

---

# OpenImageIO

*Release 2.1.20*

**Larry Gritz**

**Mar 13, 2021**



## CONTENTS

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Image I/O API Helper Classes</b>	<b>9</b>
<b>3</b>	<b>ImageOutput: Writing Images</b>	<b>41</b>
<b>4</b>	<b>ImageInput: Reading Images</b>	<b>67</b>
<b>5</b>	<b>Writing ImageIO Plugins</b>	<b>89</b>
<b>6</b>	<b>Bundled ImageIO Plugins</b>	<b>115</b>
<b>7</b>	<b>Cached Images</b>	<b>137</b>
<b>8</b>	<b>Texture Access: TextureSystem</b>	<b>151</b>
<b>9</b>	<b>ImageBuf: Image Buffers</b>	<b>171</b>
<b>10</b>	<b>ImageBufAlgo: Image Processing</b>	<b>189</b>
<b>11</b>	<b>Python Bindings</b>	<b>243</b>
<b>12</b>	<b>oiiotool: the OIIO Swiss Army Knife</b>	<b>291</b>
<b>13</b>	<b>Getting Image information With <code>iinfo</code></b>	<b>343</b>
<b>14</b>	<b>Converting Image Formats With <code>iconvert</code></b>	<b>347</b>
<b>15</b>	<b>Searching Image Metadata With <code>igrep</code></b>	<b>353</b>
<b>16</b>	<b>Comparing Images With <code>idiff</code></b>	<b>355</b>
<b>17</b>	<b>Making Tiled MIP-Map Texture Files With <code>maketx</code> or <code>oiiotool</code></b>	<b>359</b>
<b>18</b>	<b>Metadata conventions</b>	<b>367</b>
<b>19</b>	<b>Glossary</b>	<b>381</b>
	<b>Index</b>	<b>383</b>



The code that implements OpenImageIO is licensed under the BSD 3-clause (also sometimes known as “new BSD” or “modified BSD”) license (<https://github.com/OpenImageIO/oio/blob/master/LICENSE.md>):

#### BSD 3-Clause License

Copyright (c) 2008-present by Contributors to the OpenImageIO project. All Rights Reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. 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.
3. Neither the name of the copyright holder 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 HOLDER 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/>

OpenImageIO incorporates code from several other software packages with compatible licenses. Copies of their licenses are reproduced here: <https://github.com/OpenImageIO/oio/blob/master/LICENSE-THIRD-PARTY.md>



## INTRODUCTION

Welcome to OpenImageIO!

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

### 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. A variety of functions in the ImageBufAlgo namespace are available to perform common image processing operations on ImageBuf’s.

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 to several TBs) very efficiently using only a tiny amount (tens of megabytes to a few GB 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 this functionality, and therefore may read or write any image file type for which an appropriate plugin is found at runtime. Paramount among these utilities `oiio_tool`, a command-line image processing engine, and `iv`, an 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 “BSD 3-clause” 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`, `libjpeg`, 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’s support of “movie” files is limited to viewing them as a sequence of separate frames within the file, but not as movies per se (for example, no support for dealing with audio or synchronization).
- Pixel data are presented as 8- 16- or 32-bit int (signed or unsigned), 16- 32- or 64-bit float. NOTHING ELSE. No < 8 bit images, or pixel value boundaries that aren’t byte boundaries. Files with < 8 will appear to the client application 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 by default converted to grayscale or RGB. Non-spectral color models, such as XYZ, CMYK, or YUV, are converted to RGB upon reading. (There is a way to override this and ask for raw pixel values.)
- 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.

## 1.3 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 further refinements since I had to rewrite the entire implementation from scratch anyway. I think the additional changes are all improvements.

Over the years and with the help of dozens of open source contributors, OpenImageIO has expanded beyond the original simple image format input/output to encompass a wide range of image-related functionality. It has grown into a foundational technology in many products and tools, particularly for the production of animation and visual effects for motion pictures (but also many other uses and fields). 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.

## 1.4 Acknowledgments

OpenImageIO incorporates, depends upon, or dynamically links against several other open source packages, detailed below. These other packages are all distributed under licenses that allow them to be used by OpenImageIO. Where not specifically noted, they are all using the same BSD license that OpenImageIO uses. Any omissions or inaccuracies in this list are inadvertent and will be fixed if pointed out. The full original licenses can be found in the relevant parts of the source code.

OpenImageIO incorporates, distributes, or contains derived works of:

- The SHA-1 implementation we use is public domain by Dominik Reichl <http://www.dominik-reichl.de/>
- Squish © 2006 Simon Brown, MIT license. <http://sjbrown.co.uk/?code=squish>
- PugiXML © 2006-2009 by Arseny Kapoulkine (based on work © 2003 Kristen Wegner), MIT license. <http://pugixml.org/>
- DPX reader/writer © 2009 Patrick A. Palmer, BSD 3-clause license. <https://github.com/patrickpalmer/dpx>
- tinyformat.h © 2011 Chris Foster, Boost license. <http://github.com/c42f/tinyformat>
- lookup3 code by Bob Jenkins, Public Domain. <http://burtleburtle.net/bob/c/lookup3.c>
- xxhash © 2014 Yann Collet, BSD 2-clause license. <https://github.com/Cyan4973/xxHash>
- farmhash © 2014 Google, Inc., MIT license. <https://github.com/google/farmhash>
- KissFFT © 2003–2010 Mark Borgerding, 3-clause BSD license. <https://github.com/mborgerding/kissfft>
- CTPL thread pool © 2014 Vitaliy Vitsentiy, Apache License. <https://github.com/vit-vit/CTPL>
- Droid fonts from the Android SDK are distributed under the Apache license. <http://www.droidfonts.com>
- function\_view.h contains code derived from LLVM, © 2003–2018 University of Illinois at Urbana-Champaign. UIUC license (compatible with BSD) <http://llvm.org>
- FindOpenVDB.cmake © 2015 Blender Foundation, BSD license.
- FindTBB.cmake © 2015 Justus Calvin, MIT license.
- fmt library © Victor Zverovich. MIT license. <https://github.com/fmtlib/fmt>
- UTF-8 decoder © 2008-2009 Bjoern Hoehrmann, MIT license. <http://bjoern.hoehrmann.de/utf-8/decoder/dfa>
- Base-64 encoder © René Nyffenegger, Zlib license. <http://www.adp-gmbh.ch/cpp/common/base64.html>
- stb\_sprintf © 2017 Sean Barrett, public domain (or MIT license where that may not apply). <https://github.com/nothings/stb>

OpenImageIO Has the following build-time dependencies (using system installs, referencing as git submodules, or downloading as part of the build), including link-time dependencies against dynamic libraries:

- 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>
- 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/>

- Ptex © 2009 Disney Enterprises, Inc. <http://ptex.us>
- Field3D © 2009 Sony Pictures Imageworks. <http://sites.google.com/site/field3d/>
- 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/>
- FFmpeg © various authors and distributed under LGPL. <https://www.ffmpeg.org>
- FreeType © 1996-2002, 2006 by David Turner, Robert Wilhelm, and Werner Lemberg. Distributed under the FreeType license (BSD compatible).
- JPEG-Turbo © 2009–2015 D. R. Commander. Distributed under the BSD license.
- pybind11 © 2016 Wenzel Jakob. Distributed under the BSD license. <https://github.com/pybind/pybind11>
- OpenVDB © 2012-2018 DreamWorks Animation LLC, Mozilla Public License 2.0. <https://www.openvdb.org/>
- Thread Building Blocks © Intel. Apache 2.0 license. <https://www.threadingbuildingblocks.org/>
- libheif copyright 2017-2018 Struktur AG (LGPL). <https://github.com/strukturag/libheif>



## 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 class, `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~ref{sec:pythontypedesc}](#) for the corresponding Python bindings.

#### **struct `TypeDesc`**

A *`TypeDesc`* describes simple data types.

It frequently comes up (in my experience, with renderers and image handling programs) that you want a way to describe data that is passed through APIs through blind pointers. These are some simple classes that provide a simple type descriptor system. This is not meant to be comprehensive for example, there is no provision for structs, unions, pointers, const, or ‘nested’ type definitions. Just simple integer and floating point, *common* aggregates such as 3-points, and reasonably-lengthed arrays thereof.

#### **Public Types**

##### **enum `BASETYPE`**

`BASETYPE` is a simple enum describing the base data types that correspond (mostly) to the C/C++ built-in types.

*Values:*

##### **UNKNOWN**

unknown type

##### **NONE**

void/no type

##### **UINT8**

8-bit unsigned int values ranging from 0..255, (C/C++ unsigned char).

**UCHAR** = *UINT8*

**INT8**

8-bit int values ranging from -128..127, (C/C++ `char`).

**CHAR** = *INT8*

**UINT16**

16-bit int values ranging from 0..65535, (C/C++ `unsigned short`).

**USHORT** = *UINT16*

**INT16**

16-bit int values ranging from -32768..32767, (C/C++ `short`).

**SHORT** = *INT16*

**UINT32**

32-bit unsigned int values (C/C++ `unsigned int`).

**UINT** = *UINT32*

**INT32**

signed 32-bit int values (C/C++ `int`).

**INT** = *INT32*

**UINT64**

64-bit unsigned int values (C/C++ `unsigned long long` on most architectures).

**ULONGLONG** = *UINT64*

**INT64**

signed 64-bit int values (C/C++ `long long` on most architectures).

**LONGLONG** = *INT64*

**HALF**

16-bit IEEE floating point values (OpenEXR `half`).

**FLOAT**

32-bit IEEE floating point values, (C/C++ `float`).

**DOUBLE**

64-bit IEEE floating point values, (C/C++ `double`).

**STRING**

Character string.

**PTR**

A pointer value.

**LASTBASE****enum AGGREGATE**

AGGREGATE describes whether our *TypeDesc* is a simple scalar of one of the `BASETYPE`'s, or one of several simple aggregates.

Note that aggregates and arrays are different. A `TypeDesc(FLOAT, 3)` is an array of three floats, a `TypeDesc(FLOAT, VEC3)` is a single 3-component vector comprised of floats, and `TypeDesc(FLOAT, 3, VEC3)` is an array of 3 vectors, each of which is comprised of 3 floats.

*Values:*

**SCALAR** = 1

A single scalar value (such as a raw `int` or `float` in C). This is the default.

**VEC2** = 2  
2 values representing a 2D vector.

**VEC3** = 3  
3 values representing a 3D vector.

**VEC4** = 4  
4 values representing a 4D vector.

**MATRIX33** = 9  
9 values representing a 3x3 matrix.

**MATRIX44** = 16  
16 values representing a 4x4 matrix.

#### enum VECSEMANTICS

VECSEMANTICS gives hints about what the data represent (for example, if a spatial vector quantity should transform as a point, direction vector, or surface normal).

*Values:*

**NOXFORM** = 0  
No semantic hints.

**NOSEMANTICS** = 0  
No semantic hints.

**COLOR**  
Color.

**POINT**  
Point: a spatial location.

**VECTOR**  
Vector: a spatial direction.

**NORMAL**  
Normal: a surface normal.

**TIMECODE**  
indicates an `int[2]` representing the standard 4-byte encoding of an SMPTE timecode.

**KEYCODE**  
indicates an `int[7]` representing the standard 28-byte encoding of an SMPTE keycode.

**RATIONAL**  
A VEC2 representing a rational number `val[0] / val[1]`

## Public Functions

**constexpr TypeDesc** (*BASETYPE* btype = *UNKNOWN*, *AGGREGATE* agg = *SCALAR*, *VECSEMANTICS* semantics = *NOSEMANTICS*, int arraylen = 0)  
Construct from a BASETYPE and optional aggregateness, semantics, and arrayness.

**constexpr TypeDesc** (*BASETYPE* btype, int arraylen)  
Construct an array of a non-aggregate BASETYPE.

**constexpr TypeDesc** (*BASETYPE* btype, *AGGREGATE* agg, int arraylen)  
Construct an array from BASETYPE, AGGREGATE, and array length, with unspecified (or moot) semantic hints.

**TypeDesc** (*string\_view* *typestring*)

Construct from a string (e.g., “float[3]”). If no valid type could be assembled, set base to UNKNOWN.

Examples:

```
TypeDesc("int") == TypeDesc(TypeDesc::INT)           // C++ int32_t
TypeDesc("float") == TypeDesc(TypeDesc::FLOAT)       // C++ float
TypeDesc("uint16") == TypeDesc(TypeDesc::UINT16)     // C++ uint16_t
TypeDesc("float[4]") == TypeDesc(TypeDesc::FLOAT, 4) // array
TypeDesc("point") == TypeDesc(TypeDesc::FLOAT,
                               TypeDesc::VEC3, TypeDesc::POINT)
```

**constexpr TypeDesc** (const *TypeDesc* &*t*)

Copy constructor.

**const** char \***c\_str** () **const**

Return the name, for printing and whatnot. For example, “float”, “int[5]”, “normal”

**constexpr size\_t** **numelements** () **const**

Return the number of elements: 1 if not an array, or the array length. Invalid to call this for arrays of undetermined size.

**constexpr size\_t** **basevalues** () **const**

Return the number of basetype values: the aggregate count multiplied by the array length (or 1 if not an array). Invalid to call this for arrays of undetermined size.

**constexpr bool** **is\_array** () **const**

Does this *TypeDesc* describe an array?

**constexpr bool** **is\_unsized\_array** () **const**

Does this *TypeDesc* describe an array, but whose length is not specified?

**constexpr bool** **is\_sized\_array** () **const**

Does this *TypeDesc* describe an array, whose length is specified?

**size\_t** **size** () **const**

Return the size, in bytes, of this type.

**constexpr TypeDesc** **elementtype** () **const**

Return the type of one element, i.e., strip out the array-ness.

**size\_t** **elementsize** () **const**

Return the size, in bytes, of one element of this type (that is, ignoring whether it’s an array).

**constexpr TypeDesc** **scalartype** () **const**

Return just the underlying C scalar type, i.e., strip out the array-ness and the aggregateness.

**size\_t** **basesize** () **const**

Return the base type size, i.e., stripped of both array-ness and aggregateness.

**bool** **is\_floating\_point** () **const**

True if it’s a floating-point type (versus a fundamentally integral type or something else like a string).

**bool** **is\_signed** () **const**

True if it’s a signed type that allows for negative values.

**constexpr bool** **is\_unknown** () **const**

Shortcut: is it UNKNOWN?



**constexpr operator bool () const**

if (typedesc) is the same as asking whether it's not UNKNOWN.

**size\_t fromstring (string\_view tpestring)**

Set \*this to the type described in the string. Return the length of the part of the string that describes the type. If no valid type could be assembled, return 0 and do not modify \*this.

**constexpr bool operator== (const TypeDesc &t) const**

Compare two *TypeDesc* values for equality.

**constexpr bool operator!= (const TypeDesc &t) const**

Compare two *TypeDesc* values for inequality.

**constexpr bool equivalent (const TypeDesc &b) const**

Member version of equivalent.

**constexpr bool is\_vec2 (BASETYPE b = FLOAT) const**

Is this a 2-vector aggregate (of the given type, float by default)?

**constexpr bool is\_vec3 (BASETYPE b = FLOAT) const**

Is this a 3-vector aggregate (of the given type, float by default)?

**constexpr bool is\_vec4 (BASETYPE b = FLOAT) const**

Is this a 4-vector aggregate (of the given type, float by default)?

**void unarray (void)**

Demote the type to a non-array

**bool operator< (const TypeDesc &x) const**

Test for lexicographic 'less', comes in handy for lots of STL containers and algorithms.

## Public Members

**unsigned char basetype**

C data type at the heart of our type.

**unsigned char aggregate**

What kind of AGGREGATE is it?

**unsigned char vecsemantics**

Hint: What does the aggregate represent?

**unsigned char reserved**

Reserved for future expansion.

**int arraylen**

Array length, 0 = not array, -1 = unsized.

## Friends

**constexpr** bool **operator==** (const TypeDesc &*t*, BASETYPE *b*)

Compare a *TypeDesc* to a basetype (it's the same if it has the same base type and is not an aggregate or an array).

**constexpr** bool **operator!=** (const TypeDesc &*t*, BASETYPE *b*)

Compare a *TypeDesc* to a basetype (it's the same if it has the same base type and is not an aggregate or an array).

**constexpr** bool **equivalent** (const TypeDesc &*a*, const TypeDesc &*b*)

*TypeDesc*'s are equivalent if they are equal, or if their only inequality is differing vector semantics.

A number of static `constexpr TypeDesc` aliases for common types exist in the outer OpenImageIO scope:

```
TypeUnknown TypeFloat TypeColor TypePoint TypeVector TypeNormal
TypeMatrix33 TypeMatrix44 TypeMatrix TypeHalf
TypeInt TypeUInt TypeInt16 TypeUInt16 TypeInt8 TypeUInt8
TypeFloat2 TypeVector2 TypeFloat4 TypeVector2i
TypeString TypeTimeCode TypeKeyCode
TypeRational TypePointer
```

The only types 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 `sec-imageoutput-metadata` Sections `sec-imageoutput-metadata`, `sec-imageinput-metadata`, and the documentation of individual ImageIO plugins for details).

## 2.2 Non-owning string views: `string_view`

### **class** `string_view`

A *string\_view* 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\_view* parameter and accept input that is any of `char*` (C string), string literal (constant `char` array), a `std::string` (C++ string), or `OIO 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\_view* 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\_view* merely refers to characters owned by another string, so the *string\_view* may not be used outside the lifetime of the string it refers to. Thus, *string\_view* is great for parameter passing, but it's not a good idea to use a *string\_view* 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\_view* 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\_view* to a function that expects a 0-terminated string (say, `fopen`), you must call `fopen(my_string_view.c_str())`. Note that the usual case is that the *string\_view* 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.

## Public Functions

**string\_view()**

Default ctr.

**string\_view(const string\_view &copy)**

Copy ctr.

**string\_view(const charT \*chars, size\_t len)**

Construct from `char*` and length.

**string\_view(const charT \*chars)**

Construct from `char*`, use `strlen` to determine length.

**string\_view(const std::string &str)**

Construct from `std::string`. Remember that a *string\_view* doesn't have its own copy of the characters, so don't use the *string\_view* after the original string has been destroyed or altered.

**std::string str() const**

Convert a *string\_view* to a `std::string`.

**const char \*c\_str() const**

Explicitly request a 0-terminated string. USUALLY, this turns out to be just `data()`, with no significant added expense (because most uses of *string\_view* are simple wrappers of C strings, C++ `std::string`, or `ustring` all of which are 0-terminated). But in the more rare case that the *string\_view* represents a non-0-terminated substring, it will force an allocation and copy underneath.

