows:

```
progress_callback (progress_callback_data, float done)
```

where *done* gives the portion of the image (between 0.0 and 1.0) that has been written thus far.

`int` **`send_to_output`** `(const char *format, ...)`

General message passing between client and image output server. This is currently undefined and is reserved for future use.

`int` **`send_to_client`** `(const char *format, ...)`

General message passing between client and image output server. This is currently undefined and is reserved for future use.

`std::string` **`error_message`** `()`

Returns the current error string describing what went wrong if any of the public methods returned `false` indicating an error. (Hopefully the implementation plugin called `error()` with a helpful error message.)

4 Image I/O: Reading Images

4.1 Image Input Made Simple

Here is the simplest sequence required to open an image file, find out its resolution, and read the pixels (converting them into 8-bit values in memory, even if that's not the way they're stored in the file):

```
#include "imageio.h"
using namespace OpenImageIO;
...

const char *filename = "foo.jpg";
int xres, yres, channels;
unsigned char *pixels;

ImageInput *in = ImageInput::create (filename);
ImageSpec spec;
in->open (filename, spec);
xres = spec.width;
yres = spec.height;
channels = spec.nchannels;
pixels = new unsigned char [xres*yres*channels];
in->read_image (TypeDesc::UINT8, pixels);
in->close ();
delete in;
```

Here is a breakdown of what work this code is doing:

- Search for an ImageIO plugin that is capable of reading the file (`"foo.jpg"`), first by trying to deduce the correct plugin from the file extension, but if that fails, by opening every ImageIO plugin it can find until one will open the file without error. When it finds the right plugin, it creates a subclass instance of `ImageInput` that reads the right kind of file format.

  ```
  ImageInput *in = ImageInput::create (filename);
  ```

- Open the file, read the header, and put all relevant metadata about the file in a specification structure.

```
ImageSpec spec;
in->open (filename, spec);
```

- The specification contains vital information such as the dimensions of the image, number of color channels, and data type of the pixel values. This is enough to allow us to allocate enough space for the image.

```
xres = spec.width;
yres = spec.height;
channels = spec.nchannels;
pixels = new unsigned char [xres*yres*channels];
```

  Note that in this example, we don't care what data format is used for the pixel data in the file — we allocate enough space for unsigned 8-bit integer pixel values, and will rely on OpenImageIO's ability to convert to our requested format from the native data format of the file.

- Read the entire image, hiding all details of the encoding of image data in the file, whether the file is scanline- or tile-based, or what is the native format of the data in the file (in this case, we request that it be automatically converted to unsigned 8-bit integers).

```
in->read_image (TypeDesc::UINT8, pixels);
```

- Close the file, destroy and free the `ImageInput` we had created, and perform all other cleanup and release of any resources used by the plugin.

```
in->close ();
delete in;
```

4.2 Advanced Image Input

Let's walk through some of the most common things you might want to do, but that are more complex than the simple example above.

4.2.1 Reading individual scanlines and tiles

The simple example of Section 4.1 read an entire image with one call. But sometimes you want to read a large image a little at a time and do not wish to retain the entire image in memory as you process it. OpenImageIO allows you to read images one scanline at a time or one tile at a time.

Reading individual scanlines

Individual scanlines may be read using the `read_scanline()` API call:

```
...
in->open (filename, spec);
unsigned char *scanline = new unsigned char [spec.width*spec.channels];
for (int y = 0;  y < yres;  ++y) {
    in->read_scanline (y, 0, TypeDesc::UINT8, scanline);
    ... process data in scanline[0..width*channels-1] ...
}
delete [] scanline;
in->close ();
...
```

The first two arguments to `read_scanline()` specify which scanline is being read by its vertical (y) scanline number (beginning with 0) and, for volume images, its slice (z) number (the slice number should be 0 for 2D non-volume images). This is followed by a `TypeDesc` describing the data type of the pixel buffer you are supplying, and a pointer to the pixel buffer itself. Additional optional arguments describe the data stride, which can be ignored for contiguous data (use of strides is explained in Section 4.2.3).

All `ImageInput` implemenations will read scanlines in strict order (starting with scanline 0, then 1, up to `yres-1`, without skipping any). However, it's probably uncommon for an `ImageInput` to properly allow reading of out-of-order scanlines.

The full description of the `read_scanline()` function may be found in Section 4.3.

Reading individual tiles

Once you `open()` an image file, you can find out if it is a tiled image (and the tile size) by examining the `ImageSpec`'s `tile_width`, `tile_height`, and `tile_depth` fields. If they are zero, it's a scanline image and you should read pixels using `read_scanline()`, not `read_tile()`.

```
...
in->open (filename, spec);
if (spec.tile_width == 0) {
    ... read by scanline ...
} else {
    // Tiles
    int tilesize = spec.tile_width * spec.tile_height;
    unsigned char *tile = new unsigned char [tilesize * spec.channels];
    for (int y = 0;  y < yres;  y += spec.tile_height) {
        for (int x = 0;  x < xres;  x += spec.tile_width) {
            in->read_tile (x, y, 0, TypeDesc::UINT8, tile);
            ... process the pixels in tile[] ..
        }
    }
    delete [] tile;
}
in->close ();
...
```

The first three arguments to `read_tile()` specify which tile is being read by the pixel coordinates of any pixel contained in the tile: x (column), y (scanline), and z (slice, which should always be 0 for 2D non-volume images). This is followed by a `TypeDesc` describing the data format of the pixel buffer you are supplying, and a pointer to the pixel buffer. Pixel data will be written to your buffer in order of increasing slice, increasing scanline within each slice, and increasing column within each scanline. Additional optional arguments describe the data stride, which can be ignored for contiguous data (use of strides is explained in Section 4.2.3).

All `ImageInput` implemenations are required to support reading tiles in arbitrary order (i.e., not in strict order of increasing y rows, and within each row, increasing x column, without missing any tiles).

The full description of the `read_tile()` function may be found in Section 4.3.

4.2.2 Converting formats

The code examples of the previous sections all assumed that your internal pixel data is stored as unsigned 8-bit integers (i.e., 0-255 range). But OpenImageIO is significantly more flexible.

You may request that the pixels be stored in any of several formats. This is done merely by passing the `read` function the data type of your pixel buffer, as one of the enumerated type `TypeDesc`.

It is not required that the pixel data buffer passed to `read_image()`, `read_scanline()`, or `read_tile()` actually be in the same data format as the data in the file being read. OpenImageIO will automatically convert from native data type of the file to the internal data format of your choice. For example. the following code will open a TIFF and read pixels into your internal buffer represented as `float` values. This will work regardless of whether the TIFF file itself is using 8-bit, 16-bit, or float values.

```
ImageInput *in = ImageInput::create ("myfile.tif");
ImageSpec spec;
in->open (filename, spec);
...
int numpixels = spec.width * spec.height;
float pixels = new float [numpixels * channels];
...
in->read_image (TypeDesc::FLOAT, pixels);
```

Note that `read_scanline()` and `read_tile()` have a parameter that works in a corresponding manner.

You can, of course, find out the native type of the file simply by examining `spec.format`. If you wish, you may then allocate a buffer big enough for an image of that type and request the native type when reading, therefore eliminating any translation among types and seeing the actual numerical values in the file.

4.2.3 Data Strides

In the preceeding examples, we have assumed that the buffer passed to the `read` functions (i.e., the place where you want your pixels to be stored) is *contiguous*, that is:

- each pixel in memory consists of a number of data values equal to the number of channels in the file;
- successive column pixels within a row directly follow each other in memory, with the first channel of pixel x of immediately following last channel of pixel $x-1$ of the same row;
- for whole images or tiles, the data for each row immediately follows the previous one in memory (the first pixel of row y immediately follows the last column of row $y-1$);
- for 3D volumetric images, the first pixel of slice z immediately follows the last pixel of of slice $z-1$.

Please note that this implies that `read_tile()` will write pixel data into your buffer so that it is contiguous in the shape of a single tile, not just an offset into a whole image worth of pixels.

The `read_scanline()` function takes an optional `xstride` argument, and the `read_image()` and `read_tile()` functions take optional `xstride`, `ystride`, and `zstride` values that describe the distance, in *bytes*, between successive pixel columns, rows, and slices, respectively, of your pixel buffer. For any of these values that are not supplied, or are given as the special constant `AutoStride`, contiguity will be assumed.

By passing different stride values, you can achieving some surprisingly flexible functionality. A few representative examples follow:

- Flip an image vertically upon reading, by using *negative y* stride:

```
unsigned char pixels[spec.width * spec.height * spec.nchannels];
int scanlinesize = spec.width * spec.nchannels * sizeof(pixels[0]);
...
in->read_image (TypeDesc::UINT8,
                (char *)pixels + (yres-1)*scanlinesize,  // offset to last
                AutoStride,                      // default x stride
                -scanlinesize,                   // special y stride
                AutoStride);                     // default z stride
```

- Read a tile into its spot in a buffer whose layout matches a whole image of pixel data, rather than having a one-tile-only memory layout:

```
unsigned char pixels[spec.width * spec.height * spec.nchannels];
int pixelsize = spec.nchannels * sizeof(pixels[0]);
int scanlinesize = xpec.width * pixelsize;
...
in->read_tile (x, y, 0, TypeDesc::UINT8,
               (char *)pixels + y*scanlinesize + x*pixelsize,
               pixelsize,
               scanlinesize);
```

Please consult Section 4.3 for detailed descriptions of the stride parameters to each `read` function.

