

# **OpenImageIO 1.4**

## Programmer Documentation

(in progress)

Editor: Larry Gritz  
*lg@openimageio.org*

Date: 14 Apr 2014

The OpenImageIO source code and documentation are:

Copyright (c) 2008-2014 Larry Gritz, et al. All Rights Reserved.

The code that implements OpenImageIO is licensed under the BSD 3-clause (also sometimes known as “new BSD” or “modified BSD”) license:

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 manual and other text documentation about OpenImageIO are licensed under the Creative Commons Attribution 3.0 Unported License.



<http://creativecommons.org/licenses/by/3.0/>

*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

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Overview . . . . .	1
1.2	Simplifying Assumptions . . . . .	2
<b>I</b>	<b>The OpenImageIO Library APIs</b>	<b>5</b>
<b>2</b>	<b>Image I/O API Helper Classes</b>	<b>7</b>
2.1	Data Type Descriptions: TypeDesc . . . . .	7
2.2	String References: string_ref . . . . .	9
2.3	Efficient unique strings: ustring . . . . .	10
2.4	Image Specification: ImageSpec . . . . .	11
<b>3</b>	<b>ImageOutput: Writing Images</b>	<b>21</b>
3.1	Image Output Made Simple . . . . .	21
3.2	Advanced Image Output . . . . .	22
3.3	ImageOutput Class Reference . . . . .	41
<b>4</b>	<b>Image I/O: Reading Images</b>	<b>49</b>
4.1	Image Input Made Simple . . . . .	49
4.2	Advanced Image Input . . . . .	50
4.3	ImageInput Class Reference . . . . .	62
<b>5</b>	<b>Writing ImageIO Plugins</b>	<b>69</b>
5.1	Plugin Introduction . . . . .	69
5.2	Image Readers . . . . .	69
5.3	Image Writers . . . . .	78
5.4	Tips and Conventions . . . . .	90
5.5	Building ImageIO Plugins . . . . .	91
<b>6</b>	<b>Bundled ImageIO Plugins</b>	<b>93</b>
6.1	BMP . . . . .	93
6.2	Cineon . . . . .	93
6.3	DDS . . . . .	93
6.4	DPX . . . . .	94

6.5	Field3D . . . . .	96
6.6	FITS . . . . .	96
6.7	GIF . . . . .	97
6.8	HDR/RGBE . . . . .	97
6.9	ICO . . . . .	98
6.10	IFF . . . . .	98
6.11	JPEG . . . . .	98
6.12	JPEG-2000 . . . . .	99
6.13	OpenEXR . . . . .	100
6.14	PNG . . . . .	102
6.15	PNM / Netpbm . . . . .	102
6.16	PSD . . . . .	103
6.17	Ptex . . . . .	103
6.18	RLA . . . . .	103
6.19	SGI . . . . .	105
6.20	Softimage PIC . . . . .	105
6.21	RAW digital camera files . . . . .	105
6.22	Targa . . . . .	105
6.23	TIFF . . . . .	106
6.24	Webp . . . . .	109
6.25	Zfile . . . . .	109
<b>7</b>	<b>Cached Images</b>	<b>111</b>
7.1	Image Cache Introduction and Theory of Operation . . . . .	111
7.2	ImageCache API . . . . .	113
<b>8</b>	<b>Texture Access: TextureSystem</b>	<b>123</b>
8.1	Texture System Introduction and Theory of Operation . . . . .	123
8.2	Helper Classes . . . . .	123
8.3	TextureSystem API . . . . .	130
<b>9</b>	<b>Image Buffers</b>	<b>145</b>
9.1	ImageBuf Introduction and Theory of Operation . . . . .	145
9.2	Constructing, reading, and writing an ImageBuf . . . . .	147
9.3	Getting and setting basic information about an ImageBuf . . . . .	150
9.4	Copying ImageBuf's and blocks of pixels . . . . .	152
9.5	Getting and setting individual pixel values – simple but slow . . . . .	153
9.6	Miscellaneous . . . . .	154
9.7	Iterators – the fast way of accessing individual pixels . . . . .	155
9.8	Dealing with buffer data types . . . . .	158
<b>10</b>	<b>Image Processing</b>	<b>161</b>
10.1	ImageBufAlgo general principles . . . . .	161
10.2	Pattern generation . . . . .	162
10.3	Image transformations and data movement . . . . .	164
10.4	Image arithmetic . . . . .	168

10.5 Image comparison and statistics . . . . .	173
10.6 Convolutions . . . . .	178
10.7 Image Enhancement / Restoration . . . . .	180
10.8 Color manipulation . . . . .	181
10.9 Import / export . . . . .	184
<b>11 Python Bindings</b>	<b>189</b>
11.1 Overview . . . . .	189
11.2 TypeDesc . . . . .	189
11.3 ROI . . . . .	192
11.4 ImageSpec . . . . .	193
11.5 ImageInput . . . . .	199
11.6 ImageOutput . . . . .	204
11.7 ImageBuf . . . . .	208
11.8 ImageBufAlgo . . . . .	215
11.9 Miscellaneous Utilities . . . . .	228
 <b>II Image Utilities</b>	 <b>229</b>
<b>12 oiiotool: the OIIO Swiss Army Knife</b>	<b>231</b>
12.1 Overview . . . . .	231
12.2 oiiotool Tutorial / Recipes . . . . .	233
12.3 oiiotool commands: general . . . . .	237
12.4 oiiotool commands: reading and writing images . . . . .	239
12.5 oiiotool commands that change the current image metadata . . . . .	241
12.6 oiiotool commands that shuffle channels or subimages . . . . .	243
12.7 oiiotool commands that adjust the image stack . . . . .	244
12.8 oiiotool commands that make entirely new images . . . . .	245
12.9 oiiotool commands that do image processing . . . . .	246
12.10 oiiotool commands for color management . . . . .	254
 <b>13 The iv Image Viewer</b>	 <b>257</b>
 <b>14 Getting Image information With iinfo</b>	 <b>259</b>
14.1 Using iinfo . . . . .	259
14.2 iinfo command-line options . . . . .	261
 <b>15 Converting Image Formats With iconvert</b>	 <b>263</b>
15.1 Overview . . . . .	263
15.2 iconvert Recipes . . . . .	263
15.3 iconvert command-line options . . . . .	265
 <b>16 Searching Image Metadata With igrep</b>	 <b>269</b>
16.1 Using igrep . . . . .	269
16.2 igrep command-line options . . . . .	269

<b>17 Comparing Images With <code>idiff</code></b>	<b>271</b>
17.1 Overview . . . . .	271
17.2 Using <code>idiff</code> . . . . .	271
17.3 <code>idiff</code> Reference . . . . .	273
<b>18 Making Tiled MIP-Map Texture Files With <code>maketx</code></b>	<b>275</b>
18.1 Overview . . . . .	275
18.2 <code>maketx</code> command-line options . . . . .	275
 <b>III Appendices</b>	 <b>281</b>
<b>A Building OpenImageIO</b>	<b>283</b>
<b>B Metadata conventions</b>	<b>285</b>
B.1 Description of the image . . . . .	285
B.2 Display hints . . . . .	286
B.3 Disk file format info/hints . . . . .	287
B.4 Photographs or scanned images . . . . .	288
B.5 Texture Information . . . . .	288
B.6 Exif metadata . . . . .	289
B.7 GPS Exif metadata . . . . .	295
B.8 IPTC metadata . . . . .	298
B.9 SMPTE metadata . . . . .	301
B.10 Extension conventions . . . . .	301
 <b>C Glossary</b>	 <b>303</b>
 <b>Index</b>	 <b>305</b>



# 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 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.

The `ImageCache` class transparently manages a cache so that it can access truly vast amounts of image data (thousands of image files totaling hundreds of GB) very efficiently using only a tiny amount (tens of megabytes at most) of runtime memory. Additionally, a `TextureSystem` class provides filtered MIP-map texture lookups, atop the nice caching behavior of `ImageCache`.

Finally, 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. Additionally, there are programs for converting images among different formats, comparing image data between two images, and examining image metadata.

All of this is released as “open source” software using the very permissive “New 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 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 elegance 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 64-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 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 can be treated (by apps or readers/writers) as having the same data format (though there is a way to deal with per-channel formats for apps and readers/writers that truly need it).
- 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 implement 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<sup>1</sup>). 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.

---

<sup>1</sup>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

OpenImageIO incorporates or depends upon several other open source packages:

- libtiff © 1988-1997 Sam Leffler and 1991-1997 Silicon Graphics, Inc.  
<http://www.remotesensing.org/libtiff>
- IJG libjpeg © 1991-1998, Thomas G. Lane. <http://www.ijg.org>
- OpenEXR, Ilmbase, and Half © 2006, Industrial Light & Magic.  
<http://www.openexr.com>
- zlib © 1995-2005 Jean-loup Gailly and Mark Adler. <http://www.zlib.net>
- libpng © 1998-2008 Glenn Randers-Pehrson, et al. <http://www.libpng.org>
- The SHA-1 implementation we use is public domain by Dominik Reichl  
<http://www.dominik-reichl.de/>
- Boost © various authors. <http://www.boost.org>
- GLEW © 2002-2007 Milan Ikits, et al. <http://glew.sourceforge.net>
- Jasper © 2001-2006 Michael David Adams, et al.  
<http://www.ece.uvic.ca/~mdadams/jasper/>
- Squish © 2006 Simon Brown. <http://sjbrown.co.uk/?code=squish>
- PugiXML © 2006-2009 by Arseny Kapoulkine (based on work ©2003 Kristen Wegner).  
<http://pugixml.org/>
- DPX reader/writer © 2009 Patrick A. Palmer. <http://code.google.com/p/dpx>
- Ptex © 2009 Disney Enterprises, Inc. <http://ptex.us>
- Field3D © 2009 Sony Pictures Imageworks. <http://sites.google.com/site/field3d/>
- tinyformat.h © 2011 Chris Foster. <http://github.com/c42f/tinyformat>
- lookup3 code by Bob Jenkins, Public Domain. <http://burtleburtle.net/bob/c/lookup3.c>
- xxhash ©2012 Yann Collet (BSD Licensed). <http://code.google.com/p/xxhash/>
- KissFFT ©2003–2010 Mark Borgerding (BSD Licensed). <http://sourceforge.net/projects/kissfft/>
- GIFLIB ©1997 Eric S. Raymond (MIT Licensed). <http://giflib.sourceforge.net/>
- LibRaw ©2008-2013 LibRaw LLC (LGPL, CDDL, and LibRaw licenses). <http://www.libraw.org/>

These other packages are all distributed under licenses that allow them to be used by and distributed with OpenImageIO.

## **Part I**

# **The OpenImageIO Library APIs**



## 2 Image I/O API Helper Classes

### 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 and is described in the header file `OpenImageIO/typedesc.h`. A `TypeDesc` describes a base data format type, aggregation into simple vector and matrix types, and an array length (if it’s an array).

The remainder of this section describes the C++ API for `TypeDesc`. See Section 11.2 for the corresponding Python bindings.

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

<code>UINT8</code>	8-bit integer values ranging from 0..255, corresponding to the C/C++ unsigned char.
<code>INT8</code>	8-bit integer values ranging from -128..127, corresponding to the C/C++ char.
<code>UINT16</code>	16-bit integer values ranging from 0..65535, corresponding to the C/C++ unsigned short.
<code>INT16</code>	16-bit integer values ranging from -32768..32767, corresponding to the C/C++ short.
<code>UINT</code>	32-bit integer values, corresponding to the C/C++ unsigned int.
<code>INT</code>	signed 32-bit integer values, corresponding to the C/C++ int.
<code>UINT64</code>	64-bit integer values, corresponding to the C/C++ unsigned long long (on most architectures).
<code>INT64</code>	signed 64-bit integer values, corresponding to the C/C++ long long (on most architectures).
<code>FLOAT</code>	32-bit IEEE floating point values, corresponding to the C/C++ float.
<code>DOUBLE</code>	64-bit IEEE floating point values, corresponding to the C/C++ double.
<code>HALF</code>	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 <code>int</code> or <code>float</code> 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	sixteen values representing a $4 \times 4$ matrix.

And optionally, a hint about the semantics of the data, described by the enumerated type VECSEMANICS:<sup>1</sup>

NOSEMANICS	nothing special known.
COLOR	indicates a vector that is intended to represent a “color,” not a spatial quantity (and of course therefore does not undergo a transformation).
POINT	indicates a vector that represents a spatial position and should be transformed by a $4 \times 4$ matrix as if it had a 4th component of 1.
VECTOR	indicates a vector that represents a spatial direction and should be transformed by a $4 \times 4$ matrix as if it had a 4th component of 0.
NORMAL	indicates a vector that represents a surface normal and should be transformed like a vector, but using the inverse-transpose of a $4 \times 4$ matrix.
TIMECODE	indicates an <code>int[2]</code> representing the standard 4-byte encoding of an SMPTE timecode.
KEYCODE	indicates an <code>int[7]</code> representing the standard 28-byte encoding of an SMPTE keycode.

These can be combined to fully describe a complex type:

**TypeDesc** (BASETYPE btype, AGGREGATE agg, VECSEMANICS xform=NOSEMANICS)

**TypeDesc** (BASETYPE btype, AGGREGATE agg, int arraylen)

**TypeDesc** (BASETYPE btype, AGGREGATE agg, VECSEMANICS xform, int arraylen)

<sup>1</sup>It's unfortunately called `VECSEMANICS` because it used to be used strictly for 3-vectors. If we were to do it over again, it would just be `SEMANICS`.



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 String References: `string_ref`

A `string_ref` is a non-owning, non-copying, non-allocating reference to a sequence of characters. It encapsulates both a character pointer and a length.

A function that takes a string input (but does not need to alter the string in place) may use a `string_ref` parameter and accept input that is any of `char*` (C string), string literal (constant `char` array), a `std::string` (C++ string), or `OIOO::ustring`. For all of these cases, no extra allocations are performed, and no extra copies of the string contents are performed (as they would be, for example, if the function took a `const std::string&` argument but was passed a `char*` or string literal).

Furthermore, a function that returns a copy or a substring of one of its inputs (for example, a `substr()`-like function) may return a `string_ref` rather than a `std::string`, and thus generate its return value without any allocation or copying. Upon assignment to a `std::string` or `ustring`, it will properly auto-convert.

There are two important caveats to using this class:

1. The `string_ref` merely refers to characters owned by another string, so the `string_ref` may not be used outside the lifetime of the string it refers to. Thus, `string_ref` is great for parameter passing, but it's not a good idea to use a `string_ref` to store strings in a data structure (unless you are really sure you know what you're doing).
2. Because the run of characters that the `string_ref` refers to may not be 0-terminated, it is important to distinguish between the `data()` method, which returns the pointer to the characters, and the `c_str()` method, which is guaranteed to return a valid C string that is 0-terminated. Thus, if you want to pass the contents of a `string_ref` to a function that

expects a 0-terminated string (say, `fopen`), you must call `fopen(my_string_ref.c_str())`. Note that the usual case is that the `string_ref` does refer to a 0-terminated string, and in that case `c_str()` returns the same thing as `data()` without any extra expense; but in the rare case that it is not 0-terminated, `c_str()` will incur extra expense to internally allocate a valid C string.

```
string_ref (const char *chars)
string_ref (const char *chars, size_t length)
string_ref (const std::string &str)
string_ref (ustring ustr)
```

Constructs a `string_ref`. The `string_ref` doesn't have its own copy of the characters, so don't use the `string_ref` after the original string has been destroyed or altered.

Note that the version that takes a `const char*` but not a length will automatically take the `strlen(chars)` to determine the length. (All the other constructors can deduce the length without walking through all of the characters.)

```
const char* string_ref::data ()
size_t string_ref::size ()
```

The raw pointer to the characters (not necessarily 0-terminated!) and the length of the `string_ref`.

```
const char* string_ref::c_str ()
```

Return a 0-terminated `char*` version of the `string_ref` (a proper C string).

```
std::string (string_ref sr)
ustring (string_ref sr)
```

Automatic constructions of C++ `std::string` or OIIO `ustring` from a `string_ref`.

Additionally, a large portion of the usual API for `std::string` is mimicked by `string_ref`. Please consult the public `string_ref.h` header file for full details, if you wish to use `string_ref` in your own code.

## 2.3 Efficient unique strings: `ustring`

A `ustring` is an alternative to `char*` or `std::string` for storing strings, in which the character sequence is stored uniquely. If there are many copies of the same `ustring`, they all point to the same canonical characters, which themselves are stored only once. This allows `ustring`'s to be assigned to one another with only the copy of a pointer assignment (not allocation or copying of characters), and for `==` and `!=` comparisons of two `ustring` values for only the cost of comparing the pointers (not examining the characters).

OpenImageIO uses `ustring` in a few select places, where string assignment/copy and equality/inequality are the dominant string operation and we want them to be the same low cost as a

simple pointer assignment or comparison. (This primarily comes into play for the `ImageCache` and `TextureSystem`, where we refer to images by their filenames and want very fast lookups.)

Please consult the public header `ustring.h` for details, especially if you want to use `ustring` extensively in your own code. But here are the most important details to know if you are calling the `OpenImageIO` functions that take `ustring` parameters:

```
ustring (const char *chars)
ustring (const char *chars, size_t length)
ustring (const std::string &str)
ustring (string_ref sref)
```

Constructs a `ustring` from a C string (`char*`), C++ `std::string`, or a `string_ref`.

```
const char* ustring::c_str ()
const std::string& ustring::string()
string_ref ustring::operator string_ref()
```

Convert a `ustring` to a 0-terminated C string, a C++ `std::string`, or a `string_ref`. All of these are extremely inexpensive.

## 2.4 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) and origin. This specifies what is often called the “pixel data window.”
- The full size and offset of an abstract “full” or “display” window. Differing full and data windows can indicate that the pixels are a crop region or a larger image, or contain overscan pixels.
- 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*.
- 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.

The remainder of this section describes the C++ API for `ImageSpec`. See Section 11.4 for the corresponding Python bindings.

### 2.4.1 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.

```
int nchannels
```

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

```
TypeDesc format
```

```
std::vector<TypeDesc> channelformats
```

Describes 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.

If all channels of the image have the same data format, that will be described by `format` and `channelformats` will be empty (zero length).

If there are different data formats for each channel, they will be described in the `channelformats` vector, and the `format` field will indicate a single default data format for applications that don't wish to support per-channel formats (usually this will be the format of the channel that has the most precision).

```
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 represents *alpha* (pixel coverage and/or transparency). It defaults to -1 if no alpha channel is present, or if it is not known which channel represents alpha.

```
int z_channel
```

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

```
bool deep
```

If `true`, this indicates that the image describes contains “deep” data consisting of multiple samples per pixel. If `false`, it's an ordinary image with one data value (per channel) per pixel.

```
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.

### 2.4.2 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, and values are assumed to be linear.

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

Sets the format as described.

```
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*”).

```
size_t channel_bytes () const
```

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 channel_bytes (int chan, bool native=false) const
```

Returns the number of bytes needed for the single specified channel. If `native` is `false` (default), compute the size of one channel of `this->format`, but if `native` is `true`, compute the size of the channel in terms of the “native” data format of that channel as stored in the file.

```
size_t pixel_bytes (bool native=false) const
```

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

If `native` is `true`, this will be the sum of all the per-channel formats in `channelformats`. If `native` is `false` (the default), or if all channels use the same format, this will simply be the number of channels multiplied by the width (in bytes) of the `format`.

```
size_t pixel_bytes (int chbegin, int chend, bool native=false) const
```

Returns the number of bytes comprising the range of channels `[chbegin, chend)` for each pixel.

If `native` is `true`, this will be the sum of the per-channel formats in `channelformats` (for the given range of channels). If `native` is `false` (the default), or if all channels use the same format, this will simply be the number of channels multiplied by the width (in bytes) of the `format`.

```
imagesize_t scanline_bytes (bool native=false) const
```

Returns the number of bytes comprising each scanline, i.e.,

`pixel_bytes(native) * width`

This will return `std::numeric_limits<imagesize_t>::max()` in the event of an overflow where it’s not representable in an `imagesize_t`.

```
imagesize_t tile_pixels () const
```

Returns the number of tiles comprising an image tile (if it's a tiled image). This will return `std::numeric_limits<imagesize_t>::max()` in the event of an overflow where it's not representable in an `imagesize_t`.

```
imagesize_t tile_bytes (bool native=false) const
```

Returns the number of bytes comprising an image tile (if it's a tiled image), i.e., `pixel_bytes(native) * tile_width * tile_height * tile_depth`. This will return `std::numeric_limits<imagesize_t>::max()` in the event of an overflow where it's not representable in an `imagesize_t`.

```
imagesize_t image_pixels () const
```

Returns the number of pixels comprising an entire image of these dimensions. This will return `std::numeric_limits<imagesize_t>::max()` in the event of an overflow where it's not representable in an `imagesize_t`.

```
imagesize_t image_bytes (bool native=false) const
```

Returns the number of bytes comprising an entire image of these dimensions, i.e., `pixel_bytes(native) * width * height * depth`. This will return `std::numeric_limits<imagesize_t>::max()` in the event of an overflow where it's not representable in an `imagesize_t`.

```
bool size_t_safe () const
```

Return `true` if an image described by this spec can the sizes (in pixels or bytes) of its scanlines, tiles, and the entire image can be represented by a `size_t` on that platform. If this returns `false`, the client application should be very careful allocating storage!

```
void attribute (string_ref name, TypeDesc type,  
               const void *value)
```

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

```
void attribute (string_ref name, unsigned int value)  
void attribute (string_ref name, int value)  
void attribute (string_ref name, float value)  
void attribute (string_ref name, string_ref value)
```

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

```
ImageIOParameter * find_attribute (string_ref name,
                                     TypeDesc searchtype=UNKNOWN,
                                     bool casesensitive=false)
const ImageIOParameter * find_attribute (string_ref name,
                                     TypeDesc searchtype=UNKNOWN,
                                     bool casesensitive=false) const
```

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 `casesensitive` is false.

```
int get_int_attribute (string_ref name, int defaultval=0) const
```

Gets an integer metadata attribute (silently converting to `int` even if the data is really `int8`, `uint8`, `int16`, `uint16`, or `uint32`), and simply substituting the supplied default value if no such metadata exists. This is a convenience function for when you know you are just looking for a simple integer value.

```
float get_float_attribute (string_ref name,
                             float defaultval=0) const
```

Gets a float metadata attribute (silently converting to `float` even if the data is really half or double), simply substituting the supplied default value if no such metadata exists. This is a convenience function for when you know you are just looking for a simple float value.

```
string_ref get_string_attribute (string_ref name,
                                   string_ref defaultval = string_ref()) const
```

Gets a string metadata attribute, simply substituting the supplied default value if no such metadata exists. This is a convenience function for when you know you are just looking for a simple string value.

```
std::string metadata_val (const ImageIOParameter &p, bool human=true) const
```

For a given parameter (in this `ImageSpec`'s `extra_attribs` field), format the value nicely as a string. If `human` is true, use especially human-readable explanations (units, or decoding of values) for certain known metadata.

```
std::string to_xml () const
```

Saves the contents of the `ImageSpec` as XML, returning it as a string.



```
void from_xml (const char *xml) const
```

Populates the fields of the `ImageSpec` based on the XML passed in.

```
TypeDesc channelformat (int chan) const
```

Returns a `TypeDesc` describing the format of the requested channel.

```
void get_channelformats (std::vector<TypeDesc> &formats) const
```

Fill in an array of channel formats describing all channels in the image. (Note that this differs slightly from the member data `channelformats`, which is empty if there are not separate per-channel formats.)

### 2.4.3 Miscellaneous Utilities

These helper functions are not part of any other `OpenImageIO` class, they just exist in the `OpenImageIO` namespace as general utilities. (See Section 11.9 for the corresponding Python bindings.)

```
int openimageio_version ()
```

Returns a numeric value for the version of `OpenImageIO`, 10000 for each major version, 100 for each minor version, 1 for each patch. For example, `OpenImageIO 1.2.3` would return a value of 10203.

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

Returns any error string describing what went wrong if `ImageInput::create()` or `ImageOutput::create()` failed (since in such cases, the `ImageInput` or `ImageOutput` itself does not exist to have its own `geterror()` function called). This function returns the last error for this particular thread; separate threads will not clobber each other's global error messages.

```
bool attribute (string_ref name, TypeDesc type, const void *val)
```

Sets an global attribute (i.e., a property or option) of `OpenImageIO`. The `name` designates the name of the attribute, `type` describes the type of data, and `val` is a pointer to memory containing the new value for the attribute.

If the name is known, valid attribute that matches the type specified, the attribute will be set to the new value and `attribute()` will return `true`. If `name` is not recognized, or if the types do not match (e.g., `type` is `TypeDesc::TypeFloat` but the named attribute is a string), the attribute will not be modified, and `attribute()` will return `false`.

The following are the recognized attributes:

```
int threads
```

Some OpenImageIO operations can be accelerated if allowed to spawn multiple threads to parallelize the task. (Examples: decompressing multiple simultaneously-read OpenEXR scanlines, or many `ImageBuf` operations.) This attribute sets the maximum number of threads that will be spawned. The default is 1. If set to 0, it means that it should use as many threads as there are hardware cores present on the system.

`string plugin_searchpath`

A colon-separated list of directories to search for dynamically-loaded format plugins.

`string format_list`

A comma-separated list of all the names of all supported image format readers and writers. (Note: can only be retrieved by `getattribute()`, cannot be set by `attribute()`.)

`string extension_list`

For each format, the format name, followed by a colon, followed by a comma-separated list of all extensions that are presumed to be used for that format. Semi-colons separate the lists for formats. For example,

```
"tiff:tif; jpeg:jpg, jpeg; openexr:exr"
```

(Note: can only be retrieved by `getattribute()`, cannot be set by `attribute()`.)

```
bool attribute (string_ref name, int val)
bool attribute (string_ref name, float val)
bool attribute (string_ref name, const char *val)
bool attribute (string_ref name, const std::string & val)
```

Specialized versions of `attribute()` in which the data type is implied by the type of the argument.

```
bool getattribute (string_ref name, TypeDesc type, void *val)
```

Gets the current value of a global attribute. The `name` designates the name of the attribute, `type` describes the type of data, and `val` is a pointer to memory where the user would like the value placed.

If the attribute name is valid and matches the type specified, the attribute value will be stored at address `val` and `attribute()` will return `true`. If `name` is not recognized as a valid attribute name, or if the types do not match (e.g., `type` is `TypeDesc::TypeFloat`

but the named attribute is a string), no data will be written to `val`, and `attribute()` will return `false`.

The complete list of attributes can be found above, in the description of the `attribute()` function.

```
bool getattribute (string_ref name, int &val)
bool getattribute (string_ref name, float &val)
bool getattribute (string_ref name, char **val)
bool getattribute (string_ref name, std::string & val)
```

Specialized versions of `getattribute()` in which the data type is implied by the type of the argument.

For example, the following are equivalent to the example above for the general (pointer) form of `getattribute()`:

```
int maxfiles;
ts->getattribute ("max_open_files", &maxfiles);
const char *path;
ts->getattribute ("plugin_searchpath", &path);
```

```
void declare_imageio_format (const std::string &format_name,
                             ImageInput::Creator input_creator,
                             const char **input_extensions,
                             ImageOutput::Creator output_creator,
                             const char **output_extensions)
```

Register the input and output ‘create’ routines and list of file extensions for a particular format.



## 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 <OpenImageIO/imageio.h>
OIIO_NAMESPACE_USING
...

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)
    return;
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 ( $640 \times 480$ ), 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` implementations 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` implementations 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 (xbegin, xend, ybegin, yend, zbegin, zend,
                    TypeDesc::UINT8, rect);
```

The first six arguments to `write_rectangle()` specify the region of pixels that is being transmitted by supplying the minimum and one-past-maximum pixel indices in *x* (column), *y* (scanline), and *z* (slice, always 0 for 2D non-volume images).<sup>1</sup> The total number of pixels being transmitted is therefore:

$$(xend-xbegin) * (yend-ybegin) * (zend-zbegin)$$

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).

<sup>1</sup>OpenImageIO nearly always follows the C++ STL convention of specifying ranges as `[begin,end)`, that is, `begin, begin+1, ..., end-1`.



### 3.2.2 Converting data 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.

```
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.

### 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  $x$  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 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 achieve 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  $32 \times 32$  pixel crop window within an abstract  $640 \times 480$  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*<sup>2</sup> 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, if 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.

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.

---

<sup>2</sup>*Metadata* refers to data about data, in this case, data about the image that goes beyond the pixel values and description thereof.

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.

### 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:

```
ImageSpec 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,

```
ImageSpec spec (...);

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

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

### Color space 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, we have a convention for explaining what color space your image pixels are in. Each individual `ImageOutput` should document how it uses this (or not).

The `ImageSpec::extra_attribs` field should store metadata that reveals the color space of the pixels you are sending the `ImageOutput`. The `"oiio:ColorSpace"` attribute may take on any of the following values:

`"Linear"` indicates that the color pixel values are known to be linear.

`"GammaCorrected"` indicates that the color pixel values (but not alpha or  $z$ ) have already been gamma corrected (raised to the power  $1/\gamma$ ), and that the gamma exponent may be found in the `"oiio:Gamma"` metadata.

`"sRGB"` indicates that the color pixel values are in sRGB color space.

`"AdobeRGB"` indicates that the color pixel values are in Adobe RGB color space.

`"Rec709"` indicates that the color pixel values are in Rec709 color space.

`"KodakLog"` indicates that the color pixel values are in Kodak logarithmic color space.

The color space hints only describe color channels. You should always pass alpha, depth, or other non-color channels with linear values.

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.attribute ("oiio:ColorSpace", "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 should try to 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 `"oiio:ColorSpace"` unset, the values will not be transformed, since the plugin can't be sure that it's not in the correct space to begin with.

### 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 always done by rescaling the value so that 0.0 maps to integer 0 and 1.0 to the maximum value representable by the integer type, then rounded to an integer value for final output. Here is the code that implements this transformation (`T` is the final output integer type):

```
float value = quant_max * input;
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	min	max
UINT8	0	255
INT8	-128	127
UINT16	0	65535
INT16	-32768	32767
UINT	0	4294967295
INT	-2147483648	2147483647

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, and do not clamp (except to their full floating-point range).

**NEW!**

The `"oio:dither"` attribute in the `ImageSpec` is used to signal the desire for dithering when writing `float` data to a `uint8` file. Dithering can reduce visible banding when converting continuous float data to low bit depth integer images. The attribute is an integer, and if 0 (the default), it means not to dither. If nonzero, it requests that the results are dithered prior to quantization, and the specific nonzero value acts as a "seed" to the hash function that determines the per-pixel dither values (two images using the same seed will have the same dither pattern, different seeds will yield different dither patterns).



### 3.2.7 Random access and repeated transmission of pixels

All `ImageOutput` implementations that support scanlines and tiles should write pixels in 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

Some image file formats support storing multiple images within a single 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"))
    ...
```

Some image formats allow you to do the initial `open()` call without declaring the specifics of the subimages, and simply append subimages as you go. You can detect this by checking

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

In this case, all you have to do is, after writing all the pixels of one image but before calling `close()`, call `open()` again for the next subimage and pass `AppendSubimage` as the value for the *mode* 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. Here is an example:

```
const char *filename = "foo.tif";
int nsubimages;      // assume this is set
ImageSpec specs[];   // assume these are set for each subimage
unsigned char *pixels[]; // assume a buffer for each subimage

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

// Be sure we can support subimages
if (subimages > 1 && (! out->supports("multiimage") ||
                    ! out->supports("appendsubimage"))) {
```

```

        std::cerr << "Does not support appending of subimages\n";
        delete out;
        return;
    }

    // Use Create mode for the first level.
    ImageOutput::OpenMode appendmode = ImageOutput::Create;

    // Write the individual subimages
    for (int s = 0; s < nsubimages; ++s) {
        out->open (filename, specs[s], appendmode);
        out->write_image (TypeDesc::UINT8, pixels[s]);
        // Use AppendSubimage mode for subsequent levels
        appendmode = ImageOutput::AppendSubimage;
    }
    out->close ();
    delete out;

```

On the other hand, if `out->supports("appendsubimage")` returns false, then you must use a different `open()` variety that allows you to declare the number of subimages and their specifications up front.

Below is an example of how to write a multi-subimage file, assuming that you know all the image specifications ahead of time. This should be safe for any file format that supports multiple subimages, regardless of whether it supports appending, and thus is the preferred method for writing subimages, assuming that you are able to know the number and specification of the subimages at the time you first open the file.

```

const char *filename = "foo.tif";
int nsubimages;      // assume this is set
ImageSpec specs[];   // assume these are set for each subimage
unsigned char *pixels[]; // assume a buffer for each subimage

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

// Be sure we can support subimages
if (subimages > 1 && ! out->supports ("multiimage")) {
    std::cerr << "Cannot write multiple subimages\n";
    delete out;
    return;
}

// Open and declare all subimages
out->open (filename, nsubimages, specs);

// Write the individual subimages
for (int s = 0; s < nsubimages; ++s) {
    if (s > 0) // Not needed for the first, which is already open
        out->open (filename, specs[s], ImageInput::AppendSubimage);
    out->write_image (TypeDesc::UINT8, pixels[s]);
}

```

```

    }
    out->close ();
    delete out;

```

In both of these examples, 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 MIP-maps

Some image file formats support multiple copies of an image at successively lower resolutions (MIP-map levels, or an “image pyramid”). Given a created `ImageOutput`, you can query whether MIP-maps may be stored in the file:

```

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

```

If you are working with an `ImageOutput` that supports MIP-map levels, it is easy to write these levels. After writing all the pixels of one MIP-map level, call `open()` again for the next MIP level and pass `ImageInput::AppendMIPLevel` as the value for the *mode* argument, and then write the pixels of the subsequent MIP level. (See Section 3.3 for the full technical description of the arguments to `open()`.) The `close()` routine is called just once, after all subimages and MIP levels are completed.

Below is pseudocode for writing a MIP-map (a multi-resolution image used for texture mapping):

```

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 either mipmaps or subimages
if (! out->supports ("mipmap") && ! 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);

// Use Create mode for the first level.
ImageOutput::OpenMode appendmode = ImageOutput::Create;

// 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, appendmode);
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;

// For subsequent levels, change the mode argument to
// open(). If the format doesn't support MIPmaps directly,
// try to emulate it with subimages.
if (out->supports("mipmap"))
    appendmode = ImageOutput::AppendMIPLevel;
else
    appendmode = ImageOutput::AppendSubimage;
}
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 MIP level.

### 3.2.10 Per-channel formats

Some image formats allow separate per-channel data formats (for example, half data for colors and float data for depth). When this is desired, the following steps are necessary:

1. Verify that the writer supports per-channel formats by checking `supports ("channelformats")`.
2. The `ImageSpec` passed to `open()` should have its `channelformats` vector filled with the types for each channel.
3. The call to `write_scanline`, `read_scanlines`, `write_tile`, `write_tiles`, or `write_image` should pass a data pointer to the raw data, already in the native per-channel format of the file and contiguously packed, and specify that the data is of type `TypeDesc::UNKNOWN`.

For example, the following code fragment will write a 5-channel image to an OpenEXR file, consisting of R/G/B/A channels in half and a Z channel in float:

```

// Mixed data type for the pixel
struct Pixel { half r,g,b,a; float z; };
Pixel pixels[xres*yres];

ImageOutput *out = ImageOutput::create ("foo.exr");

// Double check that this format accepts per-channel formats
if (! out->supports("channelformats")) {
    delete out;
    return;
}

```

```

    }

    // Prepare an ImageSpec with per-channel formats
    ImageSpec spec (xres, yres, 5, TypeDesc::FLOAT);
    spec.channelformats.push_back (TypeDesc::HALF);
    spec.channelformats.push_back (TypeDesc::HALF);
    spec.channelformats.push_back (TypeDesc::HALF);
    spec.channelformats.push_back (TypeDesc::HALF);
    spec.channelformats.push_back (TypeDesc::FLOAT);
    spec.channelnames.clear ();
    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");

    out->open (filename, spec);
    out->write_image (TypeDesc::UNKNOWN, /* use channel formats */
                    pixels, /* data buffer */
                    sizeof(Pixel)); /* pixel stride */

```

### 3.2.11 Writing “deep” data

Some image file formats (OpenEXR only, at this time) support the concept of “deep” pixels – those containing multiple samples per pixel (and a potentially differing number of them in each pixel). You can tell if a format supports deep images by checking `supports("deepdata")`, and you can specify a deep data in an `ImageSpec` by setting its `deep` field will be true.

Deep files cannot be written with the usual `write_scanline`, `write_scanlines`, `write_tile`, `write_tiles`, `write_image` functions, due to the nature of their variable number of samples per pixel. Instead, `ImageOutput` has three special member functions used only for writing deep data:

```

bool write_deep_scanlines (int ybegin, int yend, int z,
                          const DeepData &deepdata);

bool write_deep_tiles (int xbegin, int xend, int ybegin, int yend,
                      int zbegin, int zend, const DeepData &deepdata);

bool write_deep_image (const DeepData &deepdata);

```

It is only possible to write “native” data types to deep files; that is, there is no automatic translation into arbitrary data types as there is for ordinary images. All three of these functions are passed deep data in a special `DeepData` structure, defined in `imageio.h` as follows:

```

struct DeepData {
    int npixels, nchannels;
    std::vector<TypeDesc> channeltypes; // for each channel [c]
    std::vector<unsigned int> nsamples; // for each pixel [z][y][x]
    std::vector<void *> pointers; // for each channel per pixel [z][y][x][c]
    std::vector<char> data; // for each sample [z][y][x][c][s]

```

```

DeepData ();
// Set the size and allocate the nsamples[] vector.
void init (int npix, int nchan,
           const TypeDesc *chbegin, const TypeDesc *chend);
// After nsamples[] has been filled in, allocate enough scratch space
// for data and set up all the pointers.
void alloc ();
// Clear the vectors and reset size to 0.
void clear ();
// Deallocate all space in the vectors
void free ();
// Retrieve the pointer to the first sample of the given pixel and
// channel. Return NULL if there are no samples for that pixel.
void *channel_ptr (int pixel, int channel) const;
// Retrieve sample value within a pixel, cast to a float.
float deep_value (int pixel, int channel, int sample) const;
};

```

Here is an example of using these methods to write a deep image:

```

// Prepare the spec for 'half' RGBA, 'float' z
int nchannels = 5;
ImageSpec spec (xres, yres, nchannels);
TypeDesc channeltypes[] = { TypeDesc::HALF, TypeDesc::HALF,
                             TypeDesc::HALF, TypeDesc::HALF, TypeDesc::FLOAT };
spec.z_channel = 4;
spec.channelnames[spec.z_channel] = "Z";
spec.channeltypes.assign (channeltypes+0, channeltypes+nchannels);
spec.deep = true;

// Prepare the data (sorry, complicated, but need to show the gist)
DeepData deepdata;
deepdata.init (xres*yres, 5, channeltypes+0, channeltypes+nchannels);
for (int y = 0; y < yres; ++y)
    for (int x = 0; x < xres; ++x)
        deepdata.nsamples[y*xres+x] = ...num samples for that pixel...;
deepdata.alloc (); // allocate pointers and data
int pixel = 0;
for (int y = 0; y < yres; ++y)
    for (int x = 0; x < xres; ++x, ++pixel)
        for (int chan = 0; chan < nchannels; ++chan) {
            void *ptr = deepdata.pointers[pixel*nchannels + c];
            if (chan < 4) { // RGBA -- it's HALF data
                for (int samp = 0; samp < deepdata.nsamples[pixel]; ++samp)
                    ((half *)ptr)[samp] = ...value...;
            } else {
                // z channel -- FLOAT data
                for (int samp = 0; samp < deepdata.nsamples[pixel]; ++samp)
                    ((float *)ptr)[samp] = ...value...;
            }
        }
}

```

```

    }
}

// Create the output
ImageOutput *out = ImageOutput::create (filename);
if (! out)
    return;
// Make sure the format can handle deep data and per-channel formats
if (! out->supports("deepdata") || ! out->supports("channelformats"))
    return;

// Do the I/O (this is the easy part!)
out->open (filename, spec);
out->write_deep_image (deepdata);
out->close ();
delete out;

```

### 3.2.12 Copying an entire image

Suppose you want to copy an image, perhaps with alterations to the metadata but not to the pixels. You could open an `ImageInput` and perform a `read_image()`, and open another `ImageOutput` and call `write_image()` to output the pixels from the input image. However, for compressed images, this may be inefficient due to the unnecessary decompression and subsequent re-compression. In addition, if the compression is *lossy*, the output image may not contain pixel values identical to the original input.

A special `copy_image` method of `ImageOutput` is available that attempts to copy an image from an open `ImageInput` (of the same format) to the output as efficiently as possible without altering pixel values, if at all possible.

Not all format plugins will provide an implementation of `copy_image` (in fact, most will not), but the default implementation simply copies pixels one scanline or tile at a time (with decompression/recompression) so it's still safe to call. Furthermore, even a provided `copy_image` is expected to fall back on the default implementation if the input and output are not able to do an efficient copy. Nevertheless, this method is recommended for copying images so that maximal advantage will be taken in cases where savings can be had.

The following is an example use of `copy_image` to transfer pixels without alteration while modifying the image description metadata:

```

// Open the input file
const char *input = "input.jpg";
ImageInput *in = ImageInput::create (input);
ImageSpec in_spec;
in->open (input, in_spec);

// Make an output spec, identical to the input except for metadata
ImageSpec out_spec = in_spec;
out_spec.attribute ("ImageDescription", "My Title");

```

```
// Create the output file and copy the image
const char *output = "output.jpg";
ImageOutput *out = ImageOutput::create (output);
out->open (output, out_spec);
out->copy_image (in);

// Clean up
out->close ();
delete out;
in->close ();
delete in;
```

### 3.2.13 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 *search 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 `OIIO_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);
...
```

### 3.2.14 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 message is accessible using the `ImageOutput` method `geterror()`, 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 `geterror()` function, there exists a global `geterror()` 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 <OpenImageIO/imageio.h>
OIIO_NAMESPACE_USING
...

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::geterror() << "\n";
    return;
}
ImageSpec spec (xres, yres, channels, TypeDesc::UINT8);

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

if (! out->write_image (TypeDesc::UINT8, pixels)) {
    std::cerr << "Could not write pixels to " << filename
                << ", error = " << out->geterror() << "\n";
    delete out;
    return;
}

if (! out->close ()) {
    std::cerr << "Error closing " << filename
                << ", error = " << out->geterror() << "\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 OpenImageIO 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 subimages within a single file?
- "appendsubimage" Does this format support multiple subimages that can be successively appended at will, without needing to pre-declare the number and specifications the subimages when the file is first opened?
- "mipmap" Does this format support resolutions per image/subimage (MIP-map levels)?
- "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).
- "channelformats" Does this format writer support per-channel data formats, respecting the `ImageSpec`'s `channelformats` field? (If not, it only accepts a single data format for all channels and will ignore the `channelformats` field of the spec.)
- "displaywindow" Does the image format support specifying a display ("full") window that is distinct from the pixel data window?
- "origin" Does the image format support specifying a pixel window origin (i.e., nonzero `ImageSpecx, y, z`)?
- "negativeorigin" Does the image format allow pixel and data window origins (i.e., nonzero `ImageSpecx, y, z, full_x, full_y, full_z`) to have negative values?

"deepdata" Does the image format allow “deep” data consisting of multiple values per pixel (and potentially a differing number of values from pixel to pixel)?

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,
           OpenMode mode=Create)
```

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 multiimage and mode is AppendSubimage, or if it supports MIP-maps and mode is AppendMIPLevel – this is interpreted as appending a subimage, or a MIP level to the current subimage, respectively.

```
bool open (const std::string &name, int subimages, const ImageSpec *specs)
```

Open the file with given name, expecting to have a given total number of subimages, described by specs[0..subimages-1]. Return true for success, false for failure. Upon success, the first subimage will be open and ready for transmission of pixels. Subsequent subimages will be denoted with the usual call of open(name, spec, AppendSubimage) (and MIP levels by open(name, spec, AppendMIPLevel)).

The purpose of this call is to accommodate format-writing libraries that must know the number and specifications of the subimages upon first opening the file; such formats can be detected by

```
supports("multiimage") && ! supports("appendsubimage")
```

The individual specs passed to the appending open() calls for subsequent subimages must match the ones originally passed.

```
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 the scanline that includes pixels  $(*,y,z)$  from data. For 2D non-volume images,  $z$  is ignored. 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()
```

This method automatically converts the data from the specified `format` to the actual output format of the file. If `format` is `TypeDesc::UNKNOWN`, the data is assumed to already be in the file's native format (including per-channel formats, as specified in the `ImageSpec`'s `channelformats` field, if applicable). 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_scanlines (int ybegin, int yend, int z,
                      TypeDesc format, const void *data,
                      stride_t xstride=AutoStride, stride_t ystride=AutoStride)
```

Write a block of scanlines that include pixels  $(*,y,z)$ , where  $ybegin \leq y < yend$ . This is essentially identical to `write_scanline()`, except that it can write more than one scanline at a time, which may be more efficient for certain image format writers.

For 2D non-volume images,  $z$  is ignored. The `xstride` value gives the distance between successive pixels (in bytes), and `ystride` gives the distance between successive scanlines. Strides set to the special value `AutoStride` imply contiguous data, i.e.,

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

This method automatically converts the data from the specified `format` to the actual output format of the file. If `format` is `TypeDesc::UNKNOWN`, the data is assumed to already be in the file's native format (including per-channel formats, as specified in the `ImageSpec`'s `channelformats` field, if applicable). 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 file. If `format` is `TypeDesc::UNKNOWN`, the data is assumed to already be in the file's native format (including per-channel formats, as specified in the

ImageSpec's `channelformats` field, if applicable). 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.

This function returns `true` if it successfully writes the tile, otherwise `false` for a failure. The call will fail if the image is not tiled, or if  $(x,y,z)$  is not actually a tile boundary.

```
bool write_tiles (int xbegin, int xend, int ybegin, int yend,
                 int zbegin, int zend, TypeDesc format, const void *data,
                 stride_t xstride=AutoStride, stride_t ystride=AutoStride,
                 stride_t zstride=AutoStride)
```

Write the tiles that include pixels  $xbegin \leq x < xend$ ,  $ybegin \leq y < yend$ ,  $zbegin \leq z < zend$  from data, converting if necessary from format specified into the file's native data format. If format is `TypeDesc::UNKNOWN`, the data will be assumed to already be in the native format (including per-channel formats, if applicable). 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 * spec.pixel_size()
ystride = xstride * (xend-xbegin)
zstride = ystride * (yend-ybegin)
```

The data for those tiles is assumed to be in the usual image order, as if it were just one big tile, and not "padded" to a whole multiple of the tile size.

This function returns `true` if it successfully writes the tiles, otherwise `false` for a failure. The call will fail if the image is not tiled, or if the pixel ranges do not fall along tile (or image) boundaries, or if it is not a valid tile range.

```
bool write_rectangle (int xbegin, int xend, int ybegin, int yend,
                     int zbegin, int zend, TypeDesc format, const void *data,
                     stride_t xstride=AutoStride, stride_t ystride=AutoStride,
                     stride_t zstride=AutoStride)
```

Write pixels covering the range that includes pixels  $xbegin \leq x < xend$ ,  $ybegin \leq y < yend$ ,  $zbegin \leq z < zend$ . 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*(xend-xbegin)
zstride = ystride*(yend-ybegin)
```

This method automatically converts the data from the specified format to the actual output format of the file. If format is `TypeDesc::UNKNOWN`, the data is assumed to already be in the file's native format (including per-channel formats, as specified in the ImageSpec's `channelformats` field, if applicable). 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  $\text{spec.width} \times \text{spec.height} \times \text{spec.depth}$  pixels, with the given strides and in the desired format. If `format` is `TypeDesc::UNKNOWN`, the data is assumed to already be in the file's native format (including per-channel formats, as specified in the `ImageSpec`'s `channelformats` field, if applicable). 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.

```
bool write_deep_scanlines (int ybegin, int yend, int z,
                           const DeepData &deepdata)
bool write_deep_tiles (int xbegin, int xend, int ybegin, int yend,
                      int zbegin, int zend, const DeepData &deepdata)
bool write_deep_image (const DeepData &deepdata)
```

Write deep data for a block of scanlines, a block of tiles, or an entire image (analogously to the usual `write_scanlines`, `write_tiles`, and `write_image`, but with deep data). Return `true` for success, `false` for failure.

```
bool copy_image (ImageInput *in)
```

Read the current subimage of `in`, and write it as the next subimage of `*this`, in a way that is efficient and does not alter pixel values, if at all possible. Both `in` and `this` must be a properly-opened `ImageInput` and `ImageOutput`, respectively, and their current images must match in size and number of channels. Return `true` if it works ok, `false` if for some reason the operation wasn't possible.

If a particular `ImageOutput` implementation does not supply a `copy_image` method, it will inherit the default implementation, which is to simply read scanlines or tiles from `in` and write them to `*this`. However, some file format implementations may have a special technique for directly copying raw pixel data from the input to the output, when both input and output are the same file type and the same data format. This can be more efficient than `in->read_image` followed by `out->write_image`, and avoids any unintended pixel alterations, especially for formats that use lossy compression.

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 **geterror** ()

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 <OpenImageIO/imageio.h>
OIIO_NAMESPACE_USING
...

ImageInput *in = ImageInput::open (filename);
if (! in)
    return;
const ImageSpec &spec = in->spec();
int xres = spec.width;
int yres = spec.height;
int channels = spec.nchannels;
std::vector<unsigned char> pixels (xres*yres*channels);
in->read_image (TypeDesc::UINT8, &pixels[0]);
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, and tries to fully open the file.

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

- The specification, accessible as `in->spec()`, 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.

```
const ImageSpec &spec = in->spec();
int xres = spec.width;
```

```
int yres = spec.height;
int channels = spec.nchannels;
std::vector<unsigned char> pixels (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[0]);
```

- 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.

Examining the `ImageSpec` reveals whether the file is scanline or tile-oriented: a scanline image will have `spec.tile_width` and `spec.tile_height` set to 0, whereas a tiled images will have nonzero values for the tile dimensions.