Caveats:

1. This is NOT going to be part of the C++17 `std::string_view`, so it's probably best to avoid this method if you want to have 100% drop-in compatibility with `std::string_view`.
2. It is NOT SAFE to use `c_str()` on a *string\_view* whose last char is the end of an allocation because that next char may only *coincidentally* be a `'\0'`, which will cause `c_str()` to return the string start (thinking it's a valid C string, so why not just return its address?), if there's any chance that the subsequent char could change from 0 to non-zero during the use of the result of `c_str()`, and thus break the assumption that it's a valid C str.

**operator std::string() const**

Convert a *string\_view* to a `std::string`.

**bool empty() const**

Is the *string\_view* empty, containing no characters?

**const charT &operator[] (size\_type pos) const**

Element access of an individual character (beware: no bounds checking!).

**const charT &at (size\_t pos) const**

Element access with bounds checking and exception if out of bounds.

**size\_type find (string\_view s, size\_t pos = 0) const**

Find the first occurrence of substring `s` in `*this`, starting at position `pos`.

size\_type **find**(charT c, size\_t pos = 0) **const**

Find the first occurrence of character c in \*this, starting at position pos.

size\_type **rfind**(string\_view s, size\_t pos = npos) **const**

Find the last occurrence of substring s \*this, but only those occurrences earlier than position pos.

size\_type **rfind**(charT c, size\_t pos = npos) **const**

Find the last occurrence of character c in \*this, but only those occurrences earlier than position pos.

## 2.3 Efficient unique strings: `ustring`

### **class** `ustring`

A `ustring` is an alternative to `char*` or `std::string` for storing strings, in which the character sequence is unique (allowing many speed advantages for assignment, equality testing, and inequality testing).

The implementation is that behind the scenes there is a hash set of allocated strings, so the characters of each string are unique. A `ustring` itself is a pointer to the characters of one of these canonical strings. Therefore, assignment and equality testing is just a single 32- or 64-bit int operation, the only mutex is when a `ustring` is created from raw characters, and the only malloc is the first time each canonical `ustring` is created.

The internal table also contains a `std::string` version and the length of the string, so converting a `ustring` to a `std::string` (via `ustring::string()`) or querying the number of characters (via `ustring::size()` or `ustring::length()`) is extremely inexpensive, and does not involve creation/allocation of a new `std::string` or a call to `strlen`.

We try very hard to completely mimic the API of `std::string`, including all the constructors, comparisons, iterations, etc. Of course, the characters of a `ustring` are non-modifiable, so we do not replicate any of the non-const methods of `std::string`. But in most other ways it looks and acts like a `std::string` and so most templated algorithms that would work on a “const `std::string` &” will also work on a `ustring`.

Usage guidelines:

Compared to standard strings, `ustrings` have several advantages:

- Each individual `ustring` is very small in fact, we guarantee that a `ustring` is the same size and memory layout as an ordinary `char*`.
- Storage is frugal, since there is only one allocated copy of each unique character sequence, throughout the lifetime of the program.
- Assignment from one `ustring` to another is just copy of the pointer; no allocation, no character copying, no reference counting.
- Equality testing (do the strings contain the same characters) is a single operation, the comparison of the pointer.
- Memory allocation only occurs when a new `ustring` is constructed from raw characters the FIRST time subsequent constructions of the same string just finds it in the canonical string set, but doesn't need to allocate new storage. Destruction of a `ustring` is trivial, there is no de-allocation because the canonical version stays in the set. Also, therefore, no user code mistake can lead to memory leaks.

But there are some problems, too. Canonical strings are never freed from the table. So in some sense all the strings “leak”, but they only leak one copy for each unique string that the program ever comes across. Also,

creation of unique strings from raw characters is more expensive than for standard strings, due to hashing, table queries, and other overhead.

On the whole, ustrings are a really great string representation

- if you tend to have (relatively) few unique strings, but many copies of those strings;
- if the creation of strings from raw characters is relatively rare compared to copying or comparing to existing strings;
- if you tend to make the same strings over and over again, and if it's relatively rare that a single unique character sequence is used only once in the entire lifetime of the program;
- if your most common string operations are assignment and equality testing and you want them to be as fast as possible;
- if you are doing relatively little character-by-character assembly of strings, string concatenation, or other "string manipulation" (other than equality testing).

ustrings are not so hot

- if your program tends to have very few copies of each character sequence over the entire lifetime of the program;
- if your program tends to generate a huge variety of unique strings over its lifetime, each of which is used only a short time and then discarded, never to be needed again;
- if you don't need to do a lot of string assignment or equality testing, but lots of more complex string manipulation.

## Public Functions

**ustring** (void)

Default ctor for ustring make an empty string.

**ustring** (const char \*str)

Construct a ustring from a null-terminated C string (char \*).

**ustring** (string\_view str)

Construct a ustring from a [string\\_view](#), which can be auto-converted from either a null-terminated C string (char \*) or a C++ std::string.

**ustring** (const char \*str, size\_type pos, size\_type n)

Construct a ustring from at most n characters of str, starting at position pos.

**ustring** (const char \*str, size\_type n)

Construct a ustring from the first n characters of str.

**ustring** (size\_type n, char c)

Construct a ustring from n copies of character c.

**ustring** (const std::string &str, size\_type pos, size\_type n = npos)

Construct a ustring from an indexed substring of a std::string.

**ustring** (const ustring &str)

Copy construct a ustring from another ustring.

**ustring** (const ustring &str, size\_type pos, size\_type n = npos)

Construct a ustring from an indexed substring of a ustring.

**~ustring()**  
ustring destructor.

**operator string\_view() const**  
Conversion to *string\_view*.

**operator std::string() const**  
Conversion to std::string (explicit only!).

**const ustring &assign(const ustring &str)**  
Assign a ustring to \*this.

**const ustring &assign(const ustring &str, size\_type pos, size\_type n = npos)**  
Assign a substring of a ustring to \*this.

**const ustring &assign(const std::string &str)**  
Assign a std::string to \*this.

**const ustring &assign(const std::string &str, size\_type pos, size\_type n = npos)**  
Assign a substring of a std::string to \*this.

**const ustring &assign(const char \*str)**  
Assign a null-terminated C string (char\*) to \*this.

**const ustring &assign(const char \*str, size\_type n)**  
Assign the first n characters of str to \*this.

**const ustring &assign(size\_type n, char c)**  
Assign n copies of c to \*this.

**const ustring &assign(string\_view str)**  
Assign a *string\_view* to \*this.

**const ustring &operator=(const ustring &str)**  
Assign a ustring to another ustring.

**const ustring &operator=(const char \*str)**  
Assign a null-terminated C string (char \*) to a ustring.

**const ustring &operator=(const std::string &str)**  
Assign a C++ std::string to a ustring.

**const ustring &operator=(string\_view str)**  
Assign a *string\_view* to a ustring.

**const ustring &operator=(char c)**  
Assign a single char to a ustring.

**const char \*c\_str() const**  
Return a C string representation of a ustring.

**const char \*data() const**  
Return a C string representation of a ustring.

**const std::string &string() const**  
Return a C++ std::string representation of a ustring.

**void clear(void)**  
Reset to an empty string.

`size_t length (void) const`  
Return the number of characters in the string.

`size_t hash (void) const`  
Return a hashed version of the string.

`size_t size (void) const`  
Return the number of characters in the string.

`bool empty (void) const`  
Is the string empty i.e., is it nullptr or does it point to an empty string?

`const_iterator begin () const`  
Return a `const_iterator` that references the first character of the string.

`const_iterator end () const`  
Return a `const_iterator` that references the end of a traversal of the characters of the string.

`const_reverse_iterator rbegin () const`  
Return a `const_reverse_iterator` that references the last character of the string.

`const_reverse_iterator rend () const`  
Return a `const_reverse_iterator` that references the end of a reverse traversal of the characters of the string.

`const_reference operator [] (size_type pos) const`  
Return a reference to the character at the given position. Note that it's up to the caller to be sure pos is within the size of the string.

`size_type copy (char *s, size_type n, size_type pos = 0) const`  
Dump into character array s the characters of this ustring, beginning with position pos and copying at most n characters.

`ustring substr (size_type pos = 0, size_type n = npos) const`  
Returns a substring of the ustring object consisting of n characters starting at position pos.

`int compare (string_view str) const`  
Return 0 if \*this is lexicographically equal to str, -1 if \*this is lexicographically earlier than str, 1 if \*this is lexicographically after str.

`int compare (const char *str) const`  
Return 0 if \*this is lexicographically equal to str, -1 if \*this is lexicographically earlier than str, 1 if \*this is lexicographically after str.

`bool operator== (const ustring &str) const`  
Test two ustrings for equality are they comprised of the same sequence of characters. Note that because ustrings are unique, this is a trivial pointer comparison, not a char-by-char loop as would be the case with a `char*` or a `std::string`.

`bool operator!= (const ustring &str) const`  
Test two ustrings for inequality are they comprised of different sequences of characters. Note that because ustrings are unique, this is a trivial pointer comparison, not a char-by-char loop as would be the case with a `char*` or a `std::string`.

`bool operator== (const std::string &x) const`  
Test a ustring (\*this) for lexicographic equality with `std::string` x.

`bool operator== (string_view x) const`  
Test a ustring (\*this) for lexicographic equality with `string_view` x.

bool **operator==** (const char \*x) const  
Test a ustring (*this*) for lexicographic equality with char x.

bool **operator!=** (const std::string &x) const  
Test a ustring (\*this) for lexicographic inequality with std::string x.

bool **operator!=** (string\_view x) const  
Test a ustring (\*this) for lexicographic inequality with string\_view x.

bool **operator!=** (const char \*x) const  
Test a ustring (*this*) for lexicographic inequality with char x.

bool **operator<** (const ustring &x) const  
Test for lexicographic 'less', comes in handy for lots of STL containers and algorithms.

## Public Static Functions

template<typename ...Args>  
**static ustring sprintf** (const char \*fmt, const Args&... args)  
Construct a ustring in a printf-like fashion. In other words, something like: ustring s = *ustring::sprintf*("blah %d %g", (int)foo, (float)bar); The argument list is fully typesafe. The formatting of the string will always use the classic "C" locale conventions (in particular, '.' as decimal separator for float values).

template<typename ...Args>  
**static ustring fmtformat** (const char \*fmt, const Args&... args)  
Construct a ustring in a fmt::format-like fashion. In other words, something like: ustring s = *ustring::fmtformat*("blah {} {}", (int)foo, (float)bar); The argument list is fully typesafe. The formatting of the string will always use the classic "C" locale conventions (in particular, '.' as decimal separator for float values).

template<typename ...Args>  
**static ustring format** (const char \*fmt, const Args&... args)  
NOTE: Semi-DEPRECATED! This will someday switch to behave like fmt::format (or future std::format) but for now, it is back compatible and equivalent to sprintf.

**static ustring concat** (string\_view s, string\_view t)  
Concatenate two strings, returning a ustring, implemented carefully to not perform any redundant copies or allocations. This is semantically equivalent to *ustring::sprintf("%s%s", s, t)*, but is more efficient.

**static std::string getstats** (bool verbose = true)  
Return the statistics output as a string.

**static size\_t memory** ()  
Return the amount of memory consumed by the ustring table.

**static const char \*make\_unique** (string\_view str)  
Given a *string\_view*, return a pointer to the unique version kept in the internal table (creating a new table entry if we haven't seen this sequence of characters before). N.B.: this is equivalent to *ustring(str).c\_str()*. It's also the routine that is used directly by ustring's internals to generate the canonical unique copy of the characters.

**static bool is\_unique** (const char \*str)  
Is this character pointer a unique ustring representation of those characters? Useful for diagnostics and debugging.



**static** *ustring* **from\_unique**(**const** char \**unique*)

Create a ustring from characters guaranteed to already be ustring-clean, without having to run through the hash yet again. Use with extreme caution!!!

## Friends

int **compare** (**const** std::string &*a*, **const** ustring &*b*)

Return 0 if *a* is lexicographically equal to *b*, -1 if *a* is lexicographically earlier than *b*, 1 if *a* is lexicographically after *b*.

bool **operator==** (**const** std::string &*a*, **const** ustring &*b*)

Test for lexicographic equality between std::string *a* and ustring *b*.

bool **operator==** (string\_view *a*, **const** ustring &*b*)

Test for lexicographic equality between *string\_view* *a* and ustring *b*.

bool **operator==** (**const** char \**a*, **const** ustring &*b*)

Test for lexicographic equality between char\* *a* and ustring *b*.

bool **operator!=** (**const** std::string &*a*, **const** ustring &*b*)

Test for lexicographic inequality between std::string *a* and ustring *b*.

bool **operator!=** (string\_view *a*, **const** ustring &*b*)

Test for lexicographic inequality between *string\_view* *a* and ustring *b*.

bool **operator!=** (**const** char \**a*, **const** ustring &*b*)

Test for lexicographic inequality between char\* *a* and ustring *b*.

std::ostream &**operator<<** (std::ostream &*out*, **const** ustring &*str*)

Generic stream output of a ustring.

## 2.4 Non-owning array views: `span` / `cspan`

template<typename T, ptrdiff\_t **Extent** = dynamic\_extent>

**class** `span`

`span<T>` is a non-owning, non-copying, non-allocating reference to a contiguous array of T objects known length. A ‘span’ encapsulates both a pointer and a length, and thus is a safer way of passing pointers around (because the function called knows how long the array is). A function that might ordinarily take a T\* and a length could instead just take a `span<T>`.

A `span<T>` is mutable (the values in the array may be modified). A non-mutable (i.e., read-only) reference would be `span<const T>`. Thus, a function that might ordinarily take a `const T*` and a length could instead take a `span<const T>`.

For convenience, we also define `cspan<T>` as equivalent to `span<const T>`.

A `span` may be initialized explicitly from a pointer and length, by initializing with a `std::vector<T>`, or by initializing with a constant (treated as an array of length 1). For all of these cases, no extra allocations are performed, and no extra copies of the array contents are made.

Important caveat: The `span` merely refers to items owned by another array, so the `span` should not be used beyond the lifetime of the array it refers to. Thus, `span` is great for parameter passing, but it's not a good idea to use a `span` to store values in a data structure (unless you are really sure you know what you're doing).

## Public Functions

**constexpr span** ()

Default constructor the span points to nothing.

template<class **U**, ptrdiff\_t **N**>

**constexpr span** (const *span*<**U**, **N**> &copy)

Copy constructor (copies the span pointer and length, NOT the data).

**constexpr span** (const *span* &copy)

Copy constructor (copies the span pointer and length, NOT the data).

**constexpr span** (pointer *data*, index\_type *size*)

Construct from T\* and length.

**constexpr span** (pointer *b*, pointer *e*)

Construct from begin and end pointers.

**constexpr span** (T &*data*)

Construct from a single T&.

template<size\_t **N**>

**constexpr span** (T (&*data*)[**N**])

Construct from a fixed-length C array. Template magic automatically finds the length from the declared type of the array.

template<class **Allocator**>

**constexpr span** (std::vector<T, *Allocator*> &*v*)

Construct from std::vector<T>.

template<class **Allocator**>

**span** (const std::vector<value\_type, *Allocator*> &*v*)

Construct from const std::vector<T>. This turns const std::vector<T> into a span<const T> (the span isn't const, but the data it points to will be).

template<size\_t **N**>

**constexpr span** (std::array<value\_type, **N**> &*arr*)

Construct from mutable element std::array.

template<size\_t **N**>

**constexpr span** (const std::array<value\_type, **N**> &*arr*)

Construct from read-only element std::array.

**constexpr span** (std::initializer\_list<T> *il*)

Construct an span from an initializer\_list.

**span** &**operator=** (const span &copy)

Assignment copies the pointer and length, not the data.

template<index\_type **Count**>

**constexpr span**<element\_type, *Count*> **first** () const

Subview containing the first Count elements of the span.

```
template<index_type Count>
constexpr span<element_type, Count> last () const
    Subview containing the last Count elements of the span.
```

Additionally, there is a convenience template:

```
template<typename T>
using cspan = span<const T>
    cspan<T> is a synonym for a non-mutable span<const T>.
```

## 2.5 Rectangular region of interest: ROI

### struct ROI

*ROI* is a small helper struct describing a rectangular region of interest in an image. The region is [xbegin,xend) x [ybegin,yend) x [zbegin,zend), with the “end” designators signifying one past the last pixel in each dimension, a la STL style.

### ROI data members

The data members are:

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

These describe the spatial extent [xbegin,xend) x [ybegin,yend) x [zbegin,zend) And the channel extent: [chbegin, chend)]

### Spatial size functions.

The width, height, and depth of the region.

```
constexpr int width () const
    Height.
```

```
constexpr int height () const
    Width.
```

```
constexpr int depth () const
    Depth.
```

## Public Functions

**constexpr ROI ()**

Default constructor is an undefined region. Note that this is also interpreted as *All()*.

**constexpr ROI (int xbegin, int xend, int ybegin, int yend, int zbegin = 0, int zend = 1, int chbegin = 0, int chend = 10000)**

Constructor with an explicitly defined region.

**constexpr bool defined () const**

Is a region defined?

**constexpr int nchannels () const**

Number of channels in the region. Beware this defaults to a huge number, and to be meaningful you must consider `std::min (imagebuf.nchannels(), roi.nchannels())`.

**constexpr imagesize\_t npixels () const**

Total number of pixels in the region.

**constexpr bool contains (int x, int y, int z = 0, int ch = 0) const**

Test if the coordinate is within the *ROI*.

**constexpr bool contains (const ROI &other) const**

Test if another *ROI* is entirely within our *ROI*.

## Public Static Functions

**static constexpr ROI All ()**

*All()* is an alias for the default constructor, which indicates that it means “all” of the image, or no region restriction. For example, `float myfunc (ImageBuf &buf, ROI roi = ROI::All());`; Note that this is equivalent to: `float myfunc (ImageBuf &buf, ROI roi = {});`

## Friends

**constexpr bool operator== (const ROI &a, const ROI &b)**

Test equality of two ROIs.

**constexpr bool operator!= (const ROI &a, const ROI &b)**

Test inequality of two ROIs.

`std::ostream &operator<< (std::ostream &out, const ROI &roi)`

Stream output of the range.

In addition, there are several related helper functions that involve ROI:

**constexpr ROI OIIO::roi\_union (const ROI &A, const ROI &B)**

Union of two regions, the smallest region containing both.