4.2.4 Reading metadata

The `ImageSpec` that is filled in by `ImageInput::open()` specifies all the common properties that describe an image: data format, dimensions, number of channels, tiling. However, there may be a variety of additional *metadata* that are present in the image file and could be queried by your application.

The remainder of this section explains how to query additional metadata in the `ImageSpec`. It is up to the `ImageInput` to read these from the file, if indeed the file format is able to carry additional data. Individual `ImageInput` implementations should document which metadata they read.

Channel names

In addition to specifying the number of color channels, the `ImageSpec` also stores the names of those channels in its `channelnames` field, which is a `vector<std::string>`. Its length should always be equal to the number of channels (it's the responsibility of the `ImageInput` to ensure this).

Only a few file formats (and thus `ImageInput` implementations) have a way of specifying custom channel names, so most of the time you will see that the channel names follow the default convention of being named `"R"`, `"G"`, `"B"`, and `"A"`, for red, green, blue, and alpha, respectively.

Here is example code that prints the names of the channels in an image:

```
ImageInput *in = ImageInput::create (filename);
ImageSpec spec;
in->open (filename, spec);
for (int i = 0;  i < spec.nchannels;  ++i)
    std::cout << "Channel " << i << " is "
              << spec.channelnames[i] << "\n";
```

Specially-designated channels

The `ImageSpec` contains two fields, `alpha_channel` and `z_channel`, which designate which channel numbers represent alpha and z depth, if any. If either is set to `-1`, it indicates that it is not known which channel is used for that data.

If you are doing something special with alpha or depth, it is probably safer to respect the `alpha_channel` and `z_channel` designations (if not set to `-1`) rather than merely assuming that, for example, channel 3 is always the alpha channel.

Linearity hints

We certainly hope that you are using only modern file formats that support high precision and extended range pixels (such as OpenEXR) and keeping all your images in a linear color space. But you may have to work with file formats that dictate the use of nonlinear color values. This is prevalent in formats that store pixels only as 8-bit values, since 256 values are not enough to linearly represent colors without banding artifacts in the dim values.

The ImageSpec has a field that reveals what color space the image file is using. Each individual ImageInput is responsible for setting this properly.

The ImageSpec field linearity can take on any of the following values:

ImageSpec::UnknownLinearity indicates indicates it's unkown what color space the image file is using.

ImageSpec::Linear indicates that the color pixel values are known to be linear.

ImageSpec::GammaCorrected indicates that the color pixel values (but not alpha or z) in the file have already been gamma corrected (raised to the power $1/\gamma$), and that the gamma exponent may be found in the gamma field of the ImageSpec.

ImageSpec::sRGB indicates that the color pixel values in the file are in sRGB color space.

The ImageInput sets the ImageSpec::linearity field in a purely advisory capacity — the read will not convert pixel values among color spaces. Many image file formats only support nonlinear color spaces (for example, JPEG/JFIF dictates use of sRGB). So your application should intelligently deal with gamma-corrected and sRGB input.

The linearity only describes color channels. You should assume that alpha or depth (z) channels (designated by the alpha_channel and z_channel fields, respectively) always represent linear values and should never be transformed by your application.

Arbitrary metadata

All other metadata found in the file will be stored in the ImageSpec's extra_attribs field, which is a ParamValueList, which is itself essentially a vector of ParamValue instances. Each ParamValue stores one meta-datum consisting of a name, type (specified by a TypeDesc), number of values, and data pointer.

If you know the name of a specific piece of metadata you want to use, you can find it using the ImageSpec::find_attribute() method, which returns a pointer to the matching ParamValue, or NULL if no match was found. An optional TypeDesc argument can narrow the search to only parameters that match the specified type as well as the name. Below is an example that looks for orientation information, expecting it to consist of a single integer:

```
ImageInput *in = ImageInput::create (filename);
ImageSpec spec;
in->open (filename, spec);
...
ParamValue *p = spec.find_attribute ("Orientation", TypeDesc::INT);
if (p) {
    int orientation = * (int *) p->data();
} else {
    std::cout << "No integer orientation in the file\n";
}
```

By convention, ImageInput plugins will save all integer metadata as 32-bit integers (TypeDesc::INT or TypeDesc::UINT), even if the file format dictates that a particular item is stored in the file as a 8- or 16-bit integer. This is just to keep client applications from having to deal with all

the types. Since there is relatively little metadata compared to pixel data, there's no real memory waste of promoting all integer types to int32 metadata. Floating-point metadata and string metadata may also exist, of course.

It is also possible to step through all the metadata, item by item. This can be accomplished using the technique of the following example:

```
for (size_t i = 0;  i < spec.extra_attribs.size();  ++i) {
    const ParamValue &p (spec.extra_attribs[i]);
    printf ("    \%s: ", p.name.c_str());
    if (p.type() == TypeDesc::STRING)
        printf ("\"\%s\"", *(const char **)p.data());
    else if (p.type() == TypeDesc::FLOAT)
        printf ("\%g", *(const float *)p.data());
    else if (p.type() == TypeDesc::INT)
        printf ("\%d", *(const int *)p.data());
    else if (p.type() == TypeDesc::UINT)
        printf ("\%u", *(const unsigned int *)p.data());
    else
        printf ("<unknown data type>");
    printf ("\n");
}
```

Each individual `ImageInput` implementation should document the names, types, and meanings of all metadata attributes that they understand.

4.2.5 Multi-image files and MIP-maps

Some image file formats support multiple discrete images to be stored in one file. When you `open()` an `ImageOutput`, it will by default point to the first (i.e., number 0) subimage in the file. You can switch to viewing another subimage using the `seek_subimage()` function:

```
ImageInput *in = ImageOutput::create (filename);
ImageSpec spec;
in->open (filename, spec);
...
int sub = 1;
if (in->seek_subimage (sub, spec)) {
    ...
} else {
    ... no such subimage ...
}
```

The `seek_subimage()` function takes two arguments: the index of the subimage to switch to (starting with 0), and a reference to an `ImageSpec`, into which will be stored the spec of the new subimage. The `seek_subimage()` function returns `true` upon success, and `false` if no such subimage existed. It is legal to visit subimages out of order; the `ImageInput` is responsible for making it work properly. It is also possible to find out which subimage is currently being viewed, using the `current_subimage()` function, which returns the index of the current subimage.

Below is pseudocode for reading all the levels of a MIP-map (a multi-resolution image used for texture mapping) that shows how to read multi-image files:

```
ImageInput *in = ImageInput::create (filename);
ImageSpec spec;
in->open (filename, spec);

int num_subimages = 0;
while (in->seek_subimage (num_subimages, spec)) {
    // Note: spec has the format spec of the current subimage
    int npixels = spec.width * spec.height;
    int nchannels = spec.nchannels;
    unsigned char *pixels = new unsigned char [npixels * nchannels];
    in->read_image (TypeDesc::UINT8, pixels);

    ... do whatever you want with this level, in pixels ...

    delete [] pixels;
    ++num_subimages;
}
// Note: we break out of the while loop when seek_subimage fails
// to find a next subimage.

in->close ();
delete in;
```

In this example, we have used `read_image()`, but of course `read_scanline()` and `read_tile()` work as you would expect, on the current subimage.

4.2.6 Custom search paths for plugins

Please see Section 3.2.9 for discussion about search paths for finding plugins that implement `ImageOutput`.

In a similar fashion, calls to `ImageOutput::create()` will search for plugins in each directory listed in the environment variable `IMAGEIO_LIBRARY_PATH`, in the order that they are listed. If no adequate plugin is found, then it will check the custom searchpath passed as the optional second argument to `ImageInput::create()`. Here is an example:

```
char *mysearch = "/usr/myapp/lib:\${HOME}/plugins";
ImageInput *in = ImageInput::create (filename, mysearch);
...
```

4.2.7 Error checking

Nearly every `ImageInput` API function returns a `bool` indicating whether the operation succeeded (`true`) or failed (`false`). In the case of a failure, the `ImageInput` will have saved an error message describing in more detail what went wrong, and the latest error messages is accessible using the `ImageInput` method `error_message()`, which returns the message as a `std::string`.

The exception to this rule is `ImageInput::create`, which returns `NULL` if it could not create an appropriate `ImageInput`. And in this case, since no `ImageInput` exists for which you can call its `error_message()` function, there exists a global `error_message()` function (in the `OpenImageIO` namespace) that retrieves the latest error message resulting from a call to `create`.