#### Reading scanlines

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

```
...
in = ImageInput::open (filename);
const ImageSpec &spec = in->spec();
if (spec.tile_width == 0) {
    std::vector<unsigned char> scanline (spec.width*spec.channels);
```

```

    for (int y = 0; y < yres; ++y) {
        in->read_scanline (y, 0, TypeDesc::UINT8, &scanline[0]);
        ... process data in scanline[0..width*channels-1] ...
    }
} else {
    ... handle tiles, or reject the file ...
}
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).

Nearly all `ImageInput` implementations will be most efficient reading scanlines in strict order (starting with scanline 0, then 1, up to `yres-1`, without skipping any). An `ImageInput` is required to accept `read_scanline()` requests in arbitrary order, but depending on the file format and reader implementation, out-of-order scanline reads may be inefficient.

There is also a `read_scanlines()` function that operates similarly, except that it takes a `ybegin` and `yend` that specify a range, reading all scanlines  $ybegin \leq y < yend$ . For most image format readers, this is implemented as a loop over individual scanlines, but some image format readers may be able to read a contiguous block of scanlines more efficiently than reading each one individually.

The full descriptions of the `read_scanline()` and `read_scanlines()` functions may be found in Section 4.3.

### Reading 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 = ImageInput::open (filename);
const ImageSpec &spec = in->spec();
if (spec.tile_width == 0) {
    ... read by scanline ...
} else {
    // Tiles
    int tilesize = spec.tile_width * spec.tile_height;
    std::vector<unsigned char> tile (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[0]);
            ... process the pixels in 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` implementations 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::open ("myfile.tif");
const ImageSpec &spec = in->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  $x$  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 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 achieve 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::open (filename);
const ImageSpec &spec = in->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.

##### 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::open (filename);
const ImageSpec &spec = in->spec();
...
ParamValue *p = spec.find_attribute ("Orientation", TypeDesc::TypeInt);
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.

For certain common types, there is an even simpler method for retrieving the metadata:

```
int i = spec.get_int_attribute ("Orientation", 0);
float f = spec.get_float_attribute ("PixelAspectRatio", 1.0f);
std::string s = spec.get_string_attribute ("ImageDescription", "");
```

This method simply returns the value. The second argument is the default value to use if the attribute named is not found. These versions will do automatic type conversion as well — for example, if you ask for a float and the attribute is really an int, it will return the proper float for it; or if the attribute is a `UINT16` and you call `get_int_attribute`, it will succeed, promoting to an int.

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::TypeString)
        printf ("%s", *(const char **)p.data());
    else if (p.type() == TypeDesc::TypeFloat)
        printf ("%g", *(const float *)p.data());
    else if (p.type() == TypeDesc::TypeInt)
        printf ("%d", *(const int *)p.data());
    else if (p.type() == TypeDesc::UINT)
        printf ("%u", *(const unsigned int *)p.data());
    else if (p.type() == TypeDesc::TypeMatrix) {
        const float *f = (const float *)p.data();
```

```

        printf ("%f %f %f %f %f %f %f %f "
               "%f %f %f %f %f %f %f %f",
               f[0], f[1], f[2], f[3], f[4], f[5], f[6], f[7],
               f[8], f[9], f[10], f[11], f[12], f[13], f[14], f[15]);
    }
    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.

### Color space 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::extra_attrs` field may store metadata that reveals the color space the image file in the `"oiio:ColorSpace"` attribute, which may take on any of the following values:

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

"GammaCorrected" indicates that the color pixel values (but not alpha or  $z$ ) have already been gamma corrected (raised to the power  $1/\gamma$ ), and that the gamma exponent may be found in the `"oiio:Gamma"` metadata.

"sRGB" indicates that the color pixel values are in sRGB color space.

"AdobeRGB" indicates that the color pixel values are in Adobe RGB color space.

"Rec709" indicates that the color pixel values are in Rec709 color space.

"KodakLog" indicates that the color pixel values are in Kodak logarithmic color space.

The `ImageInput` sets the `"oiio:ColorSpace"` metadata 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, at the very least.

The color space hints only describe 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.



### 4.2.5 Multi-image files and MIP-maps

Some image file formats support multiple discrete subimages to be stored in one file, and/or multiple resolutions for each image to form a MIPmap. When you `open()` an `ImageInput`, it will by default point to the first (i.e., number 0) subimage in the file, and the highest resolution (level 0) MIP-map level. You can switch to viewing another subimage or MIP-map level using the `seek_subimage()` function:

```
ImageInput *in = ImageInput::open (filename);
const ImageSpec &spec = in->spec();
...
int subimage = 1;
int miplevel = 0;
if (in->seek_subimage (subimage, miplevel, spec)) {
    ...
} else {
    ... no such subimage/miplevel ...
}
```

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

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::open (filename);
const ImageSpec &spec = in->spec();

int num_miplevels = 0;
while (in->seek_subimage (0, num_miplevels, spec)) {
    // Note: spec has the format of the current subimage/miplevel
    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_miplevels;
}
// Note: we break out of the while loop when seek_subimage fails
// to find a next MIP level.

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 and MIP level.

#### 4.2.6 Per-channel formats

Some image formats allow separate per-channel data formats (for example, half data for colors and float data for depth). If you want to read the pixels in their true native per-channel formats, the following steps are necessary:

1. Check the `ImageSpec`'s `channelformats` vector. If non-empty, the channels in the file do not all have the same format.
2. When calling `read_scanline`, `read_scanlines`, `read_tile`, `read_tiles`, or `read_image`, pass a format of `TypeDesc::UNKNOWN` to indicate that you would like the raw data in native per-channel format of the file written to your data buffer.

For example, the following code fragment will read a 5-channel image to an OpenEXR file, consisting of R/G/B/A channels in half and a Z channel in float:

```
ImageInput *in = ImageInput::open (filename);
const ImageSpec &spec = in->spec();

// Allocate enough space
unsigned char *pixels = new unsigned char [spec.image_bytes(true)];

in->read_image (TypeDesc::UNKNOWN, /* use native channel formats */
               pixels);           /* data buffer */

if (spec.channelformats.size() > 0) {
    ... the buffer contains packed data in the native
        per-channel formats ...
} else {
    ... the buffer contains all data per spec.format ...
}
```

#### 4.2.7 Reading “deep” data

Some image file formats (OpenEXR only, at this time) support the concept of “deep” pixels – those containing multiple samples per pixel (and a potentially differing number of them in each pixel). You can tell an image is “deep” from its `ImageSpec`: the `deep` field will be true.

Deep files cannot be read with the usual `read_scanline`, `read_scanlines`, `read_tile`, `read_tiles`, `read_image` functions, due to the nature of their variable number of samples per pixel. Instead, `ImageInput` has three special member functions used only for reading deep data:

```

bool read_native_deep_scanlines (int ybegin, int yend, int z,
                                int chbegin, int chend,
                                DeepData &deepdata);

bool read_native_deep_tiles (int xbegin, int xend, int ybegin, int yend,
                             int zbegin, int zend,
                             int chbegin, int chend, DeepData &deepdata);

bool read_native_deep_image (DeepData &deepdata);

```

It is only possible to read “native” data types from deep files; that is, there is no automatic translation into arbitrary data types as there is for ordinary images. All three of these functions store the resulting deep data in a special `DeepData` structure, defined in `imageio.h` as follows:

```

struct DeepData {
    int npixels, nchannels;
    std::vector<TypeDesc> channeltypes; // for each channel [c]
    std::vector<unsigned int> nsamples; // for each pixel [z][y][x]
    std::vector<void *> pointers;      // for each channel per pixel [z][y][x][c]
    std::vector<char> data;           // for each sample [z][y][x][c][s]

    DeepData ();
    // Set the size and allocate the nsamples[] vector.
    void init (int npix, int nchan,
               const TypeDesc *chbegin, const TypeDesc *chend);
    // After nsamples[] has been filled in, allocate enough scratch space
    // for data and set up all the pointers.
    void alloc ();
    // Clear the vectors and reset size to 0.
    void clear ();
    // Deallocate all space in the vectors
    void free ();
    // Retrieve the pointer to the first sample of the given pixel and
    // channel. Return NULL if there are no samples for that pixel.
    void *channel_ptr (int pixel, int channel) const;
    // Retrieve sample value within a pixel, cast to a float.
    float deep_value (int pixel, int channel, int sample) const;
};

```

Here is an example of using these methods to read a deep image from a file and print all its values:

```

ImageInput *in = ImageInput::open (filename);
if (! in)
    return;
const ImageSpec &spec = in->spec();
if (spec.deep) {
    DeepData deepdata;
    in->read_native_deep_image (deepdata);
    int p = 0; // absolute pixel number
    for (int y = 0; y < spec.height; ++y) {

```

```

    for (int x = 0; x < spec.width; ++x) {
        std::cout << "Pixel " << x << ", " << y << ":\n";
        if (deepdata.nsamples[p] == 0)
            std::cout << " no samples\n";
        else
            for (int c = 0; c < spec.nchannels; ++c) {
                TypeDesc type = deepdata.channeltypes[c];
                std::cout << " " << spec.channelnames[c] << ": ";
                void *ptr = deepdata.pointers[p*spec.nchannels+c]
                for (int s = 0; s < deepdata.nsamples[p]; ++s) {
                    if (type.basetype == TypeDesc::FLOAT)
                        std::cout << ((float *)ptr)[s] << ' ';
                    else if (type.basetype == TypeDesc::HALF)
                        std::cout << ((half *)ptr)[s] << ' ';
                    ... handle other types ...
                }
                std::cout << "\n";
            }
        }
    }
}
in->close ();
delete in;

```

#### 4.2.8 Custom search paths for plugins

Please see Section 2.4.3 for discussion about setting the plugin search path via the `attribute()` function. For example:

```

std::string mysearch = "/usr/myapp/lib:${HOME}/plugins";
OpenImageIO::attribute ("plugin_searchpath", mysearch);
ImageInput *in = ImageInput::open (filename);
...

```

#### 4.2.9 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 message is accessible using the `ImageInput` method `geterror()`, which returns the message as a `std::string`.

The exceptions to this rule are static methods such as the static `ImageInput::open` and `ImageInput::create`, which return `NULL` if it could not create an appropriate `ImageInput` (and open it, in the case of `open()`). In such a case, since no `ImageInput` is returned for which you can call its `geterror()` function, there exists a global `geterror()` function (in the `OpenImageIO` namespace) that retrieves the latest error message resulting from a call to static `open()` or `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 <OpenImageIO/imageio.h>
OIIO_NAMESPACE_USING
...

const char *filename = "foo.jpg";
int xres, yres, channels;
std::vector<unsigned char> pixels;

ImageInput *in = ImageInput::open (filename);
if (! in) {
    std::cerr << "Could not open " << filename
               << ", error = " << OpenImageIO::geterror() << "\n";
    return;
}
const ImageSpec &spec = in->spec();
xres = spec.width;
yres = spec.height;
channels = spec.nchannels;
pixels.resize (xres*yres*channels);

if (! in->read_image (TypeDesc::UINT8, pixels)) {
    std::cerr << "Could not read pixels from " << filename
               << ", error = " << in->geterror() << "\n";
    delete in;
    return;
}

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

### 4.3 ImageInput Class Reference

```
ImageInput * open (const std::string &filename,
                  const ImageSpec *config=NULL)
```

Create an `ImageInput` subclass instance that is able to read the given file and open it, returning the opened `ImageInput` if successful. If it fails, return `NULL` and set an error that can be retrieved by `OpenImageIO::geterror()`.

The `config`, if not `NULL`, points to an `ImageSpec` giving requests or special instructions. `ImageInput` implementations are free to not respond to any such requests, so the default implementation is just to ignore `config`.

The `open()` function will first try to make an `ImageInput` corresponding to the format implied by the file extension (for example, "foo.tif" will try the TIFF plugin), but if one is not found or if the inferred one does not open the file, every known `ImageInput` type will be tried until one is found that will open the file.

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

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

No queries supported at this time.

```
bool valid_file (const std::string &filename) const
```

Return `true` if the named file is a file of the type for this `ImageInput`. The implementation will try to determine this as efficiently as possible, in most cases much less expensively than doing a full `open()`. Note that a file can appear to be of the right type (i.e., `valid_file()` returning `true`) but still fail a subsequent call to `open()`, such as if the contents of the file are truncated, nonsensical, or otherwise corrupted.

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

Opens the file with given name and seek to the first subimage in the file. 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`.

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

Opens the file with given name, similarly to `open(name, newspec)`. However, in this version, any non-default fields of `config`, including metadata, will be taken to be configuration requests, preferences, or hints. The default implementation of `open(name, newspec, config)` will simply ignore `config` and calls the usual `open(name, newspec)`. But a plugin may choose to implement this version of `open` and respond in some way to the configuration requests. Supported configuration requests should be documented by each plugin.

```
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 subimage, int miplevel, ImageSpec &newspec)
```

Seek to the given subimage and MIP-map level within the open image file. The first subimage in the file has index 0, and for each subimage, the highest-resolution MIP level has index 0. Return `true` on success, `false` on failure (including that there is not a subimage or MIP level with those indices). 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 and MIP levels — in other words, if it can't randomly seek to the given subimage or MIP level, it should transparently close, reopen, and sequentially read through prior subimages and levels.

```
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 ( $z = 0$  for non-volume images), converting if necessary from the native data format of the file into the format specified. If format is `TypeDesc::UNKNOWN`, the data will be preserved in its native format (including per-channel formats, if applicable). 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 * spec.pixel_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_scanlines (int ybegin, int yend, int z,
                    TypeDesc format, void *data,
                    stride_t xstride=AutoStride, stride_t ystride=AutoStride)
bool read_scanlines (int ybegin, int yend, int z,
                    int chbegin, int chend, TypeDesc format, void *data,
                    stride_t xstride=AutoStride, stride_t ystride=AutoStride)
```

Read all the scanlines that include pixels  $(*,y,z)$ , where  $ybegin \leq y < yend$ , into data. This is essentially identical to `read_scanline()`, except that can read more than one scanline at a time, which may be more efficient for certain image format readers.

The version that specifies a channel range will read only channels `[chbegin, chend)` into the buffer.

```
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 whose upper-left origin is  $(x,y,z)$  into data ( $z = 0$  for non-volume images), converting if necessary from the native data format of the file into the format specified. If format is `TypeDesc::UNKNOWN`, the data will be preserved in its native format (including per-channel formats, if applicable). 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 * spec.pixel_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.

This function returns `true` if it successfully reads the tile, otherwise `false` for a failure. The call will fail if the image is not tiled, or if  $(x,y,z)$  is not actually a tile boundary.

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

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

```
bool read_tiles (int xbegin, int xend, int ybegin, int yend,
                 int zbegin, int zend, TypeDesc format, void *data,
                 stride_t xstride=AutoStride, stride_t ystride=AutoStride,
                 stride_t zstride=AutoStride)
bool read_tiles (int xbegin, int xend, int ybegin, int yend,
                 int zbegin, int zend, int chbegin, int chend,
                 TypeDesc format, void *data,
                 stride_t xstride=AutoStride, stride_t ystride=AutoStride,
                 stride_t zstride=AutoStride)
```

Read the tiles bounded by  $xbegin \leq x < xend$ ,  $ybegin \leq y < yend$ ,  $zbegin \leq z < zend$  into data converting if necessary from the file's native data format into the specified buffer format. If format is `TypeDesc::UNKNOWN`, the data will be preserved in its native format (including per-channel formats, if applicable). 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 * spec.pixel_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_tiles()` and then does appropriate format conversion, so there's no reason for each format plugin to override this method.

This function returns `true` if it successfully reads the tiles, otherwise `false` for a failure. The call will fail if the image is not tiled, or if the pixel ranges do not fall along tile (or image) boundaries, or if it is not a valid tile range.

The version that specifies a channel range will read only channels  $[chbegin, chend)$  into the buffer.

```
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, converting into the desired data format. If format is `TypeDesc::UNKNOWN`, the data will be preserved in its native format (including per-channel formats, if applicable). This function will automatically handle either tiles or scanlines in the file.

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

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

The function will internally either call `read_scanlines` or `read_tiles`, 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 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_scanlines (int ybegin, int yend, int z, void *data)
```

The `read_native_scanlines()` function is just like `read_native_scanline`, except that it reads a range of scanlines rather than only one scanline. It is not necessary for format plugins to override this method — a default implementation in the `ImageInput` base class simply calls `read_native_scanline` for each scanline in the range. But format plugins may optionally override this method if there is a way to achieve higher performance by reading multiple scanlines at once.

```
bool read_native_scanlines (int ybegin, int yend, int z,
int chbegin, int chend, void *data)
```

A variant of `read_native_scanlines` that reads only a subset of channels `[chbegin, chend)`. If a format reader subclass does not override this method, the default implementation will simply call the all-channel version of `read_native_scanlines` into a temporary buffer and copy the subset of channels.

```
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.

```
bool read_native_tiles (int xbegin, int xend, int ybegin, int yend,
                        int zbegin, int zend, void *data)
```

The `read_native_tiles()` function is just like `read_tiles()`, except that it keeps the data in the native format of the disk file and always read into contiguous memory (no strides). If a format reader does not override this method, the default implementation it will simply be a loop calling `read_native_tile` for each tile in the block.

```
bool read_native_tiles (int xbegin, int xend, int ybegin, int yend,
                        int zbegin, int zend, int chbegin, int chend, void *data)
```

A variant of `read_native_tiles()` that reads only a subset of channels `[chbegin, chend)`. If a format reader subclass does not override this method, the default implementation will simply call the all-channel version of `read_native_tiles` into a temporary buffer and copy the subset of channels.

```
bool read_native_deep_scanlines (int ybegin, int yend, int z,
                                int chbegin, int chend, DeepData &deepdata)
```

```
bool read_native_deep_tiles (int xbegin, int xend, int ybegin, int yend,
                             int zbegin, int zend, int chbegin, int chend, DeepData &deepdata)
```

```
bool read_native_deep_image (DeepData &deepdata)
```

Read native deep data from scanlines, tiles, or an entire image, storing the results in `deepdata` (analogously to the usual `read_scanlines`, `read_tiles`, and `read_image`, but with deep data). Only channels `[chbegin, chend)` will be read.

```
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 geterror () 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 enclose the contents of your plugin in the same namespace that the OpenImageIO library uses:

```
#include <OpenImageIO/imageio.h>
OIIO_PLUGIN_NAMESPACE_BEGIN

... everything else ...

OIIO_PLUGIN_NAMESPACE_END
```

2. Declare three public items:

- (a) An integer called `name_imageio_version` that identifies the version of the ImageIO protocol implemented by the plugin, defined in `imageio.h` as the constant `OIIO_PLUGIN_VERSION`. This allows the library to be sure it is not loading a plugin that was compiled against an incompatible 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, and we provide handy macros `OIIO_PLUGIN_EXPORTS_BEGIN` and `OIIO_PLUGIN_EXPORTS_END` to make this easy. 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 `OIIO_EXPORT` macro for this purpose, defined in `export.h`.

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

```
OIIO_PLUGIN_EXPORTS_BEGIN
    OIIO_EXPORT int jpeg_imageio_version = OIIO_PLUGIN_VERSION;
    OIIO_EXPORT JpgInput *jpeg_input_imageio_create () {
        return new JpgInput;
    }
    OIIO_EXPORT const char *jpeg_input_extensions[] = {
        "jpg", "jpe", "jpeg", NULL
    };
OIIO_PLUGIN_EXPORTS_END
```

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) `supports()`, only if your format supports any of the optional features described in Section 4.3.

- (g) `valid_file()`, if your format has a way to determine if a file is of the given format in a way that is less expensive than a full `open()`.
- (h) `seek_subimage()`, only if your format supports reading multiple subimages within a single file.
- (i) `read_native_scanlines()`, only if your format has a speed advantage when reading multiple scanlines at once. If you do not supply this function, the default implementation will simply call `read_scanline()` for each scanline in the range.
- (j) `read_native_tile()`, only if your format supports reading tiled images.
- (k) `read_native_tiles()`, only if your format supports reading tiled images and there is a speed advantage when reading multiple tiles at once. If you do not supply this function, the default implementation will simply call `read_native_tile()` for each tile in the range.
- (l) “Channel subset” versions of `read_native_scanlines()` and/or `read_native_tiles()`, only if your format has a more efficient means of reading a subset of channels. If you do not supply these methods, the default implementation will simply use `read_native_scanlines()` or `read_native_tiles()` to read into a temporary all-channel buffer and then copy the channel subset into the user’s buffer.
- (m) `read_native_deep_scanlines()` and/or `read_native_deep_tiles()`, only if your format supports “deep” data 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 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; }
};
```

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 "OpenImageIO/imageio.h"
#include "OpenImageIO/filesystem.h"
#include "OpenImageIO/fmath.h"
#include "jpeg_pvt.h"

OIIO_PLUGIN_NAMESPACE_BEGIN

// N.B. The class definition for JpgInput is in jpeg_pvt.h.

// Export version number and create function symbols
OIIO_PLUGIN_EXPORTS_BEGIN

```



```

OIIO_EXPORT int jpeg_imageio_version = OIIO_PLUGIN_VERSION;
OIIO_EXPORT ImageInput *jpeg_input_imageio_create () {
    return new JpgInput;
}
OIIO_EXPORT const char *jpeg_input_extensions[] = {
    "jpg", "jpe", "jpeg", "jif", "jfif", ".jfi", NULL
};

OIIO_PLUGIN_EXPORTS_END

static const uint8_t JPEG_MAGIC1 = 0xff;
static const uint8_t JPEG_MAGIC2 = 0xd8;

// For explanations of the error handling, see the "example.c" in the
// libjpeg distribution.

static void
my_error_exit (j_common_ptr cinfo)
{
    /* cinfo->err really points to a my_error_mgr struct, so coerce pointer */
    JpgInput::my_error_ptr myerr = (JpgInput::my_error_ptr) cinfo->err;

    /* Always display the message. */
    /* We could postpone this until after returning, if we chose. */
    // (*cinfo->err->output_message) (cinfo);
    myerr->jpginput->jpegerror (myerr, true);

    /* Return control to the setjmp point */
    longjmp(myerr->setjmp_buffer, 1);
}

static void
my_output_message (j_common_ptr cinfo)
{
    JpgInput::my_error_ptr myerr = (JpgInput::my_error_ptr) cinfo->err;
    myerr->jpginput->jpegerror (myerr, true);
}

void
JpgInput::jpegerror (my_error_ptr myerr, bool fatal)
{
    // Send the error message to the ImageInput
    char errbuf[JMSG_LENGTH_MAX];

```

```

    (*m_cinfo.err->format_message) ((j_common_ptr)&m_cinfo, errbuf);
    error ("JPEG error: %s (%s)", errbuf, filename().c_str());

    // Shut it down and clean it up
    if (fatal) {
        m_fatalerr = true;
        close ();
        m_fatalerr = true;    // because close() will reset it
    }
}

bool
JpgInput::valid_file (const std::string &filename) const
{
    FILE *fd = Filesystem::fopen (filename, "rb");
    if (! fd)
        return false;

    // Check magic number to assure this is a JPEG file
    uint8_t magic[2] = {0, 0};
    bool ok = (fread (magic, sizeof(magic), 1, fd) == 1);
    fclose (fd);

    if (magic[0] != JPEG_MAGIC1 || magic[1] != JPEG_MAGIC2) {
        ok = false;
    }
    return ok;
}

bool
JpgInput::open (const std::string &name, ImageSpec &newspec,
                const ImageSpec &config)
{
    const ImageIOParameter *p = config.find_attribute ("_jpeg:raw",
                                                       TypeDesc::TypeInt);

    m_raw = p && *(int *)p->data();
    return open (name, newspec);
}

bool
JpgInput::open (const std::string &name, ImageSpec &newspec)
{
    // Check that file exists and can be opened
    m_filename = name;
    m_fd = Filesystem::fopen (name, "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
uint8_t magic[2] = {0, 0};
if (fread (magic, sizeof(magic), 1, m_fd) != 1) {
    error ("Empty file \"%s\"", name.c_str());
    close_file ();
    return false;
}

rewind (m_fd);
if (magic[0] != JPEG_MAGIC1 || magic[1] != JPEG_MAGIC2) {
    close_file ();
    error ("%s\" is not a JPEG file, magic number doesn't match (was 0x%x)",
        name.c_str(), magic);
    return false;
}

// Set up the normal JPEG error routines, then override error_exit and
// output_message so we intercept all the errors.
m_cinfo.err = jpeg_std_error ((jpeg_error_mgr *)&m_jerr);
m_jerr.pub.error_exit = my_error_exit;
m_jerr.pub.output_message = my_output_message;
if (setjmp (m_jerr.setjmp_buffer)) {
    // Jump to here if there's a libjpeg internal error
    // Prevent memory leaks, see example.c in jpeg distribution
    jpeg_destroy_decompress (&m_cinfo);
    close_file ();
    return false;
}

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
for (int mark = 0; mark < 16; ++mark)
    jpeg_save_markers (&m_cinfo, JPEG_APP0+mark, 0xffff);
jpeg_save_markers (&m_cinfo, JPEG_COM, 0xffff);    // comment marker

// read the file parameters
if (jpeg_read_header (&m_cinfo, FALSE) != JPEG_HEADER_OK || m_fatalerr) {
    error ("Bad JPEG header for \"%s\"", filename().c_str());
    return false;
}
if (m_raw)
    m_coeffs = jpeg_read_coefficients (&m_cinfo);
else
    jpeg_start_decompress (&m_cinfo);          // start working

```

```

    if (m_fatalerr)
        return false;
    m_next_scanline = 0; // next scanline we'll read

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

    // Assume JPEG is in sRGB unless the Exif or XMP tags say otherwise.
    m_spec.attribute ("oio:ColorSpace", "sRGB");

    for (jpeg_saved_marker_ptr m = m_cinfo.marker_list; m; m = m->next) {
        if (m->marker == (JPEG_APP0+1) &&
            ! strcmp ((const char *)m->data, "Exif")) {
            // The block starts with "Exif\0\0", so skip 6 bytes to get
            // to the start of the actual Exif data TIFF directory
            decode_exif ((unsigned char *)m->data+6, m->data_length-6, m_spec);
        }
        else if (m->marker == (JPEG_APP0+1) &&
            ! strcmp ((const char *)m->data, "http://ns.adobe.com/xap/1.0/")) {
#ifdef NDEBBUG
            std::cerr << "Found APP1 XMP! length " << m->data_length << "\n";
#endif
            std::string xml ((const char *)m->data, m->data_length);
            decode_xmp (xml, m_spec);
        }
        else if (m->marker == (JPEG_APP0+13) &&
            ! strcmp ((const char *)m->data, "Photoshop 3.0"))
            jpeg_decode_iptc ((unsigned char *)m->data);
        else if (m->marker == JPEG_COM) {
            if (! m_spec.find_attribute ("ImageDescription", TypeDesc::STRING))
                m_spec.attribute ("ImageDescription",
                                   std::string ((const char *)m->data));
        }
    }

    newspec = m_spec;
    return true;
}

bool
JpgInput::read_native_scanline (int y, int z, void *data)
{
    if (m_raw)
        return false;
    if (y < 0 || y >= (int)m_cinfo.output_height) // out of range scanline
        return false;
    if (m_next_scanline > y) {
        // User is trying to read an earlier scanline than the one we're
        // up to. Easy fix: close the file and re-open.
    }
}

```

```

    ImageSpec dummyspec;
    int subimage = current_subimage();
    if (! close () ||
        ! open (m_filename, dummyspec) ||
        ! seek_subimage (subimage, 0, dummyspec))
        return false;    // Somehow, the re-open failed
    assert (m_next_scanline == 0 && current_subimage() == subimage);
}

// Set up our custom error handler
if (setjmp (m_jerr.setjmp_buffer)) {
    // Jump to here if there's a libjpeg internal error
    return false;
}

for ( ; m_next_scanline <= y; ++m_next_scanline) {
    // Keep reading until we've read the scanline we really need
    if (jpeg_read_scanlines (&m_cinfo, (JSAMPLE **)&data, 1) != 1
        || m_fatalerr) {
        error ("JPEG failed scanline read (\"%s\")", filename().c_str());
        return false;
    }
}

return true;
}

bool
JpgInput::close ()
{
    if (m_fd != NULL) {
        // unnecessary?  jpeg_abort_decompress (&m_cinfo);
        jpeg_destroy_decompress (&m_cinfo);
        close_file ();
    }
    init ();    // Reset to initial state
    return true;
}

void
JpgInput::jpeg_decode_iptc (const unsigned char *buf)
{
    // APP13 blob doesn't have to be IPTC info.  Look for the IPTC marker,
    // which is the string "Photoshop 3.0" followed by a null character.
    if (strcmp ((const char *)buf, "Photoshop 3.0"))
        return;
    buf += strlen("Photoshop 3.0") + 1;
}

```

```

// Next are the 4 bytes "8BIM"
if (strncmp ((const char *)buf, "8BIM", 4))
    return;
buf += 4;

// Next two bytes are the segment type, in big endian.
// We expect 1028 to indicate IPTC data block.
if (((buf[0] << 8) + buf[1]) != 1028)
    return;
buf += 2;

// Next are 4 bytes of 0 padding, just skip it.
buf += 4;

// Next is 2 byte (big endian) giving the size of the segment
int segmentsize = (buf[0] << 8) + buf[1];
buf += 2;

decode_iptc_iim (buf, segmentsize, m_spec);
}

OIIO_PLUGIN_NAMESPACE_END

```

### 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 `name_imageio_version` that identifies the version of the ImageIO protocol implemented by the plugin, defined in `imageio.h` as the constant `OIIO_PLUGIN_VERSION`. This allows the library to be sure it is not loading a plugin that was compiled against an incompatible 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, and we provide handy macros `OIIO_PLUGIN_EXPORTS_BEGIN` and `OIIO_PLUGIN_EXPORTS_END` to make this easy. 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 `OIIO_EXPORT` macro for this purpose, defined in `export.h`.

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

```
OIIO_PLUGIN_EXPORTS_BEGIN
    OIIO_EXPORT int jpeg_imageio_version = OIIO_PLUGIN_VERSION;
    OIIO_EXPORT JpgOutput *jpeg_output_imageio_create () {
        return new JpgOutput;
    }
    OIIO_EXPORT const char *jpeg_input_extensions[] = {
        "jpg", "jpe", "jpeg", NULL
    };
OIIO_PLUGIN_EXPORTS_END
```

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_scanlines()`, only if your format supports writing scanlines and you can get a performance improvement when outputting multiple scanlines at once. If you don’t supply `write_scanlines()`, the default implementation will simply call `write_scanline()` separately for each scanline in the range.

- (h) `write_tile()`, only if your format supports writing tiled images.
- (i) `write_tiles()`, only if your format supports writing tiled images and you can get a performance improvement when outputting multiple tiles at once. If you don't supply `write_tiles()`, the default implementation will simply call `write_tile()` separately for each tile in the range.
- (j) `write_rectangle()`, only if your format supports writing arbitrary rectangles.
- (k) `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.
- (l) `write_deep_scanlines()` and/or `write_deep_tiles()`, only if your format supports "deep" data images.

It is not strictly required, but certainly appreciated, if a file format does not support tiles, to nonetheless accept an `ImageSpec` that specifies tile sizes by allocating a full-image buffer in `open()`, providing an implementation of `write_tile()` that copies the tile of data to the right spots in the buffer, and having `close()` then call `write_scanlines` to process the buffer now that the image has been fully sent.

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 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 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` that convert a scanline, tile, or rectangular array of values from one format to the native format(s) of the file.

```
const void * to_native_scanline (TypeDesc format, const void *data,
                                   stride_t xstride, std::vector<unsigned char> &scratch,
                                   unsigned int dither=0, int yorigin=0, int zorigin=0)
```

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, unsigned int dither=0,
                               int xorigin=0, int yorigin=0, int zorigin=0)
```

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 xbegin, int xend, int ybegin, int
yend,
                                   int zbegin, int zend, TypeDesc format, const void *data,
                                   stride_t xstride, stride_t ystride, stride_t zstride,
                                   std::vector<unsigned char> &scratch, unsigned int dither=0,
                                   int xorigin=0, int yorigin=0, int zorigin=0)
```

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).

For float to 8 bit integer conversions only, if `dither` parameter is nonzero, random dither will be added to reduce quantization banding artifacts; in this case, the specific nonzero `dither` value is used as a seed for the hash function that produces the per-pixel dither amounts, and the optional `origin` parameters help it to align the pixels to the right position in the dither pattern.

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 "OpenImageIO/imageio.h"
#include "OpenImageIO/filesystem.h"
```

```

#include "OpenImageIO/fmath.h"
#include "jpeg_pvt.h"

OIIO_PLUGIN_NAMESPACE_BEGIN

#define DBG if(0)

// 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,
                      OpenMode mode=Create);
    virtual bool write_scanline (int y, int z, TypeDesc format,
                                const void *data, stride_t xstride);
    virtual bool write_tile (int x, int y, int z, TypeDesc format,
                             const void *data, stride_t xstride,
                             stride_t ystride, stride_t zstride);
    virtual bool close ();
    virtual bool copy_image (ImageInput *in);

private:
    FILE *m_fd;
    std::string m_filename;
    unsigned int m_dither;
    int m_next_scanline;           // Which scanline is the next to write?
    std::vector<unsigned char> m_scratch;
    struct jpeg_compress_struct m_cinfo;
    struct jpeg_error_mgr c_jerr;
    jvirt_barray_ptr *m_copy_coeffs;
    struct jpeg_decompress_struct *m_copy_decompressor;
    std::vector<unsigned char> m_tilebuffer;

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

OIIO_PLUGIN_EXPORTS_BEGIN

```

```

OIIO_EXPORT ImageOutput *jpeg_output_imageio_create () {
    return new JpgOutput;
}
OIIO_EXPORT const char *jpeg_output_extensions[] = {
    "jpg", "jpe", "jpeg", "jif", "jfif", "jfi", NULL
};

OIIO_PLUGIN_EXPORTS_END

bool
JpgOutput::open (const std::string &name, const ImageSpec &newspec,
                 OpenMode mode)
{
    if (mode != Create) {
        error ("%s does not support subimages or MIP levels", format_name());
        return false;
    }

    // Save name and spec for later use
    m_filename = name;
    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;
    }

    if (m_spec.nchannels != 1 && m_spec.nchannels != 3 &&
        m_spec.nchannels != 4) {
        error ("%s does not support %d-channel images\n",
              format_name(), m_spec.nchannels);
        return false;
    }

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

    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;
} 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;

if (m_copy_coeffs) {
    // Back door for copy()
    jpeg_copy_critical_parameters (m_copy_decompressor, &m_cinfo);
    DBG std::cout << "out open: copy_critical_parameters\n";
    jpeg_write_coefficients (&m_cinfo, m_copy_coeffs);
    DBG std::cout << "out open: write_coefficients\n";
} else {
    // normal write of scanlines
    jpeg_set_defaults (&m_cinfo);                // default compression
    DBG std::cout << "out open: set_defaults\n";
    int quality = newspec.get_int_attribute ("CompressionQuality", 98);
    jpeg_set_quality (&m_cinfo, quality, TRUE);    // baseline values
    DBG std::cout << "out open: set_quality\n";
    jpeg_start_compress (&m_cinfo, TRUE);         // start working
    DBG std::cout << "out open: start_compress\n";
}
m_next_scanline = 0;    // next scanline we'll write

// Write JPEG comment, if sent an 'ImageDescription'
ImageIOParameter *comment = m_spec.find_attribute ("ImageDescription",
                                                    TypeDesc::STRING);

if (comment && comment->data()) {
    const char **c = (const char **) comment->data();
    jpeg_write_marker (&m_cinfo, JPEG_COM, (JOCTET*)*c, strlen(*c) + 1);
}

if (Strutil::iequals (m_spec.get_string_attribute ("oio:ColorSpace"), "sRGB"))
    m_spec.attribute ("Exif:ColorSpace", 1);

// Write EXIF info
std::vector<char> exif;
// Start the blob with "Exif" and two nulls. That's how it

```

```

// always is in the JPEG files I've examined.
exif.push_back ('E');
exif.push_back ('x');
exif.push_back ('i');
exif.push_back ('f');
exif.push_back (0);
exif.push_back (0);
encode_exif (m_spec, exif);
jpeg_write_marker (&m_cinfo, JPEG_APP0+1, (JOCTET*)&exif[0], exif.size());

// Write IPTC IIM metadata tags, if we have anything
std::vector<char> iptc;
encode_iptc_iim (m_spec, iptc);
if (iptc.size()) {
    static char photoshop[] = "Photoshop 3.0";
    std::vector<char> head (photoshop, photoshop+strlen(photoshop)+1);
    static char _8BIM[] = "8BIM";
    head.insert (head.end(), _8BIM, _8BIM+4);
    head.push_back (4);    // 0x0404
    head.push_back (4);
    head.push_back (0);    // four bytes of zeroes
    head.push_back (0);
    head.push_back (0);
    head.push_back (0);
    head.push_back ((char)(iptc.size() >> 8)); // size of block
    head.push_back ((char)(iptc.size() & 0xff));
    iptc.insert (iptc.begin(), head.begin(), head.end());
    jpeg_write_marker (&m_cinfo, JPEG_APP0+13, (JOCTET*)&iptc[0], iptc.size());
}

// Write XMP packet, if we have anything
std::string xmp = encode_xmp (m_spec, true);
if (! xmp.empty()) {
    static char prefix[] = "http://ns.adobe.com/xap/1.0/";
    std::vector<char> block (prefix, prefix+strlen(prefix)+1);
    block.insert (block.end(), xmp.c_str(), xmp.c_str()+xmp.length());
    jpeg_write_marker (&m_cinfo, JPEG_APP0+1, (JOCTET*)&block[0], block.size());
}

m_spec.set_format (TypeDesc::UINT8); // JPG is only 8 bit
m_dither = m_spec.get_int_attribute ("oiio:dither", 0);

// If user asked for tiles -- which JPEG doesn't support, emulate it by
// buffering the whole image.
if (m_spec.tile_width && m_spec.tile_height)
    m_tilebuffer.resize (m_spec.image_bytes());

return true;
}

```

```

bool
JpgOutput::write_scanline (int y, int z, TypeDesc format,
                           const void *data, stride_t xstride)
{
    y -= m_spec.y;
    if (y != m_next_scanline) {
        error ("Attempt to write scanlines out of order to %s",
              m_filename.c_str());
        return false;
    }
    if (y >= (int)m_cinfo.image_height) {
        error ("Attempt to write too many scanlines to %s", m_filename.c_str());
        return false;
    }
    assert (y == (int)m_cinfo.next_scanline);

    // It's so common to want to write RGBA data out as JPEG (which only
    // supports RGB) than it would be too frustrating to reject it.
    // Instead, we just silently drop the alpha. Here's where we do the
    // dirty work, temporarily doctoring the spec so that
    // to_native_scanline properly contiguizes the first three channels,
    // then we restore it. The call to to_native_scanline below needs
    // m_spec.nchannels to be set to the true number of channels we're
    // writing, or it won't arrange the data properly. But if we
    // doctored m_spec.nchannels = 3 permanently, then subsequent calls
    // to write_scanline (including any surrounding call to write_image)
    // with stride=AutoStride would screw up the strides since the
    // user's stride is actually not 3 channels.
    int save_nchannels = m_spec.nchannels;
    m_spec.nchannels = m_cinfo.input_components;

    data = to_native_scanline (format, data, xstride, m_scratch,
                              m_dither, y, z);
    m_spec.nchannels = save_nchannels;

    jpeg_write_scanlines (&m_cinfo, (JSAMPLE**) &data, 1);
    ++m_next_scanline;

    return true;
}

bool
JpgOutput::write_tile (int x, int y, int z, TypeDesc format,
                       const void *data, stride_t xstride,
                       stride_t ystride, stride_t zstride)
{
    // Emulate tiles by buffering the whole image
    return copy_tile_to_image_buffer (x, y, z, format, data, xstride,

```

```

        ystride, zstride, &m_tilebuffer[0]);
    }

bool
JpgOutput::close ()
{
    if (! m_fd) {          // Already closed
        return true;
        init();
    }

    bool ok = true;

    if (m_spec.tile_width) {
        // We've been emulating tiles; now dump as scanlines.
        ASSERT (m_tilebuffer.size());
        ok &= write_scanlines (m_spec.y, m_spec.y+m_spec.height, 0,
                               m_spec.format, &m_tilebuffer[0]);
        std::vector<unsigned char>().swap (m_tilebuffer); // free it
    }

    if (m_next_scanline < spec().height && m_copy_coeffs == NULL) {
        // But if we've only written some scanlines, write the rest to avoid
        // errors
        std::vector<char> buf (spec().scanline_bytes(), 0);
        char *data = &buf[0];
        while (m_next_scanline < spec().height) {
            jpeg_write_scanlines (&m_cinfo, (JSAMPLE **)&data, 1);
            // DBG std::cout << "out close: write_scanlines\n";
            ++m_next_scanline;
        }
    }

    if (m_next_scanline >= spec().height || m_copy_coeffs) {
        DBG std::cout << "out close: about to finish_compress\n";
        jpeg_finish_compress (&m_cinfo);
        DBG std::cout << "out close: finish_compress\n";
    } else {
        DBG std::cout << "out close: about to abort_compress\n";
        jpeg_abort_compress (&m_cinfo);
        DBG std::cout << "out close: abort_compress\n";
    }

    DBG std::cout << "out close: about to destroy_compress\n";
    jpeg_destroy_compress (&m_cinfo);
    fclose (m_fd);
    m_fd = NULL;
    init();

    return ok;
}

```



```

}

bool
JpgOutput::copy_image (ImageInput *in)
{
    if (in && !strcmp(in->format_name(), "jpeg")) {
        JpgInput *jpg_in = dynamic_cast<JpgInput *> (in);
        std::string in_name = jpg_in->filename ();
        DBG std::cout << "JPG copy_image from " << in_name << "\n";

        // Save the original input spec and close it
        ImageSpec orig_in_spec = in->spec();
        in->close ();
        DBG std::cout << "Closed old file\n";

        // Re-open the input spec, with special request that the JpgInput
        // will recognize as a request to merely open, but not start the
        // decompressor.
        ImageSpec in_spec;
        ImageSpec config_spec;
        config_spec.attribute ("_jpeg:raw", 1);
        in->open (in_name, in_spec, config_spec);

        // Re-open the output
        std::string out_name = m_filename;
        ImageSpec orig_out_spec = spec();
        close ();
        m_copy_coeffs = (jvirt_barray_ptr *)jpg_in->coeffs();
        m_copy_decompressor = &jpg_in->m_cinfo;
        open (out_name, orig_out_spec);

        // Strangeness -- the write_coefficients somehow sets things up
        // so that certain writes only happen in close(), which MUST
        // happen while the input file is still open. So we go ahead
        // and close() now, so that the caller of copy_image() doesn't
        // close the input file first and then wonder why they crashed.
        close ();