**constexpr ROI OIIO::roi\_intersection (const ROI &A, const ROI &B)**

Intersection of two regions.

**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.

## 2.6 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 [ImageSpec](#) for the corresponding Python bindings.

### **class ImageSpec**

*ImageSpec* describes the data format of an image dimensions, layout, number and meanings of image channels.

The `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. The `x`, `y`, `z` fields 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.

The analogous `full_width`, `full_height`, `full_depth` and `full_x`, `full_y`, `full_z` 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.

For tiled images, `tile_width`, `tile_height`, and `tile_depth` specify that the image is stored in a file organized into rectangular *tiles* of these dimensions. The default of 0 value for these fields indicates that the image is stored in scanline order, rather than as tiles.

### 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 *x***  
origin (upper left corner) of pixel data

**int *y***  
origin (upper left corner) of pixel data

**int *z***  
origin (upper left corner) of pixel data

**int *width***  
width of the pixel data window

**int *height***  
height of the pixel data window

**int *depth***  
depth of pixel data, >1 indicates a “volume”

**int *full\_x***  
origin of the full (display) window

**int *full\_y***  
origin of the full (display) window

**int *full\_z***  
origin of the full (display) window

**int *full\_width***  
width of the full (display) window

**int *full\_height***  
height of the full (display) window

**int *full\_depth***  
depth of the full (display) window

**int *tile\_width***  
tile width (0 for a non-tiled image)

**int *tile\_height***  
tile height (0 for a non-tiled image)

**int *tile\_depth***  
tile depth (0 for a non-tiled image, 1 for a non-volume image)

**int *nchannels***  
number of image channels, e.g., 4 for RGBA

#### *TypeDesc* format

Data format of the channels. Describes the native format of the pixel data values themselves, as a *TypeDesc*. Typical values would be *TypeDesc::UINT8* for 8-bit unsigned values, *TypeDesc::FLOAT* for 32-bit floating-point values, etc.

**std::vector<*TypeDesc*> *channelformats***

Optional per-channel data formats. 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`

True if the image contains deep data. 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.

## Public Functions

`ImageSpec (TypeDesc format = TypeDesc::UNKNOWN)`

Constructor: given just the data format, set all other fields to something reasonable.

`ImageSpec (int xres, int yres, int nchans, TypeDesc fmt = TypeUInt8)`

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.

`ImageSpec (const ROI &roi, TypeDesc fmt = TypeUInt8)`

Construct an *ImageSpec* whose dimensions (both data and "full") and number of channels are given by the *ROI*, pixel data type by `fmt`, and other fields are set to their default values.

`void set_format (TypeDesc fmt)`

Set the data format, and clear any per-channel format information in `channelformats`.

`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**

Return 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**

Return the number of bytes for each pixel (counting all channels). If *native* is false (default), assume all channels are in `this->format`, but if *native* is true, compute the size of a pixel in the “native” data format of the file (these may differ in the case of per-channel formats).

**size\_t pixel\_bytes** (int *chbegin*, int *chend*, bool *native* = false) **const**

Return the number of bytes for just the subset of channels in each pixel described by [*chbegin*,*chend*). If *native* is false (default), assume all channels are in `this->format`, but if *native* is true, compute the size of a pixel in the “native” data format of the file (these may differ in the case of per-channel formats).

**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**

Return the number of pixels comprising a tile (or 0 if it is not 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, i.e., `pixel_bytes(native) * tile_width * tile_height * tile_depth`. If *native* is false (default), assume all channels are in `this->format`, but if *native* is true, compute the size of a pixel in the “native” data format of the file (these may differ in the case of per-channel formats).

**imagesize\_t image\_pixels** () **const**

Return the number of pixels for an entire 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 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<image size_t>max()` in the event of an overflow where it's not representable in an `imagesize_t`. If *native* is false (default), assume all channels are in `this->format`, but if *native* is true, compute the size of a pixel in the “native” data format of the file (these may differ in the case of per-channel formats).

**bool size\_t\_safe** () **const**

Verify that on this platform, a `size_t` is big enough to hold the number of bytes (and pixels) in a scanline, a tile, and the whole image. If this returns false, the image is much too big to allocate and read all at once, so client apps beware and check these routines for overflows!

**void attribute** (*string\_view* *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\_view* *name*, unsigned int *value*)

Add an unsigned int attribute to `extra_attribs`.



void **attribute** (*string\_view* name, int value)  
Add an int attribute to extra\_attribs.

void **attribute** (*string\_view* name, float value)  
Add a float attribute to extra\_attribs.

void **attribute** (*string\_view* name, *string\_view* value)  
Add a string attribute to extra\_attribs.

void **attribute** (*string\_view* name, *TypeDesc* type, *string\_view* value)  
Parse a string containing a textual representation of a value of the given type, and add that as an attribute to extra\_attribs. Example:

```
spec.attribute ("temperature", TypeString, "-273.15");
```

void **erase\_attribute** (*string\_view* name, *TypeDesc* searchtype = *TypeDesc::UNKNOWN*, bool casesensitive = false)  
Searches extra\_attribs for any attributes matching name (as a regular expression), removing them entirely from extra\_attribs. If searchtype is anything other than *TypeDesc::UNKNOWN*, matches will be restricted only to attributes with the given type. The name comparison will be case-sensitive if casesensitive is true, otherwise in a case-insensitive manner.

ParamValue \***find\_attribute** (*string\_view* name, *TypeDesc* searchtype = *TypeDesc::UNKNOWN*, bool casesensitive = false)  
Searches extra\_attribs for an attribute matching name, returning a pointer to the attribute record, or NULL if there was no match. If searchtype is anything other than *TypeDesc::UNKNOWN*, matches will be restricted only to attributes with the given type. The name comparison will be exact if casesensitive is true, otherwise in a case-insensitive manner if caseinsensitive is false.

**const** ParamValue \***find\_attribute** (*string\_view* name, ParamValue &tmpparam, *TypeDesc* searchtype = *TypeDesc::UNKNOWN*, bool casesensitive = false) **const**  
Search for the named attribute and return the pointer to its ParamValue record, or NULL if not found. This variety of find\_attribute() can retrieve items such as “width”, which are data members of the *ImageSpec*, but not in extra\_attribs. The tmpparam is a storage area owned by the caller, which is used as temporary buffer in cases where the information does not correspond to an actual extra\_attribs (in this case, the return value will be &tmpparam). The extra names it understands are:

- "x" "y" "z" "width" "height" "depth" "full\_x" "full\_y" "full\_z" "full\_width" "full\_height" "full\_depth"

Returns the *ImageSpec* fields of those names (despite the fact that they are technically not arbitrary named attributes in extra\_attribs). All are of type int.

- "datawindow"

Without a type, or if requested explicitly as an int[4], returns the OpenEXR-like pixel data min and max coordinates, as a 4-element integer array: { x, y, x+width-1, y+height-1 }. If instead you specifically request as an int[6], it will return the volumetric data window, { x, y, z, x+width-1, y+height-1, z+depth-1 }.

- "displaywindow"

Without a type, or if requested explicitly as an int[4], returns the OpenEXR-like pixel display min and max coordinates, as a 4-element integer array: { full\_x, full\_y, full\_x+full\_width-1, full\_y+full\_height-1 }. If instead you specifically request as an int[6], it will return the volumetric display window, { full\_x, full\_y, full\_z, full\_x+full\_width-1, full\_y+full\_height-1, full\_z+full\_depth-1 }.

## EXAMPLES

```

ImageSpec spec;           // has the info
Imath::Box2i dw;          // we want the displaywindow here
ParamValue tmp;           // so we can retrieve pseudo-values
TypeDesc int4("int[4]");  // Equivalent: TypeDesc int4(TypeDesc::INT, 4);
const ParamValue* p = spec.find_attribute ("displaywindow", int4);
if (p)
    dw = Imath::Box2i(p->get<int>(0), p->get<int>(1),
                      p->get<int>(2), p->get<int>(3));

p = spec.find_attribute("temperature", TypeFloat);
if (p)
    float temperature = p->get<float>();

```

**TypeDesc getattributetype** (*string\_view* name, bool casesensitive = false) **const**

If the named attribute can be found in the *ImageSpec*, return its data type. If no such attribute exists, return *TypeUnknown*.

This was added in version 2.1.

bool **getattribute** (*string\_view* name, *TypeDesc* type, void \*value, bool casesensitive = false)

**const**

If the *ImageSpec* contains the named attribute and its type matches type, copy the attribute value into the memory pointed to by val (it is up to the caller to ensure there is enough space) and return true. If no such attribute is found, or if it doesn't match the type, return false and do not modify val.

## EXAMPLES:

```

ImageSpec spec;
...
// Retrieving an integer attribute:
int orientation = 0;
spec.getattribute ("orientation", TypeInt, &orientation);

// Retrieving a string attribute with a char*:
const char* compression = nullptr;
spec.getattribute ("compression", TypeString, &compression);

// Alternately, retrieving a string with a ustring:
ustring compression;
spec.getattribute ("compression", TypeString, &compression);

```

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

This was added in version 2.1.

int **get\_int\_attribute** (*string\_view* name, int defaultval = 0) **const**

Retrieve the named metadata attribute and return its value as an *int*. Any integer type will convert to *int* by truncation or expansion, string data will be parsed into an *int* if its contents consist of the text representation of one integer. Floating point data will not succeed in converting to an *int*. If no such metadata exists, or are of a type that cannot be converted, the defaultval will be returned.

float **get\_float\_attribute** (*string\_view* name, float defaultval = 0) **const**

Retrieve the named metadata attribute and return its value as a *float*. Any integer or floating point type will convert to *float* in the obvious way (like a C cast), and so will string metadata if its contents consist

of the text representation of one floating point value. If no such metadata exists, or are of a type that cannot be converted, the `defaultval` will be returned.

*string\_view* **get\_string\_attribute** (*string\_view* name, *string\_view* defaultval = *string\_view*())

**const**

Retrieve any metadata attribute, converted to a string. If no such metadata exists, the `defaultval` will be returned.

std::string **serialize** (SerialFormat format, SerialVerbose verbose = SerialDetailed) **const**

Returns, as a string, a serialized version of the *ImageSpec*. The format may be either *ImageSpec::SerialText* or *ImageSpec::SerialXML*. The verbose argument may be one of: *ImageSpec::SerialBrief* (just resolution and other vital statistics, one line for *SerialText*, *ImageSpec::SerialDetailed* (contains all metadata in original form), or *ImageSpec::SerialDetailedHuman* (contains all metadata, in many cases with human-readable explanation).

std::string **to\_xml** () **const**

Converts the contents of the *ImageSpec* as an XML string.

void **from\_xml** (const char \*xml)

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

std::pair<*string\_view*, int> **decode\_compression\_metadata** (*string\_view* defaultcomp = "", int defaultqual = -1) **const**

Hunt for the “Compression” and “CompressionQuality” settings in the spec and turn them into the compression name and quality. This handles compression name/qual combos of the form “name:quality”.

bool **valid\_tile\_range** (int xbegin, int xend, int ybegin, int yend, int zbegin, int zend)

Helper function to verify that the given pixel range exactly covers a set of tiles. Also returns false if the spec indicates that the image isn’t tiled at all.

*TypeDesc* **channelformat** (int chan) **const**

Return the channelformat of the given channel. This is safe even if channelformats is not filled out.

*string\_view* **channel\_name** (int chan) **const**

Return the channel name of the given channel. This is safe even if channelnames is not filled out.

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.)

int **channelindex** (*string\_view* name) **const**

Return the index of the channel with the given name, or -1 if no such channel is present in channelnames.

*ROI* **roi** () **const**

Return pixel data window for this *ImageSpec* expressed as a *ROI*.

*ROI* **roi\_full** () **const**

Return full/display window for this *ImageSpec* expressed as a *ROI*.

void **set\_roi** (const *ROI* &r)

Set pixel data window parameters (x, y, z, width, height, depth) for this *ImageSpec* from an *ROI*. Does NOT change the channels of the spec, regardless of r.

void **set\_roi\_full** (const *ROI* &r)

Set full/display window parameters (full\_x, full\_y, full\_z, full\_width, full\_height, full\_depth) for this *ImageSpec* from an *ROI*. Does NOT change the channels of the spec, regardless of r.

void **copy\_dimensions** (const *ImageSpec* &*other*)

Copy from *other* the image dimensions (x, y, z, width, height, depth, full\*, nchannels, format) and data types. It does *not* copy arbitrary named metadata or channel names (thus, for an *ImageSpec* with lots of metadata, it is much less expensive than copying the whole thing with `operator=()`).

bool **undefined**() const

Returns `true` for a newly initialized (undefined) *ImageSpec*. (Designated by no channels and undefined data type true of the uninitialized state of an *ImageSpec*, and presumably not for any *ImageSpec* that is useful or purposefully made.)

AttrDelegate<*ImageSpec*> **operator[]** (*string\_view* name)

Array indexing by string will create an AttrDelegate that enables a convenient shorthand for adding and retrieving values from the spec:

1. Assigning to the delegate adds a metadata attribute:

```
ImageSpec spec;
spec["foo"] = 42;                // int
spec["pi"] = float(M_PI);        // float
spec["oio:ColorSpace"] = "sRGB"; // string
spec["cameratoworld"] = Imath::Matrix44(...); // matrix
```

Be very careful, the attribute's type will be implied by the C++ type of what you assign.

2. String data may be retrieved directly, and for other types, the delegate supports a `get<T>()` that retrieves an item of type T:

```
std::string colorspace = spec["oio:ColorSpace"];
int dither = spec["oio:dither"].get<int>();
```

This was added in version 2.1.

## Public Static Functions

**static** void **auto\_stride** (stride\_t &*xstride*, stride\_t &*ystride*, stride\_t &*zstride*, stride\_t *channel-size*, int *nchannels*, int *width*, int *height*)

Adjust the stride values, if set to AutoStride, to be the right sizes for contiguous data with the given format, channels, width, height.

**static** void **auto\_stride** (stride\_t &*xstride*, stride\_t &*ystride*, stride\_t &*zstride*, *TypeDesc* *format*, int *nchannels*, int *width*, int *height*)

Adjust the stride values, if set to AutoStride, to be the right sizes for contiguous data with the given format, channels, width, height.

**static** void **auto\_stride** (stride\_t &*xstride*, *TypeDesc* *format*, int *nchannels*)

Adjust *xstride*, if set to AutoStride, to be the right size for contiguous data with the given format and channels.

**static** std::string **metadata\_val** (const ParamValue &*p*, bool *human* = false)

For a given parameter *p*, 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.

## 2.7 “Deep” pixel data: DeepData

### class DeepData

A *DeepData* holds the contents of an image of “deep” pixels (multiple depth samples per pixel).

#### Public Functions

##### DeepData ()

Construct an empty *DeepData*.

##### DeepData (const ImageSpec &spec)

Construct and init from an *ImageSpec*.

##### DeepData (const DeepData &d)

Copy constructor.

##### const DeepData &operator= (const DeepData &d)

Copy assignment.

##### void clear ()

Reset the *DeepData* to be equivalent to its empty initial state.

##### void free ()

In addition to performing the tasks of *clear()*, also ensure that all allocated memory has been truly freed.

##### void init (int64\_t npix, int nchan, cspan<TypeDesc> channeltypes, cspan<std::string> channelnames)

Initialize the *DeepData* with the specified number of pixels, channels, channel types, and channel names, and allocate memory for all the data.

##### void init (const ImageSpec &spec)

Initialize the *DeepData* based on the *ImageSpec*’s total number of pixels, number and types of channels. At this stage, all pixels are assumed to have 0 samples, and no sample data is allocated.

##### bool initialized() const

Is the *DeepData* initialized?

##### bool allocated() const

Has the *DeepData* fully allocated? If no, it is still very inexpensive to call *set\_capacity()*.

##### int64\_t pixels() const

Retrieve the total number of pixels.

##### int channels() const

Retrieve the number of channels.

##### string\_view channelname (int c) const

Return the name of channel *c*.

##### TypeDesc channeltype (int c) const

Retrieve the data type of channel *c*.

##### size\_t channelsize (int c) const

Return the size (in bytes) of one sample datum of channel *c*.

##### size\_t samplesize () const

Return the size (in bytes) for all channels of one sample.

**int samples** (int64\_t *pixel*) **const**  
Retrieve the number of samples for the given pixel index.

**void set\_samples** (int64\_t *pixel*, int *samps*)  
Set the number of samples for the given pixel. This must be called after *init()*.

**void set\_all\_samples** (cspan<unsigned int> *samples*)  
Set the number of samples for all pixels. The *samples.size()* is required to match *pixels()*.

**void set\_capacity** (int64\_t *pixel*, int *samps*)  
Set the capacity of samples for the given pixel. This must be called after *init()*.

**int capacity** (int64\_t *pixel*) **const**  
Retrieve the capacity (number of allocated samples) for the given pixel index.

**void insert\_samples** (int64\_t *pixel*, int *samplepos*, int *n* = 1)  
Insert *n* samples of the specified pixel, betinning at the sample position index. After insertion, the new samples will have uninitialized values.

**void erase\_samples** (int64\_t *pixel*, int *samplepos*, int *n* = 1)  
Erase *n* samples of the specified pixel, betinning at the sample position index.

**float deep\_value** (int64\_t *pixel*, int *channel*, int *sample*) **const**  
Retrieve the value of the given pixel, channel, and sample index, cast to a *float*.

**uint32\_t deep\_value\_uint** (int64\_t *pixel*, int *channel*, int *sample*) **const**  
Retrieve the value of the given pixel, channel, and sample index, cast to a *uint32*.

**void set\_deep\_value** (int64\_t *pixel*, int *channel*, int *sample*, float *value*)  
Set the value of the given pixel, channel, and sample index, for floating-point channels.

**void set\_deep\_value** (int64\_t *pixel*, int *channel*, int *sample*, uint32\_t *value*)  
Set the value of the given pixel, channel, and sample index, for integer channels.

**void \*data\_ptr** (int64\_t *pixel*, int *channel*, int *sample*)  
Retrieve the pointer to a given pixel/channel/sample, or NULL if there are no samples for that pixel. Use with care, and note that calls to *insert\_samples* and *erase\_samples* can invalidate pointers returned by prior calls to *data\_ptr*.

**void get\_pointers** (std::vector<void\*> &*pointers*) **const**  
Fill in the vector with pointers to the start of the first channel for each pixel.