Here is another version of the simple image reading code from Section 4.1, but this time it is fully elaborated with error checking and reporting:

```
#include "imageio.h"
using namespace OpenImageIO;
...

const char *filename = "foo.jpg";
int xres, yres, channels;
unsigned char *pixels;

ImageInput *in = ImageInput::create (filename);
if (! in) {
    std::cerr << "Could not create an ImageInput for "
              << filename << ", error = "
              << OpenImageIO::error_message() << "\n";
    return;
}

ImageSpec spec;
if (! in->open (filename, spec)) {
    std::cerr << "Could not open " << filename
              << ", error = " << in->error_message() << "\n";
    delete in;
    return;
}
xres = spec.width;
yres = spec.height;
channels = spec.nchannels;
pixels = new unsigned char [xres*yres*channels];

if (! in->read_image (TypeDesc::UINT8, pixels)) {
    std::cerr << "Could read pixels from " << filename
              << ", error = " << in->error_message() << "\n";
    delete in;
    return;
}

if (! in->close ()) {
    std::cerr << "Error closing " << filename
              << ", error = " << in->error_message() << "\n";
    delete in;
    return;
}
delete in;
```

4.3 ImageInput Class Reference

```
ImageInput * create (const std::string &filename,
                             const std::string &plugin_searchpath="")
```

Create and return an `ImageInput` implementation that is able to read the given file. The `plugin_searchpath` parameter is a colon-separated list of directories to search for ImageIO plugin DSO/DLL's (not a searchpath for the image itself!). This will actually just try every ImageIO plugin it can locate, until it finds one that's able to open the file without error. This just creates the `ImageInput`, it does not open the file.

```
const char * format_name (void) const
```

Return the name of the format implemented by this class.

```
bool open (const std::string &name, ImageSpec &newspec)
```

Opens the file with given name. Various file attributes are put in `newspec` and a copy is also saved internally to the `ImageInput` (retrievable via `spec()`. From examining `newspec` or `spec()`, you can discern the resolution, if it's tiled, number of channels, native data format, and other metadata about the image. Return `true` if the file was found and opened okay, otherwise `false`.

```
const ImageSpec & spec (void) const
```

Returns a reference to the image format specification of the current subimage. Note that the contents of the spec are invalid before `open()` or after `close()`.

```
bool close ()
```

Closes an open image.

```
int current_subimage (void) const
```

Returns the index of the subimage that is currently being read. The first subimage (or the only subimage, if there is just one) is number 0.

```
bool seek_subimage (int index, ImageSpec &newspec)
```

Seek to the given subimage. The first subimage in the file has index 0. Return `true` on success, `false` on failure (including that there is not a subimage with that index). The new subimage's vital statistics are put in `newspec` (and also saved internally in a way that can be retrieved via `spec()`). The `ImageInput` is expected to give the appearance of random access to subimages — in other words, if it can't randomly seek to the given subimage, it should transparently close, reopen, and sequentially read through prior subimages.

```
bool read_scanline (int y, int z, TypeDesc format, void *data,
                    stride_t xstride=AutoStride)
```

Read the scanline that includes pixels $(*, y, z)$ into `data`, converting if necessary from the native data format of the file into the `format` specified ($z = 0$ for non-volume images). The `xstride` value gives the data spacing of adjacent pixels (in bytes). Strides set to the special value `AutoStride` imply contiguous data, i.e.,

```
xstride = spec.nchannels*format.size()
```

The `ImageInput` is expected to give the appearance of random access — in other words, if it can't randomly seek to the given scanline, it should transparently close, reopen, and sequentially read through prior scanlines. The base `ImageInput` class has a default implementation that calls `read_native_scanline()` and then does appropriate format conversion, so there's no reason for each format plugin to override this method.

```
bool read_scanline (int y, int z, float *data)
```

This simplified version of `read_scanline()` reads to contiguous float pixels.

```
bool read_tile (int x, int y, int z, TypeDesc format, void *data,
                stride_t xstride=AutoStride, stride_t ystride=AutoStride,
                stride_t zstride=AutoStride)
```

Read the tile that includes pixels $(*, y, z)$ into `data`, converting if necessary from the native data format of the file into the `format` specified ($z = 0$ for non-volume images). The stride values give the data spacing of adjacent pixels, scanlines, and volumetric slices, respectively (measured in bytes). Strides set to the special value of `AutoStride` imply contiguous data, i.e.,

```
xstride = spec.nchannels*format.size()
ystride = xstride*spec.tile_width
zstride = ystride*spec.tile_height
```

The `ImageInput` is expected to give the appearance of random access — in other words, if it can't randomly seek to the given tile, it should transparently close, reopen, and sequentially read through prior tiles. The base `ImageInput` class has a default implementation that calls read_native_tile and then does appropriate format conversion, so there's no reason for each format plugin to override this method.

```
bool read_tile (int x, int y, int z, float *data)
```

Simple version of `read_tile` that reads to contiguous float pixels.

```
bool read_image (TypeDesc format, void *data,
                 stride_t xstride=AutoStride, stride_t ystride=AutoStride,
                 stride_t zstride=AutoStride,
                 ProgressCallback progress_callback=NULL,
                 void *progress_callback_data=NULL)
```

Read the entire image of `spec.width` × `spec.height` × `spec.depth` pixels into data (which must already be sized large enough for the entire image) with the given strides and in the desired format. Read tiles or scanlines automatically.

Strides set to the special value of `AutoStride` imply contiguous data, i.e.,

```
xstride = spec.nchannels * format.size()
ystride = xstride * spec.width
zstride = ystride * spec.height
```

The function will internally either call `read_scanline` or `read_tile`, depending on whether the file is scanline- or tile-oriented.

Because this may be an expensive operation, a progres callback may be passed. Periodically, it will be called as follows:

```
progress_callback (progress_callback_data, float done)
```

where *done* gives the portion of the image (between 0.0 and 1.0) that has been read thus far.

`bool` **`read_image`** `(float *data)`

Simple version of `read_image()` reads to contiguous float pixels.

`bool` **`read_native_scanline`** `(int y, int z, void *data)`

The `read_native_scanline()` function is just like `read_scanline()`, except that it keeps the data in the native format of the disk file and always reads into contiguous memory (no strides). It's up to the user to have enough space allocated and know what to do with the data. IT IS EXPECTED THAT EACH FORMAT PLUGIN WILL OVERRIDE THIS METHOD.

`bool` **`read_native_tile`** `(int x, int y, int z, void *data)`

The `read_native_tile()` function is just like `read_tile()`, except that it keeps the data in the native format of the disk file and always read into contiguous memory (no strides). It's up to the user to have enough space allocated and know what to do with the data. IT IS EXPECTED THAT EACH FORMAT PLUGIN WILL OVERRIDE THIS METHOD IF IT SUPPORTS TILED IMAGES.

`int` **`send_to_input`** `(const char *format, ...)`

General message passing between client and image input server. This is currently undefined and is reserved for future use.

`int` **`send_to_client`** `(const char *format, ...)`

General message passing between client and image input server. This is currently undefined and is reserved for future use.

`std::string` **`error_message`** `() const`

Returns the current error string describing what went wrong if any of the public methods returned `false` indicating an error. (Hopefully the implementation plugin called `error()` with a helpful error message.)

5 Writing ImageIO Plugins

5.1 Plugin Introduction

As explained in Chapters 4 and 3, the ImageIO library does not know how to read or write any particular image formats, but rather relies on plugins located and loaded dynamically at run-time. This set of plugins, and therefore the set of image file formats that OpenImageIO or its clients can read and write, is extensible without needing to modify OpenImageIO itself.

This chapter explains how to write your own OpenImageIO plugins. We will first explain separately how to write image file readers and writers, then tie up the loose ends of how to build the plugins themselves.

5.2 Image Readers

A plugin that reads a particular image file format must implement a *subclass* of `ImageInput` (described in Chapter 4). This is actually very straightforward and consists of the following steps, which we will illustrate with a real-world example of writing a JPEG/JFIF plug-in.

1. Read the base class definition from `imageio.h`. It may also be helpful to just use the `OpenImageIO` namespace, just to make your code a little less verbose.

   ```
   #include "imageio.h"
   using namespace OpenImageIO;
   ```

2. Declare three public items:

   (a) An integer called `imageio_version` that identifies the version of the ImageIO protocol implemented by the plugin, defined in `imageio.h` as the constant `IMAGEIO_VERSION`. This allows the library to be sure it is not loading a plugin that was compiled against a different version of OpenImageIO.

   (b) A function named *name*`_input_imageio_create` that takes no arguments and returns a new instance of your `ImageInput` subclass. (Note that *name* is the name of your format, and must match the name of the plugin itself.)

   (c) An array of `char *` called *name*`_input_extensions` that contains the list of file extensions that are likely to indicate a file of the right format. The list is terminated by a `NULL` pointer.

All of these items must be inside an 'extern "C"' block in order to avoid name mangling by the C++ compiler. Depending on your compiler, you may need to use special commands to dictate that the symbols will be exported in the DSO; we provide a special DLLEXPORT macro for this purpose, defined in export.h.

Putting this all together, we get the following for our JPEG example:

```
extern "C" {
    DLLEXPORT int imageio_version = IMAGEIO_VERSION;
    DLLEXPORT JpgInput *jpeg_input_imageio_create () {
        return new JpgInput;
    }
    DLLEXPORT const char *jpeg_input_extensions[] = {
        "jpg", "jpe", "jpeg", NULL
    };
};
```

3. The definition and implementation of an ImageInput subclass for this file format. It must publicly inherit ImageInput, and must overload the following methods which are "pure virtual" in the ImageInput base class:

   (a) format_name() should return the name of the format, which ought to match the name of the plugin and by convention is strictly lower-case and contains no whitespace.