        return true;
    }

    return ImageOutput::copy_image (in);
}

OIIO_PLUGIN_NAMESPACE_END

```

## 5.4 Tips and Conventions

OpenImageIO's main goal is to hide all the pesky details of individual file formats from the client application. This inevitably leads to various mismatches between a file format's true capabilities and requests that may be made through the OpenImageIO APIs. This section outlines conventions, tips, and rules of thumb that we recommend for image file support.

### Readers

- If the file format stores images in a non-spectral color space (for example, YUV), the reader should automatically convert to RGB to pass through the OIIO APIs. In such a case, the reader should signal the file's true color space via a "Foo:colorspace" attribute in the ImageSpec.
- "Palette" images should be automatically converted by the reader to RGB.
- If the file supports thumbnail images in its header, the reader should store the thumbnail dimensions in attributes "thumbnail\_width", "thumbnail\_height", and "thumbnail\_nchannels" (all of which should be int), and the thumbnail pixels themselves in "thumbnail\_image" as an array of channel values (the array length is the total number of channel samples in the thumbnail).

### Writers

The overall rule of thumb is: try to always "succeed" at writing the file, outputting the closest approximation of the user's data as possible. But it is permissible to fail the `open()` call if it is clearly nonsensical or there is no possible way to output a decent approximation of the user's data. Some tips:

- If the client application requests a data format not directly supported by the file type, silently write the supported data format that will result in the least precision or range loss.
- It is customary to fail a call to `open()` if the ImageSpec requested a number of color channels plainly not supported by the file format. As an exception to this rule, it is permissible for a file format that does not support alpha channels to silently drop the fourth (alpha) channel of a 4-channel output request.
- If the app requests a "Compression" not supported by the file format, you may choose as a default any lossless compression supported. Do not use a lossy compression unless you are fairly certain that the app wanted a lossy compression.
- If the file format is able to store images in a non-spectral color space (for example, YUV), the writer may accept a "Foo:colorspace" attribute in the ImageSpec as a request to automatically convert and store the data in that format (but it will always be passed as RGB through the OIIO APIs).
- If the file format can support thumbnail images in its header, and the ImageSpec contain attributes "thumbnail\_width", "thumbnail\_height", "thumbnail\_nchannels", and "thumbnail\_image", the writer should attempt to store the thumbnail if possible.

## **5.5 Building ImageIO Plugins**

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



## 6 Bundled ImageIO Plugins

This chapter lists all the image format plugins that are bundled with OpenImageIO. For each plugin, we delineate any limitations, custom attributes, etc. The plugins are listed alphabetically by format name.

### 6.1 BMP

BMP is a bitmap image file format used mostly on Windows systems. BMP files use the file extension `.bmp`.

BMP is not a nice format for high-quality or high-performance images. It only supports unsigned integer 1-, 2-, 4-, and 8- bits per channel; only grayscale, RGB, and RGBA; does not support MIPmaps, multiimage, or tiles.

ImageSpec Attribute	Type	BMP header data or explanation
"XResolution"	float	hres
"YResolution"	float	vres
"ResolutionUnit"	string	always "m" (pixels per meter)

### 6.2 Cineon

Cineon is an image file format developed by Kodak that is commonly used for scanned motion picture film and digital intermediates. Cineon files use the file extension `.cin`.

### 6.3 DDS

DDS (Direct Draw Surface) is an image file format designed by Microsoft for use in Direct3D graphics. DDS files use the extension `.dds`.

DDS is an awful format, with several compression modes that are all so lossy as to be completely useless for high-end graphics. Nevertheless, they are widely used in games and graphics hardware directly supports these compression modes. Alas.

OpenImageIO currently only supports reading DDS files, not writing them.

ImageSpec Attribute	Type	DDS header data or explanation
"compression"	string	compression type
"oio:BitsPerSample"	int	bits per sample
"textureformat"	string	Set correctly to one of "Plain Texture", "Volume Texture", or "CubeFace Environment".
"texturetype"	string	Set correctly to one of "Plain Texture", "Volume Texture", or "Environment".
"dds:CubeMapSides"	string	For environment maps, which cube faces are present (e.g., "+x -x +y -y" if x & y faces are present, but not z).

## 6.4 DPX

DPX (Digital Picture Exchange) is an image file format used for motion picture film scanning, output, and digital intermediates. DPX files use the file extension .dpx.

OIO Attribute	Type	DPX header data or explanation
"ImageDescription"	string	Description of image element
"Copyright"	string	Copyright statement
"Software"	string	Creator
"DocumentName"	string	Project name
"DateTime"	string	Creation date/time
"Orientation"	int	the orientation of the DPX image data (see B.2)
"compression"	string	The compression type
"PixelAspectRatio"	float	pixel aspect ratio
"oio:BitsPerSample"	int	the true bits per sample of the DPX file.
"oio:Endian"	string	When writing, force a particular endianness for the output file ("little" or "big")
"smpte:TimeCode"	int[2]	SMPTE time code (vecsemantics will be marked as TIMECODE)
"smpte:KeyCode"	int[7]	SMPTE key code (vecsemantics will be marked as KEYCODE)

OIO Attribute	Type	DPX header data or explanation
"dpx:Transfer"	string	Transfer characteristic
"dpx:Colorimetric"	string	Colorimetric specification
"dpx:ImageDescriptor"	string	ImageDescriptor
"dpx:Packing"	string	Image packing method
"dpx:TimeCode"	int	SMPTE time code
"dpx:UserBits"	int	SMPTE user bits
"dpx:SourceDateTime"	string	source time and date
"dpx:FilmEdgeCode"	string	FilmEdgeCode
"dpx:Signal"	string	Signal ("Undefined", "NTSC", "PAL", etc.)
"dpx:UserData"	UCHAR[*]	User data (stored in an array whose length is whatever it was in the DPX file)
"dpx:EncryptKey"	int	Encryption key (-1 is not encrypted)
"dpx:DittoKey"	int	Ditto (0 = same as previous frame, 1 = new)
"dpx:LowData"	int	reference low data code value
"dpx:LowQuantity"	float	reference low quantity
"dpx:HighData"	int	reference high data code value
"dpx:HighQuantity"	float	reference high quantity
"dpx:XScannedSize"	float	X scanned size
"dpx:YScannedSize"	float	Y scanned size
"dpx:FramePosition"	int	frame position in sequence
"dpx:SequenceLength"	int	sequence length (frames)
"dpx:HeldCount"	int	held count (1 = default)
"dpx:FrameRate"	float	frame rate of original (frames/s)
"dpx:ShutterAngle"	float	shutter angle of camera (deg)
"dpx:Version"	string	version of header format
"dpx:Format"	string	format (e.g., "Academy")
"dpx:FrameId"	string	frame identification
"dpx:SlateInfo"	string	slate information
"dpx:SourceImageFileName"	string	source image filename
"dpx:InputDevice"	string	input device name
"dpx:InputDeviceSerialNumber"	string	input device serial number
"dpx:Interlace"	int	interlace (0 = noninterlace, 1 = 2:1 interlace)
"dpx:FieldNumber"	int	field number
"dpx:HorizontalSampleRate"	float	horizontal sampling rate (Hz)
"dpx:VerticalSampleRate"	float	vertical sampling rate (Hz)
"dpx:TemporalFrameRate"	float	temporal sampling rate (Hz)
"dpx:TimeOffset"	float	time offset from sync to first pixel (ms)
"dpx:BlackLevel"	float	black level code value
"dpx:BlackGain"	float	black gain
"dpx:BreakPoint"	float	breakpoint
"dpx:WhiteLevel"	float	reference white level code value
"dpx:IntegrationTimes"	float	integration time (s)

## 6.5 Field3D

Field3d is an open-source volume data file format. Field3d files commonly use the extension `.f3d`. The official Field3D site is: <http://sites.google.com/site/field3d/> Currently, OpenImageIO only reads Field3d files, and does not write them.

Fields are comprised of multiple *layers* (which appear to OpenImageIO as subimages). Each layer/subimage may have a different name, resolution, and coordinate mapping. Layers may be scalar (1 channel) or vector (3 channel) fields, and the data may be half, float, or double.

OpenImageIO always reports Field3D files as tiled. If the Field3d file has a “block size”, the block size will be reported as the tile size. Otherwise, the tile size will be the size of the entire volume.

ImageSpec Attribute	Type	Field3d header data or explanation
"ImageDescription"	string	unique layer name
"oio:subimagename"	string	unique layer name
"field3d:partition"	string	the partition name
"field3d:layer"	string	the layer (a.k.a. attribute) name
"field3d:fieldtype"	string	field type, one of: "dense", "sparse", or "MAC"
"field3d:mapping"	string	the coordinate mapping type
"field3d:localtoworld"	matrix of doubles	if a matrixMapping, the local-to-world transformation matrix
"worldtocamera"	matrix	if a matrixMapping, the world-to-local coordinate mapping

The “unique layer name” is generally the partition name + “:” + attribute name (example: “defaultfield:density”), with the following exceptions: (1) if the partition and attribute names are identical, just one is used rather than it being pointlessly concatenated (e.g., “density”, not “density:density”); (2) if there are multiple partitions + attribute combinations with identical names in the same file, “*number*” will be added after the partition name for subsequent layers (e.g., “default:density”, “default.2:density”, “default.3:density”).

## 6.6 FITS

FITS (Flexible Image Transport System) is an image file format used for scientific applications, particularly professional astronomy. FITS files use the file extension `.fits`. Official FITS specs and other info may be found at: <http://fits.gsfc.nasa.gov/>

OpenImageIO supports multiple images in FITS files, and supports the following pixel data types: UINT8, UINT16, UINT32, FLOAT, DOUBLE.

FITS files can store various kinds of arbitrary data arrays, but OpenImageIO’s support of FITS is mostly limited using FITS for image storage. Currently, OpenImageIO only supports 2D FITS data (images), not 3D (volume) data, nor 1-D or higher-dimensional arrays.



ImageSpec Attribute	Type	FITS header data or explanation
"Orientation"	int	derived from FITS "ORIENTAT" field.
"DateTime"	string	derived from the FITS "DATE" field.
"Comment"	string	FITS "COMMENT" (*)
"History"	string	FITS "HISTORY" (*)
"Hierarch"	string	FITS "HIERARCH" (*)
<i>other</i>		all other FITS keywords will be added to the ImageSpec as arbitrary named metadata.

(\*) Note: If the file contains multiple COMMENT, HISTORY, or HIERARCH fields, their text will be appended to form a single attribute (of each) in OpenImageIO's ImageSpec.

## 6.7 GIF

GIF (Graphics Interchange Format) is an image file format developed by CompuServe in 1987. Nowadays it is widely used to display basic animations despite its technical limitations.

ImageSpec Attribute	Type	GIF header data or explanation
"gif:DelayMs"	int	Delay between frames in milliseconds.
"gif:Interlacing"	int	Specifies if image is interlaced (0 or 1).
"gif:LoopCount"	int	Number of times the animation should be played (0–65535, 0 stands for infinity).
"ImageDescription"	string	The GIF comment field.

### Limitations

- GIF only supports 3-channel (RGB) images and at most 8 bits per channel.
- Each subimage can include its own palette or use global palette. Palettes contain up to 256 colors of which one can be used as background color. It is then emulated with additional Alpha channel by OpenImageIO's reader.

## 6.8 HDR/RGBE

HDR (High Dynamic Range), also known as RGBE (rgb with extended range), is a simple format developed for the Radiance renderer to store high dynamic range images. HDR/RGBE files commonly use the file extensions .hdr. The format is described in this section of the Radiance documentation: <http://radsite.lbl.gov/radiance/refer/filefmts.pdf>

RGBE does not support tiles, multiple subimages, mipmapping, true half or float pixel values, or arbitrary metadata. Only RGB (3 channel) files are supported.

RGBE became important because it was developed at a time when no standard file formats supported high dynamic range, and is still used for many legacy applications and to distribute

HDR environment maps. But newer formats with native HDR support, such as OpenEXR, are vastly superior and should be preferred except when legacy file access is required.

ImageSpec Attribute	Type	RGBE header data or explanation
"Orientation"	int	encodes the orientation (see B.2)
ImageSpec.gamma	float	the gamma correction specified in the RGBE header.

## 6.9 ICO

ICO is an image file format used for small images (usually icons) on Windows. ICO files use the file extension `.ico`.

ImageSpec Attribute	Type	ICO header data or explanation
"oiio:BitsPerSample"	int	the true bits per sample in the ICO file.
"ico:PNG"	int	if nonzero, will cause the ICO to be written out using PNG format.

### Limitations

- ICO only supports UINT8 and UINT16 formats; all output images will be silently converted to one of these.
- ICO only supports *small* images, up to  $256 \times 256$ . Requests to write larger images will fail their `open()` call.

## 6.10 IFF

IFF files are used by Autodesk Maya and use the file extension `.iff`.

OIO Attribute	Type	DPX header data or explanation
"Artist"	string	The IFF "author"
"DateTime"	string	Creation date/time
"compression"	string	The compression type
"oiio:BitsPerSample"	int	the true bits per sample of the DPX file.

## 6.11 JPEG

JPEG (Joint Photographic Experts Group), or more properly the JFIF file format containing JPEG-compressed pixel data, is one of the most popular file formats on the Internet, with applications, and from digital cameras, scanners, and other image acquisition devices. JPEG/JFIF

files usually have the file extension .jpg, .jpe, .jpeg, .jif, .jfif, or .jfi. The JFIF file format is described by <http://www.w3.org/Graphics/JPEG/jfif3.pdf>.

Although we strive to support JPEG/JFIF because it is so widely used, we acknowledge that it is a poor format for high-end work: it supports only 1- and 3-channel images, has no support for alpha channels, no support for high dynamic range or even 16 bit integer pixel data, by convention stores sRGB data and is ill-suited to linear color spaces, and does not support multiple subimages or MIPmap levels. There are newer formats also blessed by the Joint Photographic Experts Group that attempt to address some of these issues, such as JPEG-2000, but these do not have anywhere near the acceptance of the original JPEG/JFIF format.

ImageSpec Attribute	Type	JPEG header data or explanation
"ImageDescription"	string	the JPEG Comment field
"Orientation"	int	the image orientation
"XResolution", "YResolution", "ResolutionUnit"		The resolution and units from the Exif header
Exif, IPTC, XMP, GPS		Extensive Exif, IPTC, XMP, and GPS data are supported by the reader/writer, and you should assume that nearly everything described Appendix B is properly translated when using JPEG files.

### Limitations

- JPEG/JFIF only supports 1- (grayscale) and 3-channel (RGB) images. As a special case, OpenImageIO's JPEG writer will accept 4-channel image data and silently drop the alpha channel while outputting. Other channel count requests (i.e., anything other than 1, 3, and 4) will cause `open()` to fail, since it is not possible to write a JFIF file with other than 1 or 3 channels.
- Since JPEG/JFIF only supports 8 bits per channel, OpenImageIO's JPEG/JFIF writer will silently convert to UINT8 upon output, regardless of requests to the contrary from the calling program.
- OpenImageIO's JPEG/JFIF reader and writer always operate in scanline mode and do not support tiled image input or output.

## 6.12 JPEG-2000

JPEG-2000 is a successor to the popular JPEG/JFIF format, that supports better (wavelet) compression and a number of other extensions. It's geared toward photography. JPEG-2000 files use the file extensions .jp2 or .j2k. The official JPEG-2000 format specification and other helpful info may be found at <http://www.jpeg.org/JPEG2000.htm>.

JPEG-2000 is not yet widely used, so OpenImageIO's support of it is preliminary. In particular, we are not yet very good at handling the metadata robustly.

ImageSpec Attribute	Type	JPEG-2000 header data or explanation
"jpeg2000:streamformat"	string	specifies the JPEG-2000 stream format ("none" or "jpc")

### 6.13 OpenEXR

OpenEXR is an image file format developed by Industrial Light & Magic, and subsequently open-sourced. OpenEXR's strengths include support of high dynamic range imagery (half and float pixels), tiled images, explicit support of MIPmaps and cubic environment maps, arbitrary metadata, and arbitrary numbers of color channels. OpenEXR files use the file extension `.exr`. The official OpenEXR site is <http://www.openexr.com/>.

ImageSpec Attribute	Type	OpenEXR header data or explanation
width, height, x, y		dataWindow
full_width, full_height, full_x, full_y		displayWindow.
"worldtocamera"	matrix	worldToCamera
"worldtoscreen"	matrix	worldToNDC
"ImageDescription"	string	comments
"Copyright"	string	owner
"DateTime"	string	capDate
"PixelAspectRatio"	float	pixelAspectRatio
"ExposureTime"	float	expTime
"FNumber"	float	aperture
"compression"	string	one of: "none", "rle", "zip", "piz", "pxr24", "b44", or "b44a". If the writer receives a request for a compression type it does not recognize, it will use "zip" by default.
"textureformat"	string	set to "Plain Texture" for MIP-mapped OpenEXR files, "CubeFace Environment" or "Latlong Environment" for OpenEXR environment maps. Non-environment non-MIP-mapped OpenEXR files will not set this attribute.
"wrapmodes"	string	wrapmodes
"smpte:TimeCode"	int[2]	SMPTE time code (vecsemantics will be marked as TIMECODE)
"smpte:KeyCode"	int[7]	SMPTE key code (vecsemantics will be marked as KEYCODE)
"openexr:lineOrder"	string	the OpenEXR lineOrder attribute (set to "increasingY", "randomY", or "decreasingY").
"openexr:roundingmode"	int	the MIPmap rounding mode of the file.
<i>other</i>		All other attributes will be added to the ImageSpec by their name and apparent type.

### Limitations

- The OpenEXR format does not currently support multiple subimages, except for the special case of MIP-maps.
- The OpenEXR format only supports HALF, FLOAT, and UINT32 pixel data. OpenImageIO's OpenEXR writer will silently convert data in formats (including the common UINT8 and UINT16 cases) to HALF data for output.

## 6.14 PNG

PNG (Portable Network Graphics) is an image file format developed by the open source community as an alternative to the GIF, after Unisys started enforcing patents allegedly covering techniques necessary to use GIF. PNG files use the file extension `.png`.

ImageSpec Attribute	Type	PNG header data or explanation
"ImageDescription"	string	Description
"Artist"	string	Author
"DocumentName"	string	Title
"DateTime"	string	the timestamp in the PNG header
"PixelAspectRatio"	float	pixel aspect ratio
"XResolution"		resolution and units from the PNG header.
"YResolution"		
"ResolutionUnit"		

### Limitations

- PNG stupidly specifies that any alpha channel is “unassociated” (i.e., that the color channels are not “premultiplied” by alpha). This is a disaster, since it results in bad loss of precision for alpha image compositing, and even makes it impossible to properly represent certain additive glows and other desirable pixel values. OpenImageIO automatically associates alpha (i.e., multiplies colors by alpha) upon input and deassociates alpha (divides colors by alpha) upon output in order to properly conform to the OIIO convention (and common sense) that all pixel values passed through the OIIO APIs should use associated alpha.
- PNG only supports UINT8 and UINT16 output; other requested formats will be automatically converted to one of these.

## 6.15 PNM / Netpbm

The Netpbm project, a.k.a. PNM (portable “any” map) defines PBM, PGM, and PPM (portable bitmap, portable graymap, portable pixmap) files. Without loss of generality, we will refer to these all collectively as “PNM.” These files have extensions `.pbm`, `.pgm`, and `.ppm` and customarily correspond to bi-level bitmaps, 1-channel grayscale, and 3-channel RGB files, respectively, or `.pnm` for those who reject the nonsense about naming the files depending on the number of channels and bitdepth.

PNM files are not much good for anything, but because of their historical significance and extreme simplicity (that causes many “amateur” programs to write images in these formats), OpenImageIO supports them. PNM files do not support floating point images, anything other than 1 or 3 channels, no tiles, no multi-image, no MIPmapping. It’s not a smart choice unless you are sending your images back to the 1980’s via a time machine.

ImageSpec Attribute	Type	PNG header data or explanation
"oio:BitsPerSample"	int	the true bits per sample of the file (1 for true PBM files, even though OIIO will report the format as UINT8).
"pnm:binary"	int	nonzero if the file itself used the PNM binary format, 0 if it used ASCII. The PNM writer honors this attribute in the ImageSpec to determine whether to write an ASCII or binary file.

## 6.16 PSD

## 6.17 Ptex

Ptex is a special per-face texture format developed by Walt Disney Feature Animation. The format and software to read/write it are open source, and available from <http://ptex.us/>. Ptex files commonly use the file extension `.ptex`.

OpenImageIO's support of Ptex is still incomplete. We can read pixels from Ptex files, but the TextureSystem doesn't properly filter across face boundaries when using it as a texture. OpenImageIO currently does not write Ptex files at all.

ImageSpec Attribute	Type	Ptex header data or explanation
"ptex:meshType"	string	the mesh type, either "triangle" or "quad".
"ptex:hasEdits"	int	nonzero if the Ptex file has edits.
"wrapmode"	string	the wrap mode as specified by the Ptex file.
<i>other</i>		Any other arbitrary metadata in the Ptex file will be stored directly as attributes in the ImageSpec.

## 6.18 RLA

RLA (Run-Length encoded, version A) is an early CGI renderer output format, originating from Wavefront Advanced Visualizer and used primarily by software developed at Wavefront. RLA files commonly use the file extension `.rla`.

ImageSpec Attribute	Type	RLA header data or explanation
width, height, x, y		RLA “active/viewable” window.
full_width, full_height, full_x, full_y		RLA “full” window.
"rla:FrameNumber"	int	frame sequence number.
"rla:Revision"	int	file format revision number, currently "0xFFFE".
"rla:JobNumber"	int	job number ID of the file.
"rla:FieldRendered"	int	whether the image is a field-rendered (interlaced) one ("0" for false, non-zero for true).
"rla:FileName"	string	name under which the file was originally saved.
"ImageDescription"	string	RLA “Description” of the image.
"Software"	string	name of software used to save the image.
"HostComputer"	string	name of machine used to save the image.
"Artist"	string	RLA “UserName”: logon name of user who saved the image.
"rla:Aspect"	string	aspect format description string.
"rla:ColorChannel"	string	textual description of color channel data format (usually "rgb").
"rla:Time"	string	description (format not standardized) of amount of time spent on creating the image.
"rla:Filter"	string	name of post-processing filter applied to the image.
"rla:AuxData"	string	textual description of auxiliary channel data format.
"rla:AspectRatio"	float	image aspect ratio.
"rla:RedChroma"	vec2 or vec3 of floats	red point XY (vec2) or XYZ (vec3) coordinates.
"rla:GreenChroma"	vec2 or vec3 of floats	green point XY (vec2) or XYZ (vec3) coordinates.
"rla:BlueChroma"	vec2 or vec3 of floats	blue point XY (vec2) or XYZ (vec3) coordinates.
"rla:WhitePoint"	vec2 or vec3 of floats	white point XY (vec2) or XYZ (vec3) coordinates.

### Limitations

- OpenImageIO will only write 1 image to 1 file, multiple subimages are not supported by the writer (but are supported by the reader).



## 6.19 SGI

The SGI image format was a simple raster format used long ago on SGI machines. SGI files use the file extensions `sgi`, `rgb`, `rgba`, `"bw"`, `"int"`, and `"inta"`.

The SGI format is sometimes used for legacy apps, but has little merit otherwise: no support for tiles, no MIPmaps, no multi-subimage, only 8- and 16-bit integer pixels (no floating point), only 1-4 channels.

ImageSpec Attribute	Type	SGI header data or explanation
"ImageDescription"	string	image name
"Compression"	string	thee compression of the SGI file ("rle", if RLE compression is used).

## 6.20 Softimage PIC

Softimage PIC is an image file format used by the SoftImage 3D application, and some other programs that needed to be compatible with it. Softimage files use the file extension `.pic`.

The Softimage PIC format is sometimes used for legacy apps, but has little merit otherwise, so currently OpenImageIO only reads Softimage files and is unable to write them.

ImageSpec Attribute	Type	PIC header data or explanation
"ImageDescription"	string	comment
"oio:BitsPerSample"	int	the true bits per sample in the PIC file.

## 6.21 RAW digital camera files

A variety of digital camera “raw” formats are supported via this plugin that is based on the LibRaw library (<http://www.libraw.org/>).

## 6.22 Targa

Targa (a.k.a. Truevision TGA) is an image file format with little merit except that it is very simple and is used by many legacy applications. Targa files use the file extension `.tga`, or, much more rarely, `.tpic`. The official Targa format specification may be found at <http://www.dca.fee.unicamp.br/~martino/disciplinas/ea978/tgaffs.pdf>.

ImageSpec Attribute	Type	TGA header data or explanation
"ImageDescription"	string	comment
"Artist"	string	author
"DocumentName"	string	job name/ID
"Software"	string	software name
"DateTime"	string	TGA time stamp
"targa:JobTime"	string	TGA "job time."
"Compression"	string	values of "none" and "rle" are supported. The writer will use RLE compression if any unknown compression methods are requested.
"targa:ImageID"	string	Image ID
"PixelAspectRatio"	float	pixel aspect ratio
"oio:BitsPerSample"	int	the true (in the file) bits per sample.

If the TGA file contains a thumbnail, its dimensions will be stored in the attributes "thumbnail\_width", "thumbnail\_height", and "thumbnail\_nchannels", and the thumbnail pixels themselves will be stored in "thumbnail\_image" (as an array of UINT8 values, whose length is the total number of channel samples in the thumbnail).

### Limitations

- The Targa reader reserves enough memory for the entire image. Therefore it is not a good choice for high-performance image use such as would be used for ImageCache or TextureSystem.
- Targa files only support 8- and 16-bit unsigned integers (no signed, floating point, or HDR capabilities); the OpenImageIO TGA writer will silently convert all output images to UINT8 (except if UINT16 is explicitly requested).
- Targa only supports grayscale, RGB, and RGBA; the OpenImageIO TGA writer will fail its call to `open()` if it is asked create a file with more than 4 color channels.

## 6.23 TIFF

TIFF (Tagged Image File Format) is a flexible file format created by Aldus, now controlled by Adobe. TIFF supports nearly everything anybody could want in an image format (and has exactly the complexity you would expect from such a requirement). TIFF files commonly use the file extensions `.tif` or `.tiff`. Additionally, OpenImageIO associates the following extensions with TIFF files by default: `.tx`, `.env`, `.sm`, `.vsm`.

The official TIFF format specification may be found here: <http://partners.adobe.com/public/developer/tiff/index.html> The most popular library for reading TIFF directly is `libtiff`, available here: <http://www.remotesensing.org/libtiff/> OpenImageIO uses `libtiff` for its TIFF reading/writing.

We like TIFF a lot, especially since its complexity can be nicely hidden behind OIIO's simple APIs. It supports a wide variety of data formats (though unfortunately not `half`), an arbitrary number of channels, tiles and multiple subimages (which makes it our preferred texture format), and a rich set of metadata.

OpenImageIO supports the vast majority of TIFF features, including: tiled images ("`tilled`") as well as scanline images; multiple subimages per file ("`multiimage`"); MIPmapping (using multi-subimage; that means you can't use `multiimage` and `MIPmaps` simultaneously); data formats 8- 16, and 32 bit integer (both signed and unsigned), and 32- and 64-bit floating point; palette images (will convert to RGB); "miniswhite" photometric mode (will convert to "minis-black").

The TIFF plugin attempts to support all the standard Exif, IPTC, and XMP metadata if present.

ImageSpec Attribute	Type	TIFF header data or explanation
ImageSpec::x	int	XPosition
ImageSpec::y	int	YPosition
ImageSpec::full_width	int	PIXAR_IMAGEFULLWIDTH
ImageSpec::full_length	int	PIXAR_IMAGEFULLLENGTH
"ImageDescription"	string	ImageDescription
"DateTime"	string	DateTime
"Software"	string	Software
"Artist"	string	Artist
"Copyright"	string	Copyright
"Make"	string	Make
"Model"	string	Model
"DocumentName"	string	DocumentName
"HostComputer"	string	HostComputer
"XResolution" "YResolution"	float	XResolution, YResolution
"ResolutionUnit"	string	ResolutionUnit ("in" or "cm").
"Orientation"	int	Orientation
"textureformat"	string	PIXAR_TEXTUREFORMAT
"wrapmodes"	string	PIXAR_WRAPMODES
"fovcot"	float	PIXAR_FOVCOT
"worldtocamera"	matrix	PIXAR_MATRIX_WORLDTOCAMERA
"worldtoscreen"	matrix	PIXAR_MATRIX_WORLDTOSCREEN
"compression"	string	based on TIFF Compression (one of "none", "lzw", "ccittlrle", "zip", "packbits").
"tiff:compression"	int	the original integer code from the TIFF Compression tag.
"tiff:planarconfig"	string	PlanarConfiguration ("separate" or "contig"). The OpenImageIO TIFF writer will honor such a request in the ImageSpec.
"tiff:PhotometricInterpretation"	int	Photometric
"tiff:PageName"	string	PageName
"tiff:PageNumber"	int	PageNumber
"tiff:RowsPerStrip"	int	RowsPerStrip
"tiff:subfiletype"	1	SubfileType
"Exif:*"		A wide variety of EXIF data are honored, and are all prefixed with "Exif: ".
"oio:BitsPerSample"	int	The actual bits per sample in the file (may differ from ImageSpec::format).
"oio:UnassociatedAlpha"	int	Nonzero if the alpha channel contained "unassociated" alpha.

### Limitations

OpenImageIO's TIFF reader and writer have some limitations you should be aware of:

- No separate per-channel data formats (not supported by libtiff).

- Only multiples of 8 bits per pixel may be passed through OpenImageIO's APIs, e.g., 1-, 2-, and 4-bits per pixel will be reported by OIIO as 8 bit images; 12 bits per pixel will be reported as 16, etc. But the "oiio:BitsPerSample" attribute in the ImageSpec will correctly report the original bit depth of the file. Note that the TIFF specification itself does not support 16-bit floating point pixels (half data).

## 6.24 Webp

## 6.25 Zfile

Zfile is a very simple format for writing a depth (z) image, originally from Pixar's PhotoRealistic RenderMan but now supported by many other renderers. It's extremely minimal, holding only a width, height, world-to-screen and camera-to-screen matrices, and uncompressed float pixels of the z-buffer. Zfile files use the file extension `.zfile`.

ImageSpec Attribute	Type	Zfile header data or explanation
"worldtocamera"	matrix	NP
"worldtoscreen"	matrix	NI



## 7 Cached Images

### 7.1 Image Cache Introduction and Theory of Operation

`ImageCache` is a utility class that allows an application to read pixels from a large number of image files while using a remarkably small amount of memory and other resources. Of course it is possible for an application to do this directly using `ImageInput` objects. But `ImageCache` offers the following advantages:

- `ImageCache` presents an even simpler user interface than `ImageInput`— the only supported operations are asking for an `ImageSpec` describing a subimage in the file, retrieving for a block of pixels, and locking/reading/releasing individual tiles. You refer to images by filename only; you don't need to keep track of individual file handles or `ImageInput` objects. You don't need to explicitly open or close files.
- The `ImageCache` is completely thread-safe; if multiple threads are accessing the same file, the `ImageCache` internals will handle all the locking and resource sharing.
- No matter how many image files you are accessing, the `ImageCache` will maintain a reasonable number of simultaneously-open files, automatically closing files that have not been needed recently.
- No matter how large the total pixels in all the image files you are dealing with are, the `ImageCache` will use only a small amount of memory. It does this by loading only the individual tiles requested, and as memory allotments are approached, automatically releasing the memory from tiles that have not been used recently.

In short, if you have an application that will need to read pixels from many large image files, you can rely on `ImageCache` to manage all the resources for you. It is reasonable to access thousands of image files totalling hundreds of GB of pixels, efficiently and using a memory footprint on the order of 50 MB.

Below are some simple code fragments that shows ImageCache in action:

```
#include <OpenImageIO/imagecache.h>
OIIO_NAMESPACE_USING

// Create an image cache and set some options
ImageCache *cache = ImageCache::create ();
cache->attribute ("max_memory_MB", 500.0);
cache->attribute ("autotile", 64);

// Get a block of pixels from a file.
// (for brevity of this example, let's assume that 'size' is the
// number of channels times the number of pixels in the requested region)
float pixels[size];
cache->get_pixels ("file1.jpg", 0, 0, xbegin, xend, ybegin, yend,
                 zbegin, zend, TypeDesc::FLOAT, pixels);

// Get information about a file
ImageSpec spec;
bool ok = cache->get_imagespec ("file2.exr", spec);
if (ok)
    std::cout << "resolution is " << spec.width << "x"
               << "spec.height << "\n";

// Request and hold a tile, do some work with its pixels, then release
ImageCache::Tile *tile;
tile = cache->get_tile ("file2.exr", 0, 0, x, y, z);
// The tile won't be freed until we release it, so this is safe:
TypeDesc format;
void *p = cache->tile_pixels (tile, format);
// Now p points to the raw pixels of the tile, whose data format
// is given by 'format'.
cache->release_tile (tile);
// Now cache is permitted to free the tile when needed

// Note that all files were referenced by name, we never had to open
// or close any files, and all the resource and memory management
// was automatic.

ImageCache::destroy (cache);
```



## 7.2 ImageCache API

### 7.2.1 Creating and destroying an image cache

ImageCache 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 ImageCache are provided:

```
static ImageCache *ImageCache::create (bool shared=true)
```

Creates a new ImageCache and returns a pointer to it. If `shared` is `true`, `create()` will return a pointer to a shared ImageCache (so that multiple parts of an application that request an ImageCache will all end up with the same one). If `shared` is `false`, a completely unique ImageCache will be created and returned.

```
static void ImageCache::destroy (ImageCache *x, bool teardown=false)
```

Destroys an allocated ImageCache, 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).

It is safe to destroy even a shared ImageCache, as the implementation of `destroy()` will recognize a shared one and only truly release its resources if it has been requested to be destroyed as many times as shared ImageCache's were created. For a shared ImageCache, if the `teardown` parameter is `true`, it will try to truly destroy the shared cache if nobody else is still holding a reference (otherwise, it will leave it intact).

### 7.2.2 Setting options and limits for the image cache

The following member functions of ImageCache allow you to set (and in some cases retrieve) options that control the overall behavior of the image cache:

```
bool attribute (string_ref name, TypeDesc type, const void *val)
```

Sets an attribute (i.e., a property or option) of the ImageCache. The `name` designates the name of the attribute, `type` describes the type of data, and `val` is a pointer to memory containing the new value for the attribute.

If the ImageCache recognizes a valid attribute name that matches the type specified, the attribute will be set to the new value and `attribute()` will return `true`. If `name` is not recognized as a valid attribute name, or if the types do not match (e.g., `type` is `TypeDesc::FLOAT` but the named attribute is a string), the attribute will not be modified, and `attribute()` will return `false`.

Here are examples:

```
ImageCache *ts;
```

```

...
int maxfiles = 50;
ts->attribute ("max_open_files", TypeDesc::INT, &maxfiles);

const char *path = "/my/path";
ts->attribute ("searchpath", TypeDesc::STRING, &path);

```

Note that when passing a string, you need to pass a pointer to the `char*`, not a pointer to the first character. (Rationale: for an `int` attribute, you pass the address of the `int`. So for a string, which is a `char*`, you need to pass the address of the string, i.e., a `char**`).

The complete list of attributes can be found at the end of this section.

```

bool attribute (string_ref name, int val)
bool attribute (string_ref name, float val)
bool attribute (string_ref name, double val)
bool attribute (string_ref name, string_ref val)

```

Specialized versions of `attribute()` in which the data type is implied by the type of the argument.

For example, the following are equivalent to the example above for the general (pointer) form of `attribute()`:

```

ts->attribute ("max_open_files", 50);
ts->attribute ("searchpath", "/my/path");

```

```

bool getattribute (string_ref name, TypeDesc type, void *val)

```

Gets the current value of an attribute of the `ImageCache`. The `name` designates the name of the attribute, `type` describes the type of data, and `val` is a pointer to memory where the user would like the value placed.

If the `ImageCache` recognizes a valid attribute name that matches the type specified, the attribute value will be stored at address `val` and `attribute()` will return `true`. If `name` is not recognized as a valid attribute name, or if the types do not match (e.g., `type` is `TypeDesc::FLOAT` but the named attribute is a string), no data will be written to `val`, and `attribute()` will return `false`.

Here are examples:

```

ImageCache *ts;
...
int maxfiles;
ts->getattribute ("max_open_files", TypeDesc::INT, &maxfiles);

const char *path;
ts->getattribute ("searchpath", TypeDesc::STRING, &path);

```

Note that when passing a string, you need to pass a pointer to the `char*`, not a pointer to the first character. Also, the `char*` will end up pointing to characters owned by the `ImageCache`; the caller does not need to ever free the memory that contains the characters.

The complete list of attributes can be found at the end of this section.

```
bool getattribute (string_ref name, int &val)
bool getattribute (string_ref name, float &val)
bool getattribute (string_ref name, double &val)
bool getattribute (string_ref name, char **val)
bool getattribute (string_ref name, std::string & val)
```

Specialized versions of `getattribute()` in which the data type is implied by the type of the argument.

For example, the following are equivalent to the example above for the general (pointer) form of `getattribute()`:

```
int maxfiles;
ts->getattribute ("max_open_files", &maxfiles);
const char *path;
ts->getattribute ("searchpath", &path);
```

### Image cache attributes

Recognized attributes include the following:

```
int max_open_files
```

The maximum number of file handles that the image cache will hold open simultaneously. (Default = 100)

```
float max_memory_MB
```

The maximum amount of memory (measured in MB) that the image cache will use for its “tile cache.” (Default: 256.0 MB)

```
string searchpath
```

The search path for images: a colon-separated list of directories that will be searched in order for any image name that is not specified as an absolute path. (Default: no search path.)

```
string plugin_searchpath
```

The search path for plugins: a colon-separated list of directories that will be searched in order for any OIIO plugins, if not found in OIIO’s “lib” directory.) (Default: no additional search path.)

```
int autotile
int autoscanline
```

These attributes control how the image cache deals with images that are not “tiled” (i.e., are stored as scanlines).

If `autotile` is set to 0 (the default), an untiled image will be treated as if it were a single tile of the resolution of the whole image. This is simple and fast, but can lead to poor cache behavior if you are simultaneously accessing many large untiled images.

If `autotile` is nonzero (e.g., 64 is a good recommended value), any untiled images will be read and cached as if they were constructed in tiles of size:

<code>autotile × autotile</code>	if <code>autoscanline</code> is 0
<code>width × autotile</code>	if <code>autoscanline</code> is nonzero.

In both cases, this should lead more efficient caching. The `autoscanline` determines whether the “virtual tiles” in the cache are square (if `autoscanline` is 0, the default) or if they will be as wide as the image (but only `autotile` scanlines high). You should try in your application to see which leads to higher performance.

```
int automip
```

If `automip` is set to 0 (the default), an untiled single-subimage file will only be able to utilize that single subimage.

If `automip` is nonzero, any untiled, single-subimage (un-MIP-mapped) images will have lower-resolution MIP-map levels generated on-demand if pixels are requested from the lower-res subimages (that don’t really exist). Essentially this makes the `ImageCache` pretend that the file is MIP-mapped even if it isn’t.

```
int forcefloat
```

If set to nonzero, all image tiles will be converted to `float` type when stored in the image cache. This can be helpful especially for users of `ImageBuf` who want to simplify their image manipulations to only need to consider `float` data.

The default is zero, meaning that image pixels are not forced to be `float` when in cache.

```
int accept_untiled
```

When nonzero (the default), `ImageCache` accepts untiled images as usual. When set to zero, `ImageCache` will reject untiled images with an error condition, as if the file could not be properly read. This is sometimes helpful for applications that want to enforce use of tiled images only.

```
int accept_unmipped
```

When nonzero (the default), `ImageCache` accepts un-MIPmapped images as usual. When set to zero, `ImageCache` will reject un-MIPmapped images with an error condition, as if the file could not be properly read. This is sometimes helpful for applications that want to enforce use of MIP-mapped images only.

`int failure_retries`

When an `open()` or `read_tile()` calls fails, pause and try again, up to `failure_retries` times before truly returning a failure. This is meant to address spooky disk or network failures. The default is zero, meaning that failures of open or tile reading will immediately return as a failure.

`int deduplicate`

When nonzero, the `ImageCache` will notice duplicate images under different names if their headers contain a SHA-1 fingerprint (as is done with `maketx`-produced textures) and handle them more efficiently by avoiding redundant reads. The default is 1 (de-duplication turned on). The only reason to set it to 0 is if you specifically want to disable the de-duplication optimization.

`string substitute_image`

When set to anything other than the empty string, the `ImageCache` will use the named image in place of *all* other images. This allows you to run an app using OIIO and (if you can manage to get this option set) automatically substitute a grid, zone plate, or other special debugging image for all image/texture use.

`int unassociatedalpha`

When nonzero, will request that image format readers try to leave input images with unassociated alpha as they are, rather than automatically converting to associated alpha upon reading the pixels. The default is 0, meaning that the automatic conversion will take place.

`string options`

This catch-all is simply a comma-separated list of `name=value` settings of named options. For example,

```
ic->attribute ("options", "max_memory_MB=512.0,autotile=1");
```

### 7.2.3 Getting information about images

```
bool get_image_info (ustring filename, int subimage, int miplevel,
                     ustring dataname, TypeDesc datatype, void *data)
```

Retrieves information about the image named by `filename`. The `dataname` is a keyword indicating 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_image_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_image_info()` will return false.

Supported `dataname` values include:

- `exists` Return 1 if the file exists and is an image format that OpenImageIO knows how to read, otherwise return 0. The data pointer is not used.
  - `subimages` The number of subimages in the file, as an integer.
  - `resolution` The resolution of the image file, which is an array of 2 integers (described as `TypeDesc(INT, 2)`).
  - `miplevels` The number of MIPmap levels for the specified subimage (an integer).
  - `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).
  - `format` The native data format of the pixels in the file (an integer, giving the `TypeDesc::BASETYPE` of the data). Note that this is not necessarily the same as the data format stored in the image cache.
  - `cachedformat` The native data format of the pixels as stored in the image cache (an integer, giving the `TypeDesc::BASETYPE` of the data). Note that this is not necessarily the same as the native data format of the file.
  - `datawindow` Returns the pixel data window of the image, which is either an array of 4 integers (returning `xmin`, `ymin`, `xmax`, `ymax`) or an array of 6 integers (returning `xmin`, `ymin`, `zmin`, `xmax`, `ymax`, `zmax`). The `z` values may be useful for 3D/volumetric images; for 2D images they will be 0).
  - `displaywindow` Returns the display (a.k.a. full) window of the image, which is either an array of 4 integers (returning `xmin`, `ymin`, `xmax`, `ymax`) or an array of 6 integers (returning `xmin`, `ymin`, `zmin`, `xmax`, `ymax`, `zmax`). The `z` values may be useful for 3D/volumetric images; for 2D images they will be 0).
  - `viewingmatrix` The viewing matrix, which is a  $4 \times 4$  matrix (an `Imath::M44f`, described as `TypeDesc(FLOAT, MATRIX)`).
  - `projectionmatrix` The projection matrix, which is a  $4 \times 4$  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,
                    int subimage=0, int miplevel=0, bool native=false)
```

If the named image (and the specific subimage and MIP level) is found and able to be opened by an available image format plugin, and the designated subimage exists, this function copies its image specification for that subimage into `spec` and returns `true`. Otherwise, if the file is not found, could not be opened, is not of a format readable by any plugin that could be found, or the designated subimage did not exist in the file, the return value is `false` and `spec` will not be modified.

If `native` is `false` (the default), then the `spec` retrieved will accurately describe the image stored internally in the cache, whereas if `native` is `true`, the `spec` retrieved will reflect the original file. These may differ due to use of certain ImageCache settings such as "forcefloat" or "autotile".

```
const ImageSpec * imagespec (ustring filename, int subimage=0,
                              int miplevel=0, bool native=false)
```

If the named image is found and able to be opened by an available image format plugin, and the designated subimage exists, this function returns a pointer to an `ImageSpec` that describes it. Otherwise, if the file is not found, could not be opened, is not of a format readable by any plugin that could be found, or the designated subimage did not exist in the file, the return value is `NULL`.

If `native` is `false` (the default), then the `spec` retrieved will accurately describe the image stored internally in the cache, whereas if `native` is `true`, the `spec` retrieved will reflect the original file. These may differ due to use of certain ImageCache settings such as "forcefloat" or "autotile".

This method is much more efficient than `get_imagespec()`, since it just returns a pointer to the `spec` held internally by the ImageCache (rather than copying the `spec` to the user's memory). However, the caller must beware that the pointer is only valid as long as nobody (even other threads) calls `invalidate()` on the file, or `invalidate_all()`, or destroys the ImageCache.

```
std::string resolve_filename (const std::string &filename)
```

Returns the true path to the given file name, with searchpath logic applied.

### 7.2.4 Getting pixels

```
bool get_pixels (ustring filename, int subimage, int miplevel,
                 int xbegin, int xend, int ybegin, int yend,
                 int zbegin, int zend,
                 TypeDesc format, void *result)
```

Retrieve the rectangle of pixels of the designated subimage and miplevel, storing the pixel values beginning at the address specified by `result`. The pixel 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). The rectangular region to be retrieved includes `begin` but does not include `end` (much like STL `begin/end` usage). Requested pixels that are not part of the valid pixel data region of the image file will be filled with zero values.

```
bool get_pixels (ustring filename, int subimage, int miplevel,
                 int xbegin, int xend, int ybegin, int yend,
                 int zbegin, int zend, int chbegin, int chend,
                 TypeDesc format, void *result,
                 stride_t xstride, stride_t ystride, stride_t zstride)
```

Retrieve the rectangle of pixels and subset of channels of the designated `subimage` and `miplevel`, storing the pixel values beginning at the address specified by `result` and with the given strides. The pixel 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, data format, and strides). Any stride values set to `AutoStride` will be assumed to indicate a contiguous data layout. The rectangular region and channel set to be retrieved includes `begin` but does not include `end` (much like STL `begin/end` usage). Requested pixels that are not part of the valid pixel data region of the image file will be filled with zero values.

### 7.2.5 Dealing with tiles

```
ImageCache::Tile * get_tile (ustring filename, int subimage, int miplevel,
                             int x, int y, int z)
```

Find a tile given by an image `filename`, `subimage`, and pixel coordinates. An opaque pointer to the tile will be returned, or `NULL` if no such file (or tile within the file) exists or can be read. The tile will not be purged from the cache until after `release_tile()` is called on the tile pointer. This is thread-safe.

```
void release_tile (ImageCache::Tile *tile)
```

After finishing with a tile, `release_tile()` will allow it to once again be purged from the tile cache if required.

```
const void * tile_pixels (ImageCache::Tile *tile, TypeDesc &format)
```

For a tile retrieved by `get_tile()`, return a pointer to the pixel data itself, and also store in `format` the data type that the pixels are internally stored in (which may be different than the data type of the pixels in the disk file). This method should only be called on a tile that has been requested by `get_tile()` but has not yet been released with `release_tile()`.



```
void invalidate (ustring filename)
```

Invalidate any loaded tiles or open file handles associated with the filename, so that any subsequent queries will be forced to re-open the file or re-load any tiles (even those that were previously loaded and would ordinarily be reused). A client might do this if, for example, they are aware that an image being held in the cache has been updated on disk. This is safe to do even if other procedures are currently holding reference-counted tile pointers from the named image, but those procedures will not get updated pixels until they release the tiles they are holding.

```
void invalidate_all (bool force=false)
```

Invalidate all loaded tiles and open file handles, so that any subsequent queries will be forced to re-open the file or re-load any tiles (even those that were previously loaded and would ordinarily be reused). A client might do this if, for example, they are aware that an image being held in the cache has been updated on disk. This is safe to do even if other procedures are currently holding reference-counted tile pointers from the named image, but those procedures will not get updated pixels until they release the tiles they are holding. If force is true, everything will be invalidated, no matter how wasteful it is, but if force is false, in actuality files will only be invalidated if their modification times have been changed since they were first opened.

### 7.2.6 Seeding the cache

```
bool add_file (ustring filename, ImageInput::Creator creator)
```

This creates a file entry in the cache that, instead of reading from disk, uses a custom `ImageInput` to generate the image (note that it will have no effect if there's already an image by the same name in the cache). The `creator` is a factory that creates the custom `ImageInput` and will be called like this:

```
ImageInput *in = creator();
```

Once created, the `ImageCache` owns the `ImageInput` and is responsible for destroying it when done. Custom `ImageInput`'s allow "procedural" images, among other things. Also, this is the method you use to set up a "writeable" `ImageCache` images (perhaps with a type of `ImageInput` that's just a stub that does as little as possible).

```
bool add_tile (ustring filename, int subimage, int miplevel,
               int x, int y, int z, TypeDesc format, const void *buffer,
               stride_t xstride=AutoStride, stride_t ystride=AutoStride,
               stride_t zstride=AutoStride)
```

Preemptively add a tile corresponding to the named image, at the given subimage and MIP level. The tile added is the one whose corner is (x,y,z), and buffer points to the pixels (in the given format, with supplied strides) which will be copied and inserted into the cache and made available for future lookups.

### 7.2.7 Errors and statistics

`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.

`std::string getstats (int level=1)`

Returns a big string containing useful statistics about the `ImageCache` operations, suitable for saving to a file or outputting to the terminal. The `level` indicates the amount of detail in the statistics, with higher numbers (up to a maximum of 5) yielding more and more esoteric information.

`void reset_stats ()`

Reset most statistics to be as they were with a fresh `TextureSystem`. Caveat emptor: this does not flush the cache itself, so the resulting statistics from the next set of texture requests will not match the number of tile reads, etc., that would have resulted from a new `TextureSystem`.

## 8 Texture Access: TextureSystem

### 8.1 Texture System Introduction and Theory of Operation

Coming soon. FIXME

### 8.2 Helper Classes

#### 8.2.1 Imath

The texture functionality 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  $4 \times 4$  transformation matrixes. To use these yourself, we recommend that you:

```
#include <OpenEXR/ImathVec.h>
#include <OpenEXR/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 TextureOpt

`TextureOpt` is a structure that holds many options controlling single-point texture lookups. Because each texture lookup API call takes a reference to a `TextureOpt`, 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 `TextureOpt` 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`.

```
int subimage
usttring subimagename
```

Specifies the subimage or face within the file to use for the texture lookup. If `subimagename` is set (it defaults to the empty string), it will try to use the subimage that had a matching metadata `"oio:subimagename"`, otherwise the integer `subimage` will be used (which defaults to 0, i.e., the first/default subimage). Nonzero subimage indices only make sense for a texture file that supports subimages or separate images per face (such as Ptex). This will be ignored if the file does not have multiple subimages or separate per-face textures.

```
Wrap swrap, twrap
```

Specify the *wrap mode* for 2D texture lookups (and 3D volume texture lookups, using the additional `rwrap` 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.

```
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).

```
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.

```
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 4-channel lookup on a 3-channel texture, the last channel will get the fill value. (Note: this behavior is affected by the "gray\_to\_rgb" attribute described in Section 8.3.2.)

```
const float* missingcolor
```

If not NULL, indicates that a missing or broken texture should *not* be treated as an error, but rather will simply return the supplied color as the texture lookup color and `texture()` will return `true`. If the `missingcolor` field is left at its default (a NULL pointer), a missing or broken texture will be treated as an error and `texture()` will return `false`. Note: When not NULL, the data must point to *nchannels* contiguous floats.

```
float *dresultds, *dresultdt
```

If not NULL (the default), these specify locations in which to store the *derivatives* of the texture lookup, i.e., the change of the filtered texture per unit of *s* and *t*, respectively. Each must point to *nchannels* contiguous floats. If either is NULL, the derivative computations will not be performed.

```
float bias
```

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

```
int samples
```

For shadow map lookups only, the number of samples to use for the lookup.

```
Wrap rwrap
```

```
float rblur, rwidth
```

```
float *dresultdr
```

Specifies wrap, blur, width, and derivative results for the third component of 3D volume texture lookups. These are not used for 2D texture lookups.

### 8.2.3 TextureOptions

`TextureOptions` is a structure that holds many options controlling batched 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`.

`int subimage`

The subimage or face within the file to use for the texture lookup. The default is 0, and larger values only make sense for a texture file that supports subimages or separate images per face (such as Ptex). This will be ignored if the file does not have multiple subimages or separate per-face textures.

`Wrap swrap, twrap`

Specify the *wrap mode* for 2D texture lookups (and 3D volume texture lookups, using the additional `rwrap` 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> 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 4-channel lookup on a 3-channel texture, the `lsat` channel will get the fill value. (Note: this behavior is affected by the `"gray__to_rgb"` attribute described in Section 8.3.2.)

`VaryingRef<float> missingcolor`

If supplied, indicates that a missing or broken texture should *not* be treated as an error, but rather will simply return the supplied color as the texture lookup color and `texture()` will return `true`. If the `missingcolor` field is left at its default (a NULL pointer), a missing or broken texture will be treated as an error and `texture()` will return `false`.

Although this is a `VaryingRef<float>`, the data must point to *nchannels* contiguous floats, and if “varying,” the step size must be set to `nchannels*sizeof(float)`, not `sizeof(float)`.

`float *dresultds, *dresultdt`

If not NULL (the default), these specify locations in which to store the *derivatives* of the texture lookup, i.e., the change of the filtered texture per unit of *s* and *t*, respectively. Each must point to *nchannels* contiguous floats. If either is NULL, the derivative computations will not be performed.

`VaryingRef<float> bias`

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

`VaryingRef<int> samples`

For shadow map lookups only, the number of samples to use for each lookup.

`Wrap rwrap`

`VaryingRef<float> rblur, rwidth`

`float *dresultdr`

Specifies wrap, blur, width, and derivative results for the third component of 3D volume texture lookups. These are not used for 2D texture lookups.

### 8.2.4 VaryingRef: encapsulate uniform and varying

Many texture access API routines are designed to look 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 pass 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);
```

### 8.2.5 SIMD Run Flags

Many 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 beginactive`, `int endactive`. There are also `VaryingRef` parameters such as *s* and *t* that act as if they are arrays.

The `beginactive` and `endactive` indices are the first (inclusive) and last (exclusive) 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 beginactive, int endactive,
          VaryingRef<float> arg, VaryingRef <float> result)
{
    for (int i = beginactive; i < endactive; ++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 (bool share=true)
```

Creates a new TextureSystem and returns a pointer to it. If `share` is true, the TextureSystem created will share its underlying ImageCache with any other TextureSystem's or ImageCache's that requested shared caches. If `share` is false, a completely unique ImageCache will be created that is private to this particular TextureSystem.

```
static void TextureSystem::destroy (TextureSystem *x,  
                                   bool teardown_imagecache=false)
```

Destroys an allocated TextureSystem, including freeing all system resources that it holds (such as its underlying ImageCache).

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).

If `teardown_imagecache` is true, and the TextureSystem's underlying ImageCache is the *shared* one, then that ImageCache will be thoroughly destroyed, not merely releasing the reference.

### 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:

```
bool attribute (string_ref name, TypeDesc type, const void *val)
```

Sets an attribute (i.e., a property or option) of the TextureSystem. The `name` designates the name of the attribute, `type` describes the type of data, and `val` is a pointer to memory containing the new value for the attribute.

If the TextureSystem recognizes a valid attribute name that matches the type specified, the attribute will be set to the new value and `attribute()` will return true. If `name` is not recognized as a valid attribute name, or if the types do not match (e.g., `type` is `TypeDesc::FLOAT` but the named attribute is a string), the attribute will not be modified, and `attribute()` will return false.

Here are examples:

```
TextureSystem *ts;
```

```

...
int maxfiles = 50;
ts->attribute ("max_open_files", TypeDesc::INT, &maxfiles);

const char *path = "/my/path";
ts->attribute ("searchpath", TypeDesc::STRING, &path);

```

Note that when passing a string, you need to pass a pointer to the `char*`, not a pointer to the first character. (Rationale: for an `int` attribute, you pass the address of the `int`. So for a string, which is a `char*`, you need to pass the address of the string, i.e., a `char**`).

The complete list of attributes can be found at the end of this section.

```

bool attribute (string_ref name, int val)
bool attribute (string_ref name, float val)
bool attribute (string_ref name, double val)
bool attribute (string_ref name, string_ref val)

```

Specialized versions of `attribute()` in which the data type is implied by the type of the argument.

For example, the following are equivalent to the example above for the general (pointer) form of `attribute()`:

```

ts->attribute ("max_open_files", 50);
ts->attribute ("searchpath", "/my/path");

```

```

bool getattribute (string_ref name, TypeDesc type, void *val)

```

Gets the current value of an attribute of the `TextureSystem`. The name designates the name of the attribute, `type` describes the type of data, and `val` is a pointer to memory where the user would like the value placed.

If the `TextureSystem` recognizes a valid attribute name that matches the type specified, the attribute value will be stored at address `val` and `attribute()` will return `true`. If name is not recognized as a valid attribute name, or if the types do not match (e.g., `type` is `TypeDesc::FLOAT` but the named attribute is a string), no data will be written to `val`, and `attribute()` will return `false`.

Here are examples:

```

TextureSystem *ts;
...
int maxfiles;
ts->getattribute ("max_open_files", TypeDesc::INT, &maxfiles);

const char *path;
ts->getattribute ("searchpath", TypeDesc::STRING, &path);

```

Note that when passing a string, you need to pass a pointer to the `char*`, not a pointer to the first character. Also, the `char*` will end up pointing to characters owned by the

TextureSystem; the caller does not need to ever free the memory that contains the characters.

The complete list of attributes can be found at the end of this section.

```
bool getattribute (string_ref name, int &val)
bool getattribute (string_ref name, float &val)
bool getattribute (string_ref name, double &val)
bool getattribute (string_ref name, char **val)
bool getattribute (string_ref name, std::string & val)
```

Specialized versions of `getattribute()` in which the data type is implied by the type of the argument.

For example, the following are equivalent to the example above for the general (pointer) form of `getattribute()`:

```
int maxfiles;
ts->getattribute ("max_open_files", &maxfiles);
const char *path;
ts->getattribute ("searchpath", &path);
```

### Texture system attributes

Recognized attributes include the following:

```
int max_open_files
float max_memory_MB
string searchpath
string plugin_searchpath
int autotile
int autoscanline
int automip
int accept_untiled
int accept_unmipped
int failure_retries
int deduplicate
string substitute_image
```

These attributes are all passed along to the underlying `ImageCache` that is used internally by the `TextureSystem`. Please consult the `ImageCache` attribute list in Section 7.2.2 for explanations of these attributes.

```
matrix worldtocommon
```

The  $4 \times 4$  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.

`matrix commontoworld`

The  $4 \times 4$  matrix that is the inverse of `worldtocommon` — that is, it transforms points from “common” to “world” coordinates.

You do not need to set `commontoworld` and `worldtocommon` separately; just setting either one will implicitly set the other, since each is the inverse of the other.

`int gray_to_rgb`

If set to nonzero, texture lookups of single-channel (grayscale) images will replicate the sole channel’s values into the next two channels, making it behave like an RGB image that happens to have all three channels with identical pixel values. (Channels beyond the third will get the “fill” value.)

The default value of zero means that all missing channels will get the “fill” color.

`string latlong_up`

Sets the default “up” direction for latlong environment maps (only applies if the map itself doesn’t specify a format or is in a format that explicitly requires a particular orientation). The default is “y”. (Currently any other value will result in *z* being “up.”)

`string options`

This catch-all is simply a comma-separated list of `name=value` settings of named options. For example,

```
ic->attribute ("options", "max_memory_MB=512.0,autotile=1");
```

### 8.3.3 Opaque data for performance lookups

`Perthread * get_perthread_info ()`

Retrieves an opaque handle for per-thread info, to be used for `get_texture_handle()` and the texture routines that take handles directly.

```
TextureHandle * get_texture_handle (usttring filename,  
                                   Perthread *thread_info=NULL)
```

Retrieve an opaque handle for fast texture lookups. The opaque pointer `thread_info` is thread-specific information returned by `get_perthread_info()`. Return `NULL` if something has gone horribly wrong.

### 8.3.4 Texture Lookups

```
bool texture (usttring filename, TextureOpt &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.

```
int subimage
```

The subimage or face within the file. This will be ignored if the file does not have multiple subimages or separate per-face textures.

```
Wrap swrap, twrap
```

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

```
float swidth, twidth
```

For each direction, gives a multiplier for the derivatives.

```
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.

```
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 4-channel lookup on a 3-channel texture, the last channel will get the fill value. (Note: this behavior is affected by the `"gray_to_rgb"` attribute described in Section 8.3.2.)

```
const float *missingcolor
```

If not NULL, specifies the color that will be returned for missing or broken textures (rather than being an error).

```
float *dresultds, *dresultdt
```

If not NULL, specifies locations in which to store the *s* and *t* derivatives of the filtered texture lookup.

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 (TextureHandle *texture_handle, Perthread *thread_info,
               TextureOpt &options,
               float s, float t, float dsdx, float dtdx,
               float dsdy, float dtdy, float *result)
```

A slightly faster texture call for applications that are willing to do the extra housekeeping of knowing the handle of the texture they are accessing and the per-thread info for the current thread. These may be retrieved by the `get_texture_handle()` and `get_perthread_info()` methods, respectively.

```
bool texture (ustring filename, TextureOptions &options,
               Runflag *runflags, int beginactive, int endactive,
               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 many of the fields in the options.

Texture will be computed at indices *beginactive* through *endactive* (exclusive of the end), 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 Volume Texture Lookups

```
bool texture3d (ustring filename, TextureOpt &options,
                 const Imath::V3f &P, const Imath::V3f &dPdx,
                 const Imath::V3f &dPdy, const Imath::V3f &dPdZ,
                 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` and depth `z`. By knowing how `P` changes from pixel to pixel in the final image, and as we step in `z` depth, we can properly *filter* or antialias the texture lookups. This information is given via derivatives `dPdx`, `dPdy`, and `dPdz` that define the changes in `P` per unit of `x`, `y`, and `z`, 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.

The `P` coordinate and `dPdx`, `dPdy`, and `dPdz` derivatives are assumed to be in some kind of common global coordinate system (usually "world" space) and will be automatically transformed into volume local coordinates, if such a transformation is specified in the volume file itself.

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, rwrap`

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

`float swidth, twidth, rwidth`

For each direction, gives a multiplier for the derivatives.

`float sblur, tblur, rblur`

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.

`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 4-channel lookup on a 3-channel texture, the last channel will get the fill value. (Note: this behavior is affected by the "gray\_to\_rgb" attribute described in Section 8.3.2.)

`const float *missingcolor`

If not NULL, specifies the color that will be returned for missing or broken textures (rather than being an error).

`float time`

A time value to use if the volume texture specifies a time-varying local transformation (default: 0).



```
float *dresultds, *dresultdt, *dresultdr
```

If not NULL, specifies locations in which to store the  $s$ ,  $t$ , and  $r$  derivatives of the filtered texture lookup.