**bool copy\_deep\_sample** (int64\_t *pixel*, int *sample*, **const** *DeepData* &*src*, int64\_t *srcpixel*, int *src-sample*)  
Copy a deep sample from *src* to this *DeepData*. They must have the same channel layout. Return *true* if ok, *false* if the operation could not be performed.

**bool copy\_deep\_pixel** (int64\_t *pixel*, **const** *DeepData* &*src*, int64\_t *srcpixel*)  
Copy an entire deep pixel from *src* to this *DeepData*, completely eplacing any pixel data for that pixel. They must have the same channel ayout. Return *true* if ok, *false* if the operation could not be erformed.

**bool split** (int64\_t *pixel*, float *depth*)  
Split all samples of that pixel at the given depth *zsplit*. Samples that span *z* (i.e. *z* < *zsplit* < *zback*) will be split into two samples with depth ranges [*z*,*zsplit*] and [*zsplit*,*zback*] with appropriate changes to their color and alpha values. Samples not spanning *zsplit* will remain intact. This operation will have no effect if there are not *Z* and *Zback* channels present. Return *true* if any splits occurred, *false* if the pixel was not modified.

void **sort** (int64\_t *pixel*)  
Sort the samples of the pixel by their Z depth.

void **merge\_overlaps** (int64\_t *pixel*)  
Merge any adjacent samples in the pixel that exactly overlap in z range. This is only useful if the pixel has previously been split at all sample starts and ends, and sorted by Z. Note that this may change the number of samples in the pixel.

void **merge\_deep\_pixels** (int64\_t *pixel*, const *DeepData* &*src*, int *srcpixel*)  
Merge the samples of *src*'s pixel into this *DeepData*'s pixel. Return `true` if ok, `false` if the operation could not be performed.

float **opaque\_z** (int64\_t *pixel*) const  
Return the z depth at which the pixel reaches full opacity.

void **occlusion\_cull** (int64\_t *pixel*)  
Remove any samples hidden behind opaque samples.

## 2.8 Global Attributes

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

bool `OIIO::attribute` (*string\_view* *name*, *TypeDesc* *type*, const void \**val*)  
`OIIO::attribute()` 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 `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:

- `string options`

This catch-all is simply a comma-separated list of `name=value` settings of named options, which will be parsed and individually set. For example,

```
OIIO::attribute ("options", "threads=4,log_times=1");
```

Note that if an option takes a string value that must itself contain a comma, it is permissible to enclose the value in either 'single' or "double" quotes.

- `int threads`

How many threads to use for operations that can be sped up by being multithreaded. (Examples: simultaneous format conversions of multiple scanlines read together, or many `ImageBufAlgo` operations.) The default is 0, meaning to use the full available hardware concurrency detected.



Situations where the main application logic is essentially single threaded (i.e., one top-level call into OIIO at a time) should leave this at the default value, or some reasonable number of cores, thus allowing lots of threads to fill the cores when OIIO has big tasks to complete. But situations where you have many threads at the application level, each of which is expected to be making separate OIIO calls simultaneously, should set this to 1, thus having each calling thread do its own work inside of OIIO rather than spawning new threads with a high overall “fan out.”

- `int exr_threads`

Sets the internal OpenEXR thread pool size. The default is to use as many threads as the amount of hardware concurrency detected. Note that this is separate from the OIIO “threads” attribute.

- `string plugin_searchpath`

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

- `int read_chunk`

When performing a `read_image()`, this is the number of scanlines it will attempt to read at a time (some formats are more efficient when reading and decoding multiple scanlines). The default is 256. The special value of 0 indicates that it should try to read the whole image if possible.

- `float[] missingcolor, string missingcolor`

This attribute may either be an array of float values, or a string containing a comma-separated list of the values. Setting this option globally is equivalent to always passing an *ImageInput* open-with-configuration hint “oiio:missingcolor” with the value.

When set, it gives some *ImageInput* readers the option of ignoring any *missing* tiles or scanlines in the file, and instead of treating the read failure of an individual tile as a full error, will interpret it as an intentionally missing tile and proceed by simply filling in the missing pixels with the color specified. If the first element is negative, it will use the absolute value, but draw alternating diagonal stripes of the color. For example,

```
float missing[4] = { -1.0, 0.0, 0.0, 0.0 }; // striped red
OIIO::attribute ("missingcolor", TypeDesc("float[4]"), &missing);
```

Note that only some file formats support files with missing tiles or scanlines, and this is only taken as a hint. Please see `chap-bundledplugins_` for details on which formats accept a “missingcolor” configuration hint.

- `int debug`

When nonzero, various debug messages may be printed. The default is 0 for release builds, 1 for DEBUG builds (values > 1 are for OIIO developers to print even more debugging information). This attribute but also may be overridden by the `OPENIMAGEIO_DEBUG` environment variable.

- `int tiff:half`

When nonzero, allows TIFF to write half pixel data. N.B. Most apps may not read these correctly, but OIIO will. That’s why the default is not to support it.

- `int log_times`

When the “log\_times” attribute is nonzero, `ImageBufAlgo` functions are instrumented to record the number of times they were called and the total amount of time spent executing them. It can be overridden by environment variable `OPENIMAGEIO_LOG_TIMES`. If the value of `log_times` is 2 or more when the application terminates, the timing report will be printed to `stdout` upon exit.

When enabled, there is a slight runtime performance cost due to checking the time at the start and end of each of those function calls, and the locking and recording of the data structure that holds the log information. When the `log_times` attribute is disabled, there is no additional performance cost.



The report of totals can be retrieved as the value of the "timing\_report" attribute, using `OIIO::get_attribute()` call.

`bool OIIO::attribute(string_view name, int val)`

`bool OIIO::attribute(string_view name, float val)`

`bool OIIO::attribute(string_view name, string_view val)`

Shortcuts for setting an attribute to a single int, float, or string.

`bool OIIO::getattribute(string_view name, TypeDesc type, void *val)`

Get the named global attribute of OpenImageIO, store it in `*val`. Return `true` if found and it was compatible with the type specified, otherwise return `false` and do not modify the contents of `*val`. It is up to the caller to ensure that `val` points to the right kind and size of storage for the given type.

In addition to being able to retrieve all the attributes that are documented as settable by the `OIIO::attribute()` call, `getattribute()` can also retrieve the following read-only attributes:

- `string format_list`
- `string input_format_list`
- `string output_format_list`

A comma-separated list of all the names of, respectively, all supported image formats, all formats accepted as inputs, and all formats accepted as outputs.

- `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. Semicolons separate the lists for formats. For example,

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

- `string library_list`

For each format that uses a dependent library, the format name, followed by a colon, followed by the name and version of the dependency. Semicolons separate the lists for formats. For example,

```
"tiff:LIBTIFF 4.0.4;gif:gif_lib 4.2.3;openexr:OpenEXR 2.2.0"
```

- `string "timing_report"` A string containing the report of all the log\_times.
- `string hw:simd`
- `string oiio:simd (read-only)`

A comma-separated list of hardware CPU features for SIMD (and some other things). The "oiio:simd" attribute is similarly a list of which features this build of OIIO was compiled to support.

This was added in OpenImageIO 1.8.

- `float resident_memory_used_MB`

This read-only attribute can be used for debugging purposes to report the approximate process memory used (resident) by the application, in MB.

- `string timing_report`

Retrieving this attribute returns the timing report generated by the `log_timing` attribute (if it was enabled). The report is sorted alphabetically and for each named instrumentation region, prints the number of times it executed, the total runtime, and the average per call, like this:

IBA::computePixelStats	2	2.69ms	(avg 1.34ms)
IBA::make_texture	1	74.05ms	(avg 74.05ms)
IBA::mul	8	2.42ms	(avg 0.30ms)
IBA::over	10	23.82ms	(avg 2.38ms)
IBA::resize	20	0.24s	(avg 12.18ms)
IBA::zero	8	0.66ms	(avg 0.08ms)

bool **getattribute** (string\_view *name*, int &*val*)

bool **getattribute** (string\_view *name*, float &*val*)

bool **getattribute** (string\_view *name*, char \*\**val*)

bool **getattribute** (string\_view *name*, std::string &*val*)

Specialized versions of `getattribute()` in which the data type is implied by the type of the argument (for single int, float, or string). Two string versions exist: one that retrieves it as a `std::string` and another that retrieves it as a `char *`. In all cases, the return value is `true` if the attribute is found and the requested data type conversion was legal.

EXAMPLES:

```
int threads;
OIIO::getattribute ("threads", &threads);
std::string path;
OIIO::getattribute ("plugin_searchpath", &path);
```

int **get\_int\_attribute** (string\_view *name*, int *defaultvalue* = 0)

float **get\_float\_attribute** (string\_view *name*, float *defaultvalue* = 0)

string\_view **get\_string\_attribute** (string\_view *name*, string\_view *defaultvalue* = "")

Specialized versions of `getattribute()` for common types, in which the data is returned directly, and a supplied default value is returned if the attribute was not found.

EXAMPLES:

```
int threads = OIIO::getattribute ("threads", 0);
string_view path = OIIO::getattribute ("plugin_searchpath");
```

## 2.9 Miscellaneous Utilities

These helper functions are not part of any other OpenImageIO class, they just exist in the OIIO namespace as general utilities. (See [Miscellaneous Utilities](#) for the corresponding Python bindings.)

int OIIO::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. One example of how this is useful is for plugins to query the version to be sure they are linked against an adequate version of the library.

std::string OIIO::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.

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

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

## 2.10 Environment variables

There are a few special environment variables that can be used to control OpenImageIO at times that it is not convenient to set options individually from inside the source code.

### OPENIMAGEIO\_OPTIONS

Allows you to seed the global OpenImageIO-wide options.

The value of the environment variable should be a comma-separated list of *name=value* settings. If a value is a string that itself needs to contain commas, it may be enclosed in single or double quotes.

Upon startup, the contents of this environment variable will be passed to a call to:

```
OIIO::attribute ("options", value);
```

### OPENIMAGEIO\_IMAGECACHE\_OPTIONS

Allows you to seed the options for any ImageCache created.

The value of the environment variable should be a comma-separated list of *name=value* settings. If a value is a string that itself needs to contain commas, it may be enclosed in single or double quotes.

Upon creation of any ImageCache, the contents of this environment variable will be passed to a call to:

```
imagecache->attribute ("options", value);
```

### OPENIMAGEIO\_TEXTURESYSTEM\_OPTIONS

Allows you to seed the options for any TextureSystem created.

The value of the environment variable should be a comma-separated list of *name=value* settings. If a value is a string that itself needs to contain commas, it may be enclosed in single or double quotes.

Upon creation of any TextureSystem, the contents of this environment variable will be passed to a call to:

```
texturesys->attribute ("options", value);
```



## 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>
using namespace OIIO;
...

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

std::unique_ptr<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 ();
```

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.

```
std::unique_ptr<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 x 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.

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

### What happens when the file format doesn't support the spec?

The `open()` call will fail (returning an empty pointer and set an appropriate error message) if the output format cannot accommodate what is requested by the `ImageSpec`. This includes:

- Dimensions (width, height, or number of channels) exceeding the limits supported by the file format.<sup>1</sup>
- Volumetric (depth > 1) if the format does not support volumetric data.
- Tile size >1 if the format does not support tiles.
- Multiple subimages or MIP levels if not supported by the format.

However, several other mismatches between requested `ImageSpec` and file format capabilities will be silently ignored, allowing `open()` to succeed:

- If the pixel data format is not supported (for example, a request for `half` pixels when writing a JPEG/JFIF file), the format writer may substitute another data format (generally, whichever commonly-used data format supported by the file type will result in the least reduction of precision or range).
- If the `ImageSpec` requests different per-channel data formats, but the format supports only a single format for all channels, it may just choose the most precise format requested and use it for all channels.
- If the file format does not support arbitrarily-named channels, the channel names may be lost when saving the file.
- Any other metadata in the `ImageSpec` may be summarily dropped if not supported by the file format.

## 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 *Image Output Made Simple* 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 `writescanline()` 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);
}
```

(continues on next page)

---

<sup>1</sup> One exception to the rule about number of channels is that a file format that supports only RGB, but not alpha, is permitted to silently drop the alpha channel without considering that to be an error.

(continued from previous page)

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

The first two arguments to `writescanline()` 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 [Data Strides](#)).

All `ImageOutput` implementations will accept scanlines in strict order (starting with scanline 0, then 1, up to `yres-1`, without skipping any). See Section [Random access and repeated transmission of pixels](#) for details on out-of-order or repeated scanlines.

The full description of the `writescanline()` function may be found in Section [ImageOutput Class Reference](#).

## Writing individual tiles

Not all image formats (and therefore not all `ImageOutput` implementations) support tiled images. If the format does not support tiles, then `writetile()` 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:

```
...
std::unique_ptr<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 `writetile()` 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 [Data Strides](#)).

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

The full description of the `writetile()` function may be found in Section [ImageOutput Class Reference](#).

## 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:

```
...
std::unique_ptr<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).

---

**Note:** OpenImageIO nearly always follows the C++ STL convention of specifying ranges as `[begin, end)`, that is, `begin`, `begin+1`, ..., `end-1`.

---

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 [Data Strides](#)).



### 3.2.2 Converting pixel data types

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 pixels be stored in any of several data types. 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 pixel values 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 pixel data type 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 pixel data types understood by the OpenImageIO library. Each `ImageOutput` plugin implementation should document which data formats it supports. An individual `ImageOutput` implementation is expected to always succeed, but if the file format does not support the requested pixel data type, it is expected to choose a data type that is supported, usually the data type that best preserves the precision and range of the originally-requested data type.

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);
```

Quantization limits 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 (0.0 - 1.0) range. Floating-point types do not attempt to remap values, and do not clamp (except to their full floating-point range).

It is not required that the pixel data passed to `writeimage()`, `writescanline()`, `writetile()`, or `write_rectangle()` actually be in the same data type as that requested as the native pixel data type 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 `writeimage()` performing the conversion automatically:

```
std::unique_ptr<ImageOutput> out = ImageOutput::create ("myfile.tif");
ImageSpec spec (xres, yres, channels, TypeDesc::UINT16);
```

(continues on next page)

(continued from previous page)

```

out->open (filename, spec);
...
float pixels [xres*yres*channels];
...
out->write_image (TypeDesc::FLOAT, pixels);

```

Note that `writescanline()`, `writetile()`, 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 `writetile()` 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 `writescanline()` function takes an optional `xstride` argument, and the `writeimage()`, `writetile()`, 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,

```

(continues on next page)

(continued from previous page)

```
(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*channelsize];

// File representation is 1 channel
std::unique_ptr<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 [ImageOutput Class Reference](#) 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 32x32 pixel crop window within an abstract 640 x 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 window, 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!
...
std::unique_ptr<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* that should be carried along with the image or saved in the file.

---

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

---

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.

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");
```

(continues on next page)

(continued from previous page)

```
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.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");
spec.alpha_channel = 3;
spec.z_channel = 4;
```

There are 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", 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` (see Section `Color information metadata` for explanations of particular values).

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 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:

```
std::unique_ptr<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"))
    ...
```

### 3.2.7 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:

```
std::unique_ptr<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 [ImageOutput Class Reference](#) 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
std::unique_ptr<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";
    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 ();

```

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
std::unique_ptr<ImageOutput> out = ImageOutput::create (filename);

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

```

(continues on next page)



(continued from previous page)

```

}
out->close ();

```

In both of these examples, we have used `writeimage()`, but of course `writescanline()`, `writetile()`, and `write_rectangle()` work as you would expect, on the current subimage.

### 3.2.8 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:

```

std::unique_ptr<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 *ImageOutput Class Reference* 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
std::unique_ptr<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";
    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,

```

(continues on next page)

(continued from previous page)

```

    // try to emulate it with subimages.
    if (out->supports("mipmap"))
        appendmode = ImageOutput::AppendMIPLevel;
    else
        appendmode = ImageOutput::AppendSubimage;
}
out->close ();

```

In this example, we have used `writeimage()`, but of course `writescanline()`, `writetile()`, and `write_rectangle()` work as you would expect, on the current MIP level.

### 3.2.9 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 `TypeUnknown`.

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];

std::unique_ptr<ImageOutput> out = ImageOutput::create ("foo.exr");

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

```

### 3.2.10 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, described in detail in Section “*Deep*” pixel data: *DeepData*.

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 (spec);
for (int y = 0; y < yres; ++y)
    for (int x = 0; x < xres; ++x)
        deepdata.set_samples(y*xres+x, ...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)
            for (int samp = 0; samp < deepdata.samples(pixel); ++samp)
                deepdata.set_deep_value (pixel, chan, samp, ...value...);

// Create the output
std::unique_ptr<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;
```

(continues on next page)

(continued from previous page)

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

### 3.2.11 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";
std::unique_ptr<ImageInput> in = ImageInput::open (input);

// 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";
std::unique_ptr<ImageOutput> out = ImageOutput::create (output);
out->open (output, out_spec);
out->copy_image (in);

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

### 3.2.12 Custom I/O proxies (and writing the file to a memory buffer)

Some file format writers allow you to supply a custom I/O proxy object that can allow bypassing the usual file I/O with custom behavior, including the ability to fill an in-memory buffer with a byte-for-byte representation of the correctly formatted file that would have been written to disk.

Only some output format writers support this feature. To find out if a particular file format supports this feature, you can create an `ImageOutput` of the right type, and check if it supports the feature name "iopproxy":

```
auto out = ImageOutput::create (filename);
if (! out || ! out->supports ("iopproxy")) {
    return;
}
```

`ImageOutput` writers that support "iopproxy" will respond to a special attribute, "oiio:iopproxy", which passes a pointer to a `Filesystem::IOProxy*` (see OpenImageIO's `filesystem.h` for this type and its subclasses). `IOProxy` is an abstract type, and concrete subclasses include `IOFile` (which wraps I/O to an open `FILE*`) and `IOVecOutput` (which sends output to a `std::vector<unsigned char>`).