   (b) open() should open the file and return true, or should return false if unable to do so (including if the file was found but turned out not to be in the format that your plugin is trying to implement).

   (c) close() should close the file, if open.

   (d) read_native_scanline should read a single scanline from the file into the address provided, uncompressing it but keeping it in its native data format without any translation.

   (e) The virtual destructor, which should close() if the file is still open, addition to performing any other tear-down activities.

   Additionally, your ImageInput subclass may optionally choose to overload any of the following methods, which are defined in the ImageInput base class and only need to be overloaded if the default behavior is not appropriate for your plugin:

   (f) seek_subimage(), only if your format supports reading multiple subimages within a single file.

   (g) read_native_tile(), only if your format supports reading tiled images.

   Here is how the class definition looks for our JPEG example. Note that the JPEG/JFIF file format does not support multiple subimages or tiled images.

```
class JpgInput : public ImageInput {
 public:
    JpgInput () { init(); }
    virtual ~JpgInput () { close(); }
    virtual const char * format_name (void) const { return "jpeg"; }
    virtual bool open (const char *name, ImageSpec &spec);
    virtual bool read_native_scanline (int y, int z, void *data);
    virtual bool close ();
 private:
    FILE *m_fd;
    bool m_first_scanline;
    struct jpeg_decompress_struct m_cinfo;
    struct jpeg_error_mgr m_jerr;

    void init () { m_fd = NULL; }
};
```

Your subclass implementation of `open()`, `close()`, and `read_native_scanline()` are the heart of an `ImageInput` implementation. (Also `read_native_tile()` and `seek_subimage()`, for those image formats that support them.)

The remainder of this section simply lists the full implementation of our JPEG reader, which relies heavily on the open source `jpeg-6b` library to perform the actual JPEG decoding.

```
/*
  Copyright 2008 Larry Gritz and the other authors and contributors.
  All Rights Reserved.
  Based on BSD-licensed software Copyright 2004 NVIDIA Corp.

  Redistribution and use in source and binary forms, with or without
  modification, are permitted provided that the following conditions are
  met:
  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
  * Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.
  * Neither the name of the software's owners nor the names of its
    contributors may be used to endorse or promote products derived from
    this software without specific prior written permission.
  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
```

```
  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

  (This is the Modified BSD License)
*/

#include <cassert>
#include <cstdio>

extern "C" {
#include "jpeglib.h"
}

#include "imageio.h"
using namespace OpenImageIO;
#include "fmath.h"
#include "jpeg_pvt.h"



// See JPEG library documentation in /usr/share/doc/libjpeg-devel-6b


class JpgInput : public ImageInput {
 public:
    JpgInput () { init(); }
    virtual ~JpgInput () { close(); }
    virtual const char * format_name (void) const { return "jpeg"; }
    virtual bool open (const std::string &name, ImageSpec &spec);
    virtual bool read_native_scanline (int y, int z, void *data);
    virtual bool close ();
 private:
    FILE *m_fd;
    bool m_first_scanline;
    struct jpeg_decompress_struct m_cinfo;
    struct jpeg_error_mgr m_jerr;

    void init () { m_fd = NULL; }
};



// Export version number and create function symbols
extern "C" {
    DLLEXPORT int imageio_version = IMAGEIO_VERSION;
    DLLEXPORT JpgInput *jpeg_input_imageio_create () {
        return new JpgInput;
    }
    DLLEXPORT const char *jpeg_input_extensions[] = {
        "jpg", "jpe", "jpeg", NULL
    };
};
```

```
bool
JpgInput::open (const std::string &name, ImageSpec &newspec)
{
    // Check that file exists and can be opened
    m_fd = fopen (name.c_str(), "rb");
    if (m_fd == NULL) {
        error ("Could not open file \"%s\"", name.c_str());
        return false;
    }

    // Check magic number to assure this is a JPEG file
    int magic = 0;
    fread (&magic, 4, 1, m_fd);
    rewind (m_fd);
    const int JPEG_MAGIC = 0xffd8ffe0, JPEG_MAGIC_OTHER_ENDIAN =  0xe0ffd8ff;
    const int JPEG_MAGIC2 = 0xffd8ffe1, JPEG_MAGIC2_OTHER_ENDIAN =  0xe1ffd8ff;
    if (magic != JPEG_MAGIC && magic != JPEG_MAGIC_OTHER_ENDIAN &&
        magic != JPEG_MAGIC2 && magic != JPEG_MAGIC2_OTHER_ENDIAN) {
        fclose (m_fd);
        return false;
    }

    m_cinfo.err = jpeg_std_error (&m_jerr);
    jpeg_create_decompress (&m_cinfo);          // initialize decompressor
    jpeg_stdio_src (&m_cinfo, m_fd);            // specify the data source

    // Request saving of EXIF and other special tags for later spelunking
    jpeg_save_markers (&m_cinfo, JPEG_APP0+1, 0xffff);
    // FIXME - also process JPEG_COM marker

    jpeg_read_header (&m_cinfo, FALSE);         // read the file parameters
    jpeg_start_decompress (&m_cinfo);           // start working
    m_first_scanline = true;                    // start decompressor

    m_spec = ImageSpec (m_cinfo.output_width, m_cinfo.output_height,
                        m_cinfo.output_components, TypeDesc::UINT8);

    for (jpeg_saved_marker_ptr m = m_cinfo.marker_list;  m;  m = m->next) {
        if (m->marker == (JPEG_APP0+1))
            exif_from_APP1 (m_spec, (unsigned char *)m->data);
    }

    newspec = m_spec;
    return true;
}
```

```
bool
JpgInput::read_native_scanline (int y, int z, void *data)
{
    m_first_scanline = false;
    assert (y == (int)m_cinfo.output_scanline);
    assert (y < (int)m_cinfo.output_height);
    jpeg_read_scanlines (&m_cinfo, (JSAMPLE **)&data, 1); // read one scanline
    return true;
}



bool
JpgInput::close ()
{
    if (m_fd != NULL) {
        if (!m_first_scanline)
            jpeg_finish_decompress (&m_cinfo);
        jpeg_destroy_decompress (&m_cinfo);
        fclose (m_fd);
    }
    init ();   // Reset to initial state
    return true;
}
```

5.3 Image Writers

A plugin that writes a particular image file format must implement a *subclass* of `ImageOutput` (described in Chapter 3). This is actually very straightforward and consists of the following steps, which we will illustrate with a real-world example of writing a JPEG/JFIF plug-in.

1. Read the base class definition from `imageio.h`, just as with an image reader (see Section 5.2).

2. Declare three public items:

   (a) An integer called `imageio_version` that identifies the version of the ImageIO protocol implemented by the plugin, defined in `imageio.h` as the constant `IMAGEIO_VERSION`. This allows the library to be sure it is not loading a plugin that was compiled against a different version of OpenImageIO. Note that if your plugin has both a reader and writer and they are compiled as separate modules (C++ source files), you don't want to declare this in *both* modules; either one is fine.

   (b) A function named *name*`_output_imageio_create` that takes no arguments and returns a new instance of your `ImageOutput` subclass. (Note that *name* is the name of your format, and must match the name of the plugin itself.)

(c) An array of `char *` called *name*`_output_extensions` that contains the list of file extensions that are likely to indicate a file of the right format. The list is terminated by a `NULL` pointer.

All of these items must be inside an '`extern "C"`' block in order to avoid name mangling by the C++ compiler. Depending on your compiler, you may need to use special commands to dictate that the symbols will be exported in the DSO; we provide a special `DLLEXPORT` macro for this purpose, defined in `export.h`.

Putting this all together, we get the following for our JPEG example:

```
extern "C" {
    DLLEXPORT int imageio_version = IMAGEIO_VERSION;
    DLLEXPORT JpgOutput *jpeg_output_imageio_create () {
        return new JpgOutput;
    }
    DLLEXPORT const char *jpeg_input_extensions[] = {
        "jpg", "jpe", "jpeg", NULL
    };
};
```

3. The definition and implementation of an `ImageOutput` subclass for this file format. It must publicly inherit `ImageOutput`, and must overload the following methods which are "pure virtual" in the `ImageOutput` base class:

   (a) `format_name()` should return the name of the format, which ought to match the name of the plugin and by convention is strictly lower-case and contains no whitespace.

   (b) `supports()` should return `true` if its argument names a feature supported by your format plugin, `false` if it names a feature not supported by your plugin. See Section 3.3 for the list of feature names.

   (c) `open()` should open the file and return true, or should return false if unable to do so (including if the file was found but turned out not to be in the format that your plugin is trying to implement).

   (d) `close()` should close the file, if open.

   (e) `write_scanline` should write a single scanline to the file, translating from internal to native data format and handling strides properly.

   (f) The virtual destructor, which should `close()` if the file is still open, addition to performing any other tear-down activities.

   Additionally, your `ImageOutput` subclass may optionally choose to overload any of the following methods, which are defined in the `ImageOutput` base class and only need to be overloaded if the default behavior is not appropriate for your plugin:

   (g) `write_tile()`, only if your format supports writing tiled images.