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 texture3d (TextureHandle *texture_handle, Perthread *thread_info,
                TextureOpt &opt,
                const Imath::V3f &P,
                const Imath::V3f &dPdx,
                const Imath::V3f &dPdy,
                const Imath::V3f &dPdz,
                float *result)
```

A slightly faster `texture3d` call for applications that are willing to do the extra house-keeping of knowing the handle of the texture they are accessing and the per-thread info for the current thread. These may be retrieved by the `get_texture_handle()` and `get_perthread_info()` methods, respectively.

```
bool texture3d (ustring filename, TextureOptions &options,
                Runflag *runflags, int beginactive, int endactive,
                VaryingRef<Imath::V3f> P,
                VaryingRef<Imath::V3f> dPdx,
                VaryingRef<Imath::V3f> dPdy,
                VaryingRef<Imath::V3f> dPdz,
                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`, `dPdy`, and `dPdz` 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 `beginactive` through `endactive` (exclusive of the end), 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 Shadow Lookups

```
bool shadow (ustring filename, TextureOpt &opt,
              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:

`float swidth, twidth`

For each direction, gives a multiplier for the derivatives.

`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.

`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.

`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 (TextureHandle *texture_handle, Perthread *thread_info,
              TextureOpt &opt,
              const Imath::V3f &P,
              const Imath::V3f &dPdx,
              const Imath::V3f &dPdy,
              float *result)
```

A slightly faster shadow call for applications that are willing to do the extra housekeeping of knowing the handle of the texture they are accessing and the per-thread info for the current thread. These may be retrieved by the `get_texture_handle()` and `get_perthread_info()` methods, respectively.

```
bool shadow (ustring filename, TextureOptions &options,
             Runflag *runflags, int beginactive, int endactive,
             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 many the fields in the options.

Shadow lookups will be computed at indices `beginactive` through `endactive` (exclusive of the end), 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 Environment Lookups

```
bool environment (ustring filename, TextureOpt &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.

```
float swidth, twidth
```

For each direction, gives a multiplier for the derivatives.

```
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.

```
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 4-channel lookup on a 3-channel texture, the last channel will get the fill value. (Note: this behavior is affected by the "gray\_to\_rgb" attribute described in Section 8.3.2.)

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 (TextureHandle *texture_handle, Perthread *thread_info,
    TextureOpt &opt,
    const Imath::V3f &R,
    const Imath::V3f &dRdx,
    const Imath::V3f &dRdy,
    float *result)
```

A slightly faster `environment` call for applications that are willing to do the extra house-keeping of knowing the handle of the texture they are accessing and the per-thread info for the current thread. These may be retrieved by the `get_texture_handle()` and `get_perthread_info()` methods, respectively.

```
bool environment (ustring filename, TextureOptions &options,
    Runflag *runflags, int beginactive, int endactive,
    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 many the fields in the `options`.

Results will be computed at indices `beginactive` through `endactive` (exclusive of the end), 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.

### 8.3.8 Texture Metadata and Raw Texels

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

Retrieves information about the texture named by `filename`. The `dataname` is a keyword indicating 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:

`exists` Return 1 if the file exists and is an image format that OpenImageIO knows how to read, otherwise return 0. The data pointer is not used.

`subimages` The number of subimages/faces in the file, as an integer.

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

`resolution (int[3 ])` The 3D resolution of the texture file, which is an array of 3 integers (described as `TypeDesc(INT, 3)`) The third value will be 1 unless it's a volumetric (3D) image.

`miplevels` The number of MIPmap levels for the specified subimage (an integer).

`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).

`datawindow` Returns the pixel data window of the image, which is either an array of 4 integers (returning `xmin`, `ymin`, `xmax`, `ymax`) or an array of 6 integers (returning `xmin`, `ymin`, `zmin`, `xmax`, `ymax`, `zmax`). The `z` values may be useful for 3D/volumetric images; for 2D images they will be 0).

`displaywindow` Returns the display (a.k.a. full) window of the image, which is either an array of 4 integers (returning `xmin`, `ymin`, `xmax`, `ymax`) or an array of 6 integers (returning `xmin`, `ymin`, `zmin`, `xmax`, `ymax`, `zmax`). The `z` values may be useful for 3D/volumetric images; for 2D images they will be 0).

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

`projectionmatrix` The projection matrix, which is a  $4 \times 4$  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, int subimage, ImageSpec &spec)
```

If the named image is found and able to be opened by an available image format 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 plugin that could be found, the return value is `false`.

```
const ImageSpec * imagespec (ustring filename, int subimage)
```

If the named image is found and able to be opened by an available image format plugin, and the designated subimage exists, this function returns a pointer to an `ImageSpec` that describes it. Otherwise, if the file is not found, could not be opened, is not of a format readable by any plugin that could be find, or the designated subimage did not exist in the file, the return value is `NULL`.

This method is much more efficient than `get_imagespec()`, since it just returns a pointer to the `spec` held internally by the underlying `ImageCache` (rather than copying the `spec` to the user's memory). However, the caller must beware that the pointer is only valid as long as nobody (even other threads) calls `invalidate()` on the file, or `invalidate_all()`, or destroys the `TextureSystem`.

```
bool get_texels (ustring filename, TextureOpt &options, int level,
                 int xbegin, int xend, int ybegin, int yend,
                 int zbegin, int zend, TypeDesc format, void *result)
```

Retrieve a rectangle of raw unfiltered texels at the named MIP-map level, storing the texel values beginning at the address specified by `result`. Note that the face/subimage is communicated through `options.subimage`. 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). The rectangular region to be retrieved includes `begin` but does not include `end` (much like STL `begin/end` usage). Requested pixels that are not part of the valid pixel data region of the image file will be filled with zero values.

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

```
int subimage
```

The subimage to retrieve.

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.

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 4-channel lookup on a 3-channel texture, the last channel will get the fill value. (Note: this behavior is affected by the "gray\_to\_rgb" attribute described in Section 8.3.2.)

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

std::string **resolve\_filename** (const std::string &filename)

Returns the true path to the given file name, with searchpath logic applied.

### 8.3.9 Miscellaneous – Statistics, errors, flushing the cache

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.

std::string **getstats** (int level=1, bool icstats=true)

Returns a big string containing useful statistics about the ImageCache operations, suitable for saving to a file or outputting to the terminal. The `level` indicates the amount of detail in the statistics, with higher numbers (up to a maximum of 5) yielding more and more esoteric information. If `icstats` is true, the returned string will also contain all the statistics of the underlying ImageCache, but if false will only contain texture-specific statistics.

void **reset\_stats** ()

Reset most statistics to be as they were with a fresh ImageCache. Caveat emptor: this does not flush the cache itself, so the resulting statistics from the next set of texture requests will not match the number of tile reads, etc., that would have resulted from a new ImageCache.

```
void invalidate (ustring filename)
```

Invalidate any loaded tiles or open file handles associated with the filename, so that any subsequent queries will be forced to re-open the file or re-load any tiles (even those that were previously loaded and would ordinarily be reused). A client might do this if, for example, they are aware that an image being held in the cache has been updated on disk. This is safe to do even if other procedures are currently holding reference-counted tile pointers from the named image, but those procedures will not get updated pixels until they release the tiles they are holding.

```
void invalidate_all (bool force=false)
```

Invalidate all loaded tiles and open file handles, so that any subsequent queries will be forced to re-open the file or re-load any tiles (even those that were previously loaded and would ordinarily be reused). A client might do this if, for example, they are aware that an image being held in the cache has been updated on disk. This is safe to do even if other procedures are currently holding reference-counted tile pointers from the named image, but those procedures will not get updated pixels until they release the tiles they are holding. If force is true, everything will be invalidated, no matter how wasteful it is, but if force is false, in actuality files will only be invalidated if their modification times have been changed since they were first opened.



## 9 Image Buffers

### 9.1 ImageBuf Introduction and Theory of Operation

ImageBuf is a utility class that stores an entire image. It provides a nice API for reading, writing, and manipulating images as a single unit, without needing to worry about any of the details of storage or I/O.

An ImageBuf can store its pixels in one of several ways:

- Allocate “local storage” to hold the image pixels internal to the ImageBuf. This storage will be freed when the ImageBuf is destroyed.
- “Wrap” pixel memory already allocated by the calling application, which will continue to own that memory and be responsible for freeing it after the ImageBuf is destroyed.
- Be “backed” by an ImageCache, which will automatically be used to retrieve pixels when requested, but the ImageBuf will not allocate separate storage for it. This brings all the advantages of the ImageCache, but can only be used for read-only ImageBuf’s that reference a stored image file.

All I/O involving ImageBuf (that is, calls to read or write) are implemented in terms of ImageCache, ImageInput, and ImageOutput underneath, and so support all of the image file formats supported by OIIO.

The ImageBuf class definition requires that you

```
#include <OpenImageIO/imagebuf.h>
```

#### 9.1.1 Helper: ROI

ROI is a small helper struct that describes a rectangular region of pixels of an image (and a channel range). An ROI holds the following data members:

```
int xbegin, xend, ybegin, yend, zbegin, zend;  
int chbegin, chend;
```

Describes the  $x, y, z$  range of the region. The end values are *exclusive*; that is, (xbegin, ybegin, zbegin) is the first pixel included in the range, (xend-1, yend-1, zend-1) is the last pixel included in the range, and (xend, yend, zend) is *one past the last pixel* in each dimension.

Similarly, `chbegin` and `chend` describe a range of channels: `chbegin` is the first channel, `chend` is *one past the last channel*.

ROI has the following member functions and friends:

`ROI ()`

A default-constructed ROI has an *undefined* region, which is interpreted to mean all valid pixels and all valid channels of an image.

`bool ROI::defined () const`

Returns `true` if the ROI is defined, having a specified region, or `false` if the ROI is undefined.

`static ROI ROI::All ()`

Returns an undefined ROI, which is interpreted to mean all valid pixels and all valid channels of an image.

`int ROI::width () const`

`int ROI::height () const`

`int ROI::depth () const`

Returns the width, height, and depth, respectively, of a defined region. These do not return sensible values for an ROI that is not `defined()`.

`imagesize_t ROI::npixels () const`

For an ROI that is `defined()`, returns the total number of pixels included in the region, or 0 for an undefined ROI.

`imagesize_t ROI::nchannels () const`

For an ROI that is `defined()`, returns the number of channels in the channel range.

`ROI roi_union (const ROI &A, const ROI &B)`

`ROI roi_intersection (const ROI &A, const ROI &B)`

Returns the union of two ROI's (an ROI that is exactly big enough to include all the pixels of both individual ROI's) or intersection of two ROI's (an ROI that contains only the pixels that are contained in *both* ROI's).

`ROI get_roi (const ImageSpec &spec)`

`ROI get_roi_full (const ImageSpec &spec)`

Return the ROI describing `spec`'s pixel data window (the `x`, `y`, `z`, `width`, `height`, `depth` fields) or the full (display) window (the `full_x`, `full_y`, `full_z`, `full_width`, `full_height`, `full_depth` fields), respectively.

```
void set_roi (const ImageSpec &spec, const ROI &newroi)
void set_roi_full (const ImageSpec &spec, const ROI &newroi)
```

Alters the spec so to make its pixel data window or the full (display) window match newroi.

## 9.2 Constructing, reading, and writing an ImageBuf

### Default constructor of an empty ImageBuf

```
ImageBuf ()
```

The default constructor makes an uninitialized ImageBuf. There isn't much you can do with an uninitialized buffer until you call `reset()`.

```
void clear ()
```

Resets the ImageBuf to a pristine state identical to that of a freshly constructed ImageBuf using the default constructor.

### Constructing and initializing a writeable ImageBuf

```
ImageBuf (const ImageSpec &spec)
ImageBuf (const std::string &name, const ImageSpec &spec)
```

Constructs a writeable ImageBuf with the given specification (including resolution, data type, metadata, etc.), initially set to all black pixels. Optionally, you may name the ImageBuf.

```
void reset (const ImageSpec &spec)
void reset (const std::string &name, const ImageSpec &spec)
```

Destroys any previous contents of the ImageBuf and re-initializes it as a writeable all-black ImageBuf with the given specification (including resolution, data type, metadata, etc.). Optionally, you may name the ImageBuf.

### Constructing a readable ImageBuf and reading from a file

Constructing a readable ImageBuf that will hold an image to be read from disk.

```
ImageBuf (const std::string &name, int subimage=0, int miplevel=0,
          ImageCache *imagecache=NULL)
```

Construct an `ImageBuf` that will be used to read the named file (at the given subimage and MIP-level, defaulting to the first in the file). But don't read it yet! The image will actually be read when other methods need to access the spec and/or pixels, or when an explicit call to `init_spec()` or `read()` is made, whichever comes first. If `imagecache` is non-NULL, the custom `ImageCache` will be used (if applicable); otherwise, a NULL `imagecache` indicates that the global/shared `ImageCache` should be used.

```
void reset (const std::string &name, int subimage=0, int miplevel=0,
           ImageCache *imagecache = NULL)
```

Destroys any previous contents of the `ImageBuf` and re-initializes it to read the named file (but doesn't actually read yet).

```
bool read (int subimage=0, int miplevel=0, bool force=false,
           TypeDesc convert=TypeDesc::UNKNOWN,
           ProgressCallback progress_callback=NULL,
           void *progress_callback_data=NULL)
```

Explicitly reads the particular subimage and MIP level of the image. Generally, this will skip the expensive read if the file has already been read into the `ImageBuf` (at the specified subimage and MIP level). It will clear and re-allocate memory if the previously allocated space was not appropriate for the size or data type of the image being read. If `convert` is set to a specific type (not `UNKNOWN`), the `ImageBuf` memory will be allocated for that type specifically and converted upon read.

In general, `read()` will try not to do any I/O at the time of the `read()` call, but rather to have the `ImageBuf` “backed” by an `ImageCache`, which will do the file I/O on demand, as pixel values are needed, and in that case the `ImageBuf` doesn't actually allocate memory for the pixels (the data lives in the `ImageCache`). However, there are several conditions for which the `ImageCache` will be bypassed, the `ImageBuf` will allocate “local” memory, and the disk file will be read directly into allocated buffer at the time of the `read()` call: (a) if the `force` parameter is true; (b) if the `convert` parameter requests a data format conversion to a type that is not the native file type and also is not one of the internal types supported by the `ImageCache` (specifically, `FLOAT` and `UINT8`); (c) if the `ImageBuf` already has local pixel memory allocated, or “wraps” an application buffer.

If `progress_callback` is non-NULL, the underlying read, if expensive, may make several calls to

```
progress_callback(progress_callback_data, portion_done);
```

which allows you to implement some sort of progress meter. Note that if the `ImageBuf` is backed by an `ImageCache`, the progress callback will never be called, since no actual file I/O will occur at this time (`ImageCache` will load tiles or scanlines on demand, as individual pixel values are needed).

Note that `read()` is not strictly necessary. If you are happy with the filename, subimage and MIP level specified by the `ImageBuf` constructor (or the last call to `reset()`), and you want the storage to be backed by the `ImageCache` (including storing the pixels in

whatever data format that implies), then the file contents will be automatically read the first time you make any other `ImageBuf` API call that requires the spec or pixel values. The only reason to call `read()` yourself is if you are changing the filename, subimage, or MIP level, or if you want to use `force=true` or a specific `convert` value to force data format conversion.

```
bool init_spec (const std::string &filename, int subimage, int miplevel)
```

This call will read the `ImageSpec` for the given file, subimage, and MIP level into the `ImageBuf`, but will not read the pixels or allocate any local storage (until a subsequent call to `read()`). This is helpful if you have an `ImageBuf` and you need to know information about the image, but don't want to do a full read yet, and maybe won't need to do the full read, depending on what's found in the spec.

Note that `init_spec()` is not strictly necessary. If you are happy with the filename, subimage and MIP level specified by the `ImageBuf` constructor (or the last call to `reset()`), then the spec will be automatically read the first time you make any other `ImageBuf` API call that requires it. The only reason to call `read()` yourself is if you are changing the filename, subimage, or MIP level, or if you want to use `force=true` or a specific `convert` value to force data format conversion.

### Constructing an `ImageBuf` that “wraps” an application buffer

```
ImageBuf (const ImageSpec &spec, void *buffer)
ImageBuf (const std::string &name, const ImageSpec &spec, void *buffer)
```

Constructs an `ImageBuf` that “wraps” a memory buffer owned by the calling application. It can write pixels to this buffer, but can't change its resolution or data type. Optionally, it names the `ImageBuf`.

### Writing an `ImageBuf` to a file

```
bool write (const std::string &filename,
            const std::string &fileformat = std::string(),
            ProgressCallback progress_callback=NULL,
            void *progress_callback_data=NULL) const
```

Write the image to the named file in the named format (an empty format means to infer the type from the filename extension). Return `true` if all went ok, `false` if there were errors writing.

By default, it will always write a scanline-oriented file, unless the `set_write_tiles()` method has been used to override this. Also, it will use the data format of the buffer itself, unless the `set_write_format()` method has been used to override the data format.

```
bool write (ImageOutput *out,
           ProgressCallback progress_callback=NULL,
           void *progress_callback_data=NULL) const
```

Write the image to the open ImageOutputout. Return true if all went ok, false if there were errors writing. It does NOT close the file when it's done (and so may be called in a loop to write a multi-image file).

```
void set_write_format (TypeDesc format=TypeDesc::UNKNOWN)
set set_write_tiles (int width=0, int height=0, int depth=0)
```

These methods allow the caller to override the data format and tile sizing when using the write() function (the variety that does not take an open ImageOutput\*).

### 9.3 Getting and setting basic information about an ImageBuf

```
bool initialized () const
```

Returns true if the ImageBuf is initialized, false if not yet initialized.

```
IBStorage storage () const
```

Returns an enumerated type describing the type of storage currently employed by the ImageBuf: UNINITIALIZED (no storage), LOCALBUFFER (the ImageBuf has allocated and owns the pixel memory), APPBUFFER (the ImageBuf “wraps” memory owned by the calling application), or IMAGECACHE (the image is backed by an ImageCache).

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

The spec() function returns a const reference to an ImageSpec that describes the image data held by the ImageBuf.

The nativespec() function returns a const reference to an ImageSpec that describes the actual data in the file that was read.

These may differ — for example, if a data format conversion was requested, if the buffer is backed by an ImageCache which stores the pixels internally in a different data format than that of the file, or if the file had differing per-channel data formats (ImageBuf must contain a single data format for all channels).

```
const std::string & name (void) const
```

Returns the name of the buffer (name of the file, for an ImageBuf read from disk).

```
const std::string & file_format_name (void) const
```

Returns the name of the file format, for an ImageBuf read from disk (for example, "openexr").

```
int subimage () const
int nsubimages () const
int miplevel () const
int nmiplevels () const
```

The `subimage()` and `miplevel()` methods return the subimage and MIP level of the image held by the `ImageBuf` (the file it came from may hold multiple subimages and/or MIP levels, but the `ImageBuf` can only store one of those at any given time).

The `nsubimages()` method returns the total number of subimages in the file, and the `nmiplevels()` method returns the total number of MIP levels in the currently-loaded subimage.

```
int nchannels () const
```

Returns the number of channels stored in the buffer (this is equivalent to `spec().nchannels`).

```
int xbegin () const
int xend () const
int ybegin () const
int yend () const
int zbegin () const
int zend () const
```

Returns the `[begin,end)` range of the pixel data window of the buffer. These are equivalent to `spec().x`, `spec().x+spec().width`, `spec().y`, `spec().y+spec().height`, `spec().z`, and `spec().z+spec().depth`, respectively.

```
int orientation () const
int oriented_width () const
int oriented_height () const
int oriented_x () const
int oriented_y () const
int oriented_full_width () const
int oriented_full_height () const
int oriented_full_x () const
int oriented_full_y () const
```

The `orientation()` returns the interpretation of the layout (top/bottom, left/right) of the image, per the table in Section B.2.

The oriented width, height, x, and y describe the pixel data window after taking the display orientation into consideration. The *full* versions the “full” (a.k.a. display) window after taking the display orientation into consideration.

```
TypeDesc pixeltype () const
```

The data type of the pixels stored in the buffer (equivalent to `spec().format`).

```
void set_full (int xbegin, int xend, int ybegin, int yend,
              int zbegin, int zend)
```

Alters the metadata of the spec in the `ImageBuf` to reset the “full” image size (a.k.a. “display window”). This does not affect the size of the pixel data window.

```
ImageSpec & specmod ()
```

This returns a *writable* reference to the `ImageSpec` describing the buffer. It’s ok to modify most of the metadata, but if you modify the spec’s `format`, `width`, `height`, or `depth` fields, you get the pain you deserve, as the `ImageBuf` will no longer have correct knowledge of its pixel memory layout. USE WITH EXTREME CAUTION.

## 9.4 Copying ImageBuf’s and blocks of pixels

```
bool copy (const ImageBuf &src)
```

Copies `src` to this – both pixel values and all metadata.

```
void copy_metadata (const ImageBuf &src)
```

Copies all metadata (except for `format`, `width`, `height`, `depth` from `src` to this.

```
bool copy_pixels (const ImageBuf &src)
```

Copies the pixels of `src` to this, but does not change the metadata (other than `format` and `resolution`) of this.

```
void swap (ImageBuf &other)
```

Swaps the entire contents of `other` and this.

```
bool get_pixel_channels (int xbegin, int xend, int ybegin, int yend,
                        int zbegin, int zend, int chbegin, int chend,
                        TypeDesc format, void *result,
                        stride_t xstride=AutoStride,
                        stride_t ystride=AutoStride,
                        stride_t zstride=AutoStride) const
```

Retrieve the rectangle of pixels spanning  $[xbegin..xend) \times [ybegin..yend) \times [zbegin..zend)$ , channels  $[chbegin, chend)$  (all with exclusive end), specified as integer pixel coordinates, at the current subimage and MIP-map level, storing the pixel values beginning at the address specified by `result` and with the given strides (by default, `AutoStride` means the usual contiguous packing of pixels) and converting into the data type described 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. Return `true` if the operation could be completed, otherwise return `false`.



## 9.5. GETTING AND SETTING INDIVIDUAL PIXEL VALUES – SIMPLE BUT SLOW

```
bool get_pixels (int xbegin, int xend, int ybegin, int yend,
                int zbegin, int zend, TypeDesc format,
                void *result, stride_t xstride=AutoStride,
                stride_t ystride=AutoStride,
                stride_t zstride=AutoStride) const
```

Retrieve the rectangle of pixels spanning  $[xbegin..xend) \times [ybegin..yend) \times [zbegin..zend)$  (all with exclusive end), specified as integer pixel coordinates, at the current subimage and MIP-map level, storing the pixel values beginning at the address specified by `result` and with the given strides (by default, `AutoStride` means the usual contiguous packing of pixels) and converting into the data type described 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. Return `true` if the operation could be completed, otherwise return `false`.

## 9.5 Getting and setting individual pixel values – simple but slow

```
float getchannel (int x, int y, int z, int c,
                 WrapMode wrap=WrapBlack) const
```

Returns the value of pixel `x`, `y`, `z`, channel `c`.

The `wrap` describes what value should be returned if the `x`, `y`, `z` coordinates are outside the pixel data window, and may be one of: `WrapBlack`, `WrapClamp`, `WrapPeriodic`, or `WrapMirror`.

```
void getpixel (int x, int y, int z, float *pixel,
               int maxchannels=1000, WrapMode wrap=WrapBlack) const
```

Retrieves pixel  $(x, y, z)$ , placing its contents in `pixel[0..n-1]`, where  $n$  is the smaller of `maxchannels` or the actual number of channels stored in the buffer. It is up to the application to ensure that `pixel` points to enough memory to hold the required number of channels.

The `wrap` describes what value should be returned if the `x`, `y`, `z` coordinates are outside the pixel data window, and may be one of: `WrapBlack`, `WrapClamp`, `WrapPeriodic`, or `WrapMirror`.

```
void interppixel_NDC_full (float s, float t, float *pixel,
                           WrapMode wrap=WrapBlack) const
```

Given 2D floating point image-space coordinates  $(s, t)$ , linearly interpolate the surrounding pixels to yield interpolated value `pixel[0..n-1]`, where  $n$  is the smaller of `maxchannels` or the actual number of channels stored in the buffer. It is up to the application to ensure that `pixel` points to enough memory to hold the required number of channels. The coordinates  $(s, t)$  are in *image space*, where  $(0,0)$  is the upper left corner of the full (a.k.a. “display”) window and  $(1,1)$  is the lower right corner of the full/display window.

The `wrap` describes how the image function should be computed outside the boundaries of the pixel data window, and may be one of: `WrapBlack`, `WrapClamp`, `WrapPeriodic`, or `WrapMirror`.

```
void setpixel (int x, int y, int z, const float *pixel, int maxchannels=1000)
```

Set the pixel with coordinates  $(x, y, z)$  to have the values `pixel[0..n-1]`. The number of channels, `n`, is the minimum of `minchannels` and the actual number of channels in the image.

### Deep data in an ImageBuf

```
bool deep () const
```

Returns `true` if the `ImageBuf` holds a “deep” image, `false` if the `ImageBuf` holds an ordinary pixel-based image.

```
int deep_samples (int x, int y, int z=0) const
```

Returns the number of deep samples for the given pixel, or 0 if there are no deep samples for that pixel (including if the pixel coordinates are outside the data area). For non-deep images, it will always return 0.

```
const void *deep_pixel_ptr (int x, int y, int z, int c) const
```

Returns a pointer to the raw array of deep samples for channel `c` of pixel  $(x, y, z)$ . This will return `NULL` if the pixel coordinates or channel number are out of range, if the pixel/channel has no deep samples, or if the image is not deep.

```
float deep_value (int x, int y, int z, int c, int s) const
```

Return the value (as a `float`) of sample `s` of channel `c` of pixel  $(x, y, z)$ . Return 0.0 if not a deep image or if the pixel coordinates, channel number, or sample number are out of range, or if it has no deep samples.

## 9.6 Miscellaneous

```
void error (const char *format, ...) const
```

This can be used to register an error associated with this `ImageBuf`.

```
bool has_error (void) const
```

Returns `true` if the `ImageBuf` has had an error and has an error message to retrieve via `geterror()`.

```
std::string geterror (void) const
```

Return the text of all error messages issued since `geterror()` was called (or an empty string if no errors are pending). This also clears the error message for next time.

```
void *localpixels ();
const void *localpixels () const;
```

Returns a raw pointer to the “local” pixel memory, if they are fully in RAM and not backed by an `ImageCache` (in which case, `NULL` will be returned). You can also test it like a `bool` to find out if pixels are local.

```
const void *pixeladdr (int x, int y, int z=0) const
void *pixeladdr (int x, int y, int z)
```

Return the address where pixel (x,y,z) is stored in the image buffer. Use with extreme caution! Will return `NULL` if the pixel values aren’t local (for example, if backed by an `ImageCache`).

## 9.7 Iterators – the fast way of accessing individual pixels

Sometimes you need to visit every pixel in an `ImageBuf` (or at least, every pixel in a large region). Using the `getpixel` and `setpixel` for this purpose is simple but very slow. But `ImageBuf` provides templated `Iterator` and `ConstIterator` types that are very inexpensive and hide all the details of local versus cached storage.

An `Iterator` is associated with a particular `ImageBuf`. The `Iterator` has a *current pixel* coordinate that it is visiting, and an *iteration range* that describes a rectangular region of pixels that it will visit as it advances. It always starts at the upper left corner of the iteration region. We say that the iterator is *done* after it has visited every pixel in its iteration range. We say that a pixel coordinate *exists* if it is within the pixel data window of the `ImageBuf`. We say that a pixel coordinate is *valid* if it is within the iteration range of the iterator.

The `ImageBuf::ConstIterator` is identical to the `Iterator`, except that `ConstIterator` may be used on a `const ImageBuf` and may not be used to alter the contents of the `ImageBuf`. For simplicity, the remainder of this section will only discuss the `Iterator`.

The `Iterator<BUFT,USERT>` is templated based on two types: `BUFT` the type of the data stored in the `ImageBuf`, and `USERT` type of the data that you want to manipulate with your code. `USERT` defaults to `float`, since usually you will want to do all your pixel math with `float`. We will thus use `Iterator<T>` synonymously with `Iterator<T,float>`.

For the remainder of this section, we will assume that you have a `float`-based `ImageBuf`, for example, if it were set up like this:

```
ImageBuf buf ("myfile.exr");
buf.read (0, 0, true, TypeDesc::FLOAT);

Iterator<BUFT> (ImageBuf &buf, WrapMode wrap=WrapDefault)
```

Initialize an iterator that will visit every pixel in the data window of `buf`, and start it out pointing to the upper left corner of the data window. The `wrap` describes what values will be retrieved if the iterator is positioned outside the data window of the buffer.

```
Iterator<BUFT> (ImageBuf &buf, const ROI &roi, WrapMode wrap=WrapDefault)
```

Initialize an iterator that will visit every pixel of `buf` within the region described by `roi`, and start it out pointing to pixel (`roi.xbegin`, `roi.ybegin`, `roi.zbegin`). The `wrap` describes what values will be retrieved if the iterator is positioned outside the data window of the buffer.

```
Iterator<BUFT> (ImageBuf &buf, int x, int y, int z, WrapMode wrap=WrapDefault)
```

Initialize an iterator that will visit every pixel in the data window of `buf`, and start it out pointing to pixel (`x`, `y`, `z`). The `wrap` describes what values will be retrieved if the iterator is positioned outside the data window of the buffer.

```
Iterator::operator++ ()
```

The `++` operator advances the iterator to the next pixel in its iteration range. (Both prefix and postfix increment operator are supported.)

```
bool Iterator::done () const
```

Returns `true` if the iterator has completed its visit of all pixels in its iteration range.

```
ROI Iterator::range () const
```

Returns the iteration range of the iterator, expressed as an `ROI`.

```
int Iterator::x () const
```

```
int Iterator::y () const
```

```
int Iterator::z () const
```

Returns the `x`, `y`, and `z` pixel coordinates, respectively, of the pixel that the iterator is currently visiting.

```
bool Iterator::valid () const
```

Returns `true` if the iterator's current pixel coordinates are within its iteration range.

```
bool Iterator::valid (int x, int y, int z=0) const
```

Returns `true` if pixel coordinate (`x`, `y`, `z`) are within the iterator's iteration range (regardless of where the iterator itself is currently pointing).

```
bool Iterator::exists () const
```

Returns true if the iterator's current pixel coordinates are within the data window of the ImageBuf.

```
bool Iterator::exists (int x, int y, int z=0) const
```

Returns true if pixel coordinate (x, y, z) are within the pixel data window of the ImageBuf (regardless of where the iterator itself is currently pointing).

```
USERT& Iterator::operator[] (int i)
```

The value of channel i of the current pixel. (The wrap mode, set up when the iterator was constructed, determines what value is returned if the iterator points outside the pixel data window of its buffer.)

```
int Iterator::deep_samples () const
```

For deep images only, retrieves the number of deep samples for the current pixel.

```
USERT& Iterator::deep_value (int c, int s)
```

For deep images only, returns the value of channel c, sample number s, at the current pixel.

### Example: Visiting all pixels to compute an average color

```
void print_channel_averages (const std::string &filename)
{
    // Set up the ImageBuf and read the file
    ImageBuf buf (filename);
    bool ok = buf.read (0, 0, true, TypeDesc::FLOAT); // Force a float buffer
    if (! ok)
        return;

    // Initialize a vector to contain the running total
    int nc = buf.nchannels();
    std::vector<float> total (n, 0.0f);

    // Iterate over all pixels of the image, summing channels separately
    for (ImageBuf::ConstIterator<float> it (buf); ! it.done(); ++it)
        for (int c = 0; c < nc; ++c)
            total[c] += it[c];

    // Print the averages
    imagesize_t npixels = buf.spec().image_pixels();
    for (int c = 0; c < nc; ++c)
        std::cout << "Channel " << c << " avg = " (total[c] / npixels) << "\n";
}
```

**Example: Set all pixels in a region to black**

```
bool make_black (ImageBuf &buf, ROI region)
{
    if (buf.spec().format != TypeDesc::FLOAT)
        return false;    // Assume it's a float buffer

    // Clamp the region's channel range to the channels in the image
    roi.chend = std::min (roi.chend, buf.nchannels);

    // Iterate over all pixels in the region...
    for (ImageBuf::Iterator<float> it (buf, region); ! it.done(); ++it) {
        if (! it.exists())    // Make sure the iterator is pointing
            continue;        // to a pixel in the data window
        for (int c = roi.chbegin; c < roi.chend; ++c)
            it[c] = 0.0f;    // clear the value
    }
    return true;
}
```

**9.8 Dealing with buffer data types**

The previous section on iterators presented examples and discussion based on the assumption that the `ImageBuf` was guaranteed to store `float` data and that you wanted all math to also be done as `float` computations. Here we will explain how to deal with buffers and files that contain different data types.

**Strategy 1: Only have `float` data in your `ImageBuf`**

When creating your own buffers, make sure they are `float`:

```
ImageSpec spec (640, 480, 3, TypeDesc::FLOAT); // <-- float buffer
ImageBuf buf ("mybuf", spec);
```

When using `ImageCache`-backed buffers, force the `ImageCache` to convert everything to `float`:

```
// Just do this once, to set up the cache:
ImageCache *cache = ImageCache::create (true /* shared cache */);
cache->attribute ("forcefloat", 1);
...
ImageBuf buf ("myfile.exr");    // Backed by the shared cache
```

Or force the read to convert to `float` in the buffer if it's not a native type that would automatically be stored as a `float` internally to the `ImageCache`:<sup>1</sup>

```
ImageBuf buf ("myfile.exr");    // Backed by the shared cache
buf.read (0, 0, false /* don't force read to local mem */,
          TypeDesc::FLOAT /* but do force conversion to float */);
```

<sup>1</sup>`ImageCache` only supports a limited set of types internally, currently only `FLOAT` and `UINT8`, and all other data types are converted to these automatically as they are read into the cache.

Or force a read into local memory unconditionally (rather than relying on the ImageCache), and convert to float:

```
ImageBuf buf ("myfile.exr");
buf.read (0, 0, true /*force read*/,
          TypeDesc::FLOAT /* force conversion */);
```

### Strategy 2: Template your iterating functions based on buffer type

Consider the following alternate version of the `make_black` function from Section 9.7:

```
template<type BUFT>
static bool make_black_impl (ImageBuf &buf, ROI region)
{
    // Clamp the region's channel range to the channels in the image
    roi.chend = std::min (roi.chend, buf.nchannels);

    // Iterate over all pixels in the region...
    for (ImageBuf::Iterator<BUFT> it (buf, region); ! it.done(); ++it) {
        if (! it.exists()) // Make sure the iterator is pointing
            continue;    // to a pixel in the data window
        for (int c = roi.chbegin; c < roi.chend; ++c)
            it[c] = 0.0f; // clear the value
    }
    return true;
}

bool make_black (ImageBuf &buf, ROI region)
{
    if (buf.spec().format == TypeDesc::FLOAT)
        return make_black_impl<float> (buf, region);
    else if (buf.spec().format == TypeDesc::HALF)
        return make_black_impl<half> (buf, region);
    else if (buf.spec().format == TypeDesc::UINT8)
        return make_black_impl<unsigned char> (buf, region);
    else if (buf.spec().format == TypeDesc::UINT16)
        return make_black_impl<unsigned short> (buf, region);
    else {
        buf.error ("Unsupported pixel data format %s", buf.spec().format);
        return false;
    }
}
```

In this example, we make an implementation that is templated on the buffer type, and then a wrapper that calls the appropriate template specialization for each of 4 common types (and logs an error in the buffer for any other types it encounters).

In fact, `imagebufalgo_util.h` provides a macro to do this (and several variants, which will be discussed in more detail in the next chapter). You could rewrite the example even more simply:

```
#include <OpenImageIO/imagebufalgo_util.h>

template<type BUFT>
static bool make_black_impl (ImageBuf &buf, ROI region)
{
    ... same as before ...
}

bool make_black (ImageBuf &buf, ROI region)
{
    OIIO_DISPATCH_COMMON_TYPES ("make_black", make_black_impl,
                                buf.spec().format, buf, region);
}
```

This other type-dispatching helper macros will be discussed in more detail in Chapter 10.



# 10 Image Processing

## 10.1 ImageBufAlgo general principles

ImageBufAlgo is a set of image processing functions that operate on ImageBuf's. The functions are declared in the header file `OpenImageIO/imagebufalgo.h` and are declared in the namespace `ImageBufAlgo`.

### Return values and error messages

All ImageBufAlgo functions return a `bool` that is `true` if the function succeeds, `false` if the function fails. Upon failure, the *destination* ImageBuf (the one that is being altered) will have an error message set. Below is an example:

```
ImageBuf src ("input.exr");
ImageBuf dst;    // will be the output image
...
bool ok = ImageBufAlgo::crop (dst, src);
if (! ok) {
    std::string err = dst.gas_error() ? dst.geterror() : "unknown";
    std::cout << "crop error: " << err << "\n";
}
```

For a small minority of ImageBufAlgo functions, there are only input images, and no image outputs (e.g., `isMonochrome()`). In such cases, the error message should be retrieved from the first input image.

### Region of interest

Most ImageBufAlgo functions take a destination (output) ImageBuf and one or more source (input) ImageBuf's. The destination ImageBuf may or may not already be initialized (allocated and having existing pixel values). A few ImageBufAlgo functions take only a single ImageBuf parameter, which is altered in-place (and must already be initialized prior to the function call).

All ImageBufAlgo functions take an optional ROI parameter describing which subset of the image should be altered. The default value of the ROI parameter is an “undefined” ROI.

If the destination ImageBuf is already initialized, then the operation will be performed on the pixels in the destination that overlap the ROI, leaving pixels in the destination which are outside the ROI unaltered. If the ROI is also undefined, the operation will be performed on the entire destination image.

If the destination `ImageBuf` is uninitialized, it will be initialized/allocated to be the size of the `ROI`. If the `ROI` itself is undefined, it will be set to the union of the defined pixel regions of all the input images.

The usual way to use `ImageBufAlgo` functions is to have an uninitialized destination image, and pass `ROI::All()` (which is a synonym for an undefined `ROI`) for the region of interest.

Most `ImageBufAlgo` functions also respect the `chbegin` and `chend` members of the `ROI`, thus restricting the channel range on which the operation is performed. The default `ROI` constructor sets up the `ROI` to specify that the operation should be performed on all channels of the input image(s).

### Multithreading

All `ImageBufAlgo` functions take an optional `nthreads` parameter that signifies the maximum number of threads to use to parallelize the operation. The default value for `nthreads` is 0, which signifies that the number of thread should be the `OIIO` global default set by `OIIO::attribute()` (see Section 2.4.3), which itself defaults to be the detected level of hardware concurrency (number of cores available).

Generally you can ignore this parameter (or pass 0), meaning to use all the cores available in order to perform the computation as quickly as possible. The main reason to explicitly pass a different number (generally 1) is if the application is multithreaded at a high level, and the thread calling the `ImageBufAlgo` function just wants to continue doing the computation without spawning additional threads, which might tend to crowd out the other application threads.

## 10.2 Pattern generation

```
bool zero (ImageBuf &dst, ROI roi=ROI::All(), int nthreads=0)
```

Set the destination image pixels to 0 within the specified region. This operation is performed in-place. The `dst` buffer needs to already be an initialized `ImageBuf`, otherwise there's no way to know how big to make it or what data type it should have.

Examples:

```
ImageBuf dst ("myfile.exr");
...

// Zero out whole buffer, keeping it the same size
ImageBufAlgo::zero (dst);

// Zero out just a rectangle in the upper left corner
ImageBufAlgo::zero (dst, ROI (0, 100, 0, 100));

// Zero out just the green channel, leave everything else the same
ROI roi = dst.roi ();
roi.chbegin = 1; // green
roi.chend = 2;   // one past the end of the channel region
ImageBufAlgo::zero (dst, roi);
```

```
bool fill (ImageBuf &dst, const float *values,
          ROI roi=ROI::All(), int nthreads=0)
```

Set the pixels in the destination image within the specified region to the values in `values[]`. The values array must point to at least `chend` values, or the number of channels in the image, whichever is smaller.

Examples:

```
// Create a new 640x480 RGB image, fill it with pink
ImageBuf A ("myimage", ImageSpec(640, 480, 3, TypeDesc::FLOAT));
float pink[3] = { 1, 0.5, 0.5 };
ImageBufAlgo::fill (A, pink);

// Draw a red rectangle
float red[3] = { 1, 0, 0 };
ImageBufAlgo::fill (A, red, ROI(50,100, 75, 85));
```

```
bool checker (ImageBuf &dst, float width, float height, float depth,
             const float *color1, const float *color2,
             int xoffset=0, int yoffset=0, int zoffset=0,
             ROI roi=ROI::All(), int nthreads=0)
```

Set the pixels in the destination image within the specified region to a checkerboard pattern with origin given by the offset values, checker size given by the width, height, depth values, and alternating between `color1[]` and `color2[]`. The colors must point to arrays long enough to contain values for all channels in the image.

Examples:

```
// Create a new 640x480 RGB image, fill it with a two-toned gray
// checkerboard, the checkers being 64x64 pixels each.
ImageBuf A (ImageSpec(640, 480, 3, TypeDesc::FLOAT));
float dark[3] = { 0.1, 0.1, 0.1 };
float light[3] = { 0.4, 0.4, 0.4 };
ImageBufAlgo::checker (A, 64, 64, 1, dark, light, 0, 0, 0);
```

```
bool render_text (ImageBuf &dst, int x, int y,
                 const std::string &text, int fontsize=16,
                 const std::string &fontname="",
                 const float *textcolor = NULL)
```

Render a text string into the destination, essentially doing an “over” of the antialiased character into the existing pixel data. The baseline of the first character will start at position  $(x, y)$ . The font is given by `fontname` as a full pathname to the font file (defaulting to some reasonable system font if not supplied at all), and with a nominal height of `fontsize` (in pixels). The characters will be drawn in opaque white (1.0 in all channels), unless `textcolor` is supplied (and is expected to point to a float array of length at least equal to the number of channels in `dst`).

Examples:

```
ImageBuf A (ImageSpec (640, 480, 4, TypeDesc::FLOAT));

ImageBufAlgo::render_text (A, 50, 100, "Hello, world");

float red[] = { 1, 0, 0, 1 };
ImageBufAlgo::render_text (A, 100, 200, "Go Big Red!",
                           60, "Arial Bold", red);
```

### 10.3 Image transformations and data movement

```
bool channels (ImageBuf &dst, const ImageBuf &src, int nchannels,
               const int *channelorder, const float *channelvalues=NULL,
               const std::string *newchannelnames=NULL,
               bool shuffle_channel_names=false)
```

**Generic channel shuffling:** copy src to dst, but with channels in the order specified by `channelorder[0..nchannels-1]`. Does not support in-place operation. For any channel in which `channelorder[i] < 0`, it will just make dst channel `i` be a constant value set to `channelvalues[i]` (if `channelvalues` is not NULL) or 0.0 (if `channelvalues` is NULL).

If `channelorder` is NULL, it will be interpreted as `{0, 1, ..., nchannels-1}`, meaning that it's only renaming channels, not reordering them.

If `newchannelnames` is not NULL, it points to an array of new channel names. Channels for which `newchannelnames[i]` is the empty string (or all channels, if `newchannelnames == NULL`) will be named as follows: If `shuffle_channel_names` is false, the resulting dst image will have default channel names in the usual order ("R", "G", etc.), but if `shuffle_channel_names` is true, the names will be taken from the corresponding channels of the source image – be careful with this, shuffling both channel ordering and their names could result in no semantic change at all, if you catch the drift.

Examples:

```
// Copy the first 3 channels of an RGBA, drop the alpha
ImageBuf RGBA (...); // assume it's initialized, 4 chans
ImageBuf RGB;
ImageBufAlgo::channels (RGB, RGBA, 3, NULL /*default ordering*/);

// Copy just the alpha channel, making a 1-channel image
ImageBuf Alpha;
int alpha_channel[] = { 3 /* alpha channel */ };
ImageBufAlgo::channels (Alpha, RGBA, 1, &alpha_channel);

// Swap the R and B channels
ImageBuf BRGA;
int channelorder[] = { 2 /*B*/, 1 /*G*/, 0 /*R*/, 3 /*A*/ };
```

```

ImageBufAlgo::channels (BRGA, RGBA, 4, &channelorder);

// Add an alpha channel with value 1.0 everywhere to an RGB image,
// keep the other channels with their old ordering, values, and
// names.
int channelorder[] = { 0, 1, 2, -1 /*use a float value*/ };
float channelvalues[] = { 0 /*ignore*/, 0 /*ignore*/, 0 /*ignore*/, 1.0 };
std::string channelnames[] = { "", "", "", "A" };
ImageBufAlgo::channels (RGBA, RGB, 4, channelorder,
                        channelvalues, channelnames);

```

```

bool channel_append (ImageBuf &dst, const ImageBuf &A, const ImageBuf &B,
                    ROI roi=ROI::All(), int nthreads=0)

```

Append the channels of A and B together into dst over the region of interest. If the region passed is uninitialized (the default), it will be interpreted as being the union of the pixel windows of A and B (and all channels of both images). If dst is not already initialized, it will be resized to be big enough for the region.

Examples:

```

ImageBuf RGBA (...); // assume initialized, 4 channels
ImageBuf Z (...);    // assume initialized, 1 channel
ImageBuf RGBAZ;
ImageBufAlgo::channel_append (RGBAZ, RGBA, Z);

```

```

bool flatten (ImageBuf &dst, const ImageBuf &src,
              ROI roi=ROI::All(), int nthreads=0)

```

Copy pixels from *deep* image src into non-deep dst, compositing the depth samples within each pixel to yield a single “flat” value per pixel. If src is not deep, it just copies the pixels without alteration.

Examples:

```

ImageBuf Deep ("deepalpha.exr");
ImageBuf Flat;
ImageBufAlgo::flatten (Flat, Deep);

```

```

bool crop (ImageBuf &dst, const ImageBuf &src,
           ROI roi=ROI::All(), int nthreads=0)

```

Reset dst to be the specified region of src.

Note that the crop operation does not actually move the pixels on the image plane or adjust the full/display window; it merely restricts which pixels are copied from src to dst. (Note the difference compared to cut()).

Examples:

```

// Set B to be the upper left 200x100 region of A

```

```
ImageBuf A (...); // Assume initialized
ImageBuf B;
ImageBufAlgo::crop (B, A, ROI(0,200,0,100));
```

```
bool cut (ImageBuf &dst, const ImageBuf &src,
          ROI roi=ROI::All(), int nthreads=0)
```

**NEW!**

Reset dst to be the specified region of src, but repositioned at the image plane origin and with the full/display window set to exactly cover the new pixel window. (Note the difference compared to `crop()`).

Examples:

```
// Set B to be the 100x100 region of A with origin (50,200).
ImageBuf A (...); // Assume initialized
ImageBuf B;
ImageBufAlgo::cut (B, A, ROI(50,250,200,300));
// Note: B has origin at 0,0, NOT at (50,200).
```

```
bool paste (ImageBuf &dst, int xbegin, int ybegin, int zbegin, int chbegin,
            const ImageBuf &src, ROI srcroi=ROI::All(), int nthreads=0)
```

Copy into dst, beginning at (xbegin, ybegin, zbegin), the pixels of src described by srcroi. If srcroi is `ROI::All()`, the entirety of src will be used. It will copy into channels [chbegin...], as many channels as are described by srcroi.

Examples:

```
// Paste small.exr on top of big.exr at offset (100,100)
ImageBuf Big ("big.exr");
ImageBuf Small ("small.exr");
ImageBufAlgo::paste (Big, 100, 100, 0, 0, Small);
```

```
bool flip (ImageBuf &dst, const ImageBuf &src,
           ROI roi=ROI::All(), int nthreads=0)
```

Copy src (or a subregion of src) to the corresponding pixels of dst, but with the scanlines exchanged vertically.

Examples:

```
ImageBuf A ("tahoe.exr");
ImageBuf B;
ImageBufAlgo::flip (B, A);
```

```
bool flop (ImageBuf &dst, const ImageBuf &src,
           ROI roi=ROI::All(), int nthreads=0)
```

Copy `src` (or a subregion of `src`) to the corresponding pixels of `dst`, but with the columns exchanged horizontally. Examples:

```
ImageBuf A ("tahoe.exr");
ImageBuf B;
ImageBufAlgo::flop (B, A);
```

```
bool flipflop (ImageBuf &dst, const ImageBuf &src,
               ROI roi=ROI::All(), int nthreads=0)
```

Copy `src` (or a subregion of `src`) to the corresponding pixels of `dst`, but with both the rows exchanged vertically and the columns exchanged horizontally (this is equivalent to a 180 degree rotation).

Examples:

```
ImageBuf A ("tahoe.exr");
ImageBuf B;
ImageBufAlgo::flipflop (B, A);
```

```
bool transpose (ImageBuf &dst, const ImageBuf &src,
                ROI roi=ROI::All(), int nthreads=0)
```

Copy `src` (or a subregion of `src`) to the corresponding transposed ( $x \leftrightarrow y$ ) pixels of `dst`. In other words, for all  $(x,y)$  within the region, set  $dst[y,x] = src[x,y]$ .

Examples:

```
ImageBuf A ("tahoe.exr");
ImageBuf B;
ImageBufAlgo::transpose (B, A);
```

```
bool circular_shift (ImageBuf &dst, const ImageBuf &src,
                     int xshift, int yshift, int zshift=0,
                     ROI roi=ROI::All(), int nthreads=0)
```

Copy `src` (or a subregion of `src`) to the pixels of `dst`, but circularly shifting by the given amount. To clarify, the circular shift of  $[0, 1, 2, 3, 4, 5]$  by  $+2$  is  $[4, 5, 0, 1, 2, 3]$ .

Examples:

```
ImageBuf A ("tahoe.exr");
ImageBuf B;
ImageBufAlgo::circular_shift (B, A, 200, 100);
```

```
bool resize (ImageBuf &dst, const ImageBuf &src,
             const std::string filtername="", float filtersize=0,
             ROI roi=ROI::All(), int nthreads=0)
bool resize (ImageBuf &dst, const ImageBuf &src,
             Filter2D *filter,
             ROI roi=ROI::All(), int nthreads=0)
```

Set `dst`, over the region of interest, to be a resized version of the corresponding portion of `src` (mapping such that the “full” image window of each correspond to each other, regardless of resolution). If `dst` is not yet initialized, it will be sized according to `roi`.

The caller may explicitly pass a reconstruction filter, or specify one by name and size, or if the name is the empty string `resize()` will choose a reasonable high-quality default if `NULL` is passed. The filter is used to weight the `src` pixels falling underneath it for each `dst` pixel; the filter’s size is expressed in pixel units of the `dst` image.

Examples:

```
// Resize the image to 640x480, using the default filter
ImageBuf Src ("tahoe.exr");
ImageBuf Dst;
ROI roi (0, 640, 0, 480, 0, 1, /*chans:*/ 0, Src.nchannels());
ImageBufAlgo::resize (Dst, Src, "", 0, roi);
```

```
bool resample (ImageBuf &dst, const ImageBuf &src,
               bool interpolate = true,
               ROI roi=ROI::All(), int nthreads=0)
```

Set `dst`, over the region of interest, to be a resized version of the corresponding portion of `src` (mapping such that the “full” image window of each correspond to each other, regardless of resolution). If `dst` is not yet initialized, it will be sized according to `roi`.

Unlike `ImageBufAlgo::resize()`, `resample()` does not take a filter; it just samples either with a bilinear interpolation (if `interpolate` is `true`, the default) or uses the single “closest” pixel (if `interpolate` is `false`). This makes it a lot faster than a proper `resize()`, though obviously with lower quality (aliasing when downsizing, pixel replication when upsizing).

Examples:

```
// Resample quickly to 320x240, using the default filter
ImageBuf Src ("tahoe.exr");
ImageBuf Dst;
ROI roi (0, 320, 0, 240, 0, 1, /*chans:*/ 0, Src.nchannels());
ImageBufAlgo::resample (Dst, Src, NULL, roi);
```

## 10.4 Image arithmetic

```
bool add (ImageBuf &dst, const ImageBuf &A, const ImageBuf &B,
          ROI roi=ROI::All(), int nthreads=0)
bool add (ImageBuf &dst, const ImageBuf &A, float B,
          ROI roi=ROI::All(), int nthreads=0)
bool add (ImageBuf &dst, const ImageBuf &A, const float *B,
          ROI roi=ROI::All(), int nthreads=0)
```



For all pixels and channels within the designated region, set `dst` to the sum of image A and B. B is either an image, a float (added to all channels) or a per-channel float array. All of the images must have the same number of channels.

Examples:

```
// Add images A and B, assign to Sum
ImageBuf A ("a.exr");
ImageBuf B ("b.exr");
ImageBuf Sum;
ImageBufAlgo::add (Sum, A, B);

// Add 0.2 to channels 0-2 of A
ImageBuf A ("a.exr"), Sum;
ROI roi = get_roi (A.spec());
roi.chbegin = 0; roi.chend = 3;
ImageBufAlgo::add (Sum, A, 0.2, roi);
```

```
bool sub (ImageBuf &dst, const ImageBuf &A, const ImageBuf &B,
          ROI roi=ROI::All(), int nthreads=0)
bool sub (ImageBuf &dst, const ImageBuf &A, float B,
          ROI roi=ROI::All(), int nthreads=0)
bool sub (ImageBuf &dst, const ImageBuf &A, const float *B,
          ROI roi=ROI::All(), int nthreads=0)
```

For all pixels within the designated region, subtract B from A, putting the results into `dst`. B is either an image, a float (added to all channels) or a per-channel float array. All of the images must have the same number of channels.

Examples:

```
ImageBuf A ("a.exr");
ImageBuf B ("b.exr");
ImageBuf Difference;
ImageBufAlgo::sub (Difference, A, B);
```

```
bool mul (ImageBuf &dst, const ImageBuf &A, const ImageBuf &B,
          ROI roi=ROI::All(), int nthreads=0)
bool mul (ImageBuf &dst, const ImageBuf &A, float B,
          ROI roi=ROI::All(), int nthreads=0)
bool mul (ImageBuf &dst, const ImageBuf &A, const float *B,
          ROI roi=ROI::All(), int nthreads=0)
```

For all pixels within the designated region, multiply the pixel values of image A by B (channel by channel), putting the product in `dst`. B is either an image, a float (added to all channels) or a per-channel float array. All of the images must have the same number of channels.

Examples:

```

ImageBuf A ("a.exr");
ImageBuf B ("b.exr");
ImageBuf Product;
ImageBufAlgo::mul (Product, A, B);

// Reduce intensity of A's channels 0-2 by 50%
ROI roi = get_roi (A.spec());
roi.chbegin = 0;  roi.chend = 3;
ImageBufAlgo::mul (A, A, 0.5, roi);

```

```

bool pow (ImageBuf &dst, const ImageBuf &A, float b,
          ROI roi=ROI::All(), int nthreads=0)
bool pow (ImageBuf &dst, const ImageBuf &A, const float *b,
          ROI roi=ROI::All(), int nthreads=0)

```

## NEW!

For all pixels within the designated region, raise the pixel value to the power *b* (channel by channel), putting the result in *dst*. The value *b* is either a single float (applied to all channels) or a per-channel float array.

### Examples:

```

// Gamma-correct by 2.2 channels 0-2 of the image
ROI roi = get_roi (A.spec());
roi.chbegin = 0;  roi.chend = 3;
ImageBufAlgo::mul (A, A, 1.0f/2.2f, roi);

```

```

bool channel_sum (ImageBuf &dst, const ImageBuf &src,
                  const float *weights=NULL,
                  ROI roi=ROI::All(), int nthreads=0)

```

Converts a multi-channel image into a 1-channel image via a weighted sum of channels. For each pixel of *src* within the designated ROI (defaulting to all of *src*, if not defined), sum the channels designated by *roi* and store the result in channel 0 of *dst*. If *weights* is not NULL, *weight[i]* will provide a per-channel weight (rather than defaulting to 1.0 for each channel).

### Examples:

```

// Compute luminance via a weighted sum of R,G,B
// (assuming Rec709 primaries and a linear scale)
float luma_weights[3] = { .2126, .7152, .0722 };
ImageBuf A ("a.exr");
ImageBuf B;
ROI roi = A.roi();
roi.chbegin = 0;  roi.chend = 3;
ImageBufAlgo::channel_sum (B, A, luma_weights, roi);

```

```

bool clamp (ImageBuf &dst, const ImageBuf &src,
            float min = -std::numeric_limits<float>::max(),
            float max = std::numeric_limits<float>::max(),
            bool clampalpha01 = false, ROI roi=ROI::All(), int nthreads=0)
bool clamp (ImageBuf &dst, const ImageBuf &src,
            const float *min = NULL, const float *max = NULL,
            bool clampalpha01 = false, ROI roi=ROI::All(), int nthreads=0)

```

Copy pixels from `src` to `dst` (within the `roi`), clamping between the `min` and `max` values. Additionally, if `clampalpha01` is true, then any alpha channel is clamped to the 0–1 range.

For the variety of `clamp()` in which the `min` and `max` values are float, the minimum and maximum will be applied to all color channels (or, at least, the subset of channels specified by `roi`).

For the variety of `clamp()` in which the `min` and `max` parameters are pointers, they point to arrays giving per-channel minimum and maximum clamp values. If `min` is NULL, no minimum clamping is performed, and if `max` is NULL, no maximum clamping is performed.

Examples:

```

// Clamp image buffer A in-place to the [0,1] range for all pixels.
ImageBufAlgo::clamp (A, A, 0.0f, 1.0f);

// Just clamp alpha to [0,1]
ImageBufAlgo::clamp (A, A, -std::numeric_limits<float>::max(),
                    std::numeric_limits<float>::max(), true);

// Clamp R & G to [0,0.5], leave other channels alone
std::vector<float> min (A.nchannels(), -std::numeric_limits<float>::max());
std::vector<float> max (A.nchannels(), std::numeric_limits<float>::max());
min[0] = 0.0f; max[0] = 0.5f;
min[1] = 0.0f; max[1] = 0.5f;
ImageBufAlgo::clamp (A, A, &min[0], &max[0], false);

```

```

bool rangecompress (ImageBuf &dst, const ImageBuf &src, bool useluma = false,
                    ROI roi=ROI::All(), int nthreads=0)
bool rangeexpand (ImageBuf &dst, const ImageBuf &src, bool useluma = false,
                  ROI roi=ROI::All(), int nthreads=0)

```

Some image operations (such as resizing with a “good” filter that contains negative lobes) can have objectionable artifacts when applied to images with very high-contrast regions involving extra bright pixels (such as highlights in HDR captured or rendered images). One way to address this is by compressing the range of pixel values, then performing the operation (such as a resize), then re-expanding the range of the result again. This approach can yield results that are much more pleasing (even if not exactly mathematically correct).

The `rangecompress` operation does the following: For all pixels and color channels of `src` within region `roi` (defaulting to all the defined pixels of `src`), copy the pixels to `dst` while applying a logarithmic transformation on their values. Alpha and z channels are copied but not transformed.

The `rangeexpand` operation is the opposite of `rangecompress`: it copies while rescaling the logarithmic color channel values back to a linear response.

If `useluma` is true, the luma of the first three channels (presumed to be R, G, and B) are used to compute a single scale factor for all color channels, rather than scaling all channels individually (which could result in a big color shift when performing `rangecompress` and `rangeexpand`).

Examples:

```
// Resize the image to 640x480, using a Lanczos3 filter, which
// has negative lobes. To prevent those negative lobes from
// producing ringing or negative pixel values for HDR data,
// do range compression, then resize, then re-expand the range.