Here is an example of using a proxy that writes the "file" to a `std::vector<unsigned char>`:

```
// ImageSpec describing the image we want to write.
ImageSpec spec (xres, yres, channels, TypeDesc::UINT8);

std::vector<unsigned char> file_buffer; // bytes will go here
Filesystem::IOVecOutput vecout (file_buffer); // I/O proxy object
void *ptr = &vecout;
spec.attribute ("oiio:iopproxy", TypeDesc::PTR, &ptr);

auto out = ImageOutput::create ("out.exr");
out->open ("out.exr", spec);
out->write_image (...);
out->close ();

// At this point, file_buffer will contain the "file"
```

### 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";
std::unique_ptr<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 *Image Output Made Simple*, but this time it is fully elaborated with error checking and reporting:

```
#include <OpenImageIO/imageio.h>
using namespace OIIO;
...

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

std::unique_ptr<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";
    return;
}

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

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

### 3.3 ImageOutput Class Reference

#### class ImageOutput

*ImageOutput* abstracts the writing of an image file in a file format-agnostic manner.

Users don't directly declare these. Instead, you call the *create()* static method, which will return a `unique_ptr` holding a subclass of *ImageOutput* that implements writing the particular format.

#### Creating an ImageOutput

```
static unique_ptr 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 name.

**Return** A `unique_ptr` that will close and free the *ImageOutput* when it exits scope or is reset. The pointer will be empty if the required writer was not able to be created.

#### Parameters

- `filename`: The name of the file format (e.g., "openexr"), a file extension (e.g., "exr"), or a filename from which the file format can be inferred from its extension (e.g., "hawaii.exr").
- `plugin_searchpath`: An optional colon-separated list of directories to search for OpenImageIO plugin DSO/DLL's.

#### Opening and closing files for output

#### enum OpenMode

Modes passed to the *open()* call.

*Values:*

**Create**

**AppendSubimage**

**AppendMIPLevel**

```
virtual int supports(string_view feature) const
```

Given the name of a "feature", return whether this *ImageOutput* supports output of images with the given properties. Most queries will simply return 0 for "doesn't support" and 1 for "supports it," but it is acceptable to have queries return other nonzero integers to indicate varying degrees of support or limits (but should be clearly documented as such).

Feature names that *ImageOutput* implementations are expected to recognize include:

- "tiles" : Is this format writer able to write tiled images?
- "rectangles" : Does this writer accept arbitrary rectangular pixel regions (via *write\_rectangle()*)? Returning 0 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 (0 indicates that they *must* be in successive order)?

- "multiimage" : Does this format support multiple subimages within a file?
- "appendsubimage" : Does this format support multiple subimages that can be successively appended at will via `open(name, spec, AppendSubimage)`? A value of 0 means that the format requires pre-declaring the number and specifications of the subimages when the file is first opened, with `open(name, subimages, specs)`.
- "mipmap" : Does this format support multiple resolutions for an image/subimage?
- "volumes" : Does this format support "3D" pixel arrays (a.k.a. volume images)?
- "alpha" : Can this format support an alpha channel?
- "nchannels" : Can this format support arbitrary number of channels (beyond RGBA)?
- "rewrite" : May the same scanline or tile be sent more than once? Generally, this is true for plugins that implement 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 format support display ("full") windows distinct from the pixel data window?
- "origin" : Does the image format support specifying a pixel window origin (i.e., nonzero *ImageSpec* `x, y, z`)?
- "negativeorigin" : Does the image format allow data and display window origins (i.e., *ImageSpec* `x, 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)?
- "arbitrary\_metadata" : Does the image file format allow metadata with arbitrary names (and either arbitrary, or a reasonable set of, data types)? (Versus the file format supporting only a fixed list of specific metadata names/values.)
- "exif" Does the image file format support Exif camera data (either specifically, or via arbitrary named metadata)?
- "iptc" Does the image file format support IPTC data (either specifically, or via arbitrary named metadata)?
- "iopproxy" Does the image file format support writing to an `IOProxy`?

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.

**virtual bool open** (**const** std::string &name, **const** *ImageSpec* &newspec, *OpenMode* mode = *Create*) = 0

Open the file with given name, with resolution and other format data as given in newspec. 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.



**Return** `true` upon success, or `false` upon failure.

#### Parameters

- `name`: The name of the image file to open.
- `newspec`: The *ImageSpec* describing the resolution, data types, etc.
- `mode`: Specifies whether the purpose of the `open` is to create/truncate the file (default: `Create`), append another subimage (`AppendSubimage`), or append another MIP level (`AppendMIPLevel`).

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

Open a multi-subimage file with given name and specifications for each of the subimages. 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 `fmust` 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.

**Return** `true` upon success, or `false` upon failure.

#### Parameters

- `name`: The name of the image file to open.
- `subimages`: The number of subimages (and therefore the length of the `specs[]` array.
- `specs[]`: Pointer to an array of *ImageSpec* objects describing each of the expected subimages.

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

Return 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()`.

**virtual bool close** () = 0

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

## Writing pixels

Common features of all the write methods:

- The `format` parameter describes the data type of the `data[]`. The write methods automatically convert the data from the specified `format` to the actual output data type of the file (as was specified by the *ImageSpec* passed to `open()`). If `format` is `TypeUnknown`, then rather than converting from `format`, it will just copy pixels assumed to already be in the file's native data layout (including, possibly, per-channel data formats as specified by the *ImageSpec*'s `channelformats` field).
- The `stride` values describe the layout of the data buffer: `xstride` is the distance in bytes between successive pixels within each scanline. `ystride` is the distance in bytes between successive scanlines. For volumetric images `zstride` is the distance in bytes between successive "volumetric planes". Strides set to the special value `AutoStride` imply contiguous data, i.e.,

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

- Any *range* parameters (such as *ybegin* and *yend*) describe a “half open interval”, meaning that *begin* is the first item and *end* is *one past the last item*. That means that the number of items is  $\text{end} - \text{begin}$ .
- For ordinary 2D (non-volumetric) images, any *z* or *zbegin* coordinates should be 0 and any *zend* should be 1, indicating that only a single image “plane” exists.
- Scanlines or tiles must be written in successive increasing coordinate order, unless the particular output file driver allows random access (indicated by `supports("random_access")`).
- All write functions return `true` for success, `false` for failure (after which a call to `geterror()` may retrieve a specific error message).

**virtual bool write\_scanline** (int *y*, int *z*, *TypeDesc* *format*, **const** void *\*data*, stride\_t *xstride* = `AutoStride`)

Write the full scanline that includes pixels (*y*,*z*). For 2D non-volume images, *z* should be 0. The *xstride* value gives the distance between successive pixels (in bytes). Strides set to `AutoStride` imply “contiguous” data.

**Return** `true` upon success, or `false` upon failure.

#### Parameters

- *y/z*: The *y* & *z* coordinates of the scanline.
- *format*: A *TypeDesc* describing the type of data.
- *data*: Pointer to the pixel data.
- *xstride*: The distance in bytes between successive pixels in data (or `AutoStride`).

**virtual bool write\_scanlines** (int *ybegin*, int *yend*, int *z*, *TypeDesc* *format*, **const** void *\*data*, stride\_t *xstride* = `AutoStride`, stride\_t *ystride* = `AutoStride`)

Write multiple scanlines that include pixels (*y*,*z*) for all  $ybegin \leq y < yend$ , from data. This is analogous to `write_scanline(y, z, format, data, xstride)` repeatedly for each of the scanlines in turn (the advantage, though, is that some image file types may be able to write multiple scanlines more efficiently or in parallel, than it could with one scanline at a time).

**Return** `true` upon success, or `false` upon failure.

#### Parameters

- *ybegin/yend*: The *y* range of the scanlines being passed.
- *z*: The *z* coordinate of the scanline.
- *format*: A *TypeDesc* describing the type of data.
- *data*: Pointer to the pixel data.
- *xstride/ystride*: The distance in bytes between successive pixels and scanlines (or `AutoStride`).

**virtual 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. The three stride values give the distance (in bytes) between successive pixels, scanlines, and volumetric slices, respectively. Strides set to `AutoStride` imply ‘contiguous’ data in the shape of a full tile, i.e.,

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

**Return** true upon success, or false upon failure.

**Note** This call will fail if the image is not tiled, or if (x,y,z) is not the upper left corner coordinates of a tile.

#### Parameters

- x/y/z: The upper left coordinate of the tile being passed.
- format: A *TypeDesc* describing the type of data.
- data: Pointer to the pixel data.
- xstride/ystride/zstride: The distance in bytes between successive pixels, scanlines, and image planes (or *AutoStride* to indicate a “contiguous” single tile).

**virtual 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 block of multiple tiles that include all pixels in

```
[xbegin,xend) X [ybegin,yend) X [zbegin,zend)
```

This is analogous to calling `write_tile(x, y, z, ...)` for each tile in turn (but for some file formats, passing multiple tiles may allow it to write more efficiently or in parallel).

The begin/end pairs must correctly delineate tile boundaries, with the exception that it may also be the end of the image data if the image resolution is not a whole multiple of the tile size. The stride values give the data spacing of adjacent pixels, scanlines, and volumetric slices (measured in bytes). Strides set to *AutoStride* imply contiguous data in the shape of the [begin,end) region, i.e.,

```
xstride = format.size() * spec.nchannels
ystride = xstride * (xend-xbegin)
zstride = ystride * (yend-ybegin)
```

**Return** true upon success, or false upon failure.

**Note** 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.

#### Parameters

- xbegin/xend: The x range of the pixels covered by the group of tiles passed.
- ybegin/yend: The y range of the pixels covered by the tiles.
- zbegin/zend: The z range of the pixels covered by the tiles (for a 2D image, zbegin=0 and zend=1).
- format: A *TypeDesc* describing the type of data.
- data: Pointer to the pixel data.
- xstride/ystride/zstride: The distance in bytes between successive pixels, scanlines, and image planes (or *AutoStride*).

```
virtual 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 a rectangle of pixels given by the range

```
[xbegin,xend) X [ybegin,yend) X [zbegin,zend)
```

The stride values give the data spacing of adjacent pixels, scanlines, and volumetric slices (measured in bytes). Strides set to AutoStride imply contiguous data in the shape of the [*begin*,*end*) region, i.e.,

```
xstride = format.size() * spec.nchannels
ystride = xstride * (xend-xbegin)
zstride = ystride * (yend-ybegin)
```

**Return** true upon success, or false upon failure.

**Note** The call will fail for a format plugin that does not return true for supports("rectangles").

#### Parameters

- *xbegin*/*xend*: The x range of the pixels being passed.
- *ybegin*/*yend*: The y range of the pixels being passed.
- *zbegin*/*zend*: The z range of the pixels being passed (for a 2D image, *zbegin*=0 and *zend*=1).
- *format*: A *TypeDesc* describing the type of data.
- *data*: Pointer to the pixel data.
- *xstride*/*ystride*/*zstride*: The distance in bytes between successive pixels, scanlines, and image planes (or AutoStride).

```
virtual bool write_image (TypeDesc format, const void *data, stride_t xstride = AutoStride, stride_t ystride = AutoStride, stride_t zstride = AutoStride, Progress-Callback progress_callback = nullptr, void *progress_callback_data = nullptr)
```

Write the entire image of *spec.width* x *spec.height* x *spec.depth* pixels, from a buffer with the given strides and in the desired format.

Depending on the spec, this will write either all tiles or all scanlines. Assume that data points to a layout in row-major order.

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.

**Return** true upon success, or false upon failure.

#### Parameters

- *format*: A *TypeDesc* describing the type of data.
- *data*: Pointer to the pixel data.
- *xstride*/*ystride*/*zstride*: The distance in bytes between successive pixels, scanlines, and image planes (or AutoStride).
- *progress\_callback*/*progress\_callback\_data*: Optional progress callback.

**virtual bool write\_deep\_scanlines** (int *ybegin*, int *yend*, int *z*, const *DeepData* &*deepdata*)

Write deep scanlines containing pixels (\*,y,z), for all y in the range [ybegin,yend), to a deep file. This will fail if it is not a deep file.

**Return** true upon success, or false upon failure.

#### Parameters

- *ybegin/yend*: The y range of the scanlines being passed.
- *z*: The z coordinate of the scanline.
- *deepdata*: A *DeepData* object with the data for these scanlines.

**virtual bool write\_deep\_tiles** (int *xbegin*, int *xend*, int *ybegin*, int *yend*, int *zbegin*, int *zend*, const *DeepData* &*deepdata*)

Write the block of deep tiles that include all pixels in the range

[ <i>xbegin</i> , <i>xend</i> ) X [ <i>ybegin</i> , <i>yend</i> ) X [ <i>zbegin</i> , <i>zend</i> )
---

The begin/end pairs must correctly delineate tile boundaries, with the exception that it may also be the end of the image data if the image resolution is not a whole multiple of the tile size.

**Return** true upon success, or false upon failure.

**Note** 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.

#### Parameters

- *xbegin/xend*: The x range of the pixels covered by the group of tiles passed.
- *ybegin/yend*: The y range of the pixels covered by the tiles.
- *zbegin/zend*: The z range of the pixels covered by the tiles (for a 2D image, *zbegin*=0 and *zend*=1).
- *deepdata*: A *DeepData* object with the data for the tiles.

**virtual bool write\_deep\_image** (const *DeepData* &*deepdata*)

Write the entire deep image described by *deepdata*. Depending on the spec, this will write either all tiles or all scanlines.

**Return** true upon success, or false upon failure.

#### Parameters

- *deepdata*: A *DeepData* object with the data for the image.

## Public Types

**using unique\_ptr** = std::unique\_ptr<*ImageOutput*>  
unique\_ptr to an *ImageOutput*.

**typedef *ImageOutput* \*(\*Creator)()**

Call signature of a function that creates and returns an *ImageOutput*\*.

## Public Functions

**virtual const** char \***format\_name** (void) **const** = 0

Return the name of the format implemented by this class.

**virtual 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.

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 are the same file type and the same pixel data type. 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.

**Return** true upon success, or false upon failure.

### Parameters

- in: A pointer to the open *ImageInput* to read from.

std::string **geterror** () **const**

If any of the API routines returned false indicating an error, this method will return the error string (and clear any error flags). If no error has occurred since the last time `geterror()` was called, it will return an empty string.

template<typename ...Args>

void **error** (const char \*fmt, const Args&... args) **const**

Error reporting for the plugin implementation: call this with `Strutil::format`-like arguments. Use with caution! Some day this will change to be `fmt`-like rather than `printf`-like.

template<typename ...Args>

void **errorf** (const char \*fmt, const Args&... args) **const**

Error reporting for the plugin implementation: call this with `printf`-like arguments.

template<typename ...Args>

void **fmterror** (const char \*fmt, const Args&... args) **const**

Error reporting for the plugin implementation: call this with `fmt::format`-like arguments.

void **threads** (int n)

Set the threading policy for this *ImageOutput*, controlling the maximum amount of parallelizing thread “fan-out” that might occur during large write operations. The default of 0 means that the global attribute (“threads”) value should be used (which itself defaults to using as many threads as cores; see Section Global Attributes\_).

The main reason to change this value is to set it to 1 to indicate that the calling thread should do all the work rather than spawning new threads. That is probably the desired behavior in situations where the calling application has already spawned multiple worker threads.

int **threads** () **const**

Retrieve the current thread-spawning policy.

See `threads(int)`

## IMAGEINPUT: 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>
using namespace OIIO;
...

auto 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 ();
```

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. The `open()` method returns a `std::unique_ptr<ImageInput>` that will be automatically freed when it exits scope.

```
auto 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.

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

- When `in` exits its scope, the `ImageInput` will automatically be destroyed and any resources used by the plugin will be released.

## 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 Image Input Made Simple 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:

```
...
auto 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 Data Strides).



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 <= 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 ImageInput Class Reference.

## 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()`.

```
...
auto 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 Data Strides).

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 ImageInput Class Reference.

## 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.

```
std::unique_ptr<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  $x-1$  immediately following last channel of pixel  $x-1$  of the same row;
- for whole images or tiles, the data for each row immediately follows the previous one in memory (the first pixel of row  $y$  immediately follows the last column of row  $y-1$ );
- for 3D volumetric images, the first pixel of slice  $z$  immediately follows the last pixel of of slice  $z-1$ .

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

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

By passing different stride values, you can 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]);
...
```

(continues on next page)

(continued from previous page)

```
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:

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

Please consult Section ImageInput Class Reference 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 std::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:

```
auto 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 `nullptr` 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:

```
auto in = ImageInput::open (filename);
const ImageSpec &spec = in->spec();
...
ParamValue *p = spec.find_attribute ("Orientation", 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() == TypeString)
```

(continues on next page)

(continued from previous page)

```

    printf ("\%s\\", *(const char **)p.data());
else if (p.type() == TypeFloat)
    printf ("%g", *(const float *)p.data());
else if (p.type() == TypeInt)
    printf ("%d", *(const int *)p.data());
else if (p.type() == TypeDesc::UINT)
    printf ("%u", *(const unsigned int *)p.data());
else if (p.type() == 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\_attribs field may store metadata that reveals the color space the image file in the "oiio:ColorSpace" attribute (see Section Color information metadata for explanations of particular values).

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:

```

auto in = ImageInput::open (filename);
...
int subimage = 1;
int miplevel = 0;
if (in->seek_subimage (subimage, miplevel)) {
    ...
}

```

(continues on next page)

(continued from previous page)

```

} 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:

```

auto 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 ();

```

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 `TypeUnknown` 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`:

```

auto 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 subimage, int miplevel,
                                int ybegin, int yend, int z,
                                int chbegin, int chend,
                                DeepData &deepdata);

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

bool read_native_deep_image (int subimage, int miplevel,
                             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, described in detail in Section Reading “deep” data.

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

```

auto in = ImageInput::open (filename);
if (! in)
    return;
const ImageSpec &spec = in->spec();
if (spec.deep) {
    DeepData deepdata;
    in->read_native_deep_image (0, 0, deepdata);
    int p = 0; // absolute pixel number
    for (int y = 0; y < spec.height; ++y) {
        for (int x = 0; x < spec.width; ++x, ++p) {
            std::cout << "Pixel " << x << ", " << y << ":\n";

```

(continues on next page)

(continued from previous page)

```

    if (deepdata.samples(p) == 0)
        std::cout << "  no samples\n";
    else
        for (int c = 0; c < spec.nchannels; ++c) {
            TypeDesc type = deepdata.channeltype(c);
            std::cout << "    " << spec.channelnames[c] << ": ";
            void *ptr = deepdata.pointers[p*spec.nchannels+c]
            for (int s = 0; s < deepdata.samples(p); ++s) {
                if (type.basetype == TypeDesc::FLOAT ||
                    type.basetype == TypeDesc::HALF)
                    std::cout << deepdata.deep_value(p, c, s) << ' ';
                else if (type.basetype == TypeDesc::UINT32)
                    std::cout << deepdata.deep_value_uint(p, c, s) << ' ';
            }
            std::cout << "\n";
        }
    }
}
}
in->close ();

```

#### 4.2.8 Custom I/O proxies (and reading the file from a memory buffer)

Some file format readers allow you to supply a custom I/O proxy object that can allow bypassing the usual file I/O with custom behavior, including the ability to read the file from an in-memory buffer rather than reading from disk.