(h) `write_rectangle()`, only if your format supports writing arbitrary rectangles.

(i) `write_image()`, only if you have a more clever method of doing so than the default implementation that calls `write_scanline()` or `write_tile()` repeatedly.

Here is how the class definition looks for our JPEG example. Note that the JPEG/JFIF file format does not support multiple subimages or tiled images.

```
class JpgOutput : public ImageOutput {
 public:
    JpgOutput () { init(); }
    virtual ~JpgOutput () { close(); }
    virtual const char * format_name (void) const { return "jpeg"; }
    virtual bool supports (const char *property) const { return false; }
    virtual bool open (const char *name, const ImageSpec &spec,
                       bool append=false);
    virtual bool write_scanline (int y, int z, TypeDesc format,
                                 const void *data, stride_t xstride);
    bool close ();
 private:
    FILE *m_fd;
    std::vector<unsigned char> m_scratch;
    struct jpeg_compress_struct m_cinfo;
    struct jpeg_error_mgr m_jerr;

    void init () { m_fd = NULL; }
};
```

Your subclass implementation of `open()`, `close()`, and `write_scanline()` are the heart of an `ImageOutput` implementation. (Also `write_tile()`, for those image formats that support tiled output.)

An `ImageOutput` implementation must properly handle all data formats and strides passed to `write_scanline()` or `write_tile()`, unlike an `ImageInput` implementation, which only needs to read scanlines or tiles in their native format and then have the super-class handle the translation. But don't worry, all the heavy lifting can be accomplished with the following helper functions provided as protected member functions of `ImageOutput`:

```
const void * to_native_scanline (TypeDesc format, const void *data,
             stride_t xstride, std::vector<unsigned char> &scratch);
```

Convert a full scanline of pixels (pointed to by *data*) with the given *format* and strides into contiguous pixels in the native format (described by the `ImageSpec` returned by the `spec()` member function). The location of the newly converted data is returned, which may either be the original *data* itself if no data conversion was necessary and the requested layout was contiguous (thereby avoiding unnecessary memory copies), or may point into memory allocated within the *scratch* vector passed by the user. In either case, the caller doesn't need to worry about thread safety or freeing any allocated memory (other than eventually destroying the scratch vector).

```
const void * to_native_tile (TypeDesc format, const void *data,
             stride_t xstride, stride_t ystride, stride_t zstride,
             std::vector<unsigned char> &scratch);
```

Convert a full tile of pixels (pointed to by *data*) with the given *format* and strides into contiguous pixels in the native format (described by the `ImageSpec` returned by the `spec()` member function). The location of the newly converted data is returned, which may either be the original *data* itself if no data conversion was necessary and the requested layout was contiguous (thereby avoiding unnecessary memory copies), or may point into memory allocated within the *scratch* vector passed by the user. In either case, the caller doesn't need to worry about thread safety or freeing any allocated memory (other than eventually destroying the scratch vector).

```
const void * to_native_rectangle (int xmin, int xmax, int ymin, int ymax,
             int zmin, int zmax, TypeDesc format, const void *data,
             stride_t xstride, stride_t ystride, stride_t zstride,
             std::vector<unsigned char> &scratch);
```

Convert a rectangle of pixels (pointed to by *data*) with the given *format*, dimensions, and strides into contiguous pixels in the native format (described by the `ImageSpec` returned by the `spec()` member function). The location of the newly converted data is returned, which may either be the original *data* itself if no data conversion was necessary and the requested layout was contiguous (thereby avoiding unnecessary memory copies), or may point into memory allocated within the *scratch* vector passed by the user. In either case, the caller doesn't need to worry about thread safety or freeing any allocated memory (other than eventually destroying the scratch vector).

The remainder of this section simply lists the full implementation of our JPEG writer, which relies heavily on the open source `jpeg-6b` library to perform the actual JPEG encoding.

```
/*
  Copyright 2008 Larry Gritz and the other authors and contributors.
  All Rights Reserved.
  Based on BSD-licensed software Copyright 2004 NVIDIA Corp.

  Redistribution and use in source and binary forms, with or without
  modification, are permitted provided that the following conditions are
  met:
  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
  * Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.
  * Neither the name of the software's owners nor the names of its
    contributors may be used to endorse or promote products derived from
    this software without specific prior written permission.
  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
```

```
  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

  (This is the Modified BSD License)
*/


#include <cassert>
#include <cstdio>
#include <vector>

extern "C" {
#include "jpeglib.h"
}

#include "imageio.h"
using namespace OpenImageIO;
#include "fmath.h"
#include "jpeg_pvt.h"



// See JPEG library documentation in /usr/share/doc/libjpeg-devel-6b


class JpgOutput : public ImageOutput {
 public:
    JpgOutput () { init(); }
    virtual ~JpgOutput () { close(); }
    virtual const char * format_name (void) const { return "jpeg"; }
    virtual bool supports (const std::string &property) const { return false; }
    virtual bool open (const std::string &name, const ImageSpec &spec,
                       bool append=false);
    virtual bool write_scanline (int y, int z, TypeDesc format,
                                 const void *data, stride_t xstride);
    bool close ();
 private:
    FILE *m_fd;
    std::vector<unsigned char> m_scratch;
    struct jpeg_compress_struct m_cinfo;
    struct jpeg_error_mgr c_jerr;

    void init (void) { m_fd = NULL; }
```

```
};



extern "C" {
    DLLEXPORT JpgOutput *jpeg_output_imageio_create () {
        return new JpgOutput;
    }
    DLLEXPORT const char *jpeg_output_extensions[] = {
        "jpg", "jpe", "jpeg", NULL
    };
};




bool
JpgOutput::open (const std::string &name, const ImageSpec &newspec,
                 bool append)
{
    if (append) {
        error ("JPG doesn't support multiple images per file");
        return false;
    }

    // Save spec for later use
    m_spec = newspec;

    // Check for things this format doesn't support
    if (m_spec.width < 1 || m_spec.height < 1) {
        error ("Image resolution must be at least 1x1, you asked for %d x %d",
               m_spec.width, m_spec.height);
        return false;
    }
    if (m_spec.depth < 1)
        m_spec.depth = 1;
    if (m_spec.depth > 1) {
        error ("%s does not support volume images (depth > 1)", format_name());
        return false;
    }

    m_fd = fopen (name.c_str(), "wb");
    if (m_fd == NULL) {
        error ("Unable to open file \"%s\"", name.c_str());
        return false;
    }

    int quality = 98;
    // FIXME -- see if there's a quality set in the attributes

    m_cinfo.err = jpeg_std_error (&c_jerr);             // set error handler
```

```
    jpeg_create_compress (&m_cinfo);                    // create compressor
    jpeg_stdio_dest (&m_cinfo, m_fd);                   // set output stream

    // Set image and compression parameters
    m_cinfo.image_width = m_spec.width;
    m_cinfo.image_height = m_spec.height;

    if (m_spec.nchannels == 3 || m_spec.nchannels == 4) {
        m_cinfo.input_components = 3;
        m_cinfo.in_color_space = JCS_RGB;
        m_spec.nchannels = 3;  // Force RGBA -> RGB
        m_spec.alpha_channel = -1;  // No alpha channel
    } else if (m_spec.nchannels == 1) {
        m_cinfo.input_components = 1;
        m_cinfo.in_color_space = JCS_GRAYSCALE;
    }
    m_cinfo.density_unit = 2; // RESUNIT_INCH;
    m_cinfo.X_density = 72;
    m_cinfo.Y_density = 72;
    m_cinfo.write_JFIF_header = true;

    jpeg_set_defaults (&m_cinfo);                       // default compression
    jpeg_set_quality (&m_cinfo, quality, TRUE);         // baseline values
    jpeg_start_compress (&m_cinfo, TRUE);               // start working

    std::vector<char> exif;
    APP1_exif_from_spec (m_spec, exif);
    if (exif.size())
        jpeg_write_marker (&m_cinfo, JPEG_APP0+1, (JOCTET*)&exif[0], exif.size());

//    jpeg_write_marker (&m_cinfo, JPEG_COM, comment, strlen(comment) /* + 1 ? */ );

    m_spec.set_format (TypeDesc::UINT8);  // JPG is only 8 bit

    return true;
}



bool
JpgOutput::write_scanline (int y, int z, TypeDesc format,
                           const void *data, stride_t xstride)
{
    y -= m_spec.y;
    assert (y == (int)m_cinfo.next_scanline);
    assert (y < (int)m_cinfo.image_height);

    data = to_native_scanline (format, data, xstride, m_scratch);

    jpeg_write_scanlines (&m_cinfo, (JSAMPLE**)&data, 1);
```

```
    return true;
}



bool
JpgOutput::close ()
{
    if (! m_fd)          // Already closed
        return true;
    jpeg_finish_compress (&m_cinfo);
    jpeg_destroy_compress (&m_cinfo);
    fclose (m_fd);
    init();

    return true;
}
```

5.4 Building ImageIO Plugins

FIXME – spell out how to compile and link plugins on each of the major platforms.