// 1. Read the original image
ImageBuf Src ("tahoeHDR.exr");

// 2. Range compress to a logarithmic scale
ImageBuf Compressed;
ImageBufAlgo::rangecompress (Compressed, Src);

// 3. Now do the resize
ImageBuf Dst;
ROI roi (0, 640, 0, 480, 0, 1, /*chans:*/ 0, Compressed.nchannels());
ImageBufAlgo::resize (Dst, Compressed, "lanczos3", 6.0, roi);

// 4. Expand range to be linear again (operate in-place)
ImageBufAlgo::rangeexpand (Dst, Dst);
```

```
bool over (ImageBuf &dst, const ImageBuf &A, const ImageBuf &B,
          ROI roi=ROI::All(), int nthreads=0)
```

For all pixels within the designated region, combine the pixels of images A and B using the Porter-Duff “over” compositing operation, putting the result in `dst`. Image A is the “foreground,” and B is the “background.” Images A and B must have the same number of channels and must both have an alpha channel.

Examples:

```
ImageBuf A ("fg.exr");
ImageBuf B ("bg.exr");
ImageBuf Composite;
ImageBufAlgo::over (Composite, A, B);
```

```
bool zover (ImageBuf &dst, const ImageBuf &A, const ImageBuf &B,
            bool z_zeroisinf = false, ROI roi=ROI::All(), int nthreads=0)
```

For all pixels within the designated region, combine the pixels of images A and B using the Porter-Duff “over” compositing operation, putting the result in dst. A and B must have the same number of channels and must both alpha and z (depth) channels. Rather than A always being the foreground (as it would be for the `over()` function, the z channel is used to select which image is foreground and which is background for each pixel separately, with a lower z value being the foreground for that pixel. If `z_zeroisinf` is true, then `z = 0` values will be treated as if they are infinitely far away.

Examples:

```
ImageBuf A ("a.exr");
ImageBuf B ("b.exr");
ImageBuf Composite;
ImageBufAlgo::zover (Composite, A, B);
```

## 10.5 Image comparison and statistics

```
bool computePixelStats (PixelStats &stats, const ImageBuf &src,
                        ROI roi=ROI::All(), int nthreads=0)
```

Compute statistics about the ROI of the image src, storing results in stats (each of the vectors within stats will be automatically resized to the number of channels in the image). A return value of true indicates success, false indicates that it was not possible to complete the operation. The PixelStats structure is defined as follows:

```
struct PixelStats {
    std::vector<float> min;
    std::vector<float> max;
    std::vector<float> avg;
    std::vector<float> stddev;
    std::vector<imagesize_t> nancount;
    std::vector<imagesize_t> infcount;
    std::vector<imagesize_t> finitecount;
    std::vector<double> sum, sum2; // for intermediate calculation
};
```

Examples:

```
ImageBuf A ("a.exr");
ImageBufAlgo::PixelStats stats;
ImageBufAlgo::computePixelStats (stats, A);
for (int c = 0; c < A.nchannels(); ++c) {
    std::cout << "Channel " << c << ":\n";
    std::cout << "    min = " << stats.min[c] << "\n";
    std::cout << "    max = " << stats.max[c] << "\n";
    std::cout << "    average = " << stats.avg[c] << "\n";
    std::cout << "    standard deviation = " << stats.stddev[c] << "\n";
```

```

std::cout << "    # NaN values      = " << stats.nancount[c] << "\n";
std::cout << "    # Inf values       = " << stats.infcount[c] << "\n";
std::cout << "    # finite values = " << stats.finitecount[c] << "\n";
}

```

```

bool compare (const ImageBuf &A, const ImageBuf &B,
              float failthresh, float warnthresh, CompareResults &result,
              ROI roi=ROI::All(), int nthreads=0)

```

Numerically compare two images. The difference threshold (for any individual color channel in any pixel) for a “failure” is failthresh, and for a “warning” is warnthresh. The results are stored in result. If roi is defined, pixels will be compared for the pixel and channel range that is specified. If roi is not defined, the comparison will be for all channels, on the union of the defined pixel windows of the two images (for either image, undefined pixels will be assumed to be black). The CompareResults structure is defined as follows:

```

struct CompareResults {
    double meanerror, rms_error, PSNR, maxerror;
    int maxx, maxy, maxz, maxc;
    imagesize_t nwarn, nfail;
};

```

Examples:

```

ImageBuf A ("a.exr");
ImageBuf B ("b.exr");
ImageBufAlgo::CompareResults comp;
ImageBufAlgo::compare (A, B, 1.0f/255.0f, 0.0f, comp);
if (comp.nwarn == 0 && comp.nfail == 0) {
    std::cout << "Images match within tolerance\n";
} else {
    std::cout << "Image differed: " << comp.nfail << " failures, "
               << comp.nwarn << " warnings.\n";
    std::cout << "Average error was " << comp.meanerror << "\n";
    std::cout << "RMS error was " << comp.rms_error << "\n";
    std::cout << "PSNR was " << comp.PSNR << "\n";
    std::cout << "largest error was " << comp.maxerror
               << " on pixel (" << comp.maxx << ", " << comp.maxy
               << ", " << comp.maxz << "), channel " << comp.maxc << "\n";
}

```

```

bool isConstantColor (const ImageBuf &src, float *color,
                      ROI roi=ROI::All(), int nthreads=0)

```

If all pixels of src within the ROI have the same values (for the subset of channels described by roi), return true and store the values in color[roi.chbegin...roi.chend-1]. Otherwise, return false.

Examples:

```

ImageBuf A ("a.exr");
std::vector<float> color (A.nchannels());
if (ImageBufAlgo::isConstantColor (A, &color[0])) {
    std::cout << "The image has the same value in all pixels: ";
    for (int c = 0; c < A.nchannels(); ++c)
        std::cout << (c ? " " : "") << color[c];
    std::cout << "\n";
} else {
    std::cout << "The image is not a solid color.\n";
}

```

```

bool isConstantChannel (const ImageBuf &src, int channel, float val,
                        ROI roi=ROI::All(), int nthreads=0)

```

Returns true if all pixels of src within the ROI have the given channel value val.

Examples:

```

ImageBuf A ("a.exr");
int alpha = A.spec().alpha_channel;
if (alpha < 0)
    std::cout << "The image does not have an alpha channel\n";
else if (ImageBufAlgo::isConstantChannel (A, alpha, 1.0f))
    std::cout << "The image has alpha = 1.0 everywhere\n";
else
    std::cout << "The image has alpha < 1 in at least one pixel\n";

```

```

bool isMonochrome (const ImageBuf &src, ROI roi=ROI::All(), int nthreads=0)

```

Returns true if the image is monochrome within the ROI, that is, for all pixels within the region, do all channels [roi.chbegin, roi.chend) have the same value? If roi is not defined (the default), it will be understood to be all of the defined pixels and channels of source.

Examples:

```

ImageBuf A ("a.exr");
ROI roi = get_roi (A.spec());
roi.chend = std::min (3, roi.chend); // only test RGB, not alpha
if (ImageBufAlgo::isMonochrome (A, roi))
    std::cout << "a.exr is really grayscale\n";

```

```

bool color_count (const ImageBuf &src, imagesize_t *count,
                  int ncolors, const float *color, const float *eps=NULL,
                  ROI roi=ROI::All(), int nthreads=0)

```

Count how many pixels in the image (within the ROI) match a list of colors. The colors to match are in:

```

colors[0 ... nchans-1]
colors[nchans ... 2*nchans-1]
...
colors[(ncolors-1)*nchans ... (ncolors*nchans)-1]

```

and so on, a total of `ncolors` consecutively stored colors of `nchans` channels each (`nchans` is the number of channels in the image, itself, it is not passed as a parameter).

The values in `eps[0..nchans-1]` are the error tolerances for a match, for each channel. Setting `eps[c]` to `numeric_limits<float>::max()` will effectively make it ignore the channel. Passing `eps == NULL` will be interpreted as a tolerance of 0.001 for all channels (requires exact matches for 8 bit images, but allows a wee bit of imprecision for float images).

Examples:

```

ImageBuf A ("a.exr");
int n = A.nchannels();

// Try to match two colors: pure red and green
std::vector<float> colors (2*n, numeric_limits<float>::max());
colors[0] = 1.0f; colors[1] = 0.0f; colors[2] = 0.0f;
colors[n+0] = 0.0f; colors[n+1] = 1.0f; colors[n+2] = 0.0f;

const int ncolors = 2;
imagesize_t count[ncolors];
ImageBufAlgo::color_count (A, count, ncolors);
std::cout << "Number of red pixels   : " << count[0] << "\n";
std::cout << "Number of green pixels : " << count[1] << "\n";

```

```

bool color_range_check (const ImageBuf &src, imagesize_t *lowcount,
                        imagesize_t *highcount, imagesize_t *inrangecount,
                        const float *low, const float *high,
                        ROI roi=ROI::All(), int nthreads=0)

```

Count how many pixels in the image (within the ROI) are outside the value range described by `low[roi.chbegin..roi.chend-1]` and `high[roi.chbegin..roi.chend-1]` as the low and high acceptable values for each color channel.

The number of pixels containing values that fall below the lower bound will be stored in `*lowcount`, the number of pixels containing values that fall above the upper bound will be stored in `*highcount`, and the number of pixels for which all channels fell within the bounds will be stored in `*inrangecount`. Any of these may be `NULL`, which simply means that the counts need not be collected or stored.

Examples:

```

ImageBuf A ("a.exr");
ROI roi = get_roi (A.spec());
roi.chend = std::min (roi.chend, 4); // only compare RGBA

```



```

float low[] = {0, 0, 0, 0};
float high[] = {1, 1, 1, 1};

imagesize_t lowcount, highcount, inrangecount;
ImageBufAlgo::color_range_check (A, &lowcount, &highcount, &inrangecount,
                                low, high, roi);
std::cout << lowcount << " pixels had components < 0\n";
std::cout << highcount << " pixels had components > 1\n";
std::cout << inrangecount << " pixels were fully within [0,1] range\n";

std::string computePixelHashSHA1 (const ImageBuf &src,
                                   const std::string &extrainfo = "",
                                   ROI roi=ROI::All(), int blocksize=0, int nthreads=0)

```

Compute the SHA-1 byte hash for all the pixels in the specified region of the image. If `blocksize > 0`, the function will compute separate SHA-1 hashes of each `blocksize` batch of scanlines, then return a hash of the individual hashes. This is just as strong a hash, but will NOT match a single hash of the entire image (`blocksize == 0`). But by breaking up the hash into independent blocks, we can parallelize across multiple threads, given by `nthreads`. The `extrainfo` provides additional text that will be incorporated into the hash.

Examples:

```

ImageBuf A ("a.exr");
std::string hash;
hash = ImageBufAlgo::computePixelHashSHA1 (A, "", ROI::All(), 64);

```

```

bool histogram (const ImageBuf &src, int channel,
               std::vector<imagesize_t> &histogram, int bins=256,
               float min=0, float max=1, imagesize_t *submin=NULL,
               imagesize_t *supermax=NULL, ROI roi=ROI::All())

```

Computes a histogram of the given channel of image `src`, within the ROI, as follows: The vector `histogram[0..bins-1]` will contain the count of pixels whose value was in each of the equally-sized range bins between `min` and `max`; if `submin` is not NULL, it specifies storage of the number of pixels whose value was `< min`; if `supermax` is not NULL, it specifies storage of the number of pixels whose value was `> max`.

Examples:

```

ImageBuf Src ("tahoe.exr");
const int bins = 4;
std::vector<imagesize_t> hist (bins, 0);
imagesize_t submin=0, supermax=0;
ImageBufAlgo::histogram (Src, 0, hist, bins, 0.0f, 1.0f,
                        &submin, &supermax);
std::cout << "Channel 0 of the image had:\n";

```

```

float binsize = (max-min)/nbins;
for (int i = 0; i < nbins; ++i)
    hist[i] << " pixels that are >= " << (min+i*binsize) << " and "
        << (i == nbins-1 ? " <= " : " < ")
        << (min+(i+1)*binsize) << "\n";
std::cout << submin << " pixels < " << min << "\n";
std::cout << supermax << " pixels > " << max << "\n";

```

## 10.6 Convolutions

```

bool make_kernel (ImageBuf &dst, const char *name,
                  float width, float height, float depth = 1.0f,
                  bool normalize = true)

```

Initialize `dst` to be a 1-channel float image of the named kernel. The size of the `dst` image will be big enough to contain the kernel given its size ( $\text{width} \times \text{height}$ ) and rounded up to odd resolution so that the center of the kernel can be at the center of the middle pixel. The kernel image will be offset so that its center is at the  $(0, 0)$  coordinate. If `normalize` is true, the values will be normalized so that they sum to 1.0.

If  $\text{depth} > 1$ , a volumetric kernel will be created. Use with caution!

Kernel names can be: "gaussian", "sharp-gaussian", "box", "triangle", "mitchell", "blackman-harris", "b-spline", "catmull-rom", "lanczos3", "disk", "binomial". Note that "catmull-rom" and "lanczos3" are fixed-size kernels that don't scale with the width, and are therefore probably less useful in most cases.

Examples:

```

ImageBuf K;
ImageBufAlgo::make_kernel (K, "gaussian", 5.0f, 5.0f);

```

```

bool convolve (ImageBuf &dst, const ImageBuf &src,
               const ImageBuf &kernel, bool normalize = true,
               ROI roi=ROI::All(), int nthreads=0)

```

Replace the given ROI of `dst` with the convolution of `src` and a kernel. If `roi` is not defined, it defaults to the full size of `dst` (or `src`, if `dst` was uninitialized). If `dst` is uninitialized, it will be allocated to be the size specified by `roi`. If `normalize` is true, the kernel will be normalized for the convolution, otherwise the original values will be used.

Examples:

```

// Blur an image with a 5x5 Gaussian kernel
ImageBuf Src ("tahoe.exr");
ImageBuf K;
ImageBufAlgo::make_kernel (K, "gaussian", 5.0f, 5.0f);
ImageBuf Blurred;
ImageBufAlgo::convolve (Blurred, Src, K);

```

```
bool fft (ImageBuf &dst, const ImageBuf &src,
          ROI roi=ROI::All(), int nthreads=0)
bool ifft (ImageBuf &dst, const ImageBuf &src,
           ROI roi=ROI::All(), int nthreads=0)
```

The `fft()` function takes the discrete Fourier transform (DFT) of the section of `src` denoted by `roi`, storing it in `dst`. If `roi` is not defined, it will be all of `src`'s pixels. Only one channel of `src` may be transformed at a time, so it will be the first channel described by `roi` (or, again, channel 0 if `roi` is undefined). If not already in the correct format, `dst` will be re-allocated to be a 2-channel float buffer of size `roi.width() × roi.height`, with channel 0 being the “real” part and channel 1 being the “imaginary” part. The values returned are actually the unitary DFT, meaning that it is scaled by  $1/\sqrt{\text{npixels}}$ .

Examples:

```
ImageBuf Src ("tahoe.exr");

// Take the DFT of the first channel of Src
ImageBuf Freq;
ImageBufAlgo::fft (Freq, Src);

// At this point, Freq is a 2-channel float image (real, imag)
// Convert it back from frequency domain to a spatial image
ImageBuf Spatial;
ImageBufAlgo::ifft (Spatial, Freq);
```

```
bool complex_to_polar (ImageBuf &dst, const ImageBuf &src,
                       ROI roi=ROI::All(), int nthreads=0)
bool polar_to_complex (ImageBuf &dst, const ImageBuf &src,
                       ROI roi=ROI::All(), int nthreads=0)
```

The `polar_to_complex()` function transforms a 2-channel image whose channels are interpreted as complex values (real and imaginary components) into the equivalent values expressed in polar form of amplitude and phase (with phase between 0 and  $2\pi$ ).

The `complex_to_polar()` function performs the reverse transformation, converting from polar values (amplitude and phase) to complex (real and imaginary).

In either case, the section of `src` denoted by `roi` is transformed, storing the result in `dst`. If `roi` is not defined, it will be all of `src`'s pixels. Only the first two channels of `src` will be transformed.

Examples:

```
// Suppose we have a set of frequency space values expressed as
// amplitudes and phase...
ImageBuf Polar ("polar.exr");

// Convert to complex representation
ImageBuf Complex;
ImageBufAlgo::complex_to_polar (Complex, Polar);
```

**NEW!**

```
// Now, it's safe to take an IFFT of the complex image.
// Convert it back from frequency domain to a spatial image.
ImageBuf Spatial;
ImageBufAlgo::ifft (Spatial, Complex);
```

## 10.7 Image Enhancement / Restoration

```
bool fixNonFinite (ImageBuf &dst, const ImageBuf &src,
                  NonFiniteFixMode mode = NONFINITE_BOX3,
                  int *pixelsFixed = NULL,
                  ROI roi=ROI::All(), int nthreads=0)
```

Copy pixel values from `src` to `dst` (within the pixel and channel range designated by `roi`), and repair any non-finite (NaN or Inf) pixels. If `pixelsFixed` is not NULL, store in it the number of pixels that contained non-finite value.

How the non-finite values are repaired is specified by one of the following modes:

`NONFINITE_NONE` do not alter the pixels (but do count the number of nonfinite pixels in `*pixelsFixed`, if non-NULL).

`NONFINITE_BLACK` change non-finite values to 0.

`NONFINITE_BOX3` replace non-finite values by the average of any finite pixels within a 3x3 window.

This works on all pixel data types, though it's just a copy for images with pixel data types that cannot represent NaN or Inf values.

Examples:

```
ImageBuf Src ("tahoe.exr");
int pixelsFixed = 0;
ImageBufAlgo::fixNonFinite (Src, Src, ImageBufAlgo::NONFINITE_BOX3,
                           &pixelsFixed);
std::cout << "Repaired " << pixelsFixed << " non-finite pixels\n";
```

```
bool fillholes_pushpull (ImageBuf &dst, const ImageBuf &src,
                        ROI roi=ROI::All(), int nthreads=0)
```

Copy the specified ROI of `src` to `dst` and fill any holes (pixels where `alpha < 1`) with plausible values using a push-pull technique. The `src` image must have an alpha channel. The `dst` image will end up with a copy of `src`, but will have an alpha of 1.0 everywhere, and any place where the alpha of `src` was  $\neq 1$ , `dst` will have a pixel color that is a plausible “filling” of the original alpha hole.

Examples:

```
ImageBuf Src ("holes.exr");
ImageBuf Filled;
ImageBufAlgo::fillholes_pushpull (Filled, Src);
```

```
bool unsharp_mask (ImageBuf &dst, const ImageBuf &src,
    const char *kernel = "gaussian", float width = 3.0f,
    float contrast = 1.0f, float threshold = 0.0f,
    ROI roi=ROI::All(), int nthreads=0)
```

Replace the given ROI of `dst` with a sharpened version of the corresponding region of `src` using the “unsharp mask” technique. Unsharp masking basically works by first blurring the image (low pass filter), subtracting this from the original image, then adding the residual back to the original to emphasize the edges. Roughly speaking,

$$\text{dst} = \text{src} + \text{contrast} * \text{thresh}(\text{src} - \text{blur}(\text{src}))$$

The specific blur can be selected by kernel name and width. The `contrast` is a multiplier on the overall sharpening effect. The thresholding step causes all differences less than `threshold` to be squashed to zero, which can be useful for suppressing sharpening of low-contrast details (like noise) but allow sharpening of higher-contrast edges.

Examples:

```
ImageBuf Blurry ("tahoe.exr");
ImageBuf Sharp;
ImageBufAlgo::unsharp_mask (Sharp, Blurry, "gaussian", 5.0f);
```

## 10.8 Color manipulation

```
bool colorconvert (ImageBuf &dst, const ImageBuf &src,
    const char *from, const char *to, bool unpremult=false,
    ROI roi=ROI::All(), int nthreads=0)
bool colorconvert (ImageBuf &dst, const ImageBuf &src,
    const ColorProcessor *processor, bool unpremult=false,
    ROI roi=ROI::All(), int nthreads=0)
```

Copy pixels from `src` to `dst` (within the ROI), while applying a color transform to the pixel values. In-place operations (`dst` and `src` being the same image) are supported.

If `unpremult` is true, unpremultiply before color conversion, then premultiply again after the color conversion. You may want to use this flag if your image contains an alpha channel.

The first form of this function specifies the “from” and “to” color spaces by name. The second form is passed a `ColorProcessor`, which is a special object created by a `ColorConfig` (see `OpenImageIO/color.h` for details).

If OIIO was built with OpenColorIO support enabled, then the transformation may be between any two spaces supported by the active OCIO configuration, or may be a “look” transformation created by `ColorConfig::createLookTransform`. If OIIO was not built with OpenColorIO support enabled, then the only transformations available are from “sRGB” to “linear” and vice versa.

Examples:

```
#include <OpenImageIO/imagebufalgo.h>
#include <OpenImageIO/color.h>
using namespace OIIO;

ImageBuf Src ("tahoe.jpg");
ImageBuf Dst;
ColorConfig cc;
ColorProcessor *processor = cc.createColorProcessor ("vd8", "lnf");
ImageBufAlgo::colorconvert (Dst, Src, processor, true);
ColorProcessor::deleteColorProcessor (processor);

// Equivalent, though possibly less efficient if you will be
// converting many images using the same transformation:
ImageBuf Src ("tahoe.jpg");
ImageBuf Dst;
ImageBufAlgo::colorconvert (Dst, Src, "vd8", "lnf", true);
```

```
bool ociolook (ImageBuf &dst, const ImageBuf &src,
               const char *looks, const char *from, const char *to,
               bool inverse=false, bool unpremult=false,
               const char *key=NULL, const char *value=NULL,
               ROI roi=ROI::All(), int nthreads=0)
```

Copy pixels from `src` to `dst` (within the ROI), while applying an OpenColorIO “look” transform to the pixel values. In-place operations (`dst` and `src` being the same image) are supported. The `key` and `value` may optionally be used to establish a context (for example, a shot-specific transform).

If `inverse` is true, it will reverse the color transformation and look application.

If `unpremult` is true, unpremultiply before color conversion, then premultiply again after the color conversion. You may want to use this flag if your image contains an alpha channel.

Examples:

```
ImageBuf Src ("tahoe.jpg");
ImageBuf Dst;
ImageBufAlgo::ociolook (Dst, Src, "look", "vd8", "lnf", false, false,
                        "SHOT", "pe0012");
```

```
bool ociodisplay (ImageBuf &dst, const ImageBuf &src,
                  const char *display, const char *view,
                  const char *from=NULL, const char *looks=NULL,
                  bool unpremult=false,
                  const char *key=NULL, const char *value=NULL,
                  ROI roi=ROI::All(), int nthreads=0)
```

Copy pixels from `src` to `dst` (within the ROI), while applying an OpenColorIO “display” transform to the pixel values. In-place operations (`dst` and `src` being the same image) are supported. If `from` or `looks` are `NULL`, it will not override the look or source color space (subtly different than passing “”, the empty string, which means to use no look or source space). The `key` and `value` may optionally be used to establish a context (for example, a shot-specific transform).

If `inverse` is `true`, it will reverse the color transformation and look application.

If `unpremult` is `true`, unpremultiply before color conversion, then premultiply again after the color conversion. You may want to use this flag if your image contains an alpha channel.

Examples:

```
ImageBuf Src ("tahoe.exr");
ImageBuf Dst;
ImageBufAlgo::ociodisplay (Dst, Src, "sRGB", "Film", "lnt", NULL,
                           false, "SHOT", "pe0012");
```

```
bool unpremult (ImageBuf &dst, const ImageBuf &src,
                ROI roi=ROI::All(), int nthreads=0)
```

Copy pixels from `src` to `dst`, and in the process divide all color channels (those not alpha or z) by the alpha value, to “un-premultiply” them. This presumes that the image starts of as “associated alpha” a.k.a. “premultiplied.” The alterations are restricted to the pixels and channels of the supplied ROI (which defaults to all of `src`). Pixels in which the alpha channel is 0 will not be modified (since the operation is undefined in that case). This is just a copy if there is no identified alpha channel (and a no-op if `dst` and `src` are the same image).

Examples:

```
// Convert in-place from associated alpha to unassociated alpha
ImageBuf A ("a.exr");
ImageBufAlgo::unpremult (A, A);
```

```
bool premult (ImageBuf &dst, const ImageBuf &src,
              ROI roi=ROI::All(), int nthreads=0)
```

Copy pixels from `src` to `dst`, and in the process multiply all color channels (those not alpha or z) by the alpha value, to “premultiply” them. This presumes that the image starts of as “unassociated alpha” a.k.a. “non-premultiplied.” The alterations are restricted to the

pixels and channels of the supplied ROI (which defaults to all of `src`). This is just a copy if there is no identified alpha channel (and a no-op if `dst` and `src` are the same image).

Examples:

```
// Convert in-place from unassociated alpha to associated alpha
ImageBuf A ("a.exr");
ImageBufAlgo::premult (A, A);
```

## 10.9 Import / export

```
bool make_texture (MakeTextureMode mode, const ImageBuf &input,
                   const std::string &outputfilename, const ImageSpec &config,
                   std::ostream *ostream = NULL)
bool make_texture (MakeTextureMode mode, const std::string &filename,
                   const std::string &outputfilename, const ImageSpec &config,
                   std::ostream *ostream = NULL)
```

Turn an image file (either an existing `ImageBuf` or specified by `filename`) into a tiled, MIP-mapped, texture file and write to the file named by `(outputfilename)`. The `mode` describes what type of texture file we are creating and may be one of the following:

<code>MakeTxTexture</code>	Ordinary 2D texture
<code>MakeTxEnvLat1</code>	Latitude-longitude environment map
<code>MakeTxEnvLat1FromLightProbe</code>	Latitude-longitude environment map constructed from a “light probe” image.

If the `ostream` pointer is not `NULL`, it should point to a stream (for example, `&std::out`, or a pointer to a local `std::stringstream` to capture output), which is where console output and error messages will be deposited.

The `config` is an `ImageSpec` that contains all the information and special instructions for making the texture. Anything set in `config` (format, tile size, or named metadata) will take precedence over whatever is specified by the input file itself. Additionally, named metadata that starts with `"maketx: "` will not be output to the file itself, but may contain instructions controlling how the texture is created. The full list of supported configuration options is:

Named fields:

<code>format</code>	Data format of the texture file (default: UNKNOWN = same format as the input)
<code>tile_width</code>	
<code>tile_height</code>	Preferred tile size (default: 64x64x1)
<code>tile_depth</code>	

Metadata in `config.extra_attribs`:

<code>compression</code>	string Default: "zip"
--------------------------	-----------------------



fovcot	float	Default: aspect ratio of the image resolution
planarconfig	string	Default: "separate"
worldtocamera	matrix	World-to-camera matrix of the view.
worldtoscreen	matrix	World-to-screen space matrix of the view.
wrapmodes	string	Default: "black,black"
maketx:verbose	int	How much detail should go to outstream (0).
maketx:stats	int	If nonzero, print stats to outstream (0).
maketx:resize	int	If nonzero, resize to power of 2. (0)
maketx:nomipmap	int	If nonzero, only output the top MIP level (0).
maketx:updatemode	int	If nonzero, write new output only if the output file doesn't already exist, or is older than the input file. (0)
maketx:constant_color_detect	int	If nonzero, detect images that are entirely one color, and change them to be low resolution (default: 0).
maketx:monochrome_detect	int	If nonzero, change RGB images which have R==G==B everywhere to single-channel grayscale (default: 0).
maketx:opaquedetect	int	If nonzero, drop the alpha channel if alpha is 1.0 in all pixels (default: 0).
maketx:unpremult	int	If nonzero, unpremultiply color by alpha before color conversion, then multiply by alpha after color conversion (default: 0).
maketx:incolospace	string	These two together will apply a color conversion (with OpenColorIO, if compiled). Default: ""
maketx:outcolospace	string	
maketx:checknan	int	If nonzero, will consider it an error if the input image has any NaN pixels. (0)
maketx:fixnan	string	If set to "black" or "box3", will attempt to repair any NaN pixels found in the input image (default: "none").
maketx:set_full_to_pixels	int	If nonzero, doctors the full/display window of the texture to be identical to the pixel/data window and reset the origin to 0,0 (default: 0).
maketx:filtername	string	If set, will specify the name of a high-quality filter to use when resampling for MIPmap levels. Default: "", use bilinear resampling.
maketx:highlightcomp	int	If nonzero, performs highlight compensation – range compression and expansion around the resize, plus clamping negative pixel values to zero. This reduces ringing when using filters with negative lobes.

<code>maketx:nchannels</code>	<code>int</code>	If nonzero, will specify how many channels the output texture should have, padding with 0 values or dropping channels, if it doesn't the number of channels in the input. (default: 0, meaning keep all input channels)
<code>maketx:channelnames</code>	<code>string</code>	If set, overrides the channel names of the output image (comma-separated).
<code>maketx:fileformatname</code>	<code>string</code>	If set, will specify the output file format. (default: "", meaning infer the format from the output filename)
<code>maketx:prman_metadata</code>	<code>int</code>	If set, output some metadata that PRMan will need for its textures. (0)
<code>maketx:oiio_options</code>	<code>int</code>	(Deprecated; all are handled by default)
<code>maketx:prman_options</code>	<code>int</code>	If nonzero, override a whole bunch of settings as needed to make textures that are compatible with PRMan. (0)
<code>maketx:mipimages</code>	<code>string</code>	Semicolon-separated list of alternate images to be used for individual MIPmap levels, rather than simply downsizing. (default: "")
<code>maketx:full_command_line</code>	<code>string</code>	The command or program used to generate this call, will be embedded in the metadata. (default: "")
<code>maketx:ignore_unassoc</code>	<code>int</code>	If nonzero, will disbelieve any evidence that the input image is unassociated alpha. (0)
<code>maketx:read_local_MB</code>	<code>int</code>	If nonzero, will read the full input file locally if it is smaller than this threshold. Zero causes the system to make a good guess at a reasonable threshold (e.g. 1 GB). (0)
<code>maketx:forcefloat</code>	<code>int</code>	Forces a conversion through float data for the sake of ImageBuf math. (1)
<code>maketx:hash</code>	<code>int</code>	Compute the sha1 hash of the file in parallel. (1)
<code>maketx:allow_pixel_shift</code>	<code>int</code>	Allow up to a half pixel shift per mipmap level. The fastest path may result in a slight shift in the image, accumulated for each mip level with an odd resolution. (0)

**Examples:**

```
// This command line:
//   maketx in.exr --hicomp --filter lanczos3 --opaque-detect \
```

```
//          -o texture.exr
// is equivalent to:

ImageBuf Input ("in.exr");
ImageSpec config;
config.attribute ("maketx:highlightcomp", 1);
config.attribute ("maketx:filtername", "lanczos3");
config.attribute ("maketx:opaquedetect", 1);
stringstream s;
bool ok = ImageBufAlgo::make_texture (ImageBufAlgo::MakeTxTexture,
                                     Input, "texture.exr", config, &s);

if (! ok)
    std::cout << "make_texture error: " << s.str() << "\n";
```

```
bool from_IplImage (ImageBuf &dst, const IplImage *ipl,
                   TypeDesc convert = TypeDesc::UNKNOWN)
```

Convert an IplImage, used by OpenCV and Intel's Image Library, and set dst to be the same image (copying the pixels). If convert is not set to UNKNOWN, try to establish dst as holding that data type and convert the IplImage data. Return true if ok, false if it couldn't figure out how to make the conversion from IplImage to an ImageBuf. If OpenImageIO was compiled without OpenCV support, this function will return false without modifying dst.

```
IplImage* to_IplImage (const ImageBuf &src)
```

Construct an IplImage\*, used by OpenCV and Intel's Image Library, that is equivalent to the ImageBufsrc. If it is not possible, or if OpenImageIO was compiled without OpenCV support, then return NULL. The ownership of the IplImage is fully transferred to the calling application.

```
bool capture_image (ImageBuf &dst, int cameranum,
                   TypeDesc convert = TypeDesc::UNKNOWN)
```

Capture a still image from a designated camera. If able to do so, store the image in dst and return true. If there is no such device, or support for camera capture is not available (such as if OpenCV support was not enabled at compile time), return false and do not alter dst.

Examples:

```
ImageBuf WebcamImage;
ImageBufAlgo::capture_image (WebcamImage, 0, TypeDesc::UINT8);
WebcamImage.save ("webcam.jpg");
```



# 11 Python Bindings

## 11.1 Overview

OpenImageIO provides Python language bindings for much of its functionality.

You must ensure that the environment variable `PYTHONPATH` includes the `python` subdirectory of the OpenImageIO installation.

A Python program must import the OpenImageIO package:

```
import OpenImageIO
```

In most of our examples below, we assume that for the sake of brevity, we will alias the package name as follows:

```
import OpenImageIO as oiio
from OpenImageIO import ImageInput, ImageOutput
from OpenImageIO import ImageBuf, ImageSpec, ImageBufAlgo
```

## 11.2 TypeDesc

The `TypeDesc` class that describes data types of pixels and metadata, described in detail in Section 2.1, is replicated for Python.

`BASETYPE`

The `BASETYPE` enum corresponds to the C++ `TypeDesc::BASETYPE` and contains the following values:

`UNKNOWN NONE UINT8 INT8 UINT16 INT16 UINT32 INT32 UINT64 INT64  
HALF FLOAT DOUBLE STRING PTR`

These names are also exported to the OpenImageIO namespace.

`AGGREGATE`

The `AGGREGATE` enum corresponds to the C++ `TypeDesc::AGGREGATE` and contains the following values:

`SCALAR VEC2 VEC3 VEC4 MATRIX44`

These names are also exported to the OpenImageIO namespace.

## VECSEMANTICS

The `VECSEMANTICS` enum corresponds to the C++ `TypeDesc::VECSEMANTICS` and contains the following values:

`NOSEMANTICS` `COLOR` `POINT` `VECTOR` `NORMAL` `TIMECODE` `KEYCODE`

These names are also exported to the `OpenImageIO` namespace.

```
TypeDesc ()
TypeDesc (basetype)
TypeDesc (basetype, aggregate)
TypeDesc (basetype, aggregate, vecsemantics)
TypeDesc (basetype, aggregate, vecsemantics, arraylen)
TypeDesc (str)
```

Construct a `TypeDesc` object.

Examples:

```
import OpenImageIO as oiio

# make a default (UNKNOWN) TypeDesc
t = TypeDesc()

# make a TypeDesc describing an unsigned 8 bit int
t = TypeDesc(oiio.UINT8)

# make a TypeDesc describing an array of 14 'half' values
t = TypeDesc(oiio.HALF, oiio.SCALAR, oiio.NOSEMANTICS, 14)

# make a TypeDesc describing a float point
t = TypeDesc(oiio.FLOAT, oiio.VEC3, oiio.POINT)

# Some constructors from a string description
t = TypeDesc("uint8")
t = TypeDesc("half[14]")
t = TypeDesc("point")      # equiv to FLOAT, VEC3, POINT
```

```
TypeDesc.TypeFloat()
TypeDesc.TypeInt()
TypeDesc.TypeString()
TypeDesc.TypeColor()
TypeDesc.TypePoint()
TypeDesc.TypeVector()
TypeDesc.TypeNormal()
TypeDesc.TypeMatrix()
```

Pre-constructed `TypeDesc` objects for some common types.

Example:

```
t = TypeDesc.TypeFloat()
```

```
string str (TypeDesc)
```

Returns a string that describes the TypeDesc.

Example:

```
print str(TypeDesc(oio.UINT16))

> int16
```

```
TypeDesc.basetype
```

```
TypeDesc.aggregate
```

```
TypeDesc.vecsemantics
```

```
TypeDesc.arraylen
```

Access to the raw fields in the TypeDesc.

Example:

```
t = TypeDesc(...)
if t.basetype == oio.FLOAT :
    print "It's made of floats"
```

```
int TypeDesc.size ()
```

```
int TypeDesc.basesize ()
```

```
TypeDesc TypeDesc.elementtype ()
```

```
int TypeDesc.numelements ()
```

```
int TypeDesc.elementsize ()
```

The `size()` is the size in bytes, of the type described. The `basesize()` is the size in bytes of the `basetype`.

The `elementtype()` is the type of each array element, if it is an array, or just the full type if it is not an array. The `elementsize()` is the size, in bytes, of the `elementtype` (thus, returning the same value as `size()` if the type is not an array). The `numelements()` method returns `arraylen` if it is an array, or 1 if it is not an array.

Example:

```
t = TypeDesc("point[2]")
print "size =", t.size()
print "elementtype =", t.elementtype()
print "elementsize =", t.elementsize()

> size = 24
> elementtype = point
> elementsize = 12
```

```
bool typedesc == typedesc
```

```
bool typedesc != typedesc
```

```
bool TypeDesc.equivalent (typedesc)
```

Test for equality or inequality. The `equivalent()` method is more forgiving than `==`, in that it considers `POINT`, `VECTOR`, and `NORMAL` vector semantics to not constitute a difference from one another.

Example:

```
f = TypeDesc("float")
p = TypeDesc("point")
v = TypeDesc("vector")
print "float==point?", (f == p)
print "vector==point?", (v == p)
print "float.equivalent(point)?", f.equivalent(p)
print "vector.equivalent(point)?", v.equivalent(p)

> float==point? False
> vector==point? False
> float.equivalent(point)? False
> vector.equivalent(point)? True
```

### 11.3 ROI

The `ROI` class that describes an image extent or region of interest, explained in detail in Section 9.1.1, is replicated for Python.

```
ROI ()
ROI (xbegin, xend, ybegin, yend)
ROI (xbegin, xend, ybegin, yend, zbegin, zend)
ROI (xbegin, xend, ybegin, yend, zbegin, zend, chbegin, chend)
```

Construct an `ROI` with the given bounds. The constructor with no arguments makes an `ROI` that is “undefined.”

Example:

```
import OpenImageIO as oiio
...
roi = ROI (0, 640, 0, 480, 0, 1, 0, 4)    # video res RGBA
```

```
int ROI.xbegin
int ROI.xend
int ROI.ybegin
int ROI.yend
int ROI.zbegin
int ROI.zend
int ROI.chbegin
int ROI.chend
```

The basic fields of the `ROI`.



ROI **ROI.All**

A pre-constructed undefined ROI.

bool **ROI.defined**

True if the ROI is defined, False if the ROI is undefined.

int **ROI.width**

int **ROI.height**

int **ROI.depth**

int **ROI.nchannels**

The number of pixels in each dimension, and the number of channels, as described by the ROI.

int **ROI.npixels**

The total number of pixels in the region described by the ROI.

ROI **get\_roi** (imagespec)

ROI **get\_roi\_full** (imagespec)

Returns the ROI corresponding to the pixel data window of the given ImageSpec, or the display/full window, respectively.

Example:

```
spec = ImageSpec(...)
roi = oiio.get_roi(spec)
```

**set\_roi** (imagespec, roi)

**set\_roi\_full** (imagespec, roi)

Alter the ImageSpec's resolution and offset to match the passed ROI.

Example:

```
# spec is an ImageSpec
# The following sets the full (display) window to be equal to the
# pixel data window:
oiio.set_roi_full (spec, oiio.get_roi(spec))
```

## 11.4 ImageSpec

The ImageSpec class that describes an image, explained in detail in Section 2.4, is replicated for Python.

**ImageSpec** ()

**ImageSpec** (basetype)

**ImageSpec** (typedesc)

**ImageSpec** (xres, yres, nchannels, basetype)

**ImageSpec** (xres, yres, nchannels, typespec)

Constructors of an ImageSpec. These correspond directly to the constructors in the C++ bindings.

Example:

```
import OpenImageIO as oiio
...

# default ctr
s = ImageSpec()

# construct with known pixel type, unknown resolution
s = ImageSpec(oiio.UINT8)

# construct with known resolution, channels, pixel data type
s = ImageSpec(640, 480, 4, oiio.HALF)
```

**ImageSpec.width**, **ImageSpec.height**, **ImageSpec.depth**

**ImageSpec.x**, **ImageSpec.y**, **ImageSpec.z**

Resolution and offset of the image data (int values).

Example:

```
s = ImageSpec (...)
print "Data window is ({},{})-({},{}).format (s.x, s.x+s.width-1,
                                             s.y, s.y+s.height-1)
```

**ImageSpec.full\_width**, **ImageSpec.full\_height**, **ImageSpec.full\_depth**

**ImageSpec.full\_x**, **ImageSpec.full\_y**, **ImageSpec.full\_z**

Resolution and offset of the “full” display window (int values).

**ImageSpec.tile\_width**, **ImageSpec.tile\_height**, **ImageSpec.tile\_depth**

For tiled images, the resolution of the tiles (int values). Will be 0 for untiled images.

typedesc **ImageSpec.format**

A TypeDesc describing the pixel data.

int **ImageSpec.nchannels**

An int giving the number of color channels in the image.

**ImageSpec.channelnames**

A tuple of strings containing the names of each color channel.

`ImageSpec.channelformats`

If all color channels have the same format, that will be `ImageSpec.format`, and `channelformats` will be `None`. However, if there are different formats per channel, they will be stored in `channelformats` as a tuple of `BASETYPE` values, and `format` will contain the “widest” of them.

Example:

```
if spec.channelformats == None:
    print "All color channels are", str(spec.format)
else:
    print "Channel formats: "
    for i in range(len(spec.channelformats)):
        print "\t", str(TypeDesc(spec.channelformats[i]))
```

`ImageSpec.alpha_channel`

`ImageSpec.z_channel`

The channel index containing the alpha or depth channel, respectively, or -1 if either one does not exist or cannot be identified.

`ImageSpec.deep`

Hold `True` if the image is a *deep* (multiple samples per pixel) image, of `False` if it is an ordinary image.

`ImageSpec.set_format (basetype)`

`ImageSpec.set_format (typedesc)`

Given a `BASETYPE` or a `TypeDesc`, sets the `format` field and also sets all the quantize fields to the defaults for the given data format.

Example:

```
s = ImageSpec ()
s.set_format (oiio.UINT8)
```

`ImageSpec.default_channel_names ()`

Sets `channel_names` to the default names given the value of the `nchannels` field.

`ImageSpec.channel_bytes ()`

`ImageSpec.channel_bytes (channel, native=False)`

Returns the size of a single channel value, in bytes (as an `int`). (Analogous to the C++ member functions, see Section 2.4.2 for details.)

```
ImageSpec.pixel_bytes ()
ImageSpec.pixel_bytes (native)
ImageSpec.pixel_bytes (chbegin, chend)
ImageSpec.pixel_bytes (chbegin, chend, native=False)
```

Returns the size of a pixel, in bytes (as an int). (Analogous to the C++ member functions, see Section 2.4.2 for details.)

```
ImageSpec.scanline_bytes (native=False)
ImageSpec.tile_bytes (native=False)
ImageSpec.image_bytes (native=False)
```

Returns the size of a scanline, tile, or the full image, in bytes (as an int). (Analogous to the C++ member functions, see Section 2.4.2 for details.)

```
ImageSpec.tile_pixels ()
ImageSpec.image_pixels ()
```

Returns the number of pixels in a tile or the full image, respectively (as an int). (Analogous to the C++ member functions, see Section 2.4.2 for details.)