Only some input format readers support this feature. To find out if a particular file format supports this feature, you can create an ImageInput of the right type, and check if it supports the feature name "iopproxy":

```

auto in = ImageInput::create(filename);
if (!in || !in->supports("iopproxy")) {
    return;
}

```

ImageInput readers that support "iopproxy" will respond to a special attribute, "oiio:iopproxy", which passes a pointer to a Filesystem::IOProxy\* (see OpenImageIO's filesystem.h for this type and its subclasses). IOProxy is an abstract type, and concrete subclasses include IOFile (which wraps I/O to an open FILE\*) and IOMemReader (which reads input from a block of memory).

Here is an example of using a proxy that reads the "file" from a memory buffer:

```

const void *buf = ...;    // pointer to memory block
size_t size = ...;        // length of memory block

ImageSpec config; // ImageSpec describing input configuration options
Filesystem::IOMemReader memreader (buf, size); // I/O proxy object
void *ptr = &memreader;
config.attribute ("oiio:iopproxy", TypeDesc::PTR, &ptr);

auto in = ImageInput::open ("in.exr", &config);
in->read_image (...);
in->close();

// That will have read the "file" from the memory buffer

```



## 4.2.9 Custom search paths for plugins

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

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

## 4.2.10 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 an empty pointer 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 Image Input Made Simple, but this time it is fully elaborated with error checking and reporting:

```
#include <OpenImageIO/imageio.h>
using namespace OIIO;
...

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

auto in = ImageInput::open (filename);
if (! in) {
    std::cerr << "Could not open " << filename
               << ", error = " << OIIO::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[0])) {
    std::cerr << "Could not read pixels from " << filename
               << ", error = " << in->geterror() << "\n";
    return;
}

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

(continues on next page)

(continued from previous page)

```

return;
}

```

## 4.3 ImageInput Class Reference

### class ImageInput

*ImageInput* abstracts the reading of an image file in a file format-agnostic manner.

#### Creating an ImageInput

**static unique\_ptr open** (const std::string &filename, const *ImageSpec* \*config = nullptr)

Create an *ImageInput* subclass instance that is able to read the given file and open it, returning a unique\_ptr to the *ImageInput* if successful. The unique\_ptr is set up with an appropriate deleter so the *ImageInput* will be properly closed and deleted when the unique\_ptr goes out of scope or is reset. If the open fails, return an empty unique\_ptr and set an error that can be retrieved by `OIOO::geterror()`.

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

`open()` 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.

**Return** A unique\_ptr that will close and free the *ImageInput* when it exits scope or is reset. The pointer will be empty if the required writer was not able to be created.

#### Parameters

- filename: The name of the file to open.
- config: Optional pointer to an *ImageSpec* whose metadata contains "configuration hints."

**static unique\_ptr create** (const std::string &filename, bool do\_open = false, const *ImageSpec* \*config = nullptr, string\_view plugin\_searchpath = "")

Create and return an *ImageInput* implementation that is able to read the given file. If do\_open is true, fully open it if possible (using the optional config configuration spec, if supplied), otherwise just create the *ImageInput* but don't open it. The plugin\_searchpath parameter is an override of the searchpath. colon-separated list of directories to search for ImageIO plugin DSO/DLL's (not a searchpath for the image itself!). This will actually just try every imageio plugin it can locate, until it finds one that's able to open the file without error.

If the caller intends to immediately open the file, then it is often simpler to call static *ImageInput::open()*.

**Return** A unique\_ptr that will close and free the *ImageInput* when it exits scope or is reset. The pointer will be empty if the required writer was not able to be created.

#### Parameters

- filename: The name of the file to open.
- do\_open: If true, not only create but also open the file.

- `config`: Optional pointer to an *ImageSpec* whose metadata contains “configuration hints” for the *ImageInput* implementation.
- `plugin_searchpath`: An optional colon-separated list of directories to search for OpenImageIO plugin DSO/DLL’s.

## Reading pixels

Common features of all the read methods:

- The `format` parameter describes the data type of the `data[]` buffer. The read methods automatically convert the data from the data type it is stored in the file into the format of the data buffer. If `format` is `TypeUnknown` it will just copy pixels of file’s native data layout (including, possibly, per-channel data formats as specified by the *ImageSpec*’s `channelfomats` field).
- The `stride` values describe the layout of the data buffer: `xstride` is the distance in bytes between successive pixels within each scanline. `ystride` is the distance in bytes between successive scanlines. For volumetric images `zstride` is the distance in bytes between successive “volumetric planes”. Strides set to the special value `AutoStride` imply contiguous data, i.e.,

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

- Any *range* parameters (such as `ybegin` and `yend`) describe a “half open interval”, meaning that `begin` is the first item and `end` is *one past the last item*. That means that the number of items is `end - begin`.
- For ordinary 2D (non-volumetric) images, any `z` or `zbegin` coordinates should be 0 and any `zend` should be 1, indicating that only a single image “plane” exists.
- Some read methods take a channel range [`chbegin`,`chend`) to allow reading of a contiguous subset of channels (`chbegin`=0, `chend`=`spec.nchannels` reads all channels).
- *ImageInput* readers are expected to give the appearance of random access in other words, if it can’t randomly seek to the given scanline or tile, it should transparently close, reopen, and sequentially read through prior scanlines.
- All read functions return `true` for success, `false` for failure (after which a call to `geterror()` may retrieve a specific error message).

**virtual bool read\_scanline** (int `y`, int `z`, *TypeDesc* `format`, void `*data`, stride\_t `xstride` = `AutoStride`)

Read the scanline that includes pixels (`*y,z`) from the “current” subimage and MIP level. The `xstride` value gives the distance between successive pixels (in bytes). Strides set to `AutoStride` imply “contiguous” data.

**Note** This variety of `read_scanline` is not re-entrant nor thread-safe. If you require concurrent reads to the same open *ImageInput*, you should use `read_scanlines` that has the subimage and miplevel passed explicitly.

**Return** `true` upon success, or `false` upon failure.

### Parameters

- `y/z`: The `y` & `z` coordinates of the scanline. For 2D images, `z` should be 0.
- `format`: A *TypeDesc* describing the type of data.
- `data`: Pointer to the pixel data buffer.

- `xstride`: The distance in bytes between successive pixels in data (or `AutoStride`).

bool **read\_scanline** (int y, int z, float \*data)  
Simple `read_scanline` reads into contiguous float pixels.

virtual bool **read\_scanlines** (int subimage, int miplevel, int ybegin, int yend, int z, int chbegin, int chend, *TypeDesc* format, void \*data, stride\_t xstride = `AutoStride`, stride\_t ystride = `AutoStride`)

Read multiple scanlines that include pixels (\*,y,z) for all ybegin <= y < yend in the specified subimage and mip level, into data, using the strides given and converting to the requested data format (`TypeUnknown` indicates no conversion, just copy native data types). Only channels [chbegin, chend) will be read/copied (chbegin=0, chend=spec.nchannels reads all channels, yielding equivalent behavior to the simpler variant of `read_scanlines`).

This version of `read_scanlines`, because it passes explicit subimage/miplevel, does not require a separate call to `seek_subimage`, and is guaranteed to be thread-safe against other concurrent calls to any of the `read_*` methods that take an explicit subimage/miplevel (but not against any other *ImageInput* methods).

**Return** true upon success, or false upon failure.

**Note** This call was changed for OpenImageIO 2.0 to include the explicit subimage and miplevel parameters. The previous versions, which lacked subimage and miplevel parameters (thus were dependent on a prior call to `seek_subimage`) are considered deprecated.

#### Parameters

- subimage: The subimage to read from (starting with 0).
- miplevel: The MIP level to read (0 is the highest resolution level).
- ybegin/yend: The y range of the scanlines being passed.
- z: The z coordinate of the scanline.
- chbegin/chend: The channel range to read.
- format: A *TypeDesc* describing the type of data.
- data: Pointer to the pixel data.
- xstride/ystride: The distance in bytes between successive pixels and scanlines (or `AutoStride`).

virtual 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[], converting if necessary from the native data format of the file into the format specified. The stride values give the data spacing of adjacent pixels, scanlines, and volumetric slices (measured in bytes). Strides set to `AutoStride` imply ‘contiguous’ data in the shape of a full tile, i.e.,

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

**Note** This variety of `read_tile` is not re-entrant nor thread-safe. If you require concurrent reads to the same open *ImageInput*, you should use `read_tiles()` that has the subimage and miplevel passed explicitly.

**Return** true upon success, or false upon failure.

**Note** This call will fail if the image is not tiled, or if (x,y,z) is not the upper left corner coordinates of a tile.

#### Parameters

- x/y/z: The upper left coordinate of the tile being passed.
- format: A *TypeDesc* describing the type of data.
- data: Pointer to the pixel data.
- xstride/ystride/zstride: The distance in bytes between successive pixels, scanlines, and image planes (or *AutoStride* to indicate a “contiguous” single tile).

bool **read\_tile**(int x, int y, int z, float \*data)  
Simple read\_tile reads into contiguous float pixels.

**virtual** bool **read\_tiles**(int subimage, int miplevel, 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 block of multiple tiles that include all pixels in

```
[xbegin,xend) X [ybegin,yend) X [zbegin,zend)
```

This is analogous to calling `read_tile(x, y, z, ...)` for each tile in turn (but for some file formats, reading multiple tiles may allow it to read more efficiently or in parallel).

The begin/end pairs must correctly delineate tile boundaries, with the exception that it may also be the end of the image data if the image resolution is not a whole multiple of the tile size. The stride values give the data spacing of adjacent pixels, scanlines, and volumetric slices (measured in bytes). Strides set to *AutoStride* imply contiguous data in the shape of the [begin,end) region, i.e.,

```
xstride = format.size() * spec.nchannels
ystride = xstride * (xend-xbegin)
zstride = ystride * (yend-ybegin)
```

This version of `read_tiles`, because it passes explicit subimage and miplevel, does not require a separate call to `seek_subimage`, and is guaranteed to be thread-safe against other concurrent calls to any of the `read_*` methods that take an explicit subimage/miplevel (but not against any other *ImageInput* methods).

**Return** true upon success, or false upon failure.

**Note** 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.

#### Parameters

- subimage: The subimage to read from (starting with 0).
- miplevel: The MIP level to read (0 is the highest resolution level).
- xbegin/xend: The x range of the pixels covered by the group of tiles being read.
- ybegin/yend: The y range of the pixels covered by the tiles.
- zbegin/zend: The z range of the pixels covered by the tiles (for a 2D image, zbegin=0 and zend=1).
- chbegin/chend: The channel range to read.
- format: A *TypeDesc* describing the type of data.

- `data`: Pointer to the pixel data.
- `xstride/ystride/zstride`: The distance in bytes between successive pixels, scanlines, and image planes (or `AutoStride`).

**virtual bool read\_image** (int *subimage*, int *miplevel*, int *chbegin*, int *chend*, *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` x `spec.height` x `spec.depth` pixels into a buffer with the given strides and in the desired data format.

Depending on the spec, this will read either all tiles or all scanlines. Assume that data points to a layout in row-major order.

This version of `read_image`, because it passes explicit subimage and miplevel, does not require a separate call to `seek_subimage`, and is guaranteed to be thread-safe against other concurrent calls to any of the `read_*` methods that take an explicit subimage/miplevel (but not against any other *ImageInput* methods).

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.

**Return** `true` upon success, or `false` upon failure.

#### Parameters

- `subimage`: The subimage to read from (starting with 0).
- `miplevel`: The MIP level to read (0 is the highest resolution level).
- `chbegin/chend`: The channel range to read.
- `format`: A *TypeDesc* describing the type of data.
- `data`: Pointer to the pixel data.
- `xstride/ystride/zstride`: The distance in bytes between successive pixels, scanlines, and image planes (or `AutoStride`).
- `progress_callback/progress_callback_data`: Optional progress callback.

**virtual bool read\_native\_deep\_scanlines** (int *subimage*, int *miplevel*, int *ybegin*, int *yend*, int *z*, int *chbegin*, int *chend*, *DeepData* &*deepdata*)

Read deep scanlines containing pixels (\*,y,z), for all y in the range [ybegin,yend) into *deepdata*. This will fail if it is not a deep file.

**Return** `true` upon success, or `false` upon failure.

#### Parameters

- `subimage`: The subimage to read from (starting with 0).
- `miplevel`: The MIP level to read (0 is the highest resolution level).
- `chbegin/chend`: The channel range to read.
- `ybegin/yend`: The y range of the scanlines being passed.
- `z`: The z coordinate of the scanline.

- `deepdata`: A *DeepData* object into which the data for these scanlines will be placed.

**virtual bool read\_native\_deep\_tiles** (int *subimage*, int *miplevel*, int *xbegin*, int *xend*, int *ybegin*, int *yend*, int *zbegin*, int *zend*, int *chbegin*, int *chend*, *DeepData* &*deepdata*)

Read into `deepdata` the block of native deep data tiles that include all pixels and channels specified by pixel range.

**Return** `true` upon success, or `false` upon failure.

**Note** 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.

#### Parameters

- `subimage`: The subimage to read from (starting with 0).
- `miplevel`: The MIP level to read (0 is the highest resolution level).
- `xbegin/xend`: The x range of the pixels covered by the group of tiles being read.
- `ybegin/yend`: The y range of the pixels covered by the tiles.
- `zbegin/zend`: The z range of the pixels covered by the tiles (for a 2D image, `zbegin=0` and `zend=1`).
- `chbegin/chend`: The channel range to read.
- `deepdata`: A *DeepData* object into which the data for these tiles will be placed.

**virtual bool read\_native\_deep\_image** (int *subimage*, int *miplevel*, *DeepData* &*deepdata*)

Read the entire deep data image of `spec.width` x `spec.height` x `spec.depth` pixels, all channels, into `deepdata`.

**Return** `true` upon success, or `false` upon failure.

#### Parameters

- `subimage`: The subimage to read from (starting with 0).
- `miplevel`: The MIP level to read (0 is the highest resolution level).
- `deepdata`: A *DeepData* object into which the data for the image will be placed.

### Reading native pixels – implementation overloads

**Note** `read_native_*` methods are usually not directly called by user code (except for `read_native_deep_*` varieties). These are the methods that are overloaded by the *ImageInput* subclasses that implement the individual file format readers.

**virtual bool read\_native\_scanline** (int *subimage*, int *miplevel*, int *y*, int *z*, void \**data*) = 0

Read a single scanline (all channels) of native data into contiguous memory.

**virtual bool read\_native\_scanlines** (int *subimage*, int *miplevel*, int *ybegin*, int *yend*, int *z*, void \**data*)

Read a range of scanlines (all channels) of native data into contiguous memory.

**virtual bool read\_native\_scanlines** (int *subimage*, int *miplevel*, int *ybegin*, int *yend*, int *z*, int *chbegin*, int *chend*, void \**data*)

Read a range of scanlines (with optionally a subset of channels) of native data into contiguous memory.

**virtual bool read\_native\_tile** (int *subimage*, int *miplevel*, int *x*, int *y*, int *z*, void *\*data*)

Read a single tile (all channels) of native data into contiguous memory. The base class `read_native_tile` fails. A format reader that supports tiles MUST overload this virtual method that reads a single tile (all channels).

**virtual bool read\_native\_tiles** (int *subimage*, int *miplevel*, int *xbegin*, int *xend*, int *ybegin*, int *yend*, int *zbegin*, int *zend*, void *\*data*)

Read multiple tiles (all channels) of native data into contiguous memory. A format reader that supports reading multiple tiles at once (in a way that's more efficient than reading the tiles one at a time) is advised (but not required) to overload this virtual method. If an *ImageInput* subclass does not overload this, the default implementation here is simply to loop over the tiles, calling the single-tile `read_native_tile()` for each one.

**virtual bool read\_native\_tiles** (int *subimage*, int *miplevel*, int *xbegin*, int *xend*, int *ybegin*, int *yend*, int *zbegin*, int *zend*, int *chbegin*, int *chend*, void *\*data*)

Read multiple tiles (potentially a subset of channels) of native data into contiguous memory. A format reader that supports reading multiple tiles at once, and can handle a channel subset while doing so, is advised (but not required) to overload this virtual method. If an *ImageInput* subclass does not overload this, the default implementation here is simply to loop over the tiles, calling the single-tile `read_native_tile()` for each one (and copying carefully to handle the channel subset issues).

## Public Types

**using unique\_ptr** = std::unique\_ptr<*ImageInput*>  
unique\_ptr to an *ImageInput*

**typedef *ImageInput* \*(\*Creator)()**

Call signature of a function that creates and returns an *ImageInput*\*.

## Public Functions

**virtual const char \*format\_name** (void) **const** = 0

Return the name of the format implemented by this class.

**virtual int supports** (*string\_view* *feature*) **const**

Given the name of a “feature”, return whether this *ImageInput* supports output of images with the given properties. Most queries will simply return 0 for “doesn’t support” and 1 for “supports it,” but it is acceptable to have queries return other nonzero integers to indicate varying degrees of support or limits (but should be clearly documented as such).

Feature names that *ImageInput* implementations are expected to recognize include:

- "arbitrary\_metadata" : Does this format allow metadata with arbitrary names and types?
- "exif" : Can this format store Exif camera data?
- "iptc" : Can this format store IPTC data?
- "procedural" : Can this format create images without reading from a disk file?
- "ioprox" : Does this format reader support reading from an *IOProxy*?

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.



**virtual bool valid\_file (const std::string &filename) const**

Return true if the *filename* names 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 there can be false positives: 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.

**Return** true upon success, or false upon failure.

**virtual bool open (const std::string &name, ImageSpec &newspec) = 0**

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 successfully.

#### Parameters

- name: Filename to open.
- newspec: Reference to an *ImageSpec* in which to deposit a full description of the contents of the first subimage of the file.

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

Open file with given name, similar to *open(name, newspec)*. The *config* is 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* and call regular *open(name, newspec)*.

**Return** true if the file was found and opened successfully.

#### Parameters

- name: Filename to open.
- newspec: Reference to an *ImageSpec* in which to deposit a full description of the contents of the first subimage of the file.
- config: An *ImageSpec* whose metadata contains “configuration hints” for the *ImageInput* implementation.

**virtual const ImageSpec &spec (void) const**

Return a reference to the image specification of the current subimage/MIPlevel. Note that the contents of the *spec* are invalid before *open()* or after *close()*, and may change with a call to *seek\_subimage()*. It is thus not thread-safe, since the *spec* may change if another thread calls *seek\_subimage*, or any of the *read\_\*()* functions that take explicit subimage/miplevel.

**virtual ImageSpec spec (int subimage, int miplevel = 0)**

Return a full copy of the *ImageSpec* of the designated subimage and MIPlevel. This method is thread-safe, but it is potentially expensive, due to the work that needs to be done to fully copy an *ImageSpec* if there is lots of named metadata to allocate and copy. See also the less expensive *spec\_dimensions()*. Errors (such as having requested a nonexistent subimage) are indicated by returning an *ImageSpec* with *format==TypeUnknown*.

**virtual ImageSpec spec\_dimensions (int subimage, int miplevel = 0)**

Return a copy of the *ImageSpec* of the designated subimage and miplevel, but only the dimension and type fields. Just as with a call to *ImageSpec::copy\_dimensions()*, neither the channel names nor any

of the arbitrary named metadata will be copied, thus this is a relatively inexpensive operation if you don't need that information. It is guaranteed to be thread-safe. Errors (such as having requested a nonexistent subimage) are indicated by returning an *ImageSpec* with `format==TypeUnknown`.

**virtual bool close () = 0**

Close an open *ImageInput*. The call to `close()` is not strictly necessary if the *ImageInput* is destroyed immediately afterwards, since it is required for the destructor to close if the file is still open.

**Return** `true` upon success, or `false` upon failure.

**virtual 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.

**virtual int current\_miplevel (void) const**

Returns the index of the MIPmap image that is currently being read. The highest-res MIP level (or the only level, if there is just one) is number 0.

**virtual bool seek\_subimage (int subimage, int miplevel)**

Seek to the given subimage and MIP-map level within the open image file. The first subimage of the file has index 0, the highest-resolution MIP level has index 0. The new subimage's vital statistics may be retrieved by `this->spec()`. The reader 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/level, it should transparently close, reopen, and sequentially read through prior subimages and levels.

**Return** `true` upon success, or `false` upon failure. A failure may indicate that no such subimage or MIP level exists in the file.

**std::string geterror () const**

If any of the API routines returned false indicating an error, this method will return the error string (and clear any error flags). If no error has occurred since the last time `geterror()` was called, it will return an empty string.

**template<typename ...Args>**

**void error (const char \*fmt, const Args&... args) const**

Error reporting for the plugin implementation: call this with `Strutil::format`-like arguments. Use with caution! Some day this will change to be `fmt`-like rather than `printf`-like.

**template<typename ...Args>**

**void errorf (const char \*fmt, const Args&... args) const**

Error reporting for the plugin implementation: call this with `printf`-like arguments.

**template<typename ...Args>**

**void fmterror (const char \*fmt, const Args&... args) const**

Error reporting for the plugin implementation: call this with `fmt::format`-like arguments.

**void threads (int n)**

Set the threading policy for this *ImageInput*, controlling the maximum amount of parallelizing thread "fan-out" that might occur during large read operations. The default of 0 means that the global attribute("threads") value should be used (which itself defaults to using as many threads as cores; see Section Global Attributes\_).

The main reason to change this value is to set it to 1 to indicate that the calling thread should do all the work rather than spawning new threads. That is probably the desired behavior in situations where the calling application has already spawned multiple worker threads.

int **threads** () **const**

Retrieve the current thread-spawning policy.

**See** *threads(int)*

void **lock** ()

Lock the internal mutex, block until the lock is acquired.

bool **try\_lock** ()

Try to lock the internal mutex, returning true if successful, or false if the lock could not be immediately acquired.

void **unlock** ()

Unlock the internal mutex.



## WRITING IMAGEIO PLUGINS

### 5.1 Plugin Introduction

As explained in Chapters *ImageInput: Reading Images* and *ImageOutput: Writing Images*, 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 Reader Plugins

A plugin that reads a particular image file format must implement a *subclass* of ImageInput (described in Chapter *ImageInput: Reading Images*). 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 these 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. An function named `name_imageio_library_version` that identifies the underlying dependent library that is responsible for reading or writing the format (it may return `nullptr` to indicate that there is no dependent library being used for this format).
  - c. A function named `name_input_imageio_create` that takes no arguments and returns an `ImageInput *` constructed from a new instance of your ImageInput subclass and a deleter. (Note that *name* is the name of your format, and must match the name of the plugin itself.)

- d. 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 `nullptr`.

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 ImageInput *jpeg_input_imageio_create () {
        return new JpgInput;
    }
    OIIO_EXPORT const char *jpeg_input_extensions[] = {
        "jpg", "jpe", "jpeg", "jif", "jfif", "jfi", nullptr
    };
    OIIO_EXPORT const char* jpeg_imageio_library_version () {
        #define STRINGIZE2(a) #a
        #define STRINGIZE(a) STRINGIZE2(a)
        #ifdef LIBJPEG_TURBO_VERSION
            return "jpeg-turbo " STRINGIZE(LIBJPEG_TURBO_VERSION);
        #else
            return "jpeglib " STRINGIZE(JPEG_LIB_VERSION_MAJOR) "."
                STRINGIZE(JPEG_LIB_VERSION_MINOR);
        #endif
    }
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 naive 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 the section describing `ImageInput::supports`.
- 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 final : public ImageInput {
public:
    JpgInput () { init(); }
    virtual ~JpgInput () { close(); }
    virtual const char * format_name (void) const override { return "jpeg"; }
    virtual bool open (const std::string &name, ImageSpec &spec) override;
    virtual bool read_native_scanline (int y, int z, void *data) override;
    virtual bool close () override;
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-present Contributors to the OpenImageIO project.
// SPDX-License-Identifier: BSD-3-Clause
// https://github.com/OpenImageIO/oio/blob/master/LICENSE.md

#include <algorithm>
#include <cassert>
#include <cstdio>

#include <OpenImageIO/color.h>
#include <OpenImageIO/filesystem.h>
#include <OpenImageIO/fmath.h>
#include <OpenImageIO/imageio.h>
#include <OpenImageIO/tiffutils.h>

#include "jpeg_pvt.h"
```