6 Bundled ImageIO Plugins

6.1 TIFF

6.2 JPEG

6.3 OpenEXR

6.4 HDR/RGBE

6.5 PNG

7 Image Buffer

8 Texture Access: `TextureSystem`

8.1 Texture System Introduction and Theory of Operation

Coming soon.

8.2 Helper Classes

8.2.1 Imath

The texture functinality of OpenImageIO uses the excellent open source `Ilmbase` package's `Imath` types when it requires 3D vectors and transformation matrixes. Specifically, we use `Imath::V3f` for 3D positions and directions, and `Imath::M44f` for 4×4 transformation matrices. To use these yourself, we recommend that you:

```
#include <ImathVec.h>
#include <ImathMatrix.h>
```

Please refer to the `Ilmbase` and `OpenEXR` documentation and header files for more complete information about use of these types in your own application. However, note that you are not strictly required to use these classes in your application — `Imath::V3f` has a memory layout identical to `float[3]` and `Imath::M44f` has a memory layout identical to `float[16]`, so as long as your own internal vectors and matrices have the same memory layout, it's ok to just cast pointers to them when passing as arguments to `TextureSystem` methods.

8.2.2 `VaryingRef`: encapsulate uniform and varying

All of the texture access API routines are designed to loook up texture efficiently at many points at once. Therefore, many of the parameters to the API routines, and many of the fields in `TextureOptions` need to accommodate both uniform and varying values. *Uniform* means that a single value may be used for each of the many simultaneous texture lookups, whereas *varying* means that a different value is provided for each of the positions where you are sampling the texture.

Please read the comments in `"varyingref.h"` for the full gory details, but here's all you really need to know about it to use the texture functionality. Let's suppose that we have a routine whose prototype looks like this:

```
void API (int n, VaryingRef<float> x);
```

This means that parameter *x* may either be a single value for use at each of the *n* texture lookups, or it may have *n* different values of *x*.

If you want to pas a uniform value, you may do any of the following:

```
float x;   // just one value
API (n, x);   // automatically knows what to do!
API (n, &x);  // Also ok to pass the pointer to x
API (n, VaryingRef<float>(x));  // Wordy but correct
API (n, Uniform(x));  // Shorthand
```

If you want to pass a varying value, i.e., an array of values,

```
float x[n];   // One value for each of n points
API (n, VaryingRef<float>(x), sizeof(x));  // Wordy but correct
API (n, Varying(x));  // Shorthand if stride is sizeof(x)
```

You can also initialize a VaryingRef directly:

```
float x;     // just one value
float y[n];  // array of values
VaryingRef<float> r;
r.init (&x);                 // Initialize to uniform
r.init (&x, 0);              // Initialize to uniform the wordy way
r.init (&y, sizeof(float));  // Initialize to varying
...
API (n, r);
```

8.2.3 TextureOptions

`TextureOptions` is a structure that holds many options controlling individual texture lookups. Because each texture lookup API call takes a reference to a `TextureOptions`, the call signatures remain uncluttered rather than having an ever-growing list of parameters, most of which will never vary from their defaults. Here is a brief description of the data members of a `TextureOptions` structure:

`int nchannels`
`int firstchannel`

The number of color channels to look up from the texture — for example, 1 (single channel), or 3 (for an RGB triple) — and the number of channels to look up. The defaults are firstchannel = 0, nchannels = 1.

Examples: To retrieve the first three channels (typically RGB), you should have nchannels = 3, firstchannel = 0. To retrieve just the blue channel, you should have nchannels = 1, firstchannel = 2.

`Wrap swrap, twrap`

Specify the *wrap mode* for 2D texture lookups (and 3D volume texture lookups, using the additional `zwrap` field). These fields are ignored for shadow and environment lookups.

These specify what happens when texture coordinates are found to be outside the usual $[0,1]$ range over which the texture is defined. `Wrap` is an enumerated type that may take on any of the following values:

`WrapBlack` The texture is black outside the [0,1] range.

`WrapClamp` The texture coordinates will be clamped to [0,1], i.e., the value outside [0,1] will be the same as the color at the nearest point on the border.

`WrapPeriodic` The texture is periodic, i.e., wraps back to 0 after going past 1.

`WrapMirror` The texture presents a mirror image at the edges, i.e., the coordinates go from 0 to 1, then back down to 0, then back up to 1, etc.

`WrapDefault` Use whatever wrap might be specified in the texture file itself, or some other suitable default (caveat emptor).

The wrap mode does not need to be identical in the *s* and *t* directions.

`VaryingRef<float> swidth, twidth`

For each direction, gives a multiplier for the derivatives. Note that a width of 0 indicates a point sampled lookup (assuming that blur is also zero). The default width is 1, indicating that the derivatives should guide the amount of blur applied to the texture filtering (not counting any additional *blur* specified).

`VaryingRef<float> sblur, tblur`

For each direction, specifies an additional amount of pre-blur to apply to the texture (*after* derivatives are taken into account), expressed as a portion of the width of the texture. In other words, blur = 0.1 means that the texture lookup should act as if the texture was pre-blurred with a filter kernel with a width 1/10 the size of the full image. The default blur amount is 0, indicating a sharp texture lookup.

`VaryingRef<float> bias`

For shadow map lookups only, this gives the "shadow bias" amount.

`VaryingRef<float> fill`

Specifies the value that will be used for any color channels that are requested but not found in the file. For example, if you perform a 3-channel lookup on a 1-channel texture, the second two channels will get the fill value.

`VaryingRef<int> samples`

The number of samples to use for each lookup. Currently this only applies for certain types of shadow maps.

`VaryingRef<float> alpha`

Specifies a *destination* for one additional channel to be looked up, the one immediately following the return value (i.e., channel *firstchannel* + *nchannels*). The point of this is to allow a 4-channel lookup, with the 4th channel put in an entirely different variable than the 3-channel color. The default for `alpha` is to point to NULL, indicating that no extra alpha channel should be retrieved.

```
Wrap zwrap
VaryingRef<float> zblur, zwidth
```

Specifies wrap, blur, and width for 3D volume texture lookups only.

8.2.4 SIMD Run Flags

Most of the texture lookup API routines are written to accommodate queries about many points at once. Furthermore, only a subset of points may need to compute. This is all expressed using three parameters: `Runflag *runflags, int firstactive, int lastactive`. There are also `VaryingRef` parameters such as `s` and `t` that act as if they are arrays.

The `firstactive` and `lastactive` indices are the first and last (inclusive) points that should be computed, and for each point `runflags[i]` is nonzero if the point should be computed. To illustrate, here is how a routine might be written that would copy values in `arg` to `result` using runflags:

```
void copy (Runflag *runflags, int firstactive, int lastactive,
           VaryingRef<float> arg, VaryingRef <float> result)
{
    for (int i = firstactive; i <= lastactive;  ++i)
        if (runflags[i])
            result[i] = arg[i];
}
```

8.3 TextureSystem API

8.3.1 Creating and destroying texture systems

`TextureSystem` is an abstract API described as a pure virtual class. The actual internal implementation is not exposed through the external API of OpenImageIO. Because of this, you cannot construct or destroy the concrete implementation, so two static methods of `TextureSystem` are provided:

`static TextureSystem *TextureSystem::`**`create`**`()`

Creates a new `TextureSystem` and returns a pointer to it.

`static void TextureSystem::`**`destroy`** `(TextureSystem * &x)`

Destroys an allocated `TextureSystem`, including freeing all system resources that it holds.

This is necessary to ensure that the memory is freed in a way that matches the way it was allocated within the library. Note that simply using `delete` on the pointer will not always work (at least, not on some platforms in which a DSO/DLL can end up using a different allocator than the main program).

8.3.2 Setting options and limits for the texture system

The following member functions of `TextureSystem` allow you to set (and in some cases retrieve) options that control the overall behavior of the texture system:

`void` **`max_open_files`** `(int nfiles)`

Sets the maximum number of file handles that the texture system will hold open simultaneously. (Default = 100)

`void` **`max_memory_MB`** `(float size)`

Sets the maximum amount of memory (measured in MB) that the texture system will use for its "tile cache." (Default: 50 MB)

`void` **`searchpath`** `(const std::string &path)`

Sets the search path for textures: a colon-separated list of directories that will be searched in order for any texture name that is not specified as an absolute path.

`void` **`worldtocommon`** `(const float *mx)`
`void` **`worldtocommon`** `(const Imath::M44f &w2c)`

Sets the 4×4 matrix that provides the spatial transformation from "world" to a "common" coordinate system. This is used for shadow map lookups, in which the shadow map itself

encodes the world coordinate system, but positions passed to `shadow()` are expressed in "common" coordinates.

There are two versions of this method: one takes a reference to a `Imath:M44f`, the other that takes a pointer to 16 contiguous floats.

`int` **`max_open_files`** `() const`

Returns the current value of the maximum number of file handles that the texture library will hold open.

`float` **`max_memory_MB`** `() const`

Returns the current value of the maximum amount of memory (in MB) the texture library will use for the tile cache.

`std::string` **`searchpath`** `() const`

Returns the current search path for textures.