```
ImageSpec.attribute (name, int)
ImageSpec.attribute (name, float)
ImageSpec.attribute (name, string)
ImageSpec.attribute (name, typedesc, data)
```

Sets a metadata value in the `extra_attrs`. If the metadata item is a single int, float, or string, you can pass it directly. For other types, you must pass the `TypeDesc` and then the data (for aggregate types or arrays, pass multiple values as a tuple).

Example:

```
s = ImageSpec (...)
s.attribute ("foo_str", "blah")
s.attribute ("foo_int", 14)
s.attribute ("foo_float", 3.14)
s.attribute ("foo_vector", TypeDesc.TypeVector, (1, 0, 11))
s.attribute ("foo_matrix", TypeDesc.TypeMatrix,
            (1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 1, 0, 1, 2, 3, 1))
```

```
ImageSpec.get_attribute (name)
ImageSpec.get_attribute (name, typedesc)
```

Retrieves a named metadata value from `extra_attrs`. The generic `get_attribute()` function returns it regardless of type, or `None` if the attribute does not exist. The typed variety will only succeed if the attribute is actually of that type specified.

Example:

```
foo = s.get_attribute ("foo")    # None if not found
foo = s.get_attribute ("foo", oiio.FLOAT)  # None if not found AND float
```

```
ImageSpec.get_int_attribute (name, defaultval=0)
ImageSpec.get_float_attribute (name, defaultval=0.0)
ImageSpec.get_string_attribute (name, defaultval="")
```

Retrieves a named metadata value from `extra_attrs`, if it is found and is of the given type; returns the default value (or a passed value) if not found.

Example:

```
# If "foo" is not found, or if it's not an int, return 0
foo = s.get_int_attribute ("foo")

# If "foo" is not found, or if it's not a string, return "blah"
foo = s.get_string_attribute ("foo", "blah")
```

ImageSpec.**extra\_attrs**

Direct access to the `extra_attrs` named metadata, appropriate for iterating over the entire list rather than searching for a particular named value.

`len(extra_attrs)`

Returns the number of extra attributes.

`extra_attrs[i].name`

The name of the indexed attribute.

`extra_attrs[i].type`

The type of the indexed attribute, as a `TypeDesc`.

`extra_attrs[i].value`

The value of the indexed attribute.

Example:

```
s = ImageSpec(...)
...
print "extra_attrs size is", len(s.extra_attrs)
for i in range(len(s.extra_attrs)) :
    print i, s.extra_attrs[i].name, s.extra_attrs[i].type, " :"
    print "\t", s.extra_attrs[i].value
print
```

**Example: Header info**

Here is an illustrative example of the use of ImageSpec, a working Python function that opens a file and prints all the relevant header information:

```
#!/usr/bin/env python
import OpenImageIO as oiio

# Print the contents of an ImageSpec
def print_imagespec (spec, subimage=0, mip=0) :
    if spec.depth <= 1 :
        print (" resolution %dx%d%d" % (spec.width, spec.height, spec.x, spec.y))
    else :
        print (" resolution %dx%d%x%d+d%d+d" %
              (spec.width, spec.height, spec.depth, spec.x, spec.y, spec.z))
    if (spec.width != spec.full_width or spec.height != spec.full_height
        or spec.depth != spec.full_depth) :
        if spec.full_depth <= 1 :
            print (" full res %dx%d+d" %
                  (spec.full_width, spec.full_height, spec.full_x, spec.full_y))
        else :
            print (" full res %dx%d%x%d+d%d+d" %
                  (spec.full_width, spec.full_height, spec.full_depth,
                   spec.full_x, spec.full_y, spec.full_z))
    if spec.tile_width :
        print (" tile size %dx%d" %
              (spec.tile_width, spec.tile_height, spec.tile_depth))
    else :
        print " untiled"
    if mip >= 1 :
        return
    print " " + str(spec.nchannels), "channels:", spec.channelnames
    print " format = ", str(spec.format)
    if spec.channelformats :
        print " channelformats = ", spec.channelformats
    print " alpha channel = ", spec.alpha_channel
    print " z channel = ", spec.z_channel
    print " deep = ", spec.deep
    for i in range(len(spec.extra_attribs)) :
        if type(spec.extra_attribs[i].value) == str :
            print " ", spec.extra_attribs[i].name, "= \" + spec.extra_attribs[i].value + "\""
        else :
            print " ", spec.extra_attribs[i].name, "=", spec.extra_attribs[i].value

def poor_mans_iinfo (filename) :
    input = ImageInput.open (filename)
    if not input :
        print 'Could not open "' + filename + '"'
        print "\tError: ", oiio.geterror()
        return
    print 'Opened "' + filename + '" as a ' + input.format_name()
    sub = 0
    mip = 0
    while True :
        if sub > 0 or mip > 0 :
            print "Subimage", sub, "MIP level", mip, ":"
            print_imagespec (input.spec(), mip=mip)
            mip = mip + 1
            if input.seek_subimage (sub, mip) :
                continue # proceed to next MIP level
            else :
                sub = sub + 1
                mip = 0
                if input.seek_subimage (sub, mip) :
                    continue # proceed to next subimage
                break # no more MIP levels or subimages
    input.close ()
```

## 11.5 ImageInput

See Chapter 4 for detailed explanations of the C++ `ImageInput` class APIs. The Python APIs are very similar. The biggest difference is that in C++, the various `read_*` functions write the pixel values into an already-allocated array that belongs to the caller, whereas the Python versions allocate and return an array holding the pixel values (or `None` if the read failed).

```
ImageInput.open (filename)
ImageInput.open (filename, config_imagespec)
```

Creates an `ImageInput` object and opens the named file. Returns the open `ImageInput` upon success, or `None` if it failed to open the file (after which, `OpenImageIO.geterror()` will contain an error message). In the second form, the optional `ImageSpec` argument `config` contains attributes that may set certain options when opening the file.

Example:

```
input = ImageInput.open ("tahoe.jpg")
if input == None :
    print "Error:", oiio.geterror()
    return
```

```
bool ImageInput.close ()
```

Closes an open image file, returning `True` if successful, `False` otherwise.

Example:

```
input = ImageInput.open (filename)
...
input.close ()
```

```
str ImageInput.format_name ()
```

Returns the format name of the open file.

Example:

```
input = ImageInput.open (filename)
if input :
    print filename, "was a", input.format_name(), "file."
    input.close ()
```

```
ImageSpec ImageInput.spec ()
```

Returns the `ImageSpec` corresponding to the currently open subimage and MIP level of the file.

Example:

```
input = ImageInput.open (filename)
spec = input.spec()
print "resolution ", spec.width, "x", spec.height
```

```
int ImageInput.current_subimage ()
int ImageInput.current_miplevel ()
```

Returns the current subimage and/or MIP level of the file.

```
bool ImageInput.seek_subimage (subimage, miplevel)
```

Repositions the file pointer to the given subimage and MIP level within the file (starting with 0). This function returns True upon success, False upon failure (which may include the file not having the specified subimage or MIP level).

Example:

```
input = ImageInput.open (filename)
mip = 0
while True :
    ok = input.seek_subimage (0, mip)
    if not ok :
        break
    spec = input.spec()
    print "MIP level", mip, "is", spec.width, "x", spec.height
```

```
array ImageInput.read_image (type=OpenImageIO.UNKNOWN)
```

Read the entire image and return the pixels as an array of  $width \times height \times depth \times nchannels$  values (or None if an error occurred). The array will be of the type specified by the TypeDesc or BASETYPE argument, or if the argument is missing or is TypeDesc (oiio.UNKNOWN), it will choose an appropriate data type that is the “smallest” type that can hold the actual types in the file.

Example:

```
input = ImageInput.open (filename)
spec = input.spec ()
pixels = input.read_image (oiio.FLOAT)
print "The first pixel is", pixels[0:spec.nchannels]
print "The second pixel is", pixels[spec.nchannels:(2*spec.nchannels)]
input.close ()
```

```
array ImageInput.read_scanline (y, z, type=OpenImageIO.UNKNOWN)
```

Read scanline number  $y$  from depth plane  $z$  from the open file, returning it as an array of  $width \times nchannels$  values (or None if an error occurred). The array will be of the type specified by the TypeDesc or BASETYPE argument, or if the argument is missing or is TypeDesc (oiio.UNKNOWN), it will choose an appropriate data type.

Example:

```
input = ImageInput.open (filename)
spec = input.spec ()
if spec.tile_width == 0 :
    for y in range(spec.y, spec.y+spec.height) :
```



```

        pixels = input.read_scanline (y, spec.z, oiio.FLOAT)
        # process the scanline
    else :
        print "It's a tiled file"
    input.close ()

```

array ImageInput.**read\_tile** (x, y, z, type=OpenImageIO.UNKNOWN)

Read the tile whose upper left corner is pixel (x,y,z) from the open file, returning it as an array of *width* × *height* × *depth* × *nchannels* values (or None if an error occurred). The array will be of the type specified by the TypeDesc or BASETYPE argument, or if the argument is missing or TypeDesc(oiio.UNKNOWN), it will choose an appropriate data type.

Example:

```

input = ImageInput.open (filename)
spec = input.spec ()
if spec.tile_width > 0 :
    for z in range(spec.z, spec.z+spec.depth, spec.tile_depth) :
        for y in range(spec.y, spec.y+spec.height, spec.tile_height) :
            for x in range(spec.x, spec.x+spec.width, spec.tile_width) :
                pixels = input.read_tile (x, y, z, oiio.FLOAT)
                # process the tile
else :
    print "It's a scanline file"
input.close ()

```

array ImageInput.**read\_scanlines** (ybegin, yend, z, chbegin, chend,  
type=OpenImageIO.UNKNOWN)

array ImageInput.**read\_tiles** (xbegin, xend, ybegin, yend, zbegin, zend,  
chbegin, chend, type=OpenImageIO.UNKNOWN)

Similar to the C++ routines, these functions read multiple scanlines or tiles at once, which in some cases may be more efficient than reading each scanline or tile separately. Additionally, they allow you to read only a subset of channels.

Example:

```

input = ImageInput.open (filename)
spec = input.spec ()

# Read the whole image, the equivalent of
#   pixels = input.read_image (type)
# but do it using read_scanlines or read_tiles:
if spec.tile_width == 0 :
    pixels = input.read_scanlines (spec.y, spec.y+spec.height, 0,
                                   0, spec.nchannels, oiio.FLOAT)
else :
    pixels = input.read_tiles (spec.x, spec.x+spec.width,
                               spec.y, spec.y+spec.height,

```

```
spec.z, spec.z+spec.depth,  
0, spec.nchannels, oiio.FLOAT)
```

```
str ImageInput.geterror ()
```

Retrieves the error message from the latest failed operation on an ImageInput.

Example:

```
input = ImageInput.open (filename)  
if not input :  
    print "Open error:", oiio.geterror()  
    # N.B. error on open must be retrieved with the global geterror(),  
    # since there is no ImageInput object!  
else :  
    pixels = input.read_image (oiio.FLOAT)  
    if not pixels :  
        print "Read_image error:", input.geterror()  
    input.close ()
```

**Example: Reading pixel values from a file to find min/max**

```
#!/usr/bin/env python
import OpenImageIO as oiio

def find_min_max (filename) :
    input = ImageInput.open (filename)
    if not input :
        print 'Could not open "' + filename + '"'
        print "\tError: ", oiio.geterror()
        return
    spec = input.spec()
    nchans = spec.nchannels
    pixels = input.read_image(oiio.FLOAT)
    if not pixels :
        print "Could not read:", input.geterror()
        return
    input.close()      # we're done with the file at this point
    minval = pixels[0:nchans]  # initialize to the first pixel value
    maxval = pixels[0:nchans]
    i = 0      # absolute index
    for z in range(spec.depth) :
        for y in range(spec.height) :
            for x in range(spec.width) :
                for c in range(nchans) :
                    if pixels[i+c] < minval[c] :
                        minval[c] = pixels[i+c]
                    if pixels[i+c] > maxval[c] :
                        maxval[c] = pixels[i+c]
                i = i + nchans      # advance the index
    print "Min values per channel were", minval
    print "Max values per channel were", maxval
```

## 11.6 ImageOutput

See Chapter 3 for detailed explanations of the C++ `ImageOutput` class APIs. The Python APIs are very similar.

`ImageOutput` **ImageOutput.create** (fileformat, plugin\_searchpath="")

Create a new `ImageOutput` capable of writing the named file format (which may also be a file name, with the type deduced from the extension). There is an optional parameter giving an colon-separated search path for finding `ImageOutput` plugins. The function returns an `ImageOutput` object, or `None` upon error (in which case, `OpenImageIO.geterror()` may be used to retrieve the error message).

Example:

```
import OpenImageIO as oiio
output = ImageOutput.create ("myfile.tif")
if not output :
    print "Error:", oiio.geterror()
```

`str` **ImageOutput.format\_name** ()

The file format name of a created `ImageOutput`.

Example:

```
output = ImageOutput.create (filename)
if output :
    print "Created output", filename, "as a", output.format_name()
```

`bool` **ImageOutput.supports** (feature)

For a created `ImageOutput`, returns `True` if the file format supports the named feature (such as "tiles", "mipmap", etc., see Section 3.3 for the full list), or `False` if this file format does not support the feature.

Example:

```
output = ImageOutput.create (filename)
if output :
    print output.format_name(), "supports..."
    print "tiles?", output.supports("tiles")
    print "multi-image?", output.supports("multiimage")
    print "MIP maps?", output.supports("mipmap")
    print "per-channel formats?", output.supports("channelformats")
```

`bool` **ImageOutput.open** (filename, imagespec, mode)

Opens the named output file, with an `ImageSpec` describing the image to be output. The mode may be one of `OpenImageIO.Create`, `OpenImageIO.AppendSubimage`, or `OpenImageIO.AppendMIPLevel`. See Section 3.3 for details. Returns `True` upon success, `False` upon failure (error messages retrieved via `ImageOutput.geterror()`).

Example:

```
output = ImageOutput.create (filename)
if not output :
    print "Error:", oiio.geterror()
spec = ImageSpec (640, 480, 3, oiio.UINT8)
ok = output.open (filename, spec, oiio.Create)
if not ok :
    print "Could not open", filename, ":", output.geterror()
```

```
bool ImageOutput.open (filename, (imagespec, ...))
```

This variety of `open()` is used specifically for multi-subimage files. A *tuple* of `ImageSpec` objects is passed, one for each subimage that will be written to the file. After each subimage is written, then a regular call to `open(name, newspec, AppendSubimage)` moves on to the next subimage.

```
bool ImageOutput.close ()
```

Closes an open output.

```
ImageSpec ImageOutput.spec ()
```

Retrieves the `ImageSpec` of the currently-open output image.

```
bool ImageOutput.write_image (typedesc, pixels, xstride=AutoStride,
                               ystride=AutoStride, zstride=AutoStride)
```

Write the currently opened image all at once. The `pixels` parameter should be an array containing data elements of the type described by the `TypeDesc`. Optional strides describe data layout in the array (the default values of `AutoStride` are used if the data are contiguous). Returns `True` upon success, `False` upon failure.

Example:

```
# This example reads a scanline file, then converts it to tiled
# and writes to the same name.

input = ImageInput.open (filename)
spec = input.spec ()
pixels = input.read_image (oiio.FLOAT)
input.close ()

output = ImageOutput.create (filename)
if output.supports("tiles") :
    spec.tile_width = 64
    spec.tile_height = 64
    output.open (filename, spec, oiio.Create)
    output.write_image (oiio.FLOAT, pixels)
    output.close ()
```

```

bool ImageOutput.write_scanline (y, z, typesdesc, pixels,
                                xstride=AutoStride)
bool ImageOutput.write_scanlines (ybegin, yend, z, typesdesc, pixels,
                                xstride=AutoStride)

```

Write one or many scanlines to the currently open file.

Example:

```

# Copy a TIFF image to JPEG by copying scanline by scanline.
input = ImageInput.open ("in.tif")
spec = input.spec ()
output = ImageOutput.create ("out.jpg")
output.open (filename, spec, oiio.Create)
for z in range(spec.z, spec.z+spec.depth) :
    for y in range(spec.y, spec.y+spec.height) :
        pixels = input.read_scanline (y, z, oiio.FLOAT)
        output.write_scanline (y, z, oiio.FLOAT, pixels)
output.close ()
input.close ()

# The same example, but copying a whole "plane" of scanlines at a time:
...
for z in range(spec.z, spec.z+spec.depth) :
    pixels = input.read_scanlines (spec.y, spec.y+spec.height,
                                  z, oiio.FLOAT)
    output.write_scanlines (spec.y, spec.y+spec.height,
                           z, oiio.FLOAT, pixels)
...

```

```

bool ImageOutput.write_tile (x, y, z, typesdesc, pixels,
                             xstride=AutoStride, ystride=AutoStride,
                             zstride=AutoStride)
bool ImageOutput.write_tiles (xbegin, xend, ybegin, yend, zbegin, zend,
                             xstride=AutoStride, ystride=AutoStride,
                             zstride=AutoStride)

```

Write one or many tiles to the currently open file.

Example:

```

input = ImageInput.open (in_filename)
spec = input.spec ()
output = ImageOutput.create (out_filename)
output.open (out_filename, spec, oiio.Create)
for z in range(spec.z, spec.z+spec.depth, spec.tile_depth) :
    for y in range(spec.y, spec.y+spec.height, spec.tile_height) :
        for x in range(spec.x, spec.x+spec.width, spec.tile_width) :
            pixels = input.read_tile (x, y, z, oiio.FLOAT)
            output.write_tile (x, y, z, oiio.FLOAT, pixels)
output.close ()
input.close ()

```

```
# The same example, but copying a whole row of of tiles at a time:
...
for z in range(spec.z, spec.z+spec.depth, spec.tile_depth) :
    for y in range(spec.y, spec.y+spec.height, spec.tile_height) :
        pixels = input.read_tiles (spec.x, spec.x+spec.width,
                                   y, y+tile_width,
                                   z, z+tile_width, oiio.FLOAT)
        output.read_tile (spec.x, spec.x+spec.width,
                           y, y+tile_width, z, z+tile_width,
                           oiio.FLOAT, pixels)
    ...
```

```
bool ImageOutput.copy_image (imageinput)
```

Copy the current image of the open input to the open output. (The reason this may be preferred in some circumstances is that, if input and output were the same kind of input file format, they may have a special efficient technique to copy pixels unaltered, for example by avoiding the decompression/recompression round trip.)

Example:

```
input = ImageInput.open (in_filename)
spec = input.spec ()
output = ImageOutput.create (out_filename)
output.open (filename, spec, oiio.Create)
output.copy_image (input)
output.close ()
input.close ()
```

```
str ImageOutput.geterror ()
```

Retrieves the error message from the latest failed operation on an open file.

Example:

```
output = ImageOutput.create (filename)
if not output :
    print "Create error:", oiio.geterror()
    # N.B. error on create must be retrieved with the global geterror(),
    # since there is no ImageOutput object!
else :
    ok = output.open (filename, spec, oiio.Create)
    if not ok :
        print "Open error:", output.geterror()
    ok = output.write_image (oiio.FLOAT, pixels)
    if not ok :
        print "Write error:", output.geterror()
    output.close ()
```

## 11.7 ImageBuf

See Chapter 9 for detailed explanations of the C++ ImageBuf class APIs. The Python APIs are very similar.

ImageBuf **ImageBuf** ()

Construct a new, empty ImageBuf. The ImageBuf is uninitialized and is awaiting a call to `reset()` or `copy()` before it is useful.

ImageBuf **ImageBuf** (filename)

ImageBuf **ImageBuf** (filename, subimage, miplevel)

Construct a read-only ImageBuf that will read from the named file. Optionally, a specific subimage or MIP level may be specified (defaulting to 0).

Example:

```
import OpenImageIO as oiio
...
buf = ImageBuf ("grid.tif")
```

ImageBuf **ImageBuf** (imagespec)

Construct a writeable ImageBuf of the dimensions and data format specified by an ImageSpec.

Example:

```
spec = ImageSpec (640, 480, 3, oiio.FLOAT)
buf = ImageBuf (spec)
```

ImageBuf.**clear** ()

Return the ImageBuf to its pristine, uninitialized state.

Example:

```
buf = ImageBuf (...)

# The following two commands are equivalent:
buf = ImageBuf()      # 1 - assign a new blank ImageBuf
buf.clear()           # 2 - clear the existing ImageBuf
```

ImageBuf.**reset** (filename)

ImageBuf.**reset** (filename, subimage, miplevel)

ImageBuf.**reset** (imagespec)

Restore the ImageBuf to its newly-constructed state, either to read from a filename (optionally specifying a subimage and MIP level) or a writeable ImageBuf specified by an ImageSpec.



```
bool ImageBuf.read (subimage=0, miplevel=0, force=False, convert=oiio.UNKNOWN)
```

Explicitly read the image from the file (of a file-reading ImageBuf), optionally specifying a particular subimage and MIP level. If `force` is `True`, will force an allocation of memory and a full read (versus the default of relying on an underlying ImageCache). If `convert` is not the default of `UNKNOWN`, it will force the ImageBuf to convert the image to the specified data format (versus keeping it in the native format or relying on the ImageCache to make a data formatting decision).

Note that a call to `read()` is not necessary — any ImageBuf API call that accesses pixel values will trigger a file read if it has not yet been done.

The `read()` method will return `True` for success, or `False` if the read could not be performed (in which case, a `geterror()` call will retrieve the specific error message).

Example:

```
buf = ImageBuf ("mytexture.exr")
buf.read (0, 2, True)
# That forces an allocation and read of MIP level 2
```

```
bool ImageBuf.init_spec (filename, subimage=0, miplevel=0)
```

Explicitly read just the header from a file-reading ImageBuf (if the header has not yet been read), optionally specifying a particular subimage and MIP level. The `init_spec()` method will return `True` for success, or `False` if the read could not be performed (in which case, a `geterror()` call will retrieve the specific error message).

Note that a call to `init_spec()` is not necessary — any ImageBuf API call that accesses the spec will read it automatically if it has not yet been done.

```
bool ImageBuf.write (filename, fileformat="")
```

Write the contents of the ImageBuf to the named file. Optionally, `fileformat` can specify a particular file format to use (by default, it will infer it from the extension of the file name).

Example:

```
# No-frills conversion of a TIFF file to JPEG
buf = ImageBuf ("in.tif")
buf.write ("out.jpg")
```

```
bool ImageBuf.set_write_format (format=oiio.UNKNOWN)
```

```
bool ImageBuf.set_write_tiles (width=0, height=0, depth=0)
```

Override the data format or tile size in a subsequent call to `write()`.

Example:

```
# Conversion to a tiled float file
buf = ImageBuf ("in.tif")
```

```

buf.set_write_format (oiio.FLOAT)
buf.set_write_tiles (64, 64)
buf.write ("out.tif")

```

```

ImageSpec ImageBuf.spec()
ImageSpec ImageBuf.nativespec()

```

`ImageBuf.spec()` returns the `ImageSpec` that describes the contents of the `ImageBuf`. `ImageBuf.nativespec()` returns an `ImageSpec` that describes the contents of the file that the `ImageBuf` was read from (this may differ from `ImageBuf.spec()` due to format conversions, or any changes made to the `ImageBuf` after the file was read, such as adding metadata).

Handy rule of thumb: `spec()` describes the buffer, `nativespec()` describes the original file it came from.

Example:

```

buf = ImageBuf ("in.tif")
print "Resolution is", buf.spec().width, "x", buf.spec().height

```

```

ImageSpec ImageBuf.specmod()

```

`ImageBuf.specmod()` provides writeable access to the `ImageSpec` that describes the contents of the `ImageBuf`. Be very careful! It is safe to modify certain metadata, but if you change the data format or resolution fields, you will get the chaos you deserve.

Example:

```

# Read an image, add a caption metadata, write it back out in place
buf = ImageBuf ("file.tif")
buf.specmod().attribute ("ImageDescription", "my photo")
buf.write ("file.tif")

```

```

str ImageBuf.name
str ImageBuf.file_format_name

```

The file name and name of the file format of the image.

```

int ImageBuf.subimage
int ImageBuf.miplevel
int ImageBuf.nsubimages
int ImageBuf.nmiplevels

```

Several fields giving information about the current subimage and MIP level, and the total numbers thereof in the file.

```
int ImageBuf.xbegin
int ImageBuf.xend
int ImageBuf.ybegin
int ImageBuf.yend
int ImageBuf.zbegin
int ImageBuf.zend
```

The range of valid pixel data window. Remember that the end is *one past* the last pixel.

```
int ImageBuf.xmin
int ImageBuf.xmax
int ImageBuf.ymin
int ImageBuf.ymax
int ImageBuf.zmin
int ImageBuf.zmax
```

The minimum and maximum (inclusive) coordinates of the pixel data window.

```
int ImageBuf.orientation
int ImageBuf.oriented_width
int ImageBuf.oriented_height
int ImageBuf.oriented_x
int ImageBuf.oriented_y
int ImageBuf.oriented_full_width
int ImageBuf.oriented_full_height
int ImageBuf.oriented_full_x
int ImageBuf.oriented_full_y
```

The `orientation` field gives the suggested display orientation of the image (see Section B.2).

The other fields are helpers that give the width, height, and origin (as well as “full” or “display” resolution and origin), taking the intended orientation into consideration.

```
ROI ImageBuf.roi
ROI ImageBuf.roi_full
```

These fields return an ROI description of the pixel data window (`roi`) and the full (a.k.a. “display”) window (`roi_full`).

Example:

```
buf = ImageBuf ("tahoe.jpg")
print "Resolution is", buf.roi.width, "x", buf.roi.height
```

```
ImageBuf.set_full (roi)
```

Changes the “full” (a.k.a. “display”) window to the specified ROI.

Example:

```
newroi = ROI (0, 1024, 0, 768)
buf.set_full (newroi)
```

bool ImageBuf.**pixels\_valid**

Will be True if the file has already been read and the pixels are valid. (It is always True for writeable ImageBuf’s.) There should be few good reasons to access these, since the spec and pixels will be automatically be read when they are needed.

TypeDesc ImageBuf.**pixeltype**

Returns the description of the data type of the pixels stored within the ImageBuf.

bool ImageBuf.**deep**

Will be True if the file contains “deep” pixel data, or False for an ordinary images.

ImageBuf.**copy\_metadata** (other\_imagebuf)

Replaces the metadata (all ImageSpec items, except for the data format and pixel data window size) with the corresponding metadata from the other ImageBuf.

ImageBuf.**copy\_pixels** (other\_imagebuf)

Replace the pixels in this ImageBuf with the values from the other ImageBuf.

ImageBuf.**copy** (other\_imagebuf)

Make this ImageBuf a complete copy of the other ImageBuf.

Example:

```
A = ImageBuf()
B = ImageBuf("B.tif")
A.copy (B)
# Now A is a separate, duplicate copy of "B.tif"
```

ImageBuf.**swap** (other\_imagebuf)

Swaps the content of this ImageBuf and the other ImageBuf.

Example:

```
A = ImageBuf("A.tif")
B = ImageBuf("B.tif")
A.swap (B)
# Now B contains the "A.tif" image and A contains the "B.tif" image
```

```
tuple ImageBuf.getpixel (x, y, z=0, wrap=oiio.WrapBlack)
```

Retrieves pixel (x,y,z) from the buffer and return it as a tuple of float values, one for each color channel. The x, y, z values are int pixel coordinates. The optional wrap parameter describes what should happen if the coordinates are outside the pixel data window (and may be: WrapBlack, WrapClamp, WrapPeriodic, WrapMirror).

Example:

```
buf = ImageBuf ("tahoe.jpg")
p = buf.getpixel (50, 50)
print p

> (0.37, 0.615, 0.97)
```

```
float ImageBuf.getchannel (x, y, z, channel, wrap=oiio.WrapBlack)
```

Retrieves just a single channel value from pixel (x,y,z) from the buffer and returns it as a float value. The optional wrap parameter describes what should happen if the coordinates are outside the pixel data window (and may be: WrapBlack, WrapClamp, WrapPeriodic, WrapMirror).

Example:

```
buf = ImageBuf ("tahoe.jpg")
green = buf.getchannel (50, 50, 0, 1)
```

```
tuple ImageBuf.interpixel (x, y, wrap=oiio.WrapBlack)
```

Interpolates the image value at coordinates (x,y) and return it as a tuple of float values, one for each color channel. The x, y values are continuous float coordinates in “pixel space.” The optional wrap parameter describes what should happen if the coordinates are outside the pixel data window (and may be: WrapBlack, WrapClamp, WrapPeriodic, WrapMirror).

Example:

```
buf = ImageBuf ("tahoe.jpg")
midx = float(buf.xbegin + buf.xend) / 2.0
midy = float(buf.ybegin + buf.yend) / 2.0
p = buf.interpixel (midx, midy)
# Now p is the interpolated value from right in the center of
# the data window
```

```
tuple ImageBuf.interpixel_NDC_full (x, y, wrap=oiio.WrapBlack)
```

Interpolates the image value at coordinates (x,y) and return it as a tuple of float values, one for each color channel. The x, y values are continuous, normalized float coordinates in “NDC space,” where (0,0) is the upper left corner of the full (a.k.a. “display”) window, and (1,1) is the lower right corner of the full/display window. The wrap parameter describes what should happen if the coordinates are outside the pixel data window (and may be: WrapBlack, WrapClamp, WrapPeriodic, WrapMirror).

Example:

```
buf = ImageBuf ("tahoe.jpg")
p = buf.interpixel_NDC_full (0.5, 0.5)
# Now p is the interpolated value from right in the center of
# the display window
```

```
ImageBuf.setpixel (x, y, pixel_value)
ImageBuf.setpixel (x, y, z, pixel_value)
```

Sets pixel (x,y,z) to be the pixel\_value, expressed as a tuple of floats (one for each color channel).

Example:

```
buf = ImageBuf (ImageSpec (640, 480, 3, oiio.UINT8))

# Set the whole image to red (the dumb slow way, but it works):
for y in range(buf.ybegin, buf.yend) :
    for x in range(buf.xbegin, buf.xend) :
        buf.setpixel (x, y, (1.0, 0.0, 0.0))
```

```
bool ImageBuf.has_error
str ImageBuf.geterror ()
```

The ImageBuf.has\_error field will be True if an error has occurred in the ImageBuf, in which case the geterror() method will retrieve the error message (and clear it afterwards).

Example:

```
buf = ImageBuf ("in.tif")
buf.read () # force a read
if buf.has_error :
    print "Error reading the file:", buf.geterror()
buf.write ("out.jpg")
if buf.has_error :
    print "Could not convert the file:", buf.geterror()
```

## 11.8 ImageBufAlgo

The C++ ImageBufAlgo functions are described in detail in Chapter 10. They are also exposed to Python. For the majority of ImageBufAlgo functions, their use in Python is identical to C++; in those cases, we will keep our descriptions of the Python bindings minimal and refer you to Chapter 10, saving the extended descriptions for those functions that differ from the C++ counterparts.

A few things about the parameters of the ImageBufAlgo function calls are identical among the functions, so we will explain once here rather than separately for each function:

- `dst` is an existing ImageBuf, which will be modified (it may be an uninitialized ImageBuf, but it must be an ImageBuf).
- `src` parameter is an initialized ImageBuf, which will not be modified (unless it happens to refer to the same image as `dst`).
- `roi`, if supplied, is an ROI specifying a region of interest over which to operate. If omitted, the region will be the entire size of the source image(s).
- `nthreads` is the maximum number of threads to use. If not supplied, it defaults to 0, meaning to use as many threads as hardware cores available.

Just as with the C++ ImageBufAlgo functions, if `dst` is an uninitialized ImageBuf, it will be sized to reflect the `roi` (which, in turn, if undefined, will be sized to be the union of the ROI's of the source images).

### 11.8.1 Pattern generation

```
bool ImageBufAlgo.zero (dst, roi=ROI.All, nthreads=0)
```

Zero out the destination buffer (or a specific region of it).

Example:

```
# Initialize buf to a 640x480 3-channel FLOAT buffer, zero it out
buf = ImageBuf (ImageSpec (640, 480, 3, oio.FLOAT))
ImageBufAlgo.zero (buf)
```

```
bool ImageBufAlgo.fill (dst, values, roi=ROI.All, nthreads=0)
```

Set the pixels in `dst` within the ROI to the values in the tuple `values`.

Examples:

```
# Draw a red rectangle into buf
buf = ImageBuf (ImageSpec(640, 480, 3, TypeDesc.FLOAT))
ImageBufAlgo.fill (buf, (1,0,0), ROI(50, 100, 75, 85))
```

```
bool ImageBufAlgo.checker (dst, width, height, depth, color1, color2,
                           xoffset=0, yoffset=0, zoffset=0, roi=ROI.All, nthreads=0)
```

Fill `dst` with a checkerboard pattern. The colors are specified as tuples giving the values for each color channel.

Examples:

```
buf = ImageBuf(ImageSpec(640, 480, 3, oio.UINT8))
ImageBufAlgo.checker (buf, 64, 64, 1, (0.1,0.1,0.1), (0.4,0.4,0.4))
```

```
bool ImageBufAlgo.render_text (dst, x, y, text, fontsize=16,
                                fontname="", textcolor=(1,1,1,1))
```

Render antialiased text into `dst`. The `textcolor` (if supplied) is a tuple giving the per-channel colors.

Examples:

```
buf = ImageBuf(ImageSpec (640, 480, 4, oio.FLOAT))
ImageBufAlgo.render_text (buf, 100, 200, "Go Big Red!",
                           60, "Arial Bold", (1,0,0,1))
```

### 11.8.2 Image transformations and data movement

```
bool ImageBufAlgo.channels (dst, src, channelorder, newchannelnames=(),
                             shuffle_channel_names=False)
```

Copy `src` to `dst`, but with channels in the order specified by the tuple `channelorder`. The length of `channelorder` specifies the number of channels to copy. Each element in the tuple `channelorder` may be one of the following:

- `int` : specifies the index (beginning at 0) of the channel to copy.
- `str` : specifies the name of the channel to copy.
- `float` : specifies a constant value to use for that channel.

Does not support in-place operation – that is, `dst` and `src` must be different `ImageBuf`'s.

If `newchannelnames` is supplied, it is a tuple of new channel names. (See the C++ version for more full explanation.)

Examples:

```
# Copy the first 3 channels of an RGBA, drop the alpha
RGBA = ImageBuf("rgba.tif")
RGB = ImageBuf()
ImageBufAlgo.channels (RGB, RGBA, (0,1,2))

# Copy just the alpha channel, making a 1-channel image
Alpha = ImageBuf()
ImageBufAlgo.channels (Alpha, RGBA, ("alpha",))
```



```
# Swap the R and B channels
BGRA = ImageBuf()
ImageBufAlgo.channels (BGRA, RGBA, (2, 1, 0, 3))

# Add an alpha channel with value 1.0 everywhere to an RGB image
ImageBufAlgo.channels (RGBA, RGB, ("R", "G", "B", 1.0))
```

```
bool ImageBufAlgo.channel_append (dst, A, B, roi=ROI.All, nthreads=0)
```

Append the channels of images A and B together into dst over the region of interest.

Examples:

```
RGBA = ImageBuf ("rgba.exr")
Z = ImageBuf ("z.exr")
RGBAZ = ImageBuf()
ImageBufAlgo.channel_append (RGBAZ, RGBA, Z)
```

```
bool ImageBufAlgo.flatten (dst, src, roi=ROI.All, nthreads=0)
```

Composite the depth samples within each pixel of “deep” ImageBuf src and store the single resulting value per pixel in “flat” ImageBufdst.

Examples:

```
Flat = ImageBuf()
ImageBufAlgo.flatten (Flat, ImageBuf("deepalpha.exr"))
```

```
bool ImageBufAlgo.crop (dst, src, roi=ROI.All, nthreads=0)
```

Reset dst to be the specified region of src.

Examples:

```
# Set B to be the upper left 200x100 region of A
A = ImageBuf ("a.tif")
B = ImageBuf()
ImageBufAlgo.crop (B, A, ROI(0,200,0,100))
```

```
bool ImageBufAlgo.paste (dst, xbegin, ybegin, zbegin, chbegin,
                        src, ROI srcroi=ROI.All, nthreads=0)
```

Copy the specified region of src into dst beginning at offset (xbegin, ybegin, zbegin).

Examples:

```
# Paste small.exr on top of big.exr at offset (100,100)
Big = ImageBuf ("big.exr")
Small = ImageBuf ("small.exr")
ImageBufAlgo.paste (Big, 100, 100, 0, 0, Small)
```

```
bool ImageBufAlgo.flip (dst, src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.flop (dst, src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.flipflop (dst, src, roi=ROI.All, nthreads=0)
```

Copy while reversing orientation vertically (flip), horizontally (flop), or both vertically and horizontally (flipflop).

Examples:

```
A = ImageBuf ("tahoe.exr")
B = ImageBuf()
ImageBufAlgo.flip (B, A)
```

```
bool ImageBufAlgo.transpose (dst, src, oi=ROI.All, nthreads=0)
```

Copy while transposing ( $x \leftrightarrow y$ ) pixels.

Examples:

```
A = ImageBuf ("tahoe.exr")
B = ImageBuf()
ImageBufAlgo.transpose (B, A)
```

```
bool ImageBufAlgo.circular_shift (dst, src, xshift, yshift, zshift=0,
    roi=ROI.All, nthreads=0)
```

Copy while circularly shifting by the given amount.

Examples:

```
A = ImageBuf ("tahoe.exr")
B = ImageBuf()
ImageBufAlgo.circular_shift (B, A, 200, 100)
```

```
bool ImageBufAlgo.resize (dst, src, filtername="", filtersize=0.0,
    roi=ROI.All, nthreads=0)
```

Set dst, over the ROI, to be a high-quality resized version of the corresponding portion of src. If the filter and size are not specified, an appropriate default will be chosen.

Examples:

```
# Resize the image to 640x480, using the default filter
Src = ImageBuf ("tahoe.exr")
Dst = ImageBuf (ImageSpec (640, 480, 3, OpenImageIO.FLOAT))
ImageBufAlgo.resize (Dst, Src)
```

```
bool ImageBufAlgo.resample (dst, src, interpolate=True,
                             roi=ROI.All, nthreads=0)
```

Set `dst`, over the ROI, to be a low-quality (but fast) resized version of the corresponding portion of `src`, either using a simple “closest pixel” choice or by bilinearly interpolating (depending on `interpolate`).

Examples:

```
# Resample quickly to 320x240 to make a low-quality thumbnail
Src = ImageBuf ("tahoe.exr")
Dst = ImageBuf (ImageSpec (320, 240, 3, OpenImageIO.UINT8))
ImageBufAlgo.resample (Dst, Src)
```

### 11.8.3 Image arithmetic

```
bool ImageBufAlgo.add (dst, A, B, roi=ROI.All, nthreads=0)
```

Compute `dst = A + B`. `A` is an `ImageBuf`, and `B` may be an `ImageBuf`, a float (added to all channels) or a tuple giving a float for each color channel.

Examples:

```
# Add two images
buf = ImageBuf ()
ImageBufAlgo.add (buf, ImageBuf("a.exr"), ImageBuf("b.exr"))

# Add 0.2 to channels 0-2
ImageBufAlgo.add (buf, buf, (0.2, 0.2, 0.2, 0))
```

```
bool ImageBufAlgo.sub (dst, A, B, roi=ROI.All, nthreads=0)
```

Compute `dst = A - B`. `A` is an `ImageBuf`, and `B` may be an `ImageBuf`, a float (added to all channels) or a tuple giving a float for each color channel.

Examples:

```
buf = ImageBuf ()
ImageBufAlgo.sub (buf, ImageBuf("a.exr"), ImageBuf("b.exr"))
```

```
bool ImageBufAlgo.mul (dst, A, B, roi=ROI.All, nthreads=0)
```

Compute `dst = A * B` (channel-by-channel multiplication). `A` is an `ImageBuf`, and `B` may be an `ImageBuf`, a float (scaling all channels) or a tuple giving a float for each color channel.

Examples:

```
# Multiply the two images
buf = ImageBuf ()
ImageBufAlgo.mul (buf, ImageBuf("a.exr"), ImageBuf("b.exr"))

# Reduce intensity of buf's channels 0-2 by 50%, in place
ImageBufAlgo.mul (buf, buf, (0.5, 0.5, 0.5, 1))
```

**NEW!**

```
bool ImageBufAlgo.pow (dst, A, B, roi=ROI.All, nthreads=0)
```

Compute `dst = pow (A, B)` (channel-by-channel exponentiation). `A` is an `ImageBuf`, and `B` may be a float (a single power for all channels) or a tuple giving a float for each color channel.

Examples:

```
# Linearize a 2.2 gamma-corrected image (channels 0-2 only)
buf = ImageBuf ()
ImageBufAlgo.pow (buf, ImageBuf("a.exr"), (2.2, 2.2, 2.2, 1.0))
```

```
bool ImageBufAlgo.channel_sum (dst, src, weights=(),
                               roi=ROI.All, nthreads=0)
```

Converts a multi-channel image into a 1-channel image via a weighted sum of channels. The `weights` is a tuple providing the weight for each channel (if not supplied, all channels will have weight 1.0).

Examples:

```
# Compute luminance via a weighted sum of R,G,B
# (assuming Rec709 primaries and a linear scale)
luma_weights = (.2126, .7152, .0722)
ImageBufAlgo.channel_sum (luma, ImageBuf("a.exr"), luma_weights)
```

```
bool ImageBufAlgo.clamp (dst, src, min, max, bool clampalpha01=False,
                        roi=ROI.All, nthreads=0)
```

Copy pixels while clamping between the min and max values. The min and max may either be tuples (one min and max value per channel), or single floats (same value for all channels). Additionally, if `clampalpha01` is `True`, then any alpha channel is clamped to the 0–1 range.

Examples:

```
# Clamp image buffer A in-place to the [0,1] range for all channels.
ImageBufAlgo.clamp (A, A, 0.0, 1.0)
```

```
bool ImageBufAlgo.rangecompress (dst, src, useluma=False,
                                 roi=ROI.All, nthreads=0)
```

```
bool ImageBufAlgo.rangeexpand (dst, src, useluma=False,
                               roi=ROI.All, nthreads=0)
```

Copy from `src` to `dst`, compressing (logarithmically) or expanding (by the inverse of the compressive transformation) the range of pixel values. Alpha and z channels are copied but not transformed.

If `useluma` is `True`, the luma of the first three channels (presumed to be R, G, and B) are used to compute a single scale factor for all color channels, rather than scaling all channels

individually (which could result in a big color shift when performing `rangecompress` and `rangeexpand`).

Examples:

```
# Resize the image to 640x480, using a Lanczos3 filter, which
# has negative lobes. To prevent those negative lobes from
# producing ringing or negative pixel values for HDR data,
# do range compression, then resize, then re-expand the range.

# 1. Read the original image
Src = ImageBuf ("tahoeHDR.exr")

# 2. Range compress to a logarithmic scale
Compressed = ImageBuf ()
ImageBufAlgo.rangecompress (Compressed, Src)

# 3. Now do the resize
Dst = ImageBuf ()
roi = ROI (0, 640, 0, 480, 0, 1, 0, Compressed.nchannels)
ImageBufAlgo.resize (Dst, Compressed, "lanczos3", 6.0, roi)

# 4. Expand range to be linear again (operate in-place)
ImageBufAlgo.rangeexpand (Dst, Dst)
```

```
bool ImageBufAlgo.over (dst, A, B, roi=ROI.All, nthreads=0)
```

Composite ImageBuf A *over* ImageBuf B.

Examples:

```
Composite = ImageBuf()
ImageBufAlgo.over (Composite, ImageBuf("fg.exr"), ImageBuf("bg.exr"))
```

```
bool ImageBufAlgo.zover (dst, A, B, bool z_zeroisinf=False,
                        roi=ROI.All, nthreads=0)
```

Composite ImageBuf A and ImageBuf B using their respective Z channels to decide which is in front on a pixel-by-pixel basis.

Examples:

```
Composite = ImageBuf()
ImageBufAlgo.zover (Composite, ImageBuf("fg.exr"), ImageBuf("bg.exr"))
```

#### 11.8.4 Image comparison and statistics

```
bool ImageBufAlgo.compare (A, B, failthresh, warnthresh, result,
                        roi=ROI.All, nthreads=0)
```

Numerically compare two ImageBuf's, A and B. The failthresh and warnthresh supply failure and warning difference thresholds. The result parameter must refer to a CompareResults object, which is defined as a class having the following members:

```
meanerror, rms_error, PSNR, maxerror  # error statistics
maxx, maxy, maxz, maxc                # pixel of biggest difference
nwarn, nfail                          # number of warnings and failures
```

Examples:

```
A = ImageBuf ("a.exr")
B = ImageBuf ("b.exr")
comp = OpenImageIO.CompareResults()
ImageBufAlgo.compare (A, B, 1.0/255.0, 0.0, comp)
if comp.nwarn == 0 and comp.nfail == 0 :
    print "Images match within tolerance"
else :
    print comp.nfail, "failures,", comp.nwarn, " warnings."
    print "Average error was " , comp.meanerror
    print "RMS error was" , comp.rms_error
    print "PSNR was" , comp.PSNR
    print "largest error was ", comp.maxerror
    print "  on pixel", (comp.maxx, comp.maxy, comp.maxz)
    print "  channel", comp.maxc
```

```
tuple ImageBufAlgo.isConstantColor (src, roi=ROI.All, nthreads=0)
```

If all pixels of src within the ROI have the same values (for the subset of channels described by roi), return a tuple giving that color (one float for each channel), otherwise return None.

Examples:

```
A = ImageBuf ("a.exr")
color = ImageBufAlgo.isConstantColor (A)
if color != None :
    print "The image has the same value in all pixels:", color
else :
    print "The image is not a solid color."
```

```
bool ImageBufAlgo.isConstantChannel (src, channel, val,
                                     roi=ROI.All, nthreads=0)
```

Returns True if all pixels of src within the ROI have the given channel value val.

Examples:

```
A = ImageBuf ("a.exr")
alpha = A.spec.alpha_channel
if alpha < 0 :
    print "The image does not have an alpha channel"
elif ImageBufAlgo.isConstantChannel (A, alpha, 1.0) :
```

```

        print "The image has alpha = 1.0 everywhere"
    else :
        print "The image has alpha < 1 in at least one pixel"

```

```
bool ImageBufAlgo.isMonochrome (src, roi=ROI.All, nthreads=0)
```

Returns True if the image is monochrome within the ROI.

Examples:

```

A = ImageBuf ("a.exr")
roi = A.roi
roi.chend = min (3, roi.chend) # only test RGB, not alpha
if ImageBufAlgo.isMonochrome (A, roi) :
    print "a.exr is really grayscale"

```

```
std::string ImageBufAlgo.computePixelHashSHA1 (src, extrainfo = "",
                                                roi=ROI.All, blocksize=0, nthreads=0)
```

Compute the SHA-1 byte hash for all the pixels in the ROI of src.

Examples:

```

A = ImageBuf ("a.exr")
hash = ImageBufAlgo.computePixelHashSHA1 (A, blocksize=64)

```

### 11.8.5 Convolutions

```
bool ImageBufAlgo.make_kernel (dst, name, width, height,
                                depth=1.0, normalize=True)
```

Initialize dst to be a 1-channel float image of the named kernel and dimensions. If normalize is True, the values will be normalized so that they sum to 1.0.

If depth > 1, a volumetric kernel will be created. Use with caution!

Kernel names can be: "gaussian", "sharp-gaussian", "box", "triangle", "mitchell", "blackman-harris", "b-spline", "catmull-rom", "lanczos3", "disk", "binomial".

Note that "catmull-rom" and "lanczos3" are fixed-size kernels that don't scale with the width, and are therefore probably less useful in most cases.

Examples:

```

K = ImageBuf()
ImageBufAlgo.make_kernel (K, "gaussian", 5.0, 5.0)

```

```
bool ImageBufAlgo.convolve (dst, src, kernel, normalize=True,
                             roi=ROI.All, nthreads=0)
```

Replace the given ROI of `dst` with the convolution of `src` and a kernel (also an `ImageBuf`).

Examples:

```
# Blur an image with a 5x5 Gaussian kernel
Src = ImageBuf ("tahoe.exr")
K = ImageBuf ()
ImageBufAlgo.make_kernel (K, "gaussian", 5.0, 5.0)
Blurred = ImageBuf ()
ImageBufAlgo.convolve (Blurred, Src, K)
```

```
bool ImageBufAlgo.fft (dst, src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.ifft (dst, src, roi=ROI.All, nthreads=0)
```

Compute the forward or inverse discrete Fourier Transform.

Examples:

```
Src = ImageBuf ("tahoe.exr")

# Take the DFT of the first channel of Src
Freq = ImageBuf ()
ImageBufAlgo.fft (Freq, Src)

# At this point, Freq is a 2-channel float image (real, imag)
# Convert it back from frequency domain to a spatial image
Spatial = ImageBuf ()
ImageBufAlgo.ifft (Spatial, Freq)
```

```
bool ImageBufAlgo.complex_to_polar (dst, src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.polar_to_complex (dst, src, roi=ROI.All, nthreads=0)
```

**NEW!**

Transform a 2-channel image from complex (real, imaginary) representation to polar (amplitude, phase), or vice versa.

Examples:

```
Polar = ImageBuf ("polar.exr")

Complex = ImageBuf ()
ImageBufAlgo.polar_to_complex (Complex, Polar)

# At this point, Complex is a 2-channel complex image (real, imag)
# Convert it back from frequency domain to a spatial image
Spatial = ImageBuf ()
ImageBufAlgo.ifft (Spatial, Complex)
```



### 11.8.6 Image Enhancement / Restoration

```
bool ImageBufAlgo.fixNonFinite (dst, src, mode=NONFINITE_BOX3,
                                roi=ROI.All, nthreads=0)
```

Copy pixel values from `src` to `dst` (within the pixel and channel range designated by `roi`), and repair any non-finite (NaN or Inf) pixels.

How the non-finite values are repaired is specified by one of the following modes:

```
OpenImageIO.NONFINITE_NONE,
OpenImageIO.NONFINITE_BLACK
OpenImageIO.NONFINITE_BOX3
```

Examples:

```
Src = ImageBuf ("tahoe.exr")
ImageBufAlgo.fixNonFinite (Src, Src, OpenImageIO.NONFINITE_BOX3)
```

```
bool ImageBufAlgo.fillholes_pushpull (dst, src, roi=ROI.All, nthreads=0)
```

Copy the specified ROI of `src` to `dst` and fill any holes (pixels where  $\alpha < 1$ ) with plausible values using a push-pull technique. The `src` image must have an alpha channel. The `dst` image will end up with a copy of `src`, but will have an alpha of 1.0 everywhere, and any place where the alpha of `src` was  $\neq 1$ , `dst` will have a pixel color that is a plausible “filling” of the original alpha hole.

Examples:

```
Src = ImageBuf ("holes.exr")
Filled = ImageBuf ()
ImageBufAlgo.fillholes_pushpull (Filled, Src)
```

```
bool ImageBufAlgo.unsharp_mask (dst, src, kernel="gaussian",
                                width=3.0, contrast=1.0, threshold=0.0,
                                roi=ROI.All, nthreads=0)
```

Replace the given ROI of `dst` with a sharpened version of the corresponding region of `src` using the “unsharp mask” technique.

Examples:

```
Blurry = ImageBuf ("tahoe.exr")
Sharp = ImageBuf ()
ImageBufAlgo.unsharp_mask (Sharp, Blurry, "gaussian", 5.0)
```

### 11.8.7 Color manipulation

```
bool ImageBufAlgo.colorconvert (dst, src, from, to, unpremult=False,
                                roi=ROI.All, nthreads=0)
```

Copy pixels from `src` to `dst` (within the ROI), while applying a color transform to the pixel values. In-place operations (`dst` and `src` being the same image) are supported.

Examples:

```
Src = ImageBuf ("tahoe.jpg")
Dst = ImageBuf ()
ImageBufAlgo.colorconvert (Dst, Src, "vd8", "lfn")
```

```
bool ImageBufAlgo.ociolook (dst, src, looks, from, to,
                             inverse=False, unpremult=False, key="", value="",
                             roi=ROI.All, nthreads=0)
```

Copy pixels from `src` to `dst` (within the ROI), while applying an OpenColorIO “look” transform to the pixel values. In-place operations (`dst` and `src` being the same image) are supported.

Examples:

```
Src = ImageBuf ("tahoe.jpg")
Dst = ImageBuf ()
ImageBufAlgo.ociolook (Dst, Src, "look", "vd8", "lfn", False, False,
                       key="SHOT", value="pe0012")
```

```
bool ImageBufAlgo.ociodisplay (dst, src, display, view,
                                from=None, looks=None, unpremult=False, key="", value="",
                                roi=ROI.All, nthreads=0)
```

Copy pixels from `src` to `dst` (within the ROI), while applying an OpenColorIO “display” transform to the pixel values. In-place operations (`dst` and `src` being the same image) are supported.

Examples:

```
Src = ImageBuf ("tahoe.exr")
Dst = ImageBuf ()
ImageBufAlgo.ociodisplay (Dst, Src, "sRGB", "Film", "lfn", False,
                           key="SHOT", value="pe0012")
```