(continues on next page)

(continued from previous page)

```

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 const char*
jpeg_imageio_library_version()
{
#define STRINGIZE2(a) #a
#define STRINGIZE(a) STRINGIZE2(a)
#ifdef LIBJPEG_TURBO_VERSION
    return "jpeg-turbo " STRINGIZE(LIBJPEG_TURBO_VERSION) "/" STRINGIZE(
        JPEG_LIB_VERSION);
#else
    return "jpeglib " STRINGIZE(JPEG_LIB_VERSION_MAJOR) "." STRINGIZE(
        JPEG_LIB_VERSION_MINOR);
#endif
}

OIIO_EXPORT ImageInput*
jpeg_input_imageio_create()
{
    return new JpgInput;
}

OIIO_EXPORT const char* jpeg_input_extensions[]
    = { "jpg", "jpe", "jpeg", "jif", "jfif", "jfi", nullptr };

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);

```

(continues on next page)



(continued from previous page)

```

    /* 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;

    // Create the message
    char buffer[JMSG_LENGTH_MAX];
    (*cinfo->err->format_message)(cinfo, buffer);
    myerr->jpginput->jpegerror(myerr, true);

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

static std::string
comp_info_to_attr(const jpeg_decompress_struct& cinfo)
{
    // Compare the current 6 samples with our known definitions
    // to determine the corresponding subsampling attr
    std::vector<int> comp;
    comp.push_back(cinfo.comp_info[0].h_samp_factor);
    comp.push_back(cinfo.comp_info[0].v_samp_factor);
    comp.push_back(cinfo.comp_info[1].h_samp_factor);
    comp.push_back(cinfo.comp_info[1].v_samp_factor);
    comp.push_back(cinfo.comp_info[2].h_samp_factor);
    comp.push_back(cinfo.comp_info[2].v_samp_factor);
    size_t size = comp.size();

    if (std::equal(JPEG_444_COMP, JPEG_444_COMP + size, comp.begin()))
        return JPEG_444_STR;
    else if (std::equal(JPEG_422_COMP, JPEG_422_COMP + size, comp.begin()))
        return JPEG_422_STR;
    else if (std::equal(JPEG_420_COMP, JPEG_420_COMP + size, comp.begin()))
        return JPEG_420_STR;
    else if (std::equal(JPEG_411_COMP, JPEG_411_COMP + size, comp.begin()))
        return JPEG_411_STR;
    return "";
}

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);
    errorf("JPEG error: %s (%s)", errbuf, filename());
}

```

(continues on next page)

(continued from previous page)

```

    // 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, Filesystem::IOProxy* io) const
{
    // Check magic number to assure this is a JPEG file
    uint8_t magic[2] = { 0, 0 };
    bool ok = true;

    if (io) {
        ok = (io->pread(magic, sizeof(magic), 0) == sizeof(magic));
    } else {
        FILE* fd = Filesystem::fopen(filename, "rb");
        if (!fd)
            return false;
        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)
{
    auto p = config.find_attribute("_jpeg:raw", TypeInt);
    m_raw = p && *(int*)p->data();
    p = config.find_attribute("oiio:ioproxy", TypeDesc::PTR);
    if (p)
        m_io = p->get<Filesystem::IOProxy*>();
    m_config.reset(new ImageSpec(config)); // save config spec
    return open(name, newspec);
}

bool
JpgInput::open(const std::string& name, ImageSpec& newspec)
{
    m_filename = name;

    if (m_io) {
        // If an IOProxy was passed, it had better be a File or a

```

(continues on next page)

(continued from previous page)

```

    // MemReader, that's all we know how to use with jpeg.
    std::string proxytype = m_io->proxytype();
    if (proxytype != "file" && proxytype != "memreader") {
        errorf("JPEG reader can't handle proxy type %s", proxytype);
        return false;
    }
} else {
    // If no proxy was supplied, create a file reader
    m_io = new Filesystem::IOFile(name, Filesystem::IOProxy::Mode::Read);
    m_local_io.reset(m_io);
}
if (!m_io || m_io->mode() != Filesystem::IOProxy::Mode::Read) {
    errorf("Could not open file \"%s\"", name);
    return false;
}

// Check magic number to assure this is a JPEG file
uint8_t magic[2] = { 0, 0 };
if (m_io->pread(magic, sizeof(magic), 0) != sizeof(magic)) {
    errorf("Empty file \"%s\"", name);
    close_file();
    return false;
}

if (magic[0] != JPEG_MAGIC1 || magic[1] != JPEG_MAGIC2) {
    close_file();
    errorf(
        "\"%s\" is not a JPEG file, magic number doesn't match (was 0x%x%x)",
        name, int(magic[0]), int(magic[1]));
    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;
}

// initialize decompressor
jpeg_create_decompress(&m_cinfo);
m_decomp_create = true;
// specify the data source
if (!strcmp(m_io->proxytype(), "file")) {
    auto fd = ((Filesystem::IOFile*)m_io)->handle();
    jpeg_stdio_src(&m_cinfo, fd);
} else {
    auto buffer = ((Filesystem::IOMemReader*)m_io)->buffer();
    jpeg_mem_src(&m_cinfo, const_cast<unsigned char*>(buffer.data()),
        buffer.size());
}

```

(continues on next page)

(continued from previous page)

```

// 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) {
    errorf("Bad JPEG header for \"%s\"", filename());
    return false;
}

int nchannels = m_cinfo.num_components;

if (m_cinfo.jpeg_color_space == JCS_CMYK
    || m_cinfo.jpeg_color_space == JCS_YCCK) {
    // CMYK jpegs get converted by us to RGB
    m_cinfo.out_color_space = JCS_CMYK; // pre-convert YCbCrK->CMYK
    nchannels                = 3;
    m_cmyk                   = true;
}

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, nchannels,
    TypeDesc::UINT8);

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

if (m_cinfo.jpeg_color_space == JCS_CMYK)
    m_spec.attribute("jpeg:ColorSpace", "CMYK");
else if (m_cinfo.jpeg_color_space == JCS_YCCK)
    m_spec.attribute("jpeg:ColorSpace", "YCbCrK");

// If the chroma subsampling is detected and matches something
// we expect, then set an attribute so that it can be preserved
// in future operations.
std::string subsampling = comp_info_to_attr(m_cinfo);
if (!subsampling.empty())
    m_spec.attribute(JPEG_SUBSAMPLING_ATTR, subsampling);

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(string_view((char*)m->data + 6, m->data_length - 6),
            m_spec);
    } else if (m->marker == (JPEG_APP0 + 1)
        && !strcmp((const char*)m->data,

```

(continues on next page)

(continued from previous page)

```

        "http://ns.adobe.com/xap/1.0/")) {
#ifdef NDEBUG
        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,
                    m->data_length));
    }
}

// Handle density/pixelaspect. We need to do this AFTER the exif is
// decoded, in case it contains useful information.
float xdensity = m_spec.get_float_attribute("XResolution");
float ydensity = m_spec.get_float_attribute("YResolution");
if (!xdensity || !ydensity) {
    xdensity = float(m_cinfo.X_density);
    ydensity = float(m_cinfo.Y_density);
    if (xdensity && ydensity) {
        m_spec.attribute("XResolution", xdensity);
        m_spec.attribute("YResolution", ydensity);
    }
}
if (xdensity && ydensity) {
    float aspect = ydensity / xdensity;
    if (aspect != 1.0f)
        m_spec.attribute("PixelAspectRatio", aspect);
    switch (m_cinfo.density_unit) {
    case 0: m_spec.attribute("ResolutionUnit", "none"); break;
    case 1: m_spec.attribute("ResolutionUnit", "in"); break;
    case 2: m_spec.attribute("ResolutionUnit", "cm"); break;
    }
}

read_icc_profile(&m_cinfo, m_spec); /// try to read icc profile

newspec = m_spec;
return true;
}

bool
JpgInput::read_icc_profile(j_decompress_ptr cinfo, ImageSpec& spec)
{
    int num_markers = 0;
    std::vector<unsigned char> icc_buf;
    unsigned int total_length = 0;
    const int MAX_SEQ_NO = 255;
    unsigned char marker_present
        [MAX_SEQ_NO

```

(continues on next page)

(continued from previous page)

```

        + 1]; // one extra is used to store the flag if marker is found, set to one_
↪if marker is found
    unsigned int data_length[MAX_SEQ_NO + 1]; // store the size of each marker
    unsigned int data_offset[MAX_SEQ_NO + 1]; // store the offset of each marker
    memset(marker_present, 0, (MAX_SEQ_NO + 1));

    for (jpeg_saved_marker_ptr m = cinfo->marker_list; m; m = m->next) {
        if (m->marker == (JPEG_APP0 + 2)
            && !strcmp((const char*)m->data, "ICC_PROFILE")) {
            if (num_markers == 0)
                num_markers = GETJOCTET(m->data[13]);
            else if (num_markers != GETJOCTET(m->data[13]))
                return false;
            int seq_no = GETJOCTET(m->data[12]);
            if (seq_no <= 0 || seq_no > num_markers)
                return false;
            if (marker_present[seq_no]) // duplicate marker
                return false;
            marker_present[seq_no] = 1; // flag found marker
            data_length[seq_no] = m->data_length - ICC_HEADER_SIZE;
        }
    }
    if (num_markers == 0)
        return false;

    // checking for missing markers
    for (int seq_no = 1; seq_no <= num_markers; seq_no++) {
        if (marker_present[seq_no] == 0)
            return false; // missing sequence number
        data_offset[seq_no] = total_length;
        total_length += data_length[seq_no];
    }

    if (total_length == 0)
        return false; // found only empty markers

    icc_buf.resize(total_length * sizeof(JOCTET));

    // and fill it in
    for (jpeg_saved_marker_ptr m = cinfo->marker_list; m; m = m->next) {
        if (m->marker == (JPEG_APP0 + 2)
            && !strcmp((const char*)m->data, "ICC_PROFILE")) {
            int seq_no = GETJOCTET(m->data[12]);
            memcpy(&icc_buf[0] + data_offset[seq_no], m->data + ICC_HEADER_SIZE,
                data_length[seq_no]);
        }
    }
    spec.attribute(ICC_PROFILE_ATTR, TypeDesc(TypeDesc::UINT8, total_length),
        &icc_buf[0]);
    return true;
}

static void
cmyk_to_rgb(int n, const unsigned char* cmyk, size_t cmyk_stride,
    unsigned char* rgb, size_t rgb_stride)

```

(continues on next page)

(continued from previous page)

```

{
    for (; n; --n, cmyk += cmyk_stride, rgb += rgb_stride) {
        // JPEG seems to store CMYK as 1-x
        float C = convert_type<unsigned char, float>(cmyk[0]);
        float M = convert_type<unsigned char, float>(cmyk[1]);
        float Y = convert_type<unsigned char, float>(cmyk[2]);
        float K = convert_type<unsigned char, float>(cmyk[3]);
        float R = C * K;
        float G = M * K;
        float B = Y * K;
        rgb[0] = convert_type<float, unsigned char>(R);
        rgb[1] = convert_type<float, unsigned char>(G);
        rgb[2] = convert_type<float, unsigned char>(B);
    }
}

bool
JpgInput::read_native_scanline(int subimage, int miplevel, int y, int z,
                               void* data)
{
    if (!seek_subimage(subimage, miplevel))
        return false;
    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.
        // Don't forget to save and restore any configuration settings.
        ImageSpec configsave;
        if (m_config)
            configsave = *m_config;
        ImageSpec dummyspec;
        int subimage = current_subimage();
        if (!close() || !open(m_filename, dummyspec, configsave)
            || !seek_subimage(subimage, 0))
            return false; // Somehow, the re-open failed
        OIIO_DASSERT(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;
    }

    void* readdata = data;
    if (m_cmyk) {
        // If the file's data is CMYK, read into a 4-channel buffer, then
        // we'll have to convert.
        m_cmyk_buf.resize(m_spec.width * 4);
        readdata = &m_cmyk_buf[0];
        OIIO_DASSERT(m_spec.nchannels == 3);
    }
}

```

(continues on next page)

(continued from previous page)

```

    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**) &readdata, 1) != 1
            || m_fatalerr) {
            errorf("JPEG failed scanline read (\"%s\")", filename());
            return false;
        }
    }

    if (m_cmyk)
        cmyk_to_rgb(m_spec.width, (unsigned char*) readdata, 4,
                    (unsigned char*) data, 3);

    return true;
}

bool
JpgInput::close()
{
    if (m_io) {
        // unnecessary?  jpeg_abort_decompress (&m_cinfo);
        if (m_decomp_create)
            jpeg_destroy_decompress(&m_cinfo);
        m_decomp_create = false;
        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;
}