8.3.3 Texture Lookups

```
bool texture (ustring filename, TextureOptions &options,
              float s, float t, float dsdx, float dtdx,
              float dsdy, float dtdy, float *result)
```

Perform a filtered 2D texture lookup on a position centered at 2D coordinates (`s`, `t`) from the texture identified by `filename`, and using relevant texture `options`. The filtered results will be stored in `result[]`.

We assume that this lookup will be part of an image that has pixel coordinates `x` and `y`. By knowing how `s` and `t` change from pixel to pixel in the final image, we can properly *filter* or antialias the texture lookups. This information is given via derivatives `dsdx` and `dtdx` that define the change in `s` and `t` per unit of `x`, and `dsdy` and `dtdy` that define the change in `s` and `t` per unit of `y`. If it is impossible to know the derivatives, you may pass 0 for them, but in that case you will not receive an antialiased texture lookup.

Fields within `options` that are honored for 2D texture lookups include the following:

`int nchannels`

The number of color channels to look up from the texture.

`int firstchannel`

The index of the first channel to look up from the texture.

`Wrap swrap, twrap`

Specify the *wrap mode* for each direction, one of: `WrapBlack`, `WrapClamp`, `WrapPeriodic`, `WrapMirror`, or `WrapDefault`.

`VaryingRef<float> swidth, twidth`

For each direction, gives a multiplier for the derivatives.

`VaryingRef<float> sblur, tblur`

For each direction, specifies an additional amount of pre-blur to apply to the texture (*after* derivatives are taken into account), expressed as a portion of the width of the texture.

`VaryingRef<float> fill`

Specifies the value that will be used for any color channels that are requested but not found in the file.

`VaryingRef<float> alpha`

Specifies a *destination* for one additional channel to be looked up, the one immediately following the return value (i.e., channel *firstchannel* + *nchannels*). The point of this is to allow a 4-channel lookup, with the 4th channel put in an entirely different variable than the 3-channel color. The default for `alpha` is to point to NULL, indicating that no extra alpha channel should be retrieved.

This function returns `true` upon success, or `false` if the file was not found or could not be opened by any available ImageIO plugin.

```
bool texture (ustring filename, TextureOptions &options,
              Runflag *runflags, int firstactive, int lastactive,
              VaryingRef<float> s, VaryingRef<float> t,
              VaryingRef<float> dsdx, VaryingRef<float> dtdx,
              VaryingRef<float> dsdy, VaryingRef<float> dtdy,
              float *result)
```

Perform filtered 2D texture lookups on a collection of positions all at once, which may be much more efficient than repeatedly calling the single-point version of `texture()`. The parameters `s`, `t`, `dsdx`, `dtdx`, and `dsdy`, `dtdy` are now `VaryingRef`'s that may refer to either a single or an array of values, as are all the fields in the `options`.

Texture will be computed at indices `firstactive` through `lastactive`, inclusive, but only at indices where `runflags[i]` is nonzero. Results will be stored at corresponding positions of `result`, that is, `result[i*n ... (i+1)*n-1]` where *n* is the number of channels requested by `options.nchannels`.

This function returns `true` upon success, or `false` if the file was not found or could not be opened by any available ImageIO plugin.

8.3.4 Volume Texture Lookups

```
bool texture (ustring filename, TextureOptions &options,
              const Imath::V3f &P,
              const Imath::V3f &dPdx,
              const Imath::V3f &dPdy,
              float *result)
```

Perform a filtered 3D volumetric texture lookup on a position centered at 3D position `P` from the texture identified by `filename`, and using relevant texture `options`. The filtered results will be stored in `result[]`.

We assume that this lookup will be part of an image that has pixel coordinates `x` and `y`. By knowing how `P` changes from pixel to pixel in the final image, we can properly *filter* or antialias the texture lookups. This information is given via derivatives `dPdx` and `dPdy` that define the changes in `P` per unit of `x` and `y`, respectively. If it is impossible to know the derivatives, you may pass 0 for them, but in that case you will not receive an antialiased texture lookup.

Fields within `options` that are honored for 3D texture lookups include the following:

`int nchannels`

The number of color channels to look up from the texture.

`int firstchannel`

The index of the first channel to look up from the texture.

`Wrap swrap, twrap, zwrap`

Specify the wrap modes for each direction, one of: `WrapBlack`, `WrapClamp`, `WrapPeriodic`, `WrapMirror`, or `WrapDefault`.

`VaryingRef<float> swidth, twidth, zwidth`

For each direction, gives a multiplier for the derivatives.

`VaryingRef<float> sblur, tblur, zblur`

For each direction, specifies an additional amount of pre-blur to apply to the texture (*after* derivatives are taken into account), expressed as a portion of the width of the texture.

`VaryingRef<float> fill`

Specifies the value that will be used for any color channels that are requested but not found in the file.

`VaryingRef<float> alpha`

Specifies a *destination* for one additional channel to be looked up, the one immediately following the return value (i.e., channel *firstchannel* + *nchannels*). The point of this is to allow a 4-channel lookup, with the 4th channel put in an entirely different variable than the 3-channel color. The default for `alpha` is to point to NULL, indicating that no extra alpha channel should be retrieved.

This function returns `true` upon success, or `false` if the file was not found or could not be opened by any available ImageIO plugin.

```
bool texture (ustring filename, TextureOptions &options,
              Runflag *runflags, int firstactive, int lastactive,
              VaryingRef<Imath::V3f> P,
              VaryingRef<Imath::V3f> dPdx,
              VaryingRef<Imath::V3f> dPdy,
              float *result)
```

Perform filtered 3D volumetric texture lookups on a collection of positions all at once, which may be much more efficient than repeatedly calling the single-point version of `texture()`. The parameters `P`, `dPdx`, and `dPdy` are now `VaryingRef`'s that may refer to either a single or an array of values, as are all the fields in the `options`.

Texture will be computed at indices `firstactive` through `lastactive`, inclusive, but only at indices where `runflags[i]` is nonzero. Results will be stored at corresponding positions of `result`, that is, `result[i*n ... (i+1)*n-1]` where *n* is the number of channels requested by `options.nchannels`.

This function returns `true` upon success, or `false` if the file was not found or could not be opened by any available ImageIO plugin.

8.3.5 Shadow Lookups

```
bool shadow (ustring filename, TextureOptions &options,
             const Imath::V3f &P,
             const Imath::V3f &dPdx,
             const Imath::V3f &dPdy,
             float *result)
```

Perform a shadow map lookup on a position centered at 3D coordinate `P` (in a designated "common" space) from the shadow map identified by `filename`, and using relevant texture `options`. The filtered results will be stored in `result[]`.

We assume that this lookup will be part of an image that has pixel coordinates `x` and `y`. By knowing how `P` changes from pixel to pixel in the final image, we can properly *filter* or antialias the texture lookups. This information is given via derivatives `dPdx` and `dPdy` that define the changes in `P` per unit of `x` and `y`, respectively. If it is impossible to know the derivatives, you may pass 0 for them, but in that case you will not receive an antialiased texture lookup.

Fields within `options` that are honored for 2D texture lookups include the following:

`VaryingRef<float> swidth, twidth`

For each direction, gives a multiplier for the derivatives.

`VaryingRef<float> sblur, tblur`

For each direction, specifies an additional amount of pre-blur to apply to the texture (*after* derivatives are taken into account), expressed as a portion of the width of the texture.

`VaryingRef<float> bias`

Specifies the amount of *shadow bias* to use — this effectively ignores shadow occlusion that is closer than the bias amount to the surface, helping to eliminate self-shadowing artifacts.

`VaryingRef<int> samples`

Specifies the number of samples to use when evaluating the shadow map. More samples will give a smoother, less noisy, appearance to the shadows, but may also take longer to compute.

This function returns `true` upon success, or `false` if the file was not found or could not be opened by any available ImageIO plugin.

```
bool shadow (ustring filename, TextureOptions &options,
             Runflag *runflags, int firstactive, int lastactive,
             VaryingRef<Imath::V3f> P,
             VaryingRef<Imath::V3f> dPdx,
             VaryingRef<Imath::V3f> dPdy,
             float *result)
```

Perform filtered shadow map lookups on a collection of positions all at once, which may be much more efficient than repeatedly calling the single-point version of `shadow()`. The parameters `P`, `dPdx`, and `dPdy` are now `VaryingRef`'s that may refer to either a single or an array of values, as are all the fields in the `options`.

Shadow lookups will be computed at indices `firstactive` through `lastactive`, inclusive, but only at indices where `runflags[i]` is nonzero. Results will be stored at corresponding positions of `result`, that is, `result[i*n ... (i+1)*n-1]` where *n* is the number of channels requested by `options.nchannels`.

This function returns `true` upon success, or `false` if the file was not found or could not be opened by any available ImageIO plugin.

8.3.6 Environment Lookups

```
bool environment (ustring filename, TextureOptions &options,
                  const Imath::V3f &R,
                  const Imath::V3f &dRdx,
                  const Imath::V3f &dRdy,
                  float *result)
```

Perform a filtered directional environment map lookup in the direction of vector `R`, from the texture identified by `filename`, and using relevant texture `options`. The filtered results will be stored in `result[]`.