```
bool ImageBufAlgo.unpremult (dst, src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.premult (dst, src, roi=ROI.All, nthreads=0)
```

Copy pixels from `src` to `dst`, and un-premultiply (or premultiply) the colors by alpha.

Examples:

```
# Convert in-place from associated alpha to unassociated alpha
A = ImageBuf ("a.exr")
ImageBufAlgo.unpremult (A, A)
```

### 11.8.8 Import / export

```
bool ImageBufAlgo.make_texture (mode, input,
                                outputfilename, config=ImageSpec())
```

Turn an input image (either an `ImageBuf` or a string giving a filename) into a tiled, MIP-mapped, texture file and write to the file named by (`outputfilename`). The `mode` describes what type of texture file we are creating and may be one of the following:

```
OpenImageIO.MakeTxTexture
OpenImageIO.MakeTxEnvLat1
OpenImageIO.MakeTxEnvLat1FromLightProbe
```

The `config`, if supplied, is an `ImageSpec` that contains all the information and special instructions for making the texture. The full list of supported configuration options is given in Section 10.9.

Examples:

```
# This command line:
#   maketx in.exr --hicomp --filter lanczos3 --opaque-detect \
#   -o texture.exr
# is equivalent to:

Input = ImageBuf ("in.exr")
config = ImageSpec()
config.attribute ("maketx:highlightcomp", 1)
config.attribute ("maketx:filtername", "lanczos3")
config.attribute ("maketx:opaquedetect", 1)
ImageBufAlgo.make_texture (oiio.MakeTxTexture, Input,
                          "texture.exr", config)
```

```
bool ImageBufAlgo::capture_image (dst, cameranum,
                                    convert = OpenImageIO.UNKNOWN)
```

Capture a still image from a designated camera.

Examples:

```
WebcamImage = ImageBuf()
ImageBufAlgo.capture_image (WebcamImage, 0, OpenImageIO.UINT8)
WebcamImage.save ("webcam.jpg")
```

## 11.9 Miscellaneous Utilities

In the main `OpenImageIO` module, there are a number of values and functions that are useful. These correspond to the C++ API functions explained in Section 2.4.3, please refer there for details.

`int openimageio_version`

The `OpenImageIO` version number, 10000 for each major version, 100 for each minor version, 1 for each patch. For example, `OpenImageIO 1.2.3` would return a value of 10203.

`str geterror ()`

Retrieves the latest global error.

`bool attribute (name, typedesc, value)`

`bool attribute (name, int_value)`

`bool attribute (name, float_value)`

`bool attribute (name, str_value)`

Sets a global attribute (see Section 2.4.3 for details), returning `True` upon success, or `False` if it was not a recognized attribute.

Example:

```
oio.attribute ("threads", 0)
```

## **Part II**

# **Image Utilities**



# 12 oiiotool: the OIO Swiss Army Knife

## 12.1 Overview

The `oiiotool` program will read images (from any file format for which an `ImageInput` plugin can be found), perform various operations on them, and write images (in any format for which an `ImageOutput` plugin can be found).

The `oiiotool` utility is invoked as follows:

```
oiiotool args
```

`oiiotool` maintains an *image stack*, with the top image in the stack also called the *current image*. The stack begins containing no images.

`oiiotool` arguments consist of image names, or commands. When an image name is encountered, that image is pushed on the stack and becomes the new *current image*.

Most other commands either alter the current image (replacing it with the alteration), or in some cases will pull more than one image off the stack (such as the current image and the next item on the stack) and then push a new image.

### Optional arguments

Some commands stand completely on their own (like `--flip`), others take one or more arguments (like `--resize` or `-o`):

```
oiiotool foo.jpg --flip --resize 640x480 -o out.tif
```

A few commands take optional modifiers for options that are so rarely-used or confusing that they should not be required arguments. In these cases, they are appended to the command name, after a colon (":"), and with a *name=value* format. As an example:

```
oiiotool --capture:camera=1 -o out.tif
```

### Frame sequences

It is also possible to have `oiiotool` operate on numbered sequences of images. In effect, this will execute the `oiiotool` command several times, making substitutions to the sequence arguments in turn.

Image sequences are specified by having filename arguments to oiiotool use either a numeric range wildcard (designated such as “1-10#” or a printf-like notation “1-10%d”), or spelling out a more complex pattern with --frames. For example:

```
oiiotool big.1-3#.tif --resize 100x100 -o small.1-3#.tif

oiiotool big.1-3%04d.tif --resize 100x100 -o small.1-3%04d.tif

oiiotool --frames 1-3 big.#.tif --resize 100x100 -o small.#.tif

oiiotool --frames 1-3 big.%04d.tif --resize 100x100 -o small.%04d.tif
```

Any of those will be the equivalent of having issued the following sequence of commands:

```
oiiotool big.0001.tif --resize 100x100 -o small.0001.tif
oiiotool big.0002.tif --resize 100x100 -o small.0002.tif
oiiotool big.0003.tif --resize 100x100 -o small.0003.tif
```

The frame range may be forwards (1-5) or backwards (5-1), and may give a step size to skip frames (1-5x2 means 1, 3, 5) or take the complement of the step size set (1-5y2 means 2, 4) and may combine subsequences with a comma.

If you are using the # or @ wildcards, then the wildcard characters themselves specify how many digits to pad with leading zeroes, with # indicating 4 digits and @ indicating one digit (these may be combined: #@ means 6 digits). An optional --framepadding can also be used to override the number of padding digits. For example,

```
oiiotool --framepadding 3 --frames 3,4,10-20x2 blah.#.tif
```

would match blah.003.tif, blah.004.tif, blah.010.tif, blah.012.tif, blah.014.tif, blah.016.tif, blah.018.tif, blah.020.tif.

Alternately, you can use the printf notation, such as

```
oiiotool --frames 3,4,10-20x2 blah.%03d.tif
```

### Stereo wildcards

oiiotool can also handle image sequences with separate left and right images per frame using views. The %V wildcard will match the full name of all views and %v will match the first character of each view. View names default to “left” and “right”, but may be overridden using the --views option. For example,

```
oiiotool --frames 1-5 blah_%V.#.tif
```

would match blah\_left.0001.tif, blah\_right.0001.tif, blah\_left.0002.tif, blah\_right.0002.tif, blah\_left.0003.tif, blah\_right.0003.tif, blah\_left.0004.tif, blah\_right.0004.tif, blah\_left.0005.tif, blah\_right.0005.tif, and

```
oiiotool --frames 1-5 blah_%v.#.tif
```

would match blah\_l.0001.tif, blah\_r.0001.tif, blah\_l.0002.tif, blah\_r.0002.tif, blah\_l.0003.tif, blah\_r.0003.tif, blah\_l.0004.tif, blah\_r.0004.tif, blah\_l.0005.tif, blah\_r.0005.tif, but

```
oiiotool --views left --frames 1-5 blah_%v.#.tif
```

would only match blah\_l.0001.tif, blah\_l.0002.tif, blah\_l.0003.tif, blah\_l.0004.tif, blah\_l.0005.tif.



## 12.2 oiiotool Tutorial / Recipes

This section will give quick examples of common uses of `oiiotool` to get you started. They should be fairly intuitive, but you can read the subsequent sections of this chapter for all the details on every command.

### Printing information about images

To print the name, format, resolution, and data type of an image (or many images):

```
oiiotool --info *.tif
```

To also print the full metadata about each input image, use both `--info` and `-v`:

```
oiiotool --info -v *.tif
```

To print info about all subimages and/or MIP-map levels of each input image, use the `-a` flag:

```
oiiotool --info -v -a mipmap.exr
```

To print statistics giving the minimum, maximum, average, and standard deviation of each channel of an image, as well as other information about the pixels:

```
oiiotool --stats img_2012.jpg
```

The `--info`, `--stats`, `-v`, and `-a` flags may be used in any combination.

### Converting between file formats

It's a snap to convert among image formats supported by OpenImageIO (i.e., for which `ImageInput` and `ImageOutput` plugins can be found). The `oiiotool` utility will simply infer the file format from the file extension. The following example converts a PNG image to JPEG:

```
oiiotool lena.png -o lena.jpg
```

The first argument (`lena.png`) is a filename, causing `oiiotool` to read the file and makes it the current image. The `-o` command outputs the current image to the filename specified by the next argument.

Thus, the above command should be read to mean, “Read `lena.png` into the current image, then output the current image as `lena.jpg` (using whatever file format is traditionally associated with the `.jpg` extension).”

### Comparing two images

To print a report of the differences between two images of the same resolution:

```
oiitool old.tif new.tif --diff
```

If you also want to save an image showing just the differences:

```
oiitool old.tif new.tif --diff --sub --abs -o diff.tif
```

This looks complicated, but it's really simple: read `old.tif`, read `new.tif` (pushing `old.tif` down on the image stack), report the differences between them, subtract `new.tif` from `old.tif` and replace them both with the difference image, replace that with its absolute value, then save that image to `diff.tif`.

### Changing the data format or bit depth

Just use the `-d` option to specify a pixel data format for all subsequent outputs. For example, assuming that `in.tif` uses 16-bit unsigned integer pixels, the following will convert it to an 8-bit unsigned pixels:

```
oiitool in.tif -d uint8 -o out.tif
```

For formats that support per-channel data formats, you can override the format for one particular channel using `-d CHNAME=TYPE`. For example, assuming `rgbaz.exr` is a float RGBAZ file, and we wish to convert it to be half for RGBA, and float for Z. That can be accomplished with the following command:

```
oiitool rgbaz.tif -d half -d Z=float -o rgbaz2.exr
```

### Changing the compression

The following command converts writes a TIFF file, specifically using LZW compression:

```
oiitool in.tif --compression lzw -o compressed.tif
```

The following command writes its results as a JPEG file at a compression quality of 50 (pretty severe compression):

```
oiitool big.jpg --quality 50 -o small.jpg
```

### Converting between scanline and tiled images

Convert a scanline file to a tiled file with  $16 \times 16$  tiles:

```
oiitool s.tif --tile 16 16 -o t.tif
```

Convert a tiled file to scanline:

```
oiitool t.tif --scanline -o s.tif
```

### Adding captions or metadata

```
oiiotool foo.jpg --caption "Hawaii vacation" -o bar.jpg
oiiotool foo.jpg --keyword "volcano,lava" -o bar.jpg
oiiotool in.exr --attrib "FStop" 22.0 -o out.exr
```

### Scale the values in an image

Reduce the brightness of the R, G, and B channels by 10%, but leave the A channel at its original value:

```
oiiotool original.exr --cmul 0.9,0.9,0.9,1.0 -o out.exr
```

### Remove gamma-correction from an image

Convert a gamma-corrected image (with gamma = 2.2) to linear values.

```
oiiotool corrected.exr --cpow 2.2,2.2,2.2,1.0 -o linear.exr
```

### Resize an image

Resize to a specific resolution:

```
oiiotool original.tif --resize 1024x768 -o specific.tif
```

Resize by a known scale factor:

```
oiiotool original.tif --resize 200% -o big.tif
oiiotool original.tif --resize 25% -o small.tif
```

Resize to a known resolution in one dimension, with the other dimension automatically computed to preserve aspect ratio (just specify 0 as the resolution in the dimension to be automatically computed):

```
oiiotool original.tif --resize 200x0 -o out.tif
oiiotool original.tif --resize 0x1024 -o out.tif
```

Resize to fit into a given resolution, keeping the original aspect ratio and padding with black where necessary to fit into the specified resolution:

```
oiiotool original.tif --fit 640x480 -o fit.tif
```

### Color convert an image

This command linearizes a JPEG assumed to be in sRGB, saving as an HDRI OpenEXR file:

```
oiiotool photo.jpg --colorconvert sRGB linear -o output.exr
```

And the other direction:

```
oiiotool render.exr --colorconvert linear sRGB -o fortheweb.png
```

This converts between two named color spaces (presumably defined by your facility's OpenColorIO configuration):

```
oiiotool in.dpx --colorconvert lg10 lnf -o out.exr
```

**Channel reordering and padding**

Turn a single channel image into gray RGB:

```
oiiotool gray.tif --ch 0,0,0 -o rgb.tif
```

Copy just the color from an RGBA file, truncating the A, yielding RGB only:

```
oiiotool rgba.tif --ch R,G,B -o rgb.tif
```

Zero out the red and green channels:

```
oiiotool rgb.tif --ch =0,=0,B -o justblue.tif
```

Swap the red and blue channels from an RGBA image:

```
oiiotool rgba.tif --ch 2,1,0,3 -o bgra.tif
```

Extract just the named channels from a complicated many-channel image, and add an alpha channel that is 1 everywhere:

```
oiiotool allmyaovs.exr --ch spec.R,spec.G,spec.B,=1 -o spec.exr
```

Add a channel to an RGBA image, setting it to 3.0 everywhere, and naming it “Z” so it will be recognized as a z channel:

```
oiiotool rgba.exr --ch R,G,B,A,Z=3.0 -o rgbaz.exr
```

**Composite a sequence of images**

Composite foreground images over background images for a series of files with frame numbers in their names:

```
oiiotool fg.1-50%04d.exr bg.1-50%04d.exr --over -o comp.1-50%04d.exr
```

Or,

```
oiiotool --frames 1-50 fg.%04d.exr bg.%04d.exr --over -o comp.%04d.exr
```

## 12.3 oiiotool commands: general

`--help`

Prints usage information to the terminal.

`-v`

Verbose status messages — print out more information about what `oiiotool` is doing at every step.

`-q`

Quiet mode — print out less information about what `oiiotool` is doing (only errors).

`--runstats`

Print timing and memory statistics about the work done by `oiiotool`.

`-a`

Performs all operations on all subimages and/or MIPmap levels of each input image. Without `-a`, generally each input image will really only read the top-level MIPmap of the first subimage of the file.

`--info`

Prints information about each input image as it is read. If verbose mode is turned on (`-v`), all the metadata for the image is printed. If verbose mode is not turned on, only the resolution and data format are printed.

`--metamatch regex`

`--no-metamatch regex`

Regular expressions to restrict which metadata are output when using `oiiotool --info -v`. The `--metamatch` expression causes only metadata whose name matches to print; non-matches are not output. The `--no-metamatch` expression causes metadata whose name matches to be suppressed; others (non-matches) are printed. It is not advised to use both of these options at the same time (probably nothing bad will happen, but it's hard to reason about the behavior in that case).

`--stats`

Prints detailed statistical information about each input image as it is read.

`--hash`

Print the SHA-1 hash of the pixels of each input image.

--dumpdata

Print to the console detailed information about the values in every pixel.

Optional appended arguments include:

empty=0|1    If 0, will cause deep images to skip printing of information about pixels with no samples.

--diff

This command computes the difference of the current image and the next image on the stack, and prints a report of those differences (how many pixels differed, the maximum amount, etc.). This command does not alter the image stack.

**NEW!**

--pdiff

This command computes the difference of the current image and the next image on the stack using a perceptual metric, and prints whether or not they match according to that metric. This command does not alter the image stack.

--colorcount *r1,g1,b1,...:r2,g2,b2,...:...*

Given a list of colors separated by colons or semicolons, where each color is a list of comma-separated values (for each channel), examine all pixels of the current image and print a short report of how many pixels matched each of the colors.

Optional appended arguments include:

eps=*r,g,b,...*    Tolerance for matching colors (default: 0.001 for all channels).

Examples:

```
oiiotool test.tif --colorcount "0.792,0,0,1;0.722,0,0,1"
```

might produce the following output:

```
10290  0.792,0,0,1
11281  0.722,0,0,1
```

Notice that use of double quotes (" ") around the list of color arguments, in order to make sure that the command shell does not interpret the semicolon (;) as a statement separator. An alternate way to specify multiple colors is to separate them with a colon (:), for example:

```
oiiotool test.tif --colorcount 0.792,0,0,1:0.722,0,0,1
```

Another example:

```
oiiotool test.tif --colorcount:eps=.01,.01,.01,1000 "0.792,0,0,1"
```

This example sets a larger epsilon for the R, G, and B channels (so that, for example, a pixel with value [0.795,0,0] would also match), and by setting the epsilon to 1000 for the alpha channel, it effectively ensures that alpha will not be considered in the matching of pixels to the color value.

`--rangecheck Rlow, Glow, Blow, ... Rhi, Bhi, Ghi, ...`

Given a two colors (each a comma-separated list of values for each channel), print a count of the number of pixels in the image that has channel values outside the [low,hi] range. Any channels not specified will assume a low of 0.0 and high of 1.0.

Example:

```
oiitool test.exr --rangecheck 0,0,0 1,1,1
```

might produce the following output:

```
0    < 0,0,0
221  > 1,1,1
65315 within range
```

`--no-clobber`

Sets “no clobber” mode, in which existing images on disk will never be overridden, even if the `-o` command specifies that file.

`--threads n`

Use *n* execution threads if it helps to speed up image operations. The default (also if *n* = 0) is to use as many threads as there are cores present in the hardware.

`--frames seq`

`--framepadding n`

Describes the frame range to substitute for the # or %0Nd numeric wildcards. The sequence is a comma-separated list of subsequences; each subsequence is a single frame (e.g., 100), a range of frames (100-150), or a frame range with step (100-150x4 means 100,104,108, ...).

The frame padding is the number of digits (with leading zeroes applied) that the frame numbers should have. It defaults to 4.

For example,

```
oiitool --framepadding 3 --frames 3,4,10-20x2 blah.#.tif
```

would match `blah.003.tif, blah.004.tif, blah.010.tif, blah.012.tif, blah.014.tif, blah.016.tif, blah.018.tif, blah.020.tif`.

## 12.4 oiitool commands: reading and writing images

The commands described in this section read images, write images, or control the way that subsequent images will be written upon output.

*filename*

If a command-line option is the name of an image file, that file will be read and will become the new *current image*, with the previous current image pushed onto the image stack.

-o *filename*

Outputs the current image to the named file. This does not remove the current image, it merely saves a copy of it.

-d *datatype*

-d *channelname=datatype*

Attempts to set the pixel data type of all subsequent outputs. If no channel is named, sets *all* channels to be the specified data type. If a specific channel is named, then the data type will be overridden for just that channel (multiple -d commands may be used).

Valid types are: uint8, sint8, uint16, sint16, half, float, double. The types uint10 and uint12 may be used to request 10- or 12-bit unsigned integers. If the output file format does not support them, uint16 will be substituted.

If the -d option is not supplied, the output data type will be the same as the data format of the input files, if possible.

In any case, if the output file type does not support the requested data type, it will instead use whichever supported data type results in the least amount of precision lost.

--scanline

Requests that subsequent output files be scanline-oriented, if scanline orientation is supported by the output file format. By default, the output file will be scanline if the input is scanline, or tiled if the input is tiled.

--tile *x y*

Requests that subsequent output files be tiled, with the given  $x \times y$  tile size, if tiled images are supported by the output format. By default, the output file will take on the tiledness and tile size of the input file.

--compression *method*

Sets the compression method for subsequent output images. Each ImageOutput plugin will have its own set of methods that it supports. By default, the output image will use the same compression technique as the input image (assuming it is supported by the output format, otherwise it will use the default compression method of the output plugin).

--quality *q*

Sets the compression quality, on a 1–100 floating-point scale. This only has an effect if the particular compression method supports a quality metric (as JPEG does).

--dither

**NEW!**

Turns on *dither* when outputting to 8-bit image files (does not affect other data types). This adds just a bit of noise that reduces visible banding artifacts.



`--planarconfig config`

Sets the planar configuration of subsequent outputs (if supported by their formats). Valid choices are: `config` for contiguous (or interleaved) packing of channels in the file (e.g., `RGBRGBRGB...`), `separate` for separate channel planes (e.g., `RRRR...GGGG...BBBB...`), or `default` for the default choice for the given format. This command will be ignored for output files whose file format does not support the given choice.

`--adjust-time`

When this flag is present, after writing each output, the resulting file's modification time will be adjusted to match any "DateTime" metadata in the image. After doing this, a directory listing will show file times that match when the original image was created or captured, rather than simply when `oiio-tool` was run. This has no effect on image files that don't contain any "DateTime" metadata.

`--noautocrop`

For subsequent outputs, do *not* automatically crop images whose formats don't support separate pixel data and full/display windows. Without this, the default is that outputs will be cropped or padded with black as necessary when written to formats that don't support the concepts of pixel data windows and full/display windows. This is a non-issue for file formats that support these concepts, such as OpenEXR.

`--autotrim`

For subsequent outputs, if the output format supports separate pixel data and full/display windows, automatically trim the output so that it writes the minimal data window that contains all the non-zero valued pixels. In other words, trim off any all-black border rows and columns before writing the file.

## **12.5 oiio-tool commands that change the current image metadata**

This section describes `oiio-tool` commands that alter the metadata of the current image, but do not alter its pixel values. Only the current (i.e., top of stack) image is affected, not any images further down the stack.

If the `-a` flag has previously been set, these commands apply to all subimages or MIPmap levels of the current top image. Otherwise, they only apply to the highest-resolution MIPmap level of the first subimage of the current top image.

`--attrib name value`

Adds or replaces metadata with the given *name* to have the specified *value*.

It will try to infer the type of the metadata from the value: if the value contains only numerals (with optional leading minus sign), it will be saved as `int` metadata; if it also

contains a decimal point, it will be saved as `float` metadata; otherwise, it will be saved as a `string` metadata.

For example, you could explicitly set the IPTC location metadata fields with:

```
oiio tool --attrib "IPTC:City" "Berkeley" in.jpg out.jpg
```

`--sattrib name value`

Adds or replaces metadata with the given *name* to have the specified *value*, forcing it to be interpreted as a `string`. This is helpful if you want to set a `string` metadata to a value that the `--attrib` command would normally interpret as a number.

`--caption text`

Sets the image metadata "ImageDescription". This has no effect if the output image format does not support some kind of title, caption, or description metadata field. Be careful to enclose *text* in quotes if you want your caption to include spaces or certain punctuation!

`--keyword text`

Adds a keyword to the image metadata "Keywords". Any existing keywords will be preserved, not replaced, and the new keyword will not be added if it is an exact duplicate of existing keywords. This has no effect if the output image format does not support some kind of keyword field.

Be careful to enclose *text* in quotes if you want your keyword to include spaces or certain punctuation. For image formats that have only a single field for keywords, OpenImageIO will concatenate the keywords, separated by semicolon (;), so don't use semicolons within your keywords.

`--clear-keywords`

Clears all existing keywords in the current image.

`--orientation orient`

Explicitly sets the image's "Orientation" metadata to a numeric value (see Section B.2 for the numeric codes). This only changes the metadata field that specifies how the image should be displayed, it does NOT alter the pixels themselves, and so has no effect for image formats that don't support some kind of orientation metadata.

`--rotcw`

`--rotccw`

`--rot180`

Adjusts the image's "Orientation" metadata by rotating it 90° clockwise, 90° degrees counter-clockwise, or 180°, respectively, compared to its current setting. This only changes the metadata field that specifies how the image should be displayed, it does NOT alter the pixels themselves, and so has no effect for image formats that don't support some kind of orientation metadata.

`--origin offset`

Set the pixel data window origin, essentially translating the existing pixel data window to a different position on the image plane. The offset is in the form

`[+-]x[+-]y`

Examples:

```
--origin +20+10      x=20, y=10
--origin +0-40       x=0, y=-40
```

`--fullsize size`

Set the display/full window size and/or offset. The size is in the form

`width x height [+-]xoffset [+-]yoffset`

If either the offset or resolution is omitted, it will remain unchanged.

Examples:

```
--fullsize 1920x1080      resolution w=1920, h=1080, offset unchanged
--fullsize -20-30         resolution unchanged, x=-20, y=-30
--fullsize 1024x768+100+0 resolution w=1024, h=768, offset x=100, y=0
```

`--fullpixels`

Set the full/display window range to exactly cover the pixel data window.

`--chnames name-list`

Rename some or all of the channels of the top image to the given comma-separated list. Any completely empty channel names in the list will not be changed. For example,

```
oiiotool in.exr --chnames ",,,A,Z" -o out.exr
```

will rename channel 3 to be "A" and channel 4 to be "Z", but will leave channels 0–3 with their old names.

## 12.6 oiiotool commands that shuffle channels or subimages

`--selectmip level`

If the current image is MIP-mapped, replace the current image with a new image consisting of only the given *level* of the MIPmap. Level 0 is the highest resolution version, level 1 is the next-lower resolution version, etc.

`--unmip`

If the current image is MIP-mapped, discard all but the top level (i.e., replacing the current image with a new image consisting of only the highest-resolution level). Note that this is equivalent to `--selectmip 0`.

`--subimage n`

If the current image has multiple subimages, replace the current image with a new image consisting of only the given subimage.

`--siappend n`

Replaces the two two images on the stack with a new image comprised of the subimages of both images appended together.

`--ch channellist`

Replaces the top image with a new copy whose channels have been reordered as given by the *channellist*. The *channellist* is a comma-separated list of channel names and/or numbers (e.g., "R,G,B", "A", "B,G,R", "4,5,6,A"). Channel numbers outside the valid range of input channels, or unknown names, will be replaced by black channels.

A channel designation beginning with the = character and followed by a number is a *literal value* that will be used to fill that channel. If the *channellist* is shorter than the number of channels in the source image, unspecified channels will be omitted.

`--chappend`

Replaces the top two images on the stack with a new image comprised of the channels of both images appended together.

`--flatten`

If the top image is “deep,” then “flatten” it by compositing the depth samples in each pixel.

## 12.7 oiio tool commands that adjust the image stack

`--pop`

Pop the image stack, discarding the current image and thereby making the next image on the stack into the new current image.

`--dup`

Duplicate the current image and push the duplicate on the stack. Note that this results in both the current and the next image on the stack being identical copies.

`--swap`

Swap the current image and the next one on the stack.

`--label name`

Gives a name to (and saves) the current image at the top of the stack. Thereafter, the label name may be used to refer to that saved image, in the usual manner that an ordinary input image would be specified by filename.

**NEW!**

## 12.8 oiiotool commands that make entirely new images

`--create size channels`

Create new black image with the given size and number of channels, pushing it onto the image stack and making it the new current image.

The *size* is in the form

*width* x *height* [+*xoffset*] [+*yoffset*]

If the offset is omitted, it will be  $x = 0, y = 0$ .

Examples:

```
--create 1920x1080 3      RGB with w=1920, h=1080, x=0, y=0
--create 1024x768+100+0 4  RGBA with w=1024, h=768, x=100, y=0
```

`--pattern patternname size channels`

Create new image with the given size and number of channels, initialize its pixels to the named pattern, and push it onto the image stack to make it the new current image.

The *size* is in the form

*width* x *height* [+*xoffset*] [+*yoffset*]

If the offset is omitted, it will be  $x = 0, y = 0$ .

The patterns recognized include the following:

black	A black image (all pixels 0.0)
constant	A constant color image, defaulting to white, but the color can be set with the optional <code>:color=r,g,b,...</code> arguments giving a numerical value for each channel.
checker	A black and white checkerboard pattern. The optional argument <code>:width=</code> sets width of the checkers (defaulting to 8 pixels).

Examples:

```
--pattern constant:color=0.3,0.5,0.1,1.0 640x480 4
      A constant 4-channel, 640 × 480 image with all pixels (0.5, 0.5, 0.1, 1).
```

```
--pattern checker:width=16 512x512 3
```

An  $512 \times 512$  RGB image with a 16-pixel-wide checker pattern.

```
--kernel name size
```

Create new 1-channel float image big enough to hold the named kernel and size (size is expressed as *widthxheight*, e.g.  $5 \times 5$ ). The *width* and *height* are allowed to be floating-point numbers. The kernel image will have its origin offset so that the kernel center is at (0,0), and will be normalized (the sum of all pixel values will be 1.0).

Kernel names can be: gaussian, sharp-gaussian, box, triangle, blackman-harris, mitchell, b-spline, disk. There are also catmull-rom and lanczos3, but they are fixed-size kernels that don't scale with the width, and are therefore probably less useful in most cases.

Examples:

```
oiio tool --kernel gaussian 11x11 -o gaussian.exr
```

```
--capture
```

Capture a frame from a camera device, pushing it onto the image stack and making it the new current image. Optional appended arguments include:

`camera=num`      Select which camera number to capture (default: 0).

Examples:

```
--capture                      Capture from the default camera.
--capture:camera=1            Capture from camera 2.
```

## 12.9 oiio tool commands that do image processing

```
--add
```

Replace the *two* top images with a new image that is the sum of those images.

```
--sub
```

Replace the *two* top images with a new image that is the difference between the first and second images.

```
--mul
```

Replace the *two* top images with a new image that the pixel-by-pixel, channel-by-channel multiplicative product of the first and second images.

--abs

Replace the current image with a new image that has each pixel consisting of the *absolute value* of the old pixel value.

--cadd *value*

--cadd *value0,value1,value2...*

Add a constant value to all the pixels in the current image. If a single constant value is given, it will be added to all color channels. Alternatively, a series of comma-separated constant values (with no spaces!) may be used to specify a different value to add to each channel in the image, respectively.

--cmul *value*

--cmul *value0,value1,value2...*

Multiply all the pixel values in the top image by a constant value. If a single constant value is given, all color channels will have their values multiplied by the same value. Alternatively, a series of comma-separated constant values (with no spaces!) may be used to specify a different multiplier for each channel in the image, respectively.

--cpow *value*

--cpow *value0,value1,value2...*

Raise all the pixel values in the top image to a constant power value. If a single constant value is given, all color channels will have their values raised to this power. Alternatively, a series of comma-separated constant values (with no spaces!) may be used to specify a different exponent for each channel in the image, respectively.

**NEW!**

--chsum

Replaces the top image by a copy that contains only 1 color channel, whose value at each pixel is the sum of all channels of the original image. Using the optional *weight* allows you to customize the weight of each channel in the sum.

*weight=r,g,...*      Specify the weight of each channel (default: 1).

Example:

```
oiiotool RGB.tif --chsum:weight=.2126,.7152,.0722 -o luma.tif
```

--paste *location*

Takes two images – the first is the “foreground” and the second is the “background” – and uses the pixels of the foreground to replace those of the background beginning at the upper left *location* (expressed as *+xpos+ypos*, e.g., +100+50, or of course using – for negative offsets).

`--mosaic size`

Removes *wxh* images, dictated by the *size*, and turns them into a single image mosaic. Optional appended arguments include:

`pad=num`      Select the number of pixels of black padding to add between images (default: 0).

Examples:

```
oiio tool left.tif right.tif --mosaic:pad=16 2x1 -o out.tif
```

```
oiio tool 0.tif 1.tif 2.tif 3.tif 4.tif --mosaic:pad=16 2x2 -o out.tif
```

`--over`

Replace the *two* top images with a new image that is the Porter/Duff “over” composite with the first image as the foreground and the second image as the background. Both input images must have the same number and order of channels and must contain an alpha channel.

`--zover`

Replace the *two* top images with a new image that is a *depth composite* of the two images – the operation is the Porter/Duff “over” composite, but each pixel individually will choose which of the two images is the foreground and which background, depending on the “Z” channel values for that pixel (larger Z means farther away). Both input images must have the same number and order of channels and must contain both depth/Z and alpha channels. Optional appended arguments include:

`zeroisinf=num`    If nonzero, indicates that  $z = 0$  pixels should be treated as if they were infinitely far away. (The default is 0, meaning that “zero means zero.”).

`--flip`

Replace the current image with a new image that is flipped vertically, with the top scanline becoming the bottom, and vice versa.

`--flop`

Replace the current image with a new image that is flopped horizontally, with the leftmost column becoming the rightmost, and vice versa.

`--flipflop`

Replace the current image with a new image that is both flipped and flopped, which is the same as a 180 degree rotation.



--transpose

Replace the current image with a new image that is trasposed about the *xy* axis (x and coordinates and size are flipped).

--cshift *offset*

Circularly shift the pixels of the image by the given offset (expressed as +10+100 to move by 10 pixels horizontally and 100 pixels vertically, or +50-30 to move by 50 pixels horizontally and -30 pixels vertically. *Circular* shifting means that the pixels wrap to the other side as they shift.

--crop *size*

Replace the current image with a new copy with the given *size*, cropping old pixels no longer needed, padding black pixels where they previously did not exist in the old image, and adjusting the offsets if requested.

The size is in the form

*width x height [+ -] xoffset [+ -] yoffset*

or *xmin,ymin,xmax,ymax*

Examples:

--crop 100x120+35+40                      resolution w=100, h=120, offset x=35, y=40

--crop 35,40,134,159                      resolution w=100, h=120, offset x=35, y=40

Note that *crop* does not *reposition* pixels, it only trims or pads to reset the image's pixel data window to the specified region.

--croptofull

Replace the current image with a new image that is cropped or padded as necessary to make the pixel data window exactly cover the full/display window.

--cut *size*

Replace the current image with a new copy with the given *size*, cropping old pixels no longer needed, padding black pixels where they previously did not exist in the old image, repositioning the cut region at the image origin (0,0) and resetting the full/display window to be identical to the new pixel data window. (In other words, --cut is equivalent to --crop followed by --origin +0+0 --fullpixels.)

The size is in the form

*width x height [+ -] xoffset [+ -] yoffset*

or *xmin,ymin,xmax,ymax*

Examples:

--cut 100x120+35+40                      resolution w=100, h=120, offset x=35, y=40

--cut 35,40,134,159                      resolution w=100, h=120, offset x=35, y=40

**NEW!**

`--resample size`

Replace the current image with a new image that is resampled to the given pixel data resolution rapidly, but at a low quality, by simply copying the “closest” pixel. The *size* is in the form

*width* x *height*  
or *scale*%

if *width* or *height* is 0, that dimension will be automatically computed so as to preserve the original aspect ratio.

Examples (suppose that the original image is 640x480):

```
--resample 1024x768      new resolution w=1024, h=768
--resample 50%           reduce resolution to 320x240
--resample 300%          increase resolution to 1920x1440
--resample 400x0         new resolution will be 400x300
```

`--resize size`

Replace the current image with a new image that is resized to the given pixel data resolution. The *size* is in the form

*width* x *height*  
or *scale*%

if *width* or *height* is 0, that dimension will be automatically computed so as to preserve the original aspect ratio.

Optional appended arguments include:

`filter=name`      Filter name. The default is `blackman-harris` when increasing resolution, `lanczos3` when decreasing resolution.

Examples (suppose that the original image is 640x480):

```
--resize 1024x768      new resolution w=1024, h=768
--resize 50%           reduce resolution to 320x240
--resize 300%          increase resolution to 1920x1440
--resize 400x0         new resolution will be 400x300
```

`--fit size`

Replace the current image with a new image that is resized to fit into the given pixel data resolution, keeping the original aspect ratio and padding with black pixels if the requested image size does not have the same aspect ratio. The *size* is in the form

*width* x *height*  
or *width* x *height* [+]*xorigin* [+]*yorigin*

Optional appended arguments include:

<code>filter=<i>name</i></code>	Filter name. The default is <code>blackman-harris</code> when increasing resolution, <code>lanczos3</code> when decreasing resolution.
<code>pad=<i>p</i></code>	If the argument is nonzero, will pad with black pixels to make the resulting image exactly the size specified, if the source and desired size are not the same aspect ratio.

**Examples:**

```
oiiotool in.exr --fit:pad=1 640x480 -o out.exr

oiiotool in.exr --fit 1024x1024 -o out.exr
```

**--convolve**

Use the top image as a kernel to convolve the next image farther down the stack, replacing both with the result.

**Examples:**

```
# Use a kernel image already prepared
oiiotool image.exr kernel.exr --convolve -o output.exr

# Construct a kernel image on the fly with --kernel
oiiotool image.exr --kernel gaussian 5x5 --convolve -o blurred.exr
```

**--blur *size***

Blur the top image with a blur kernel of the given size expressed as *widthxheight*. (The sizes may be floating point numbers.)

Optional appended arguments include:

<code>kernel=<i>name</i></code>	Kernel name. The default is <code>gaussian</code> .
---------------------------------	---

**Examples:**

```
oiiotool image.jpg --blur 5x5 -o blurred.jpg

oiiotool image.jpg --blur:kernel=bspline 7x7 -o blurred.jpg
```

**--unsharp**

Unblur the top image using an “unsharp mask.”

Optional appended arguments include:

<code>kernel=<i>name</i></code>	Name of the blur kernel (default: <code>gaussian</code> ).
<code>width=<i>w</i></code>	Width of the blur kernel (default: 3).
<code>contrast=<i>c</i></code>	Contrast scale (default: 1.0)
<code>threshold=<i>t</i></code>	Threshold for applying the difference (default: 0)

**Examples:**

```
oiiotool image.jpg --unsharp -o sharper.jpg

oiiotool image.jpg --unsharp:width=5:contrast=1.5 -o sharper.jpg
```

```
--fft
--ifft
```

Performs forward and inverse unitized discrete Fourier transform. The forward FFT always transforms only the first channel of the top image on the stack, and results in a 2-channel image (with real and imaginary channels). The inverse FFT transforms the first two channels of the top image on the stack (assuming they are real and imaginary, respectively) and results in a single channel result (with the real component only of the spatial domain result).

Examples:

```
# Select the blue channel and take its DCT
oiio tool image.jpg --ch 2 --fft -o fft.exr

# Reconstruct from the FFT
oiio tool fft.exr --ifft -o reconstructed.exr

# Output the power spectrum: real^2 + imag^2
oiio tool fft.exr --dup --mul --chsum -o powerspectrum.exr
```

```
--polar
--unpolar
```

**NEW!**

The `--polar` transforms a 2-channel image whose channels are interpreted as complex values (real and imaginary components) into the equivalent values expressed in polar form of amplitude and phase (with phase between 0 and  $2\pi$ ).

The `--unpolar` performs the reverse transformation, converting from polar values (amplitude and phase) to complex (real and imaginary).

Examples:

```
oiio tool complex.exr --polar -o polar.exr
oiio tool polar.exr --unpolar -o complex.exr
```

```
--fixnan strategy
```

Replace the top image with a copy in which any pixels that contained NaN or Inf values (hereafter referred to collectively as “nonfinite”) are repaired. If *strategy* is `black`, nonfinite values will be replaced with 0. If *strategy* is `box3`, nonfinite values will be replaced by the average of all the finite values within a  $3 \times 3$  region surrounding the pixel.

```
--clamp
```

Replace the top image with a copy in which pixel values have been clamped. Optional arguments include:

Optional appended arguments include:

<code>min=val</code>	Specify a minimum value for all channels.
<code>min=val0,val1,...</code>	Specify minimum value for each channel individually.
<code>max=val</code>	Specify a maximum value for all channels.
<code>max=val0,val1,...</code>	Specify maximum value for each channel individually.
<code>clampalpha=val</code>	If <i>val</i> is nonzero, will additionally clamp the alpha channel to [0,1]. (Default: 0, no additional alpha clamp.)

If no value is given for either the minimum or maximum, it will NOT clamp in that direction. For the variety of minimum and maximum that specify per-channel values, a missing value indicates that the corresponding channel should not be clamped.

Examples:

<code>--clamp:min=0</code>	Clamp all channels to a minimum of 0 (all negative values are changed to 0).
<code>--clamp:min=0:max=1</code>	Clamp all channels to [0,1].
<code>--clamp:clampalpha=1</code>	Clamp the designated alpha channel to [0,1].
<code>--clamp:min=,,0:max=,,3.0</code>	Clamp the third channel to [0,3], do not clamp other channels.

`--rangecompress`

`--rangeexpand`

Range compression re-maps input values to a logarithmic scale. Range expansion is the inverse mapping back to a linear scale. Range compression and expansion only applies to color channels; alpha or z channels will not be modified.

By default, this transformation will happen to each color channel independently. But if the optional `luma` argument is nonzero and the image has at least 3 channels and the first three channels are not alpha or depth, they will be assumed to be RGB and the pixel scaling will be done using the luminance and applied equally to all color channels. This can help to preserve color even when remapping intensity.

Optional appended arguments include:

<code>luma=val</code>	If <i>val</i> is 0, turns off the luma behavior.
-----------------------	--

Range compression and expansion can be useful in cases where high contrast super-white ( $> 1$ ) pixels (such as very bright highlights in HDR captured or rendered images) can produce undesirable artifacts, such as if you resize an HDR image using a filter with negative lobes – which could result in objectionable ringing or even negative result pixel values. For example,

```
oiiotool hdr.exr --rangecompress --resize 512x512 --rangeexpand -o resized.exr
```

`--fillholes`

Replace the top image with a copy in which any pixels that had  $\alpha < 1$  are “filled” in a smooth way using data from surrounding  $\alpha > 0$  pixels, resulting in an image that is  $\alpha = 1$  and plausible color everywhere. This can be used both to fill internal “holes” as well as to extend an image out.

`--text words`

Draw (rasterize) text overtop of the current image.

<code>x=xpos</code>	<code>x</code> position (in pixel coordinates) of the text
<code>y=ypos</code>	<code>y</code> position (in pixel coordinates) of the text
<code>size=size</code>	font size (height, in pixels)
<code>font=name</code>	font name, full path to the font file on disk (use double quotes "name" if the path name includes spaces)
<code>color=r,g,b,...</code>	specify the color of the text

The default positions the text starting at the center of the image, drawn 16 pixels high in opaque white in all channels (1,1,1,...), and using a default font (which may be system dependent).

Examples:

```
oiiotool in.exr --text:x=10:y=400:size=40 "Hello world" -o out.exr
```

## 12.10 oiiotool commands for color management

`--iscolospace colorspace`

Alter the metadata of the current image so that it thinks its pixels are in the named color space. This does not alter the pixels of the image, it only changes oiiotool's understanding of what color space those those pixels are in.

`--colorconvert fromspace tospace`

Replace the current image with a new image whose pixels are transformed from the named *fromspace* color space into the named *tospace* (disregarding any notion it may have previously had about the color space of the current image).

If OIIO has been compiled with OpenColorIO support and the environment variable \$OCIO is set to point to a valid OpenColorIO configuration file, you will have access to all the color spaces that are known by that OCIO configuration. If \$OCIO does not point to a valid configuration file or OIIO was not compiled with OCIO support, then the only color space transformats available are linear to Rec709 (and vice versa) and linear to sRGB (and vice versa).

If you ask for oiiotool help (oiiotool --help), at the very bottom you will see the list of all color spaces that oiiotool knows about.

`--tocolospace tospace`

Replace the current image with a new image whose pixels are transformed from their existing color space (as best understood or guessed by OIIO) into the named *tospace*. This is equivalent to a use of oiiotool --colorconvert where the *fromspace* is automatically deduced.

`--ociolook lookname`

Replace the current image with a new image whose pixels are transformed using the named OpenColorIO look description. Optional appended arguments include:

<code>from=<i>val</i></code>	Assume the image is in the named color space. If no <code>from=</code> is supplied, it will try to deduce it from the image's metadata or previous <code>--iscolorspace</code> directives.
<code>to=<i>val</i></code>	Convert to the named space after applying the look.
<code>inverse=<i>val</i></code>	If <i>val</i> is nonzero, inverts the color transformation and look application.
<code>key=<i>name</i></code>	
<code>value=<i>str</i></code>	Adds a key/value pair to the “context” that OpenColorIO will used when applying the look.

This command is only meaningful if OIIO was compiled with OCIO support and the environment variable `$OCIO` is set to point to a valid OpenColorIO configuration file. If you ask for `oiitool help` (`oiitool --help`), at the very bottom you will see the list of all looks that `oiitool` knows about.

Examples:

```
oiitool in.jpg --ociolook:from=vd8:to=vd8:key=SHOT:value=pe0012 match -o cc.jpg
```

`--ociodisplay displayname viewname`

Replace the current image with a new image whose pixels are transformed using the named OpenColorIO “display” transformation given by the *displayname* and *viewname*. Optional appended arguments include:

<code>from=<i>val</i></code>	Assume the image is in the named color space. If no <code>from=</code> is supplied, it will try to deduce it from the image's metadata or previous <code>--iscolorspace</code> directives.
<code>key=<i>name</i></code>	
<code>value=<i>str</i></code>	Adds a key/value pair to the “context” that OpenColorIO will used when applying the look.

This command is only meaningful if OIIO was compiled with OCIO support and the environment variable `$OCIO` is set to point to a valid OpenColorIO configuration file. If you ask for `oiitool help` (`oiitool --help`), at the very bottom you will see the list of all looks that `oiitool` knows about.

Examples:

```
oiitool in.exr --ociodisplay:from=lnf:key=SHOT:value=pe0012 sRGB Film -o cc.jpg
```

`--unpremult`

Divide all color channels (those not alpha or z) of the current image by the alpha value, to “un-premultiply” them. This presumes that the image starts of as “associated alpha,” a.k.a. “premultiplied.” Pixels in which the alpha channel is 0 will not be modified (since the operation is undefined in that case). This is a no-op if there is no identified alpha channel.

--premult

Multiply all color channels (those not alpha or z) of the current image by the alpha value, to “premultiply” them. This presumes that the image starts of as “unassociated alpha,” a.k.a. “non-premultiplied.”



## 13 The `iv` Image Viewer

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

More documentation on this later.



## 14 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 `iinfo` is built on top on OpenImageIO, it will print information about images of any formats readable by ImageInput plugins on hand.

### 14.1 Using `iinfo`

The `iinfo` utility is invoked as follows:

```
iinfo [options] filename ...
```

Where *filename* (and any following strings) names the image file(s) whose information should be printed. The image files may be of any format recognized by OpenImageIO (i.e., for which ImageInput plugins are available).

In its most basic usage, it simply prints the resolution, number of channels, pixel data type, and file format type of each of the files listed:

```
$ iinfo img_6019m.jpg grid.tif lenna.png

img_6019m.jpg : 1024 x 683, 3 channel, uint8 jpeg
grid.tif      : 512 x 512, 3 channel, uint8 tiff
lenna.png     : 120 x 120, 4 channel, uint8 png
```

The `-s` flag also prints the uncompressed sizes of each image file, plus a sum for all of the images:

```
$ iinfo -s img_6019m.jpg grid.tif lenna.png

img_6019m.jpg : 1024 x 683, 3 channel, uint8 jpeg (2.00 MB)
grid.tif      : 512 x 512, 3 channel, uint8 tiff (0.75 MB)
lenna.png     : 120 x 120, 4 channel, uint8 png (0.05 MB)
Total size: 2.81 MB
```

The `-v` option turns on *verbose mode*, which exhaustively prints all metadata about each image:

```
$ iinfo -v img_6019m.jpg

img_6019m.jpg : 1024 x 683, 3 channel, uint8 jpeg
  channel list: R, G, B
  Color space: sRGB
  ImageDescription: "Family photo"
  Make: "Canon"
  Model: "Canon EOS DIGITAL REBEL XT"
  Orientation: 1 (normal)
  XResolution: 72
  YResolution: 72
  ResolutionUnit: 2 (inches)
  DateTime: "2008:05:04 19:51:19"
  Exif:YCbCrPositioning: 2
  ExposureTime: 0.004
  FNumber: 11
  Exif:ExposureProgram: 2 (normal program)
  Exif:ISOSpeedRatings: 400
  Exif:DateTimeOriginal: "2008:05:04 19:51:19"
  Exif:DateTimeDigitized: "2008:05:04 19:51:19"
  Exif:ShutterSpeedValue: 7.96579 (1/250 s)
  Exif:ApertureValue: 6.91887 (f/11)
  Exif:ExposureBiasValue: 0
  Exif:MeteringMode: 5 (pattern)
  Exif:Flash: 16 (no flash, flash supression)
  Exif:FocalLength: 27 (27 mm)
  Exif:ColorSpace: 1
  Exif:PixelXDimension: 2496
  Exif:PixelYDimension: 1664
  Exif:FocalPlaneXResolution: 2855.84
  Exif:FocalPlaneYResolution: 2859.11
  Exif:FocalPlaneResolutionUnit: 2 (inches)
  Exif:CustomRendered: 0 (no)
  Exif:ExposureMode: 0 (auto)
  Exif:WhiteBalance: 0 (auto)
  Exif:SceneCaptureType: 0 (standard)
  Keywords: "Carly; Jack"
```

If the input file has multiple subimages, extra information summarizing the subimages will be printed:

```
$ iinfo img_6019m.tx

img_6019m.tx : 1024 x 1024, 3 channel, uint8 tiff (11 subimages)

$ iinfo -v img_6019m.tx

img_6019m.tx : 1024 x 1024, 3 channel, uint8 tiff
  11 subimages: 1024x1024 512x512 256x256 128x128 64x64 32x32 16x16 8x8 4x4 2x2 1x1
  channel list: R, G, B
  tile size: 64 x 64
```

...

Furthermore, the `-a` option will print information about all individual subimages:

```
$ iinfo -a ../sample-images/img_6019m.tx

img_6019m.tx : 1024 x 1024, 3 channel, uint8 tiff (11 subimages)
subimage 0: 1024 x 1024, 3 channel, uint8 tiff
subimage 1: 512 x 512, 3 channel, uint8 tiff
subimage 2: 256 x 256, 3 channel, uint8 tiff
subimage 3: 128 x 128, 3 channel, uint8 tiff
subimage 4: 64 x 64, 3 channel, uint8 tiff
subimage 5: 32 x 32, 3 channel, uint8 tiff
subimage 6: 16 x 16, 3 channel, uint8 tiff
subimage 7: 8 x 8, 3 channel, uint8 tiff
subimage 8: 4 x 4, 3 channel, uint8 tiff
subimage 9: 2 x 2, 3 channel, uint8 tiff
subimage 10: 1 x 1, 3 channel, uint8 tiff

$ iinfo -v -a img_6019m.tx
img_6019m.tx : 1024 x 1024, 3 channel, uint8 tiff
  11 subimages: 1024x1024 512x512 256x256 128x128 64x64 32x32 16x16 8x8 4x4 2x2 1x1
subimage 0: 1024 x 1024, 3 channel, uint8 tiff
  channel list: R, G, B
  tile size: 64 x 64
...
subimage 1: 512 x 512, 3 channel, uint8 tiff
  channel list: R, G, B
...
...
```

## 14.2 **iinfo** command-line options

`--help`

Prints usage information to the terminal.

`-v`

Verbose output — prints all metadata of the image files.

`-a`

Print information about all subimages in the file(s).

`-f`

Print the filename as a prefix to every line. For example,

```
$ iinfo -v -f img_6019m.jpg

img_6019m.jpg : 1024 x 683, 3 channel, uint8 jpeg
img_6019m.jpg : channel list: R, G, B
img_6019m.jpg : Color space: sRGB
img_6019m.jpg : ImageDescription: "Family photo"
img_6019m.jpg : Make: "Canon"
...
```

`-m pattern`

Match the *pattern* (specified as an extended regular expression) against data metadata field names and print only data fields whose names match. The default is to print all data fields found in the file (if `-v` is given).

For example,

```
$ iinfo -v -f -m ImageDescription test*.jpg

test3.jpg :      ImageDescription: "Birthday party"
test4.jpg :      ImageDescription: "Hawaii vacation"
test5.jpg :      ImageDescription: "Bob's graduation"
test6.jpg :      ImageDescription: <unknown>
```

Note: the `-m` option is probably not very useful without also using the `-v` and `-f` options.

`--hash`

Displays a SHA-1 hash of the pixel data of the image (and of each subimage if combined with the `-a` flag).

`-s`

Show the image sizes, including a sum of all the listed images.

# 15 Converting Image Formats With `iconvert`

## 15.1 Overview

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 correction, switch between tiled and scanline orientation, or alter or add certain metadata to the image.

The `iconvert` utility is invoked as follows:

```
iconvert [options] input output
```

Where *input* and *output* name the input image and desired output filename. The image files may be of any format recognized by OpenImageIO (i.e., for which `ImageInput` plugins are available). The file format of the output image will be inferred from the file extension of the output filename (e.g., "`foo.tif`" will write a TIFF file).

Alternately, any number of files may be specified as follows:

```
iconvert --inplace [options] file1 file2 ..
```

When the `--inplace` option is used, any number of file names  $\geq 1$  may be specified, and the image conversion commands are applied to each file in turn, with the output being saved under the original file name. This is useful for applying the same conversion to many files, or simply if you want to replace the input with the output rather than create a new file with a different name.

## 15.2 `iconvert` Recipes

This section will give quick examples of common uses of `iconvert`.

### Converting between file formats

It's a snap to converting among image formats supported by OpenImageIO (i.e., for which `ImageInput` and `ImageOutput` plugins can be found). The `iconvert` utility will simply infer the file format from the file extension. The following example converts a PNG image to JPEG:

```
iconvert lena.png lena.jpg
```

### Changing the data format or bit depth

Just use the `-d` option to specify a pixel data format. For example, assuming that `in.tif` uses 16-bit unsigned integer pixels, the following will convert it to an 8-bit unsigned pixels:

```
iconvert -d uint8 in.tif out.tif
```

### Changing the compression

The following command converts writes a TIFF file, specifically using LZW compression:

```
iconvert --compression lzw in.tif out.tif
```

The following command writes its results as a JPEG file at a compression quality of 50 (pretty severe compression):

```
iconvert --quality 50 big.jpg small.jpg
```

### Gamma-correcting an image

The following gamma-corrects the pixels, raising all pixel values to  $x^{1/2.2}$  upon writing:

```
iconvert -g 2.2 in.tif out.tif
```

### Converting between scanline and tiled images

Convert a scanline file to a tiled file with  $16 \times 16$  tiles:

```
iconvert --tile 16 16 s.tif t.tif
```

Convert a tiled file to scanline:

```
iconvert --scanline t.tif s.tif
```

### Converting images in place

You can use the `--inplace` flag to cause the output to *replace* the input file, rather than create a new file with a different name. For example, this will re-compress all of your TIFF files to use ZIP compression (rather than whatever they currently are using):

```
iconvert --inplace --compression zip *.tif
```



### Change the file modification time to the image capture time

Many image formats (including JPEGs from digital cameras) contain an internal time stamp indicating when the image was captured. But the time stamp on the file itself (what you'd see in a directory listing from your OS) most likely shows when the file was last copied, not when it was created or captured. You can use the following command to re-stamp your files so that the file system modification time matches the time that the digital image was originally captured:

```
iconvert --inplace --adjust-time *.jpg
```

### Add captions, keywords, IPTC tags

For formats that support it, you can add a caption/image description, keywords, or arbitrary string metadata:

```
iconvert --inplace --adjust-time --caption "Hawaii vacation" *.jpg
```

```
iconvert --inplace --adjust-time --keyword "John" img18.jpg img21.jpg
```

```
iconvert --inplace --adjust-time --attrib IPTC:State "HI" \  
--attrib IPTC:City "Honolulu" *.jpg
```

## 15.3 iconvert command-line options

--help

Prints usage information to the terminal.

-v

Verbose status messages.

--threads *n*

Use *n* execution threads if it helps to speed up image operations. The default (also if *n* = 0) is to use as many threads as there are cores present in the hardware.

--inplace

Causes the output to *replace* the input file, rather than create a new file with a different name.

Without this flag, `iconvert` expects two file names, which will be used to specify the input and output files, respectively.

But when `--inplace` option is used, any number of file names  $\geq 1$  may be specified, and the image conversion commands are applied to each file in turn, with the output being

saved under the original file name. This is useful for applying the same conversion to many files.

For example, the following example will add the caption “Hawaii vacation” to all JPEG files in the current directory:

```
iconvert --inplace --adjust-time --caption "Hawaii vacation" *.jpg
```

#### **-d** *datatype*

Attempt to sets the output pixel data type to one of: uint8, sint8, uint16, sint16, half, float, double.

The types uint10 and uint12 may be used to request 10- or 12-bit unsigned integers. If the output file format does not support them, uint16 will be substituted.

If the -d option is not supplied, the output data type will be the same as the data format of the input file, if possible.

In any case, if the output file type does not support the requested data type, it will instead use whichever supported data type results in the least amount of precision lost.

#### **-g** *gamma*

Applies a gamma correction of  $1/\text{gamma}$  to the pixels as they are output.

#### **--sRGB**

Explicitly tags the image as being in sRGB color space. Note that this does not alter pixel values, it only marks which color space those values refer to (and only works for file formats that understand such things). An example use of this command is if you have an image that is not explicitly marked as being in any particular color space, but you know that the values are sRGB.

#### **--tile** *x y*

Requests that the output file be tiled, with the given  $x \times y$  tile size, if tiled images are supported by the output format. By default, the output file will take on the tiledness and tile size of the input file.

#### **--scanline**

Requests that the output file be scanline-oriented (even if the input file was tile-oriented), if scanline orientation is supported by the output file format. By default, the output file will be scanline if the input is scanline, or tiled if the input is tiled.

`--separate`

`--contig`

Forces either “separate” (e.g., RRR...GGG...BBB) or “contiguous” (e.g., RGBRGBRGB...) packing of channels in the file. If neither of these options are present, the output file will have the same kind of channel packing as the input file. Of course, this is ignored if the output file format does not support a choice or does not support the particular choice requested.

`--compression method`

Sets the compression method for the output image. Each `ImageOutput` plugin will have its own set of methods that it supports.

By default, the output image will use the same compression technique as the input image (assuming it is supported by the output format, otherwise it will use the default compression method of the output plugin).

`--quality q`

Sets the compression quality, on a 1–100 floating-point scale. This only has an effect if the particular compression method supports a quality metric (as JPEG does).

`--no-copy-image`

Ordinarily, `iconvert` will attempt to use `ImageOutput::copy_image` underneath to avoid de/recompression or alteration of pixel values, unless other settings clearly contradict this (such as any settings that must alter pixel values). The use of `--no-copy-image` will force all pixels to be decompressed, read, and compressed/written, rather than copied in compressed form. We’re not exactly sure when you would need to do this, but we put it in just in case.

`--adjust-time`

When this flag is present, after writing the output, the resulting file’s modification time will be adjusted to match any “DateTime” metadata in the image. After doing this, a directory listing will show file times that match when the original image was created or captured, rather than simply when `iconvert` was run. This has no effect on image files that don’t contain any “DateTime” metadata.

`--caption text`

Sets the image metadata “ImageDescription”. This has no effect if the output image format does not support some kind of title, caption, or description metadata field. Be careful to enclose *text* in quotes if you want your caption to include spaces or certain punctuation!

`--keyword text`

Adds a keyword to the image metadata "Keywords". Any existing keywords will be preserved, not replaced, and the new keyword will not be added if it is an exact duplicate of existing keywords. This has no effect if the output image format does not support some kind of keyword field.

Be careful to enclose *text* in quotes if you want your keyword to include spaces or certain punctuation. For image formats that have only a single field for keywords, OpenImageIO will concatenate the keywords, separated by semicolon (;), so don't use semicolons within your keywords.

`--clear-keywords`

Clears all existing keywords in the image.

`--attrib name text`

Sets the named image metadata attribute to a string given by *text*. For example, you could explicitly set the IPTC location metadata fields with:

```
iconvert --attrib "IPTC:City" "Berkeley" in.jpg out.jpg
```

`--orientation orient`

Explicitly sets the image's "Orientation" metadata to a numeric value (see Section B.2 for the numeric codes). This only changes the metadata field that specifies how the image should be displayed, it does NOT alter the pixels themselves, and so has no effect for image formats that don't support some kind of orientation metadata.

`--rotcw`

`--rotccw`

`--rot180`

Adjusts the image's "Orientation" metadata by rotating it 90° clockwise, 90° degrees counter-clockwise, or 180°, respectively, compared to its current setting. This only changes the metadata field that specifies how the image should be displayed, it does NOT alter the pixels themselves, and so has no effect for image formats that don't support some kind of orientation metadata.

# 16 Searching Image Metadata With **igrep**

The `igrep` program search one or more image files for metadata that match a string or regular expression.

## 16.1 Using **igrep**

The `igrep` utility is invoked as follows:

```
igrep [options] pattern filename ...
```

Where *pattern* is a POSIX.2 regular expression (just like the Unix/Linux `grep(1)` command), and *filename* (and any following names) specify images or directories that should be searched. An image file will “match” if any of its metadata contains values contain substring that are recognized regular expression. The image files may be of any format recognized by OpenImageIO (i.e., for which ImageInput plugins are available).

Example:

```
$ igrep Jack *.jpg
bar.jpg: Keywords = Carly; Jack
foo.jpg: Keywords = Jack
test7.jpg: ImageDescription = Jack on vacation
```

## 16.2 **igrep** command-line options

`--help`

Prints usage information to the terminal.

`-d`

Print directory names as it recurses. This only happens if the `-r` option is also used.

`-E`

Interpret the pattern as an extended regular expression (just like `egrep` or `grep -E`).

-f

Match the expression against the filename, as well as the metadata within the file.

-i

Ignore upper/lower case distinctions. Without this flag, the expression matching will be case-sensitive.

-l

Simply list the matching files by name, suppressing the normal output that would include the metadata name and values that matched. For example:

```
$ igrep Jack *.jpg
bar.jpg: Keywords = Carly; Jack
foo.jpg: Keywords = Jack
test7.jpg: ImageDescription = Jack on vacation
```

```
$ igrep -l Jack *.jpg
bar.jpg
foo.jpg
test7.jpg
```

-r

Recurse into directories. If this flag is present, any files specified that are directories will have any image file contained therein to be searched for a match (and so on, recursively).

-v

Invert the sense of matching, to select image files that *do not* match the expression.

# 17 Comparing Images With `idiff`

## 17.1 Overview

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.

## 17.2 Using `idiff`

The `idiff` utility is invoked as follows:

```
idiff [options] image1 image2
```

Where *input1* and *input2* are the names of two image files that should be compared. They may be of any format recognized by OpenImageIO (i.e., for which image-reading plugins are available).

If the two input images are not the same resolutions, or do not have the same number of channels, the comparison will return FAILURE immediately and will not attempt to compare the pixels of the two images. If they are the same dimensions, the pixels of the two images will be compared, and a report will be printed including the mean and maximum error, how many pixels were above the warning and failure thresholds, and whether the result is PASS, WARNING, or FAILURE. For example:

```
$ idiff a.jpg b.jpg

Comparing "a.jpg" and "b.jpg"
Mean error = 0.00450079
RMS error = 0.00764215
Peak SNR = 42.3357
Max error = 0.254902 @ (700, 222, B)
574062 pixels (82.1%) over 1e-06
574062 pixels (82.1%) over 1e-06
FAILURE
```

The “mean error” is the average difference (per channel, per pixel). The “max error” is the largest difference in any pixel channel, and will point out on which pixel and channel it was found. It will also give a count of how many pixels were above the warning and failure thresholds.

The metadata of the two images (e.g., the comments) are not currently compared; only differences in pixel values are taken into consideration.

### Raising the thresholds

By default, if any pixels differ between the images, the comparison will fail. You can allow *some* differences to still pass by raising the failure thresholds. The following example will allow images to pass the comparison test, as long as no more than 10% of the pixels differ by 0.004 (just above a 1/255 threshold):

```
idiff -fail 0.004 -failpercent 10 a.jpg b.jpg
```

But what happens if a just a few pixels are very different? Maybe you want that to fail, also. The following adjustment will fail if at least 10% of pixels differ by 0.004, or if *any* pixel differs by more than 0.25:

```
idiff -fail 0.004 -failpercent 10 -hardfail 0.25 a.jpg b.jpg
```

If none of the failure criteria are met, and yet some pixels are still different, it will still give a **WARNING**. But you can also raise the warning threshold in a similar way:

```
idiff -fail 0.004 -failpercent 10 -hardfail 0.25 \  
-warn 0.004 -warnpercent 3 a.jpg b.jpg
```

The above example will **PASS** as long as fewer than 3% of pixels differ by more than 0.004. If it does, it will be a **WARNING** as long as no more than 10% of pixels differ by 0.004 and no pixel differs by more than 0.25, otherwise it is a **FAILURE**.

### Output a difference image

Ordinary text output will tell you how many pixels failed or were warnings, and which pixel had the biggest difference. But sometimes you need to see visually where the images differ. You can get `idiff` to save an image of the differences between the two input images:

```
idiff -o diff.tif -abs a.jpg b.jpg
```

The `-abs` flag saves the absolute value of the differences (i.e., all positive values or zero). If you omit the `-abs`, pixels in which `a.jpg` have smaller values than `b.jpg` will be negative in the difference image (be careful in this case of using a file format that doesn’t support negative values).

You can also scale the difference image with the `-scale`, making them easier to see. And the `-od` flag can be used to output a difference image only if the comparison fails, but not if the images pass within the designated threshold (thus saving you the trouble and space of saving a black image).



## 17.3 idiff Reference

The various command-line options are discussed below:

### General options

--help

Prints usage information to the terminal.

-v

Verbose output — more detail about what it finds when comparing images, even when the comparison does not fail.

-a

Compare all subimages. Without this flag, only the first subimage of each file will be compared.

### Thresholds and comparison options

-fail *A*

-failpercent *B*

-hardfail *C*

Sets the threshold for FAILURE: if more than *B*% of pixels (on a 0-100 floating point scale) are greater than *A* different, or if *any* pixels are more than *C* different. The defaults are to fail if more than 0% (any) pixels differ by more than 0.00001 (1e-6), and *C* is infinite.

-warn *A*

-warnpercent *B*

-hardwarn *C*

Sets the threshold for WARNING: if more than *B*% of pixels (on a 0-100 floating point scale) are greater than *A* different, or if *any* pixels are more than *C* different. The defaults are to warn if more than 0% (any) pixels differ by more than 0.00001 (1e-6), and *C* is infinite.

-p

Does an additional test on the images to attempt to see if they are *perceptually* different (whether you are likely to discern a difference visually), using Hector Yee's metric. If this option is enabled, the statistics will additionally show a report on how many pixels failed the perceptual test, and the test overall will fail if more than the "fail percentage" failed the perceptual test.

## Difference image output

### `-o outfile`

Outputs a *difference image* to the designated file. This difference image pixels consist are each of the value of the corresponding pixel from *image1* minus the value of the pixel *image2*.

The file extension of the output file is used to determine the file format to write (e.g., "out.tif" will write a TIFF file, "out.jpg" will write a JPEG, etc.). The data format of the output file will be format of whichever of the two input images has higher precision (or the maximum precision that the designated output format is capable of, if that is less than either of the input images).

Note that pixels whose value is lower in *image1* than in *image2*, this will result in negative pixels (which may be clamped to zero if the image format does not support negative values)), unless the `-abs` option is also used.

### `-abs`

Will cause the output image to consist of the *absolute value* of the difference between the two input images (so all values in the difference image  $\geq 0$ ).

### `-scale factor`

Scales the values in the difference image by the given (floating point) factor. The main use for this is to make small actual differences more visible in the resulting difference image by giving a large scale factor.

### `-od`

Causes a difference image to be produce *only* if the image comparison fails. That is, even if the `-o` option is used, images that are within the comparison threshold will not write out a useless black (or nearly black) difference image.

## Process return codes

The `idiff` program will return a code that can be used by scripts to indicate the results:

- 0 OK: the images match within the warning and error thresholds.
- 1 Warning: the errors differ a little, but within error thresholds.
- 2 Failure: the errors differ a lot, outside error thresholds.
- 3 The images weren't the same size and couldn't be compared.
- 4 File error: could not find or open input files, etc.

# 18 Making Tiled MIP-Map Texture Files With `maketx`

## 18.1 Overview

The `maketx` program will read an image (from any file format for which an `ImageInput` plugin can be found) and then write it in a form in which it will have high performance when used by `TextureSystem` (Chapter 8). This involves converting it to tiled (versus scanline) orientation, writing multiple subimages at different resolutions (MIP-map), and setting a variety of header or metadata fields appropriately for texture maps.

The `maketx` utility is invoked as follows:

```
maketx [options] input... -o output
```

Where *input* and *output* name the input image and desired output filename. The input files may be of any image format recognized by `OpenImageIO` (i.e., for which `ImageInput` plugins are available). The file format of the output image will be inferred from the file extension of the output filename (e.g., "`foo.tif`" will write a TIFF file).

## 18.2 `maketx` command-line options

`--help`

Prints usage information to the terminal.

`-v`

Verbose status messages, including runtime statistics and timing.

`-o outputname`

Sets the name of the output texture.

`--threads n`

Use *n* execution threads if it helps to speed up image operations. The default (also if *n* = 0) is to use as many threads as there are cores present in the hardware.

`--format formatname`

Specifies the image format of the output file (e.g., “tiff”, “OpenEXR”, etc.). If `--format` is not used, `maketx` will guess based on the file extension of the output filename; if it is not a recognized format extension, TIFF will be used by default.

`-d datatype`

Attempt to sets the output pixel data type to one of: `uint8`, `sint8`, `uint16`, `sint16`, `half`, `float`, `double`.

If the `-d` option is not supplied, the output data type will be the same as the data format of the input file.

In either case, the output file format itself (implied by the file extension of the output filename) may trump the request if the file format simply does not support the requested data type.

`--tile x y`

Specifies the tile size of the output texture. If not specified, `maketx` will make  $64 \times 64$  tiles.

`--separate`

Forces “separate” (e.g., RRR...GGG...BBB) packing of channels in the output file. Without this option specified, “contiguous” (e.g., RGBRGBRGB...) packing of channels will be used for those file formats that support it.

`--compression method`

Sets the compression method for the output image (the default is to try to use “zip” compression, if it is available).

`--update`

Ordinarily, textures are created unconditionally (which could take several seconds for large input files if read over a network) and will be stamped with the current time.

The `--update` option enables *update mode* I if the output file already exists and has the same time stamp as the input file, the texture will not be recreated. If the output file does not exist or has a different time than the input file, then the texture will be created be given the time stamp of the input file.

`--wrap wrapmode`

`--swrap wrapmode --twrap wrapmode`

Sets the default *wrap mode* for the texture, which determines the behavior when the texture is sampled outside the  $[0, 1]$  range. Valid wrap modes are: `black`, `clamp`, `periodic`, `mirror`. The default, if none is set, is `black`. The `--wrap` option sets the wrap mode in both directions simultaneously, while the `--swrap` and `--twrap` may be used to set them individually in the *s* (horizontal) and *t* (vertical) directions.

Although this sets the default wrap mode for a texture, note that the wrap mode may have an override specified in the texture lookup at runtime.

#### `--resize`

Causes the highest-resolution level of the MIP-map to be a power-of-two resolution in each dimension (by rounding up the resolution of the input image). There is no good reason to do this for the sake of OIIO's texture system, but some users may require it in order to create MIP-map images that are compatible with both OIIO and other texturing systems that require power-of-2 textures.

#### `--filter name`

By default, the resizing step that generates successive MIP levels uses a triangle filter to bilinearly combine pixels (for MIP levels with even number of pixels, this is also equivalent to a box filter, which merely averages groups of 4 adjacent pixels). This is fast, but for source images with high frequency content, can result in aliasing or other artifacts in the lower-resolution MIP levels.

The `--filter` option selects a high-quality filter to use when resizing to generate successive MIP levels. Choices include `lanczos3` (our best recommendation for highest-quality filtering, a 3-lobe Lanczos filter), `box`, `triangle`, `catrom`, `blackman-harris`, `gaussian`, `mittchell`, `bspline`, `radial-lanczos3`, `disk`, `sinc`.

If you select a filter with negative lobes (including `lanczos3`, `sinc`, `lanczos3`, or `catrom`), and your source image is an HDR image with very high contrast regions that include pixels with values  $> 1$ , you may also wish to use the `--rangecompress` option to avoid ringing artifacts.

#### `--hicomp`

Perform highlight compensation. When HDR input data with high-contrast highlights is turned into a MIP-mapped texture using a high-quality filter with negative lobes (such as `lanczos3`), objectionable ringing could appear near very high-contrast regions with pixel values  $> 1$ . This option improves those areas by using range compression (transforming values from a linear to a logarithmic scale that greatly compresses the values  $> 1$ ) prior to each image filtered-resize step, and then expanded back to a linear format after the resize, and also clamping resulting pixel values to be non-negative. This can result in some loss of energy, but often this is a preferable alternative to ringing artifacts in your upper MIP levels.

`--nomipmap`

Causes the output to *not* be MIP-mapped, i.e., only will have the highest-resolution level.

`--nchannels n`

Sets the number of output channels. If *n* is less than the number of channels in the input image, the extra channels will simply be ignored. If *n* is greater than the number of channels in the input image, the additional channels will be filled with 0 values.

`--chnames a,b,...`

Renames the channels of the output image. All the channel names are concatenated together, separated by commas. A “blank” entry will cause the channel to retain its previous value (for example, `--chnames ,,,A` will rename channel 3 to be “A” and leave channels 0–2 as they were).

`--checknan`

Checks every pixel of the input image to ensure that no NaN or Inf values are present. If such non-finite pixel values are found, an error message will be printed and `maketx` will terminate without writing the output image (returning an error code).

`--fixnan strategy`

Repairs any pixels in the input image that contained NaN or Inf values (hereafter referred to collectively as “nonfinite”). If *strategy* is `black`, nonfinite values will be replaced with 0. If *strategy* is `box3`, nonfinite values will be replaced by the average of all the finite values within a  $3 \times 3$  region surrounding the pixel.

`--fullpixels`

Resets the “full” (or “display”) pixel range to be the “data” range. This is used to deal with input images that appear, in their headers, to be crop windows or overscanned images, but you want to treat them as full 0–1 range images over just their defined pixel data.

`--Mcamera ...16 floats...`

`--Mscreen ...16 floats...`

Sets the camera and screen matrices (sometimes called `N1` and `NP`, respectively, by some renderers) in the texture file, overriding any such matrices that may be in the input image (and would ordinarily be copied to the output texture).

`--prman-metadata`

Causes metadata “`PixarTextureFormat`” to be set, which is useful if you intend to create an OpenEXR texture or environment map that can be used with PRMan as well as OIIO.

`--constant-color-detect`

Detects images in which all pixels are identical, and outputs the texture at a reduced resolution equal to the tile size, rather than filling endless tiles with the same constant color. That is, by substituting a low-res texture for a high-res texture if it's a constant color, you could save a lot of save disk space, I/O, and texture cache size. It also sets the "ImageDescription" to contain a special message of the form "ConstantColor=[r,g,...]".

`--monochrome-detect`

Detects multi-channel images in which all color components are identical, and outputs the texture as a single-channel image instead. That is, it changes RGB images that are gray into single-channel gray scale images.

Use with caution! This is a great optimization if such textures will only have their first channel accessed, but may cause unexpected behavior if the "client" application will attempt to access those other channels that will no longer exist.

`--opaque-detect`

Detects images that have a designated alpha channel for which the alpha value for all pixels is 1.0 (fully opaque), and omits the alpha channel from the output texture. So, for example, an RGBA input texture where A=1 for all pixels will be output just as RGB. The purpose is to save disk space, texture I/O bandwidth, and texturing time for those textures where alpha was present in the input, but clearly not necessary.

Use with caution! This is a great optimization only if your use of such textures will assume that missing alpha channels are equivalent to textures whose alpha is 1.0 everywhere.

`--ignore-unassoc`

Ignore any header tags in the input images that indicate that the input has "unassociated" alpha. When this option is used, color channels with unassociated alpha will not be automatically multiplied by alpha to turn them into associated alpha. This is also a good way to fix input images that really are associated alpha, but whose headers incorrectly indicate that they are unassociated alpha.

`--prman`

PRMan is will crash in strange ways if given textures that don't have its quirky set of tile sizes and other specific metadata. If you want `maketx` to generate textures that may be used with either `OpenImageIO` or `PRMan`, you should use the `--prman` option, which will set several options to make `PRMan` happy, overriding any contradictory settings on the command line or in the input texture.

Specifically, this option sets the tile size (to 64x64 for 8 bit, 64x32 for 16 bit integer, and 32x32 for float or half images), uses "separate" planar configuration (`--separate`), and sets `PRMan`-specific metadata (`--prman-metadata`). It also outputs `sint16` textures

if `uint16` is requested (because PRMan for some reason does not accept true `uint16` textures).

OpenImageIO will happily accept textures that conform to PRMan’s expectations, but not vice versa. But OpenImageIO’s `TextureSystem` has better performance with textures that use `maketx`’s default settings rather than these oddball choices. You have been warned!

`--oiio`

This sets several options that we have determined are the optimal values for OpenImageIO’s `TextureSystem`, overriding any contradictory settings on the command line or in the input texture.

Specifically, this is the equivalent to using

`--separate --tile 64 64 --hash.`

`--colorconvert inspace outspace`

Convert the color space of the input image from *inspace* to *ospace*. If `OpenColorIO` is installed and finds a valid configuration, it will be used for the color conversion. If `OCIO` is not enabled (or cannot find a valid configuration, `OIO` will at least be able to convert among linear, sRGB, and Rec709.

`--unpremult`

When undergoing color some conversions, it is helpful to “un-premultiply” the alpha before converting color channels, and then re-multiplying by alpha. Caveat emptor – if you don’t know exactly when to use this, you probably shouldn’t be using it at all.

`--mipimage filename`

Specifies the name of an image file to use as a custom MIP-map level, instead of simply downsizing the last one. This option may be used multiple times to specify multiple levels. For example:

```
maketx 256.tif --miplevel 128.tif --miplevel 64.tif -o out.tx
```

This will make a texture with the first MIP level taken from `256.tif`, the second level from `128.tif`, the third from `64.tif`, and then subsequent levels will be the usual downsizings of `64.tif`.

`--envlatl`

Creates a latitude-longitude environment map, rather than an ordinary texture map.

`--lightprobe`

Creates a latitude-longitude environment map, but in contrast to `--envlatl`, the original input image is assumed to be formatted as a *light probe* image<sup>1</sup>.

---

<sup>1</sup>See <http://www.pauldebevec.com/Probes/> for examples and an explanation of the geometric layout.



**Part III**

**Appendices**



## **A Building OpenImageIO**



## B Metadata conventions

The `ImageSpec` class, described thoroughly in Section 2.4, provides the basic description of an image that are essential across all formats — resolution, number of channels, pixel data format, etc. Individual images may have additional data, stored as name/value pairs in the `extra_attrs` field. Though literally *anything* can be stored in `extra_attrs` — it’s specifically designed for format- and user-extensibility — this chapter establishes some guidelines and lays out all of the field names that `OpenImageIO` understands.

### B.1 Description of the image

`"ImageDescription" : string`

The image description, title, caption, or comments.

`"Keywords" : string`

Semicolon-separated keywords describing the contents of the image. (Semicolons are used rather than commas because of the common case of a comma being part of a keyword itself, e.g., “Kurt Vonnegut, Jr.” or “Washington, DC.”)

`"Artist" : string`

The artist, creator, or owner of the image.

`"Copyright" : string`

Any copyright notice or owner of the image.

`"DateTime" : string`

The creation date of the image, in the following format: `YYYY:MM:DD HH:MM:SS` (exactly 19 characters long, not including a terminating `NULL`). For example, 7:30am on Dec 31, 2008 is encoded as `"2008:12:31 07:30:00"`.

`"DocumentName" : string`

The name of an overall document that this image is a part of.

"Software" : string

The software that was used to create the image.

"HostComputer" : string

The name or identity of the computer that created the image.

## B.2 Display hints

"oio:ColorSpace" : string

The name of the color space of the color channels. Values include: "Linear", "sRGB", "GammaCorrected", "AdobeRGB", "Rec709", and "KodakLog".

"oio:Gamma" : float

If the color space is "GammaCorrected", this value is the gamma exponent.

"oio:BorderColor" : float[nchannels]

The color presumed to be filling any parts of the display/full image window that are not overlapping the pixel data window. If not supplied, the default is black (0 in all channels).

"Orientation" : int

By default, image pixels are ordered from the top of the display to the bottom, and within each scanline, from left to right (i.e., the same ordering as English text and scan progression on a CRT). But the "Orientation" field can suggest that it should be displayed with a different orientation, according to the TIFF/EXIF conventions:

- 1 normal (top to bottom, left to right)
- 2 flipped horizontally (top to bottom, right to left)
- 3 rotate 180° (bottom to top, right to left)
- 4 flipped vertically (bottom to top, left to right)
- 5 transposed (left to right, top to bottom)
- 6 rotated 90° clockwise (right to left, top to bottom)
- 7 transverse (right to left, bottom to top)
- 8 rotated 90° counter-clockwise (left to right, bottom to top)

"PixelAspectRatio" : float

The aspect ratio ( $x/y$ ) of the individual pixels, with square pixels being 1.0 (the default).

```
"XResolution" : float
"YResolution" : float
"ResolutionUnit" : string
```

The number of horizontal (*x*) and vertical (*y*) pixels per resolution unit. This ties the image to a physical size (where applicable, such as with a scanned image, or an image that will eventually be printed).

Different file formats may dictate different resolution units. For example, the TIFF ImageIO plugin supports "none", "in", and "cm".

### B.3 Disk file format info/hints

```
"oiio:BitsPerSample" : int
```

Number of bits per sample *in the file*.

Note that this may not match the reported `ImageSpec::format`, if the plugin is translating from an unsupported format. For example, if a file stores 4 bit grayscale per channel, the "oiio:BitsPerSample" may be 4 but the `format` field may be `TypeDesc::UINT8` (because the OpenImageIO APIs do not support fewer than 8 bits per sample).

```
"oiio:UnassociatedAlpha" : int
```

Whether the data in the file stored alpha channels (if any) that were unassociated with the color (i.e., color not “premultiplied” by the alpha coverage value).

```
"planarconfig" : string
```

"contig" indicates that the file has contiguous pixels (RGB RGB RGB...), whereas "separate" indicate that the file stores each channel separately (RRR...GGG...BBB...).

Note that only contiguous pixels are transmitted through the OpenImageIO APIs, but this metadata indicates how it is (or should be) stored in the file, if possible.

```
"compression" : string
```

Indicates the type of compression the file uses. Supported compression modes will vary from `ImageInput` plugin to plugin, and each plugin should document the modes it supports. If `ImageInput::open` is called with an `ImageSpec` that specifies an compression mode not supported by that `ImageInput`, it will choose a reasonable default. As an example, the TIFF `ImageInput` plugin supports "none", "lzw", "ccittlrle", "zip" (the default), "packbits".

```
"CompressionQuality" : int
```

Indicates the quality of compression to use (0–100), for those plugins and compression methods that allow a variable amount of compression, with higher numbers indicating higher image fidelity.

## B.4 Photographs or scanned images

The following metadata items are specific to photos or captured images.

"Make" : string

For captured or scanned image, the make of the camera or scanner.

"Model" : string

For captured or scanned image, the model of the camera or scanner.

"ExposureTime" : float

The exposure time (in seconds) of the captured image.

"FNumber" : float

The f/stop of the camera when it captured the image.

## B.5 Texture Information

Several standard metadata are very helpful for images that are intended to be used as textures (especially for OpenImageIO's TextureSystem).

"textureformat" : string

The kind of texture that this image is intended to be. We suggest the following names:

"Plain Texture"	Ordinary 2D texture
"Volume Texture"	3D volumetric texture
"Shadow"	Ordinary z-depth shadow map
"CubeFace Shadow"	Cube-face shadow map
"Volume Shadow"	Volumetric ("deep") shadow map
"LatLong Environment"	Latitude-longitude (rectangular) environment map
"CubeFace Environment"	Cube-face environment map

"wrapmodes" : string

Give the intended texture *wrap mode* indicating what happens with texture coordinates outside the  $[0..1]$  range. We suggest the following names: "black", "periodic", "clamp", "mirror". If the wrap mode is different in each direction, they should simply be separated by a comma. For example, "black" means black wrap in both directions, whereas "clamp,periodic" means to clamp in  $u$  and be periodic in  $v$ .

"fovcot" : float

The cotangent ( $x/y$ ) of the field of view of the original image (which may not be the same as the aspect ratio of the pixels of the texture, which may have been resized).



"worldtocamera" : matrix44

For shadow maps or rendered images this item (of type `TypeDesc::PT_MATRIX`) is the world-to-camera matrix describing the camera position.

"worldtoscreen" : matrix44

For shadow maps or rendered images this item (of type `TypeDesc::PT_MATRIX`) is the world-to-screen matrix describing the full projection of the 3D view onto a  $[-1...1] \times [-1...1]$  2D domain.

"oiio:updirection" : string

For environment maps, indicates which direction is “up” (valid values are "y" or "z"), to disambiguate conventions for environment map orientation.

"oiio:sampleborder" : int

If not present or 0 (the default), the conversion from pixel integer coordinates  $(i, j)$  to texture coordinates  $(s, t)$  follows the usual convention of  $s = (i + 0.5)/xres$  and  $t = (j + 0.5)/yres$ . However, if this attribute is present and nonzero, the first and last row/column of pixels lie exactly at the  $s$  or  $t = 0$  or  $1$  boundaries, i.e.,  $s = i/(xres - 1)$  and  $t = j/(yres - 1)$ .

## B.6 Exif metadata

The following Exif metadata tags correspond to items in the “standard” set of metadata.

Exif tag	OpenImageIO metadata convention
ColorSpace	(reflected in "oiio:ColorSpace")
ExposureTime	"ExposureTime"
FNumber	"FNumber"

The other remaining Exif metadata tags all include the “Exif:” prefix to keep it from clashing with other names that may be used for other purposes.

"Exif:ExposureProgram" : int

The exposure program used to set exposure when the picture was taken:

- 0 unknown
- 1 manual
- 2 normal program
- 3 aperture priority
- 4 shutter priority
- 5 Creative program (biased toward depth of field)
- 6 Action program (biased toward fast shutter speed)
- 7 Portrait mode (closeup photo with background out of focus)
- 8 Landscape mode (background in focus)

"Exif:SpectralSensitivity" : string

The camera's spectral sensitivity, using the ASTM conventions.

"Exif:ISOSpeedRatings" : int

The ISO speed and ISO latitude of the camera as specified in ISO 12232.

"Exif:DateTimeOriginal" : string

Date and time that the original image data was generated (in "YYYY:MM:DD HH:MM:SS" format).

"Exif:DateTimeDigitized" : string

Date and time that the image was stored as digital data (in "YYYY:MM:DD HH:MM:SS" format).

"Exif:CompressedBitsPerPixel" : float

The compression mode used, measured in compressed bits per pixel.

"Exif:ShutterSpeedValue" : float

Shutter speed, in APEX units:  $-\log_2(\text{exposuretime})$

"Exif:ApertureValue" : float

Aperture, in APEX units:  $2\log_2(f\text{number})$

"Exif:BrightnessValue" : float

Brightness value, assumed to be in the range of  $-99.99 - 99.99$ .

"Exif:ExposureBiasValue" : float

Exposure bias, assumed to be in the range of  $-99.99 - 99.99$ .

"Exif:MaxApertureValue" : float

Smallest F number of the lens, in APEX units:  $2\log_2(f\text{number})$

"Exif:SubjectDistance" : float

Distance to the subject, in meters.

"Exif:MeteringMode" : int

The metering mode:

0	unknown
1	average
2	center-weighted average
3	spot
4	multi-spot
5	pattern
6	partial
255	other

"Exif:LightSource" : int

The kind of light source:

0	unknown
1	daylight
2	tungsten (incandescent light)
4	flash
9	fine weather
10	cloudy weather
11	shade
12	daylight fluorescent (D 5700-7100K)
13	day white fluorescent (N 4600-5400K)
14	cool white fluorescent (W 3900 - 4500K)
15	white fluorescent (WW 3200 - 3700K)
17	standard light A
18	standard light B
19	standard light C
20	D55
21	D65
22	D75
23	D50
24	ISO studio tungsten
255	other light source

"Exif:Flash" int

A sum of:

1	if the flash fired
0	no strobe return detection function
4	strobe return light was not detected
6	strobe return light was detected
8	compulsary flash firing
16	compulsary flash supression
24	auto-flash mode
32	no flash function (0 if flash function present)
64	red-eye reduction supported (0 if no red-eye reduction mode)

"Exif:FocalLength" : float

Actual focal length of the lens, in mm.

"Exif:SecurityClassification" : string

Security classification of the image: 'C' = confidential, 'R' = restricted, 'S' = secret, 'T' = top secret, 'U' = unclassified.

"Exif:ImageHistory" : string

Image history.

"Exif:SubsecTime" : string

Fractions of a second to augment the "DateTime" (expressed as text of the digits to the right of the decimal).

"Exif:SubsecTimeOriginal" : string

Fractions of a second to augment the "Exif:DateTimeOriginal" (expressed as text of the digits to the right of the decimal).

"Exif:SubsecTimeDigitized" : string

Fractions of a second to augment the "Exif:DateTimeDigital" (expressed as text of the digits to the right of the decimal).

"Exif:PixelXDimension" : int

"Exif:PixelYDimension" : int

The  $x$  and  $y$  dimensions of the valid pixel area. FIXME – better explanation?

"Exif:FlashEnergy" : float

Strobe energy when the image was captures, measured in Beam Candle Power Seconds (BCPS).

```
"Exif:FocalPlaneXResolution" : float
"Exif:FocalPlaneYResolution" : float
"Exif:FocalPlaneResolutionUnit" : int
```

The number of pixels in the  $x$  and  $y$  dimension, per resolution unit. The codes for resolution units are:

- |   |               |
|---|---------------|
| 1 | none          |
| 2 | inches        |
| 3 | cm            |
| 4 | mm            |
| 5 | $\mu\text{m}$ |

```
"Exif:ExposureIndex" : float
```

The exposure index selected on the camera.

```
"Exif:SensingMethod" : int
```

The image sensor type on the camera:

- |   |                              |
|---|------------------------------|
| 1 | undefined                    |
| 2 | one-chip color area sensor   |
| 3 | two-chip color area sensor   |
| 4 | three-chip color area sensor |
| 5 | color sequential area sensor |
| 7 | trilinear sensor             |
| 8 | color trilinear sensor       |

```
"Exif:FileSource" : int
```

The source type of the scanned image, if known:

- |   |                          |
|---|--------------------------|
| 1 | film scanner             |
| 2 | reflection print scanner |
| 3 | digital camera           |

```
"Exif:SceneType" : int
```

Set to 1, if a directly-photographed image, otherwise it should not be present.

```
"Exif:CustomRendered" : int
```

Set to 0 for a normal process, 1 if some custom processing has been performed on the image data.

"Exif:ExposureMode" : int

The exposure mode:

- |   |              |
|---|--------------|
| 0 | auto         |
| 1 | manual       |
| 2 | auto-bracket |

"Exif:WhiteBalance" : int

Set to 0 for auto white balance, 1 for manual white balance.

"Exif:DigitalZoomRatio" : float

The digital zoom ratio used when the image was shot.

"Exif:FocalLengthIn35mmFilm" : int

The equivalent focal length of a 35mm camera, in mm.

"Exif:SceneCaptureType" : int

The type of scene that was shot:

- |   |             |
|---|-------------|
| 0 | standard    |
| 1 | landscape   |
| 2 | portrait    |
| 3 | night scene |

"Exif:GainControl" : float

The degree of overall gain adjustment:

- |   |                |
|---|----------------|
| 0 | none           |
| 1 | low gain up    |
| 2 | high gain up   |
| 3 | low gain down  |
| 4 | high gain down |

"Exif:Contrast" : int

The direction of contrast processing applied by the camera:

- |   |        |
|---|--------|
| 0 | normal |
| 1 | soft   |
| 2 | hard   |

"Exif:Saturation" : int

The direction of saturation processing applied by the camera:

- 0 normal
- 1 low saturation
- 2 high saturation

"Exif:Sharpness" : int

The direction of sharpness processing applied by the camera:

- 0 normal
- 1 soft
- 2 hard

"Exif:SubjectDistanceRange" : int

The distance to the subject:

- 0 unknown
- 1 macro
- 2 close
- 3 distant

"Exif:ImageUniqueID" : string

A unique identifier for the image, as 16 ASCII hexadecimal digits representing a 128-bit number.

## B.7 GPS Exif metadata

The following GPS-related Exif metadata tags correspond to items in the “standard” set of metadata.

"GPS:LatitudeRef" : string

Whether the "GPS:Latitude" tag refers to north or south: "N" or "S".

"GPS:Latitude" : float[3]

The degrees, minutes, and seconds of latitude (see also "GPS:LatitudeRef").

"GPS:LongitudeRef" : string

Whether the "GPS:Longitude" tag refers to east or west: "E" or "W".

"GPS:Longitude" : float[3]

The degrees, minutes, and seconds of longitude (see also "GPS:LongitudeRef").

"GPS:AltitudeRef" : string

A value of 0 indicates that the altitude is above sea level, 1 indicates below sea level.

"GPS:Altitude" : float

Absolute value of the altitude, in meters, relative to sea level (see "GPS:AltitudeRef" for whether it's above or below sea level).

"GPS:TimeStamp" : float[3]

Gives the hours, minutes, and seconds, in UTC.

"GPS:Satellites" : string

Information about what satellites were visible.

"GPS:Status" : string

"A" indicates a measurement in progress, "V" indicates measurement interoperability.

"GPS:MeasureMode" : string

"2" indicates a 2D measurement, "3" indicates a 3D measurement.

"GPS:DOP" : float

Data degree of precision.

"GPS:SpeedRef" : string

Indicates the units of the related "GPS:Speed" tag: "K" for km/h, "M" for miles/h, "N" for knots.

"GPS:Speed" : float

Speed of the GPS receiver (see "GPS:SpeedRef" for the units).

"GPS:TrackRef" : string

Describes the meaning of the "GPS:Track" field: "T" for true direction, "M" for magnetic direction.

"GPS:Track" : float

Direction of the GPS receiver movement (from 0–359.99). The related "GPS:TrackRef" indicate whether it's true or magnetic.



"GPS:ImgDirectionRef" : string

Describes the meaning of the "GPS:ImgDirection" field: "T" for true direction, "M" for magnetic direction.

"GPS:ImgDirection" : float

Direction of the image when captured (from 0–359.99). The related "GPS:ImgDirectionRef" indicate whether it's true or magnetic.

"GPS:MapDatum" : string

The geodetic survey data used by the GPS receiver.

"GPS:DestLatitudeRef" : string

Whether the "GPS:DestLatitude" tag refers to north or south: "N" or "S".

"GPS:DestLatitude" : float[3]

The degrees, minutes, and seconds of latitude of the destination (see also "GPS:DestLatitudeRef").

"GPS:DestLongitudeRef" : string

Whether the "GPS:DestLongitude" tag refers to east or west: "E" or "W".

"GPS:DestLongitude" : float[3]

The degrees, minutes, and seconds of longitude of the destination (see also "GPS:DestLongitudeRef").

"GPS:DestBearingRef" : string

Describes the meaning of the "GPS:DestBearing" field: "T" for true direction, "M" for magnetic direction.

"GPS:DestBearing" : float

Bearing to the destination point (from 0–359.99). The related "GPS:DestBearingRef" indicate whether it's true or magnetic.

"GPS:DestDistanceRef" : string

Indicates the units of the related "GPS:DestDistance" tag: "K" for km, "M" for miles, "N" for knots.

"GPS:DestDistance" : float

Distance to the destination (see "GPS:DestDistanceRef" for the units).

"GPS:ProcessingMethod" : string

Processing method information.

"GPS:AreaInformation" : string

Name of the GPS area.

"GPS:DateStamp" : string

Date according to the GPS device, in format "YYYY:MM:DD".

"GPS:Differential" : int

If 1, indicates that differential correction was applied.

"GPS:HPositioningError" : float

Positioning error.

## B.8 IPTC metadata

The IPTC (International Press Telecommunications Council) publishes conventions for storing image metadata, and this standard is growing in popularity and is commonly used in photo-browsing programs to record captions and keywords.

The following IPTC metadata items correspond exactly to metadata in the OpenImageIO conventions, so it is recommended that you use the standards and that plugins supporting IPTC metadata respond likewise:

<b>IPTC tag</b>	<b>OpenImageIO metadata convention</b>
Caption	"ImageDescription"
Keyword	IPTC keywords should be concatenated, separated by semicolons (;), and stored as the "Keywords" attribute.
ExposureTime	"ExposureTime"
CopyrightNotice	"Copyright"
Creator	"Artist"

The remainder of IPTC metadata fields should use the following names, prefixed with "IPTC:" to avoid conflicts with other plugins or standards.

"IPTC:ObjectTypeReference" : string

Object type reference.

"IPTC:ObjectAttributeReference" : string

Object attribute reference.

"IPTC:ObjectName" : string

The name of the object in the picture.

"IPTC:EditStatus" : string

Edit status.

"IPTC:SubjectReference" : string

Subject reference.

"IPTC:Category" : string

Category.

"IPTC:ContentLocationCode" : string

Code for content location.

"IPTC:ContentLocationName" : string

Name of content location.

"IPTC:ReleaseDate" : string

"IPTC:ReleaseTime" : string

Release date and time.

"IPTC:ExpirationDate" : string

"IPTC:ExpirationTime" : string

Expiration date and time.

"IPTC:Instructions" : string

Special instructions for handling the image.

"IPTC:ReferenceService" : string

"IPTC:ReferenceDate" : string

"IPTC:ReferenceNumber" : string

Reference date, service, and number.

"IPTC:DateCreated" : string

"IPTC:TimeCreated" : string

Date and time that the image was created.

"IPTC:DigitalCreationDate" : string

"IPTC:DigitalCreationTime" : string

Date and time that the image was digitized.

"IPTC:ProgramVersion" : string

The version number of the creation software.

"IPTC:AuthorsPosition" : string

The job title or position of the creator of the image.

"IPTC:City" : string

"IPTC:Sublocation" : string

"IPTC:State" : string

"IPTC:Country" : string

"IPTC:CountryCode" : string

The city, sublocation within the city, state, country, and country code of the location of the image.

"IPTC:Headline" : string

Any headline that is meant to accompany the image.

"IPTC:Provider" : string

The provider of the image, or credit line.

"IPTC:Source" : string

The source of the image.

"IPTC:Contact" : string

The contact information for the image (possibly including name, address, email, etc.).

"IPTC:CaptionWriter" : string

The name of the person who wrote the caption or description of the image.

"IPTC:JobID" : string

"IPTC:MasterDocumentID" : string

"IPTC:ShortDocumentID" : string

"IPTC:UniqueDocumentID" : string

"IPTC:OwnerID" : string

Various identification tags.

```
"IPTC:Prefs" : string
"IPTC:ClassifyState" : string
"IPTC:SimilarityIndex" : string
```

Who knows what the heck these are?

```
"IPTC:DocumentNotes" : string
```

Notes about the image or document.

```
"IPTC:DocumentHistory" : string
```

The history of the image or document.

## B.9 SMPTE metadata

```
"smpte:TimeCode" : int[2]
```

SMPTE time code, encoded as an array of 2 32-bit integers (as a TypeDesc, it will be tagged with vecsemantics TIMECODE).

```
"smpte:KeyCode" : int[7]
```

SMPTE key code, encoded as an array of 7 32-bit integers (as a TypeDesc, it will be tagged with vecsemantics KEYCODE).

## B.10 Extension conventions

To avoid conflicts with other plugins, or with any additional standard metadata names that may be added in future versions of OpenImageIO, it is strongly advised that writers of new plugins should prefix their metadata with the name of the format, much like the "Exif:" and "IPTC:" metadata.



## C Glossary

**Channel** One of several data values present 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.

**Image Format Plugin** A DSO/DLL that implements the `ImageInput` and `ImageOutput` classes for a particular image file format.

**Format Plugin** See *image format plugin*.

**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.

**Plugin** See *image format plugin*.

**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.



# Index

- add, 169, 219
- attribute, 17, 113, 130
- BMP, 93**
  - channels, 164, 216
  - checker, 163, 216
  - Cineon, 93**
  - clamp, 171, 220
  - colorconvert, 181, 226
  - compare, 174, 222
  - composite, 248
  - computePixelHashSHA1, 177, 223
  - computePixelStats, 173
  - convolve, 178, 224
  - crop, 165, 217
  - crop windows, 27
  - cut, 166
- data formats, 7
- DDS, 93**
- deep data, 37, 58
- deep images, 165, 217, 244
- depth composite, 248
- DPX, 94**
- environment, 139
- error checking, 17, 40, 47, 60, 68, 122, 143
- Exif metadata, 289
- extension\_list, 18
- fft, 179, 224
- Field3D, 96**
- fill, 163, 215
- FITS, 96**
- fixNonFinite, 180, 225
- flatten, 165, 217
- flip, 166, 218
- flipflop, 167, 218
- flop, 167, 218
- format\_list, 18
- getAttribute, 18, 114, 131
- geterror, 17
- GIF, 97**
- globalattribute, 17
- GPS metadata, 295
- HDR, 97**
- histogram, 177
- ICO, 98**
- iconvert, 263
- idiff, 271
- IFF, 98**
- ifft, 179, 224
- igrep, 269
- iinfo, 259
- Image Buffers, 145–160**
- Image Cache, 111–122**
- Image I/O API, 21–47, 49–68**
- Image I/O API Helper Classes, 7–19**
- Image Processing, 161–187**
- ImageBuf, 145–160**
- ImageBufAlgo, 161–187**
  - add, 169, 219
  - capture\_image, 187, 227
  - channel\_append, 165, 217
  - channel\_sum, 170, 220
  - channels, 164, 216
  - checker, 163, 216
  - circular\_shift, 167, 218
  - clamp, 171, 220
  - color\_count, 175
  - color\_range\_check, 176
  - colorconvert, 181, 226
  - compare, 174, 222

- complex\_to\_polar, 179, 224
- computePixelHashSHA1, 177, 223
- computePixelStats, 173
- convolve, 178, 224
- crop, 165, 217
- cut, 166
- fft, 179, 224
- fill, 163, 215
- fillholes\_pushpull, 180, 225
- fixNonFinite, 180, 225
- flatten, 165, 217
- flip, 166, 218
- flipflop, 167, 218
- flop, 167, 218
- from\_IplImage, 187
- histogram, 177
- ifft, 179, 224
- isConstantChannel, 175, 222
- isConstantColor, 174, 222
- isMonochrome, 175, 223
- make\_kernel, 178, 223
- make\_texture, 184, 227
- mul, 169, 219
- ociodisplay, 183, 226
- ociolook, 182, 226
- over, 172, 221
- paste, 166, 217
- polar\_to\_complex, 179, 224
- pow, 170, 220
- premult, 183, 227
- rangecompress, 171, 220
- rangeexpand, 171, 220
- render\_text, 163, 216
- resample, 168, 219
- resize, 168, 218
- sub, 169, 219
- to\_IplImage, 187
- transpose, 167, 218
- unpremult, 183, 227
- unsharp\_mask, 181, 225
- zero, 162, 215
- zover, 173, 221
- ImageOutput, 21
- ImageSpec, 11
- Intel Image Library, 187
- IplImage, 187
- IPTC metadata, 298
- isConstantChannel, 175, 222
- isConstantColor, 174, 222
- isMonochrome, 175, 223
- iv, 257
- JPEG, 98
- Jpeg 2000, 99
- maketx, 275
- mul, 169, 219
- ociodisplay, 183, 226
- ociolook, 182, 226
- oiio tool, 231
- OpenCV, 187
- OpenEXR, 100
- Orientation, 286
- over, 172, 221
- overscan, 27
- paste, 166, 217
- plugin\_searchpath, 18
- Plugins
  - bundled, 93–109
- PNG, 102
- PNM, 102
- pow, 170, 220
- premult, 183, 227
- PSD, 103
- Ptex, 103
- rangecompress, 171, 220
- rangeexpand, 171, 220
- RAW digital camera files, 105
- region of interest, 145
- resample, 168, 219
- resize, 168, 218
- RGBE, 97
- RLA, 103
- ROI, 145
- SGI files, 105
- shadow, 138
- SMPTE metadata, 301
- Softimage PIC, 105

sub, 169, 219

Targa, 105

texture, 134

Texture System, 123–144

texture3d, 136

TextureOpt, 123

threads, 17

TIFF, 106

transpose, 167, 218

unpremult, 183, 227

ustring, 10

version, 17

WebP, 109

zero, 162, 215

Zfile, 109

zover, 173, 221