We assume that this lookup will be part of an image that has pixel coordinates `x` and `y`. By knowing how `R` changes from pixel to pixel in the final image, we can properly *filter* or antialias the texture lookups. This information is given via derivatives `dRdx` and `dRdy` that define the changes in `R` per unit of `x` and `y`, respectively. If it is impossible to know the derivatives, you may pass 0 for them, but in that case you will not receive an antialiased texture lookup.

Fields within `options` that are honored for 3D texture lookups include the following:

`int nchannels`

The number of color channels to look up from the texture.

`int firstchannel`

The index of the first channel to look up from the texture.

`VaryingRef<float> swidth, twidth`

For each direction, gives a multiplier for the derivatives.

`VaryingRef<float> sblur, tblur`

For each direction, specifies an additional amount of pre-blur to apply to the texture (*after* derivatives are taken into account), expressed as a portion of the width of the texture.

`VaryingRef<float> fill`

Specifies the value that will be used for any color channels that are requested but not found in the file.

`VaryingRef<float> alpha`

Specifies a *destination* for one additional channel to be looked up, the one immediately following the return value (i.e., channel *firstchannel* + *nchannels*). The point of this is to allow a 4-channel lookup, with the 4th channel put in an entirely different variable than the 3-channel color. The default for `alpha` is to point to NULL, indicating that no extra alpha channel should be retrieved.

This function returns `true` upon success, or `false` if the file was not found or could not be opened by any available ImageIO plugin.

```
bool environment (ustring filename, TextureOptions &options,
            Runflag *runflags, int firstactive, int lastactive,
            VaryingRef<Imath::V3f> R,
            VaryingRef<Imath::V3f> dRdx,
            VaryingRef<Imath::V3f> dRdy,
            float *result)
```

Perform filtered directional environment map lookups on a collection of directions all at once, which may be much more efficient than repeatedly calling the single-point version of `environment()`. The parameters `R`, `dRdx`, and `dRdy` are now `VaryingRef`'s that may refer to either a single or an array of values, as are all the fields in the `options`.

Results will be computed at indices `firstactive` through `lastactive`, inclusive, but only at indices where `runflags[i]` is nonzero. Results will be stored at corresponding positions of `result`, that is, `result[i*n ... (i+1)*n-1]` where n is the number of channels requested by `options.nchannels`.

This function returns `true` upon success, or `false` if the file was not found or could not be opened by any available ImageIO plugin.

8.3.7 Texture Metadata, Raw Texels, and Errors

```
bool get_texture_info (ustring filename, ustring dataname,
            TypeDesc datatype, void *data)
```

Retrieves information about the texture named by `filename`. The `dataname` is a keyword indcating what information should be retrieved, `datatype` is the type of data expected, and `data` points to caller-owned memory where the results should be placed. It is up to the caller to ensure that `data` contains enough space to hold an item of the requested `datatype`.

The return value is `true` if `get_texture_info()` is able to find the requested `dataname` and it matched the requested `datatype`. If the requested data was not found, or was not of the right data type, `get_texture_info()` will return `false`.

Supported `dataname` values include:

`resolution` The resolution of the texture file, which is an array of 2 integers (described as `TypeDesc(INT,2)`).

`texturetype` A string describing the type of texture of the given file, which describes how the texture may be used (also which texture API call is probably the right one for it). This currently may return one of: `"unknown"`, `"Plain Texture"`, `"Volume Texture"`, `"Shadow"`, or `"Environment"`.

`textureformat` A string describing the format of the given file, which describes the kind of texture stored in the file. This currently may return one of: `"unknown"`, `"Plain Texture"`, `"Volume Texture"`, `"Shadow"`, `"CubeFace Shadow"`, `"Volume Shadow"`, `"LatLong Environment"`, or `"CubeFace Environment"`. Note that there are several kinds of shadows and environment maps, all accessible through the same API calls.

`channels` The number of color channels in the file (an integer).

`viewingmatrix` The viewing matrix, which is a 4×4 matrix (an `Imath::M44f`, described as `TypeDesc(FLOAT,MATRIX)`).

`projectionmatrix` The projection matrix, which is a 4×4 matrix (an `Imath::M44f`, described as `TypeDesc(FLOAT,MATRIX)`).

Anything else – For all other data names, the the metadata of the image file will be searched for an item that matches both the name and data type.

```
bool get_imagespec (ustring filename, ImageSpec &spec)
```

If the named image is found and able to be opened by an available ImageIO plugin, this function copies its image specification into `spec` and returns `true`. Otherwise, if the file is not found, could not be opened, or is not of a format readable by any ImageIO plugin that could be find, the return value is `false`.

```
bool get_texels (ustring filename, TextureOptions &options,
                 int xmin, int xmax, int ymin, int ymax,
                 int zmin, int zmax, int level,
                 TypeDesc format, void *result)
```

Retrieve the rectangle of raw unfiltered texels spanning (xmin, ymin, zmin) through (xmax, ymax, zmax) (inclusive, specified as integer pixel coordinates), at the named MIP-map level, storing the texel values beginning at the address specified by result. The texel values will be converted to the type specified by format. It is up to the caller to ensure that result points to an area of memory big enough to accommodate the requested rectangle (taking into consideration its dimensions, number of channels, and data format).

Fields within `options` that are honored for raw texel retieval include the following:

`int nchannels`

The number of color channels to look up from the texture.

`int firstchannel`

The index of the first channel to look up from the texture.

`VaryingRef<float> fill`

Specifies the value that will be used for any color channels that are requested but not found in the file.

Return true if the file is found and could be opened by an available ImageIO plugin, otherwise return false.

```
std::string geterror ()
```

If any other API routines return `false`, indicating that an error has occurred, this routine will retrieve the error and clear the error status. If no error has occurred since the last time `geterror()` was called, it will return an empty string.

Part II

Image Utilities

9 The `iv` Image Viewer

The `iv` program is a great interactive image viewer. Because `iv` is built on top on OpenImageIO, it can display images of any formats readable by `ImageInput` plugins on hand.

More documentation on this later.

10 Getting Image information With `iinfo`

The `iinfo` program will print either basic information (name, resolution, format) or detailed information (including all metadata) found in images. Because `idiff` is built on top on OpenImageIO, it will print information about images of any formats readable by `ImageInput` plugins on hand.

More documentation on this later.

11 Converting Image Formats With `iconvert`

The `iconvert` program will read an image (from any file format for which an `ImageInput` plugin can be found) and then write the image to a new file (in any format for which an `ImageOutput` plugin can be found). In the process, `iconvert` can optionally change the file format or data format (for example, converting floating-point data to 8-bit integers), apply gamma dorrection, switch between tiled and scanline orientation, or alter or add metadata to the image.

More documentation on this later.

12 Creating MIP-mapped texture files with `maketx`

The `maketx` program will read an image (from any file format for which an `ImageInput` plugin can be found) and then write the image as a tiled, mip-mapped texture, environment, or shadow map file that can be accessed efficiently by OpenImageIO's texture API.

More documentation on this later.

13 Comparing Images With `idiff`

The `idiff` program compares two images, printing a report about how different they are and optionally producing a third image that records the pixel-by-pixel differences between them. There are a variety of options and ways to compare (absolute pixel difference, various thresholds for warnings and errors, and also an optional perceptual difference metric).

Because `idiff` is built on top on OpenImageIO, it can compare two images of any formats readable by `ImageInput` plugins on hand. They may have any (or different) file formats, data formats, etc.

More documentation on this later.

Part III

Appendices

A Building OpenImageIO

B Glossary

Channel One of several data values persent in each pixel. Examples include red, green, blue, alpha, etc. The data in one channel of a pixel may be represented by a single number, whereas the pixel as a whole requires one number for each channel.

Client A client (as in "client application") is a program or library that uses OpenImageIO or any of its constituent libraries.

Data format The type of numerical representation used to store a piece of data. Examples include 8-bit unsigned integers, 32-bit floating-point numbers, etc.

Image File Format The specification and data layout of an image on disk. For example, TIFF, JPEG/JFIF, OpenEXR, etc.

Metadata Data about data. As used in OpenImageIO, this means Information about an image, beyond describing the values of the pixels themselves. Examples include the name of the artist that created the image, the date that an image was scanned, the camera settings used when a photograph was taken, etc.

Native data format The *data format* used in the disk file representing an image. Note that with OpenImageIO, this may be different than the data format used by an application to store the image in the computer's RAM.

Pixel One pixel element of an image, consisting of one number describing each *channel* of data at a particular location in an image.

Scanline A single horizontal row of pixels of an image. See also *tile*.

Scanline Image An image whose data layout on disk is organized by breaking the image up into horizontal scanlines, typically with the ability to read or write an entire scanline at once. See also *tiled image*.

Tile A rectangular region of pixels of an image. A rectangular tile is more spatially coherent than a scanline that stretches across the entire image — that is, a pixel's neighbors are most likely in the same tile, whereas a pixel in a scanline image will typically have most of its immediate neighbors on different scanlines (requiring additional scanline reads in order to access them).

Tiled Image An image whose data layout on disk is organized by breaking the image up into rectangular regions of pixels called *tiles*. All the pixels in a tile can be read or written at once, and individual tiles may be read or written separately from other tiles.

Volume Image A 3-D set of pixels that has not only horizontal and vertical dimensions, but also a ”depth” dimension.