```

(continues on next page)



(continued from previous page)

```

// 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 *ImageOutput: Writing Images*). 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 *Image Reader Plugins*).
2. Declare four 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 an `ImageOutput *` constructed from a new instance of your `ImageOutput` subclass and a deleter. (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 `nullptr` 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 ImageOutput *jpeg_output_imageio_create () {
        return new JpgOutput;
    }
    OIIO_EXPORT const char *jpeg_input_extensions[] = {
        "jpg", "jpe", "jpeg", nullptr
    };
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 the description of `ImageOutput::supports()` 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 final : public ImageOutput {
public:
    JpgOutput () { init(); }
    virtual ~JpgOutput () { close(); }
    virtual const char * format_name (void) const override { return "jpeg"; }
    virtual int supports (string_view property) const override { return_
↪false; }
    virtual bool open (const std::string &name, const ImageSpec &spec,
                      bool append=false) override;
    virtual bool write_scanline (int y, int z, TypeDesc format,
                                const void *data, stride_t xstride)
↪override;
```

(continues on next page)

(continued from previous page)

```

    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, Type-
                                Desc 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-present Contributors to the OpenImageIO project.
// SPDX-License-Identifier: BSD-3-Clause
// https://github.com/OpenImageIO/oio/blob/master/LICENSE.md

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

#include <OpenImageIO/filesystem.h>
#include <OpenImageIO/fmath.h>
#include <OpenImageIO/imageio.h>
#include <OpenImageIO/tiffutils.h>

#include "jpeg_pvt.h"

OIIO_PLUGIN_NAMESPACE_BEGIN

#define DBG if (0)

// References:
// * JPEG library documentation: /usr/share/doc/libjpeg-devel-6b
// * JFIF spec: https://www.w3.org/Graphics/JPEG/jfif3.pdf
// * ITU T.871 (aka ISO/IEC 10918-5):
//   https://www.itu.int/rec/T-REC-T.871-201105-I/en

class JpgOutput final : public ImageOutput {
public:
    JpgOutput() { init(); }
    virtual ~JpgOutput() { close(); }
    virtual const char* format_name(void) const override { return "jpeg"; }
    virtual int supports(string_view feature) const override
    {
        return (feature == "exif" || feature == "iptc");
    }
    virtual bool open(const std::string& name, const ImageSpec& spec,
                     OpenMode mode = Create) override;
    virtual bool write_scanline(int y, int z, TypeDesc format, const void* data,
                               stride_t xstride) override;
    virtual bool write_tile(int x, int y, int z, TypeDesc format,
                            const void* data, stride_t xstride,
                            stride_t ystride, stride_t zstride) override;
    virtual bool close() override;
    virtual bool copy_image(ImageInput* in) override;

private:
    FILE* m_fd;
    std::string m_filename;
    unsigned int m_dither;
    int m_next_scanline; // Which scanline is the next to write?
```

(continues on next page)

(continued from previous page)

```

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;
}

void set_subsampling(const int components[])
{
    jpeg_set_colorspace(&m_cinfo, JCS_YCbCr);
    m_cinfo.comp_info[0].h_samp_factor = components[0];
    m_cinfo.comp_info[0].v_samp_factor = components[1];
    m_cinfo.comp_info[1].h_samp_factor = components[2];
    m_cinfo.comp_info[1].v_samp_factor = components[3];
    m_cinfo.comp_info[2].h_samp_factor = components[4];
    m_cinfo.comp_info[2].v_samp_factor = components[5];
}

// Read the XResolution/YResolution and PixelAspectRatio metadata, store
// in density fields m_cinfo.X_density,Y_density.
void resmeta_to_density();
};

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", nullptr };

OIIO_PLUGIN_EXPORTS_END

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

    // Save name and spec for later use

```

(continues on next page)

(continued from previous page)

```

m_filename = name;
m_spec      = newspec;

// Check for things this format doesn't support
if (m_spec.width < 1 || m_spec.height < 1) {
    errorf("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) {
    errorf("%s does not support volume images (depth > 1)", format_name());
    return false;
}

m_fd = Filesystem::fopen(name, "wb");
if (m_fd == NULL) {
    errorf("Could not open \"%s\"", name);
    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;

// JFIF can only handle grayscale and RGB. Do the best we can with this
// limited format by truncating to 3 channels if > 3 are requested,
// truncating to 1 channel if 2 are requested.
if (m_spec.nchannels >= 3) {
    m_cinfo.input_components = 3;
    m_cinfo.in_color_space   = JCS_RGB;
} else {
    m_cinfo.input_components = 1;
    m_cinfo.in_color_space   = JCS_GRAYSCALE;
}

resmeta_to_density();

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
    // Careful -- jpeg_set_defaults overwrites density
    resmeta_to_density();
    DBG std::cout << "out open: set_defaults\n";
}

```

(continues on next page)

(continued from previous page)

```

    auto compqual = m_spec.decode_compression_metadata("jpeg", 98);
    if (Strutil::iequals(compqual.first, "jpeg"))
        jpeg_set_quality(&m_cinfo, clamp(compqual.second, 1, 100), TRUE);
    else
        jpeg_set_quality(&m_cinfo, 98, TRUE); // not jpeg? default qual

    if (m_cinfo.input_components == 3) {
        std::string subsampling = m_spec.get_string_attribute(
            JPEG_SUBSAMPLING_ATTR);
        if (subsampling == JPEG_444_STR)
            set_subsampling(JPEG_444_COMP);
        else if (subsampling == JPEG_422_STR)
            set_subsampling(JPEG_422_COMP);
        else if (subsampling == JPEG_420_STR)
            set_subsampling(JPEG_420_COMP);
        else if (subsampling == JPEG_411_STR)
            set_subsampling(JPEG_411_COMP);
    }
    DBG std::cout << "out open: set_colorspace\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'
ParamValue* 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";

```

(continues on next page)

(continued from previous page)

```

    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

// Write ICC profile, if we have anything
const ParamValue* icc_profile_parameter = m_spec.find_attribute(
    ICC_PROFILE_ATTR);
if (icc_profile_parameter != NULL) {
    unsigned char* icc_profile
        = (unsigned char*)icc_profile_parameter->data();
    unsigned int icc_profile_length = icc_profile_parameter->type().size();
    if (icc_profile && icc_profile_length) {
        /* Calculate the number of markers we'll need, rounding up of course */
        int num_markers = icc_profile_length / MAX_DATA_BYTES_IN_MARKER;
        if ((unsigned int)(num_markers * MAX_DATA_BYTES_IN_MARKER)
            != icc_profile_length)
            num_markers++;
        int curr_marker = 1; /* per spec, count starts at 1 */
        size_t profile_size = MAX_DATA_BYTES_IN_MARKER + ICC_HEADER_SIZE;
        std::vector<unsigned char> profile(profile_size);
        while (icc_profile_length > 0) {
            // length of profile to put in this marker
            unsigned int length
                = std::min(icc_profile_length,
                           (unsigned int)MAX_DATA_BYTES_IN_MARKER);
            icc_profile_length -= length;
            // Write the JPEG marker header (APP2 code and marker length)
            strncpy((char*)&profile[0], "ICC_PROFILE", profile_size);
            profile[11] = 0;
            profile[12] = curr_marker;
            profile[13] = (unsigned char)num_markers;
            memcpy(&profile[0] + ICC_HEADER_SIZE,
                  icc_profile + length * (curr_marker - 1), length);
            jpeg_write_marker(&m_cinfo, JPEG_APP0 + 2, &profile[0],
                             ICC_HEADER_SIZE + length);
        }
    }
}

```

(continues on next page)



(continued from previous page)

```

        curr_marker++;
    }
}

m_dither = m_spec.get_int_attribute("oio: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;
}

void
JpgOutput::resmeta_to_density()
{
    string_view resunit = m_spec.get_string_attribute("ResolutionUnit");
    if (Strutil::iequals(resunit, "none"))
        m_cinfo.density_unit = 0;
    else if (Strutil::iequals(resunit, "in"))
        m_cinfo.density_unit = 1;
    else if (Strutil::iequals(resunit, "cm"))
        m_cinfo.density_unit = 2;
    else
        m_cinfo.density_unit = 0;

    int X_density = int(m_spec.get_float_attribute("XResolution"));
    int Y_density = int(m_spec.get_float_attribute("YResolution", X_density));
    const float aspect = m_spec.get_float_attribute("PixelAspectRatio", 1.0f);
    if (aspect != 1.0f && X_density <= 1 && Y_density <= 1) {
        // No useful [XY]Resolution, but there is an aspect ratio requested.
        // Arbitrarily pick 72 dots per undefined unit, and jigger it to
        // honor it as best as we can.
        //
        // Here's where things get tricky. By logic and reason, as well as
        // the JFIF spec and ITU T.871, the pixel aspect ratio is clearly
        // ydensity/xdensity (because aspect is xlength/ylength, and density
        // is 1/length). BUT... for reasons lost to history, a number of
        // apps get this exactly backwards, and these include PhotoShop,
        // Nuke, and RV. So, alas, we must replicate the mistake, or else
        // all these common applications will misunderstand the JPEG files
        // written by OIIO and vice versa.
        Y_density = 72;
        X_density = int(Y_density * aspect + 0.5f);
        m_spec.attribute("XResolution", float(Y_density * aspect + 0.5f));
        m_spec.attribute("YResolution", float(Y_density));
    }
    while (X_density > 65535 || Y_density > 65535) {
        // JPEG header can store only UINT16 density values. If we
        // overflow that limit, punt and knock it down to <= 16 bits.
        X_density /= 2;
        Y_density /= 2;
    }
}

```

(continues on next page)

(continued from previous page)

```

    m_cinfo.X_density = X_density;
    m_cinfo.Y_density = Y_density;
}

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) {
        errorf("Attempt to write scanlines out of order to %s", m_filename);
        return false;
    }
    if (y >= (int)m_cinfo.image_height) {
        errorf("Attempt to write too many scanlines to %s", m_filename);
        return false;
    }
    assert(y == (int)m_cinfo.next_scanline);

    // Here's where we do the dirty work of conforming to JFIF's limitation
    // of 1 or 3 channels, by temporarily doctoring the spec so that
    // to_native_scanline properly contiguizes the first 1 or 3 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 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 1 or 3 channels.
    m_spec.auto_stride(xstride, format, m_spec.nchannels);
    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]);
}

```

(continues on next page)

(continued from previous page)

```

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.
        OIIO_DASSERT(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";
    }
}

```

(continues on next page)

(continued from previous page)

```

    // 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.



## 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
<code>compression</code>	string	Compression type
<code>oiio:BitsPerSample</code>	int	bits per sample
<code>textureformat</code>	string	Set correctly to one of "Plain Texture", "Volume Texture", or "CubeFace Environment".
<code>texturetype</code>	string	Set correctly to one of "Plain Texture", "Volume Texture", or "Environment".
<code>dds:CubeMapSides</code>	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 DICOM

DICOM (Digital Imaging and Communications in Medicine) is the standard format used for medical images. DICOM files usually have the extension `.dcm`.

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

ImageSpec Attribute	Type	DDS header data or explanation
<code>oiio:BitsPerSample</code>	int	Bits per sample.
<code>dicom:*</code>	<i>any</i>	DICOM header information and metadata is currently all preceded by the <code>dicom:</code> prefix.

## 6.5 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`.

### Configuration settings for DPX input

When opening a DPX ImageInput with a *configuration* (see Section `sec-inputwithconfig`), the following special configuration options are supported:



Input Configuration Attribute	Type	Meaning
oiio:RawColor	int	If nonzero, reading images with non-RGB color models (such as YCbCr) will return unaltered pixel values (versus the default OIIO behavior of automatically converting to RGB).

### Configuration settings for DPX output

When opening a DPX ImageOutput, the following special metadata tokens control aspects of the writing itself:

Output configuration Attribute	Type	Meaning
oiio:RawColor	int	If nonzero, writing images with non-RGB color models (such as YCbCr) will keep unaltered pixel values (versus the default OIIO behavior of automatically converting from RGB to the designated color space as the pixels are written).

### DPX Attributes

ImageSpec 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 metadata:orientation)
compression	string	The compression type
PixelAspectRatio	float	pixel aspect ratio
oiio:BitsPerSample	int	the true bits per sample of the DPX file.
oiio:Endian	string	When writing, force a particular endianness for the output "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)
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

continues on next page

Table 1 – continued from previous page

ImageSpec Attribute	Type	DPX header data or explanation
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)
dpx:EndOfLinePadding	int	Padded bytes at the end of each line
dpx:EndOfImagePadding	int	Padded bytes at the end of each image

## 6.6 Field3D

Field3d is an open-source volume data file format. Field3d files commonly use the extension `.f3d`. The official Field3D site is: <https://github.com/imageworks/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
worldtolocal	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").

## 6.7 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.

## 6.8 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
<code>gif:Interlacing</code>	int	Specifies if image is interlaced (0 or 1).
<code>FramesPerSecond</code>	int[2] (ratio- nal)	Frames per second
<code>oiio:Movie</code>	int	If nonzero, indicates that it's an animated GIF.
<code>gif:LoopCount</code>	int	Number of times the animation should be played (0-65535, 0 stands for infinity).
<code>ImageDescription</code>	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.

## 6.9 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
<code>Orientation</code>	int	encodes the orientation (see Section <i>Display hints</i> )
<code>oiio:ColorSpace</code>	string	Color space (see Section <i>Color information</i> ).
<code>oiio:Gamma</code>	float	the gamma correction specified in the RGBE header (if it's gamma corrected).

## 6.10 HEIF/HEIC

HEIF is a container format for images compressed with the HEIC compression standard (same compression as HEVC/H.265). It is used commonly for iPhone camera pictures, but it is not Apple-specific and will probably become more popular on other platforms in coming years. HEIF files usually use the file extension `.HEIC`.

HEIC compression is lossy, but is higher visual quality than JPEG while taking only half the file size. Currently, OIIO's HEIF reader supports reading files as RGB or RGBA, uint8 pixel values. Multi-image files are currently supported for reading, but not yet writing. All pixel data is uint8, though we hope to add support for HDR (more than 8 bits) in the future.

### Configuration settings for HEIF output

When opening an HEIF ImageOutput, the following special metadata tokens control aspects of the writing itself:

ImageSpec Attribute	Type	HEIF header data or explanation
Compression	string	If supplied, must be "heic", but may optionally have a quality value appended, like "heic:90". Quality can be 1-100, with 100 meaning lossless. The default is 75.

## 6.11 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
oio: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 x 256. Requests to write larger images will fail their `open()` call.

## 6.12 IFF

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

ImageSpec Attribute	Type	IFF header data or explanation
Artist	string	The IFF "author"
DateTime	string	Creation date/time
compression	string	The compression type ("none" or "rle" [default])
oio:BitsPerSample	int	the true bits per sample of the IFF file.

## 6.13 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
Compression	string	If supplied, must be "jpeg", but may optionally have a quality value appended, like "jpeg:90". Quality can be 1-100, with 100 meaning lossless.
ICCProfile	uint8[]	The ICC color profile
jpeg:subsampling	string	Describes the chroma subsampling, e.g., "4:2:0" (the default), "4:4:4", "4:2:2", "4:2:1".
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 <a href="#">Appendix Metadata conventions</a> 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 n-channel image data, but will only output the first 3 channels (if  $n \geq 3$ ) or the first channel (if  $n \leq 2$ ), silently drop any extra channels from the output.
- 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.

## 6.14 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
<code>jpeg2000:streamformat</code>	string	specifies the JPEG-2000 stream format ("none" or "jpc")

## 6.15 Movie formats (using ffmpeg)

The **ffmpeg**-based reader is capable of reading the individual frames from a variety of movie file formats, including:

Format	Extensions
AVI	<code>.avi</code>
QuickTime	<code>.qt</code> , <code>.mov</code>
MPEG-4	<code>.mp4</code> , <code>.m4a</code> , <code>.m4v</code>
3GPP files	<code>.3gp</code> , <code>.3g2</code>
Motion JPEG-2000	<code>.mj2</code>
Apple M4V	<code>.m4v</code>
MPEG-1/MPEG-2	<code>.mpg</code>

Currently, these files may only be read. Write support may be added in a future release. Also, currently, these files simply look to OIIO like simple multi-image files and not much support is given to the fact that they are technically *movies* (for example, there is no support for reading audio information).

Some special attributes are used for movie files:

ImageSpec Attribute	Type	Header data or explanation
<code>oiio:Movie</code>	int	Nonzero value for movie files
<code>oiio:subimages</code>	int	The number of frames in the movie, positive if it can be known without reading the entire file. Zero or not present if the number of frames cannot be determined from reading from just the file header.
<code>FramesPerSecond</code>	int[2] (ratio- nal)	Frames per second

## 6.16 Null format

The `nullptr` reader/writer is a mock-up that does not perform any actual I/O. The reader just returns constant-colored pixels, and the writer just returns directly without saving any data. This has several uses:

- Benchmarking, if you want to have OIIO's input or output truly take as close to no time whatsoever.
- “Dry run” of applications where you don't want it to produce any real output (akin to a Unix command that you redirect output to `/dev/null`).
- Make “fake” input that looks like a file, but the file doesn't exist (if you are happy with constant-colored pixels).

The filename allows a REST-ful syntax, where you can append modifiers that specify things like resolution (of the non-existent file), etc. For example:

```
foo.null?RES=640x480&CHANNELS=3
```

would specify a null file with resolution 640x480 and 3 channels. Token/value pairs accepted are:

RES=1024x1024	Set resolution (3D example: 256x256x100)
CHANNELS=4	Set number of channels
TILES=64x64	Makes it look like a tiled image with tile size
TYPE=uint8	Set the pixel data type
PIXEL=r,g,b,...	Set pixel values (comma separates channel values)
TEX=1	Make it look like a full MIP-mapped texture
attrib=value	Anything else will set metadata

## 6.17 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/>.

### Attributes



ImageSpec Attribute	Type	OpenEXR header data or explanation
width,height,x,y	int	dataWindow
full_width, full_height, full_x, full_y	int	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", "zips", "piz", "pxr24", "b44", "b44a", "dwa", or "dwab". If the writer receives a request for a compression type it does not recognize or is not supported by the version of OpenEXR on the system, it will use "zip" by default. For "dwa" and "dwab", the dwaCompressionLevel may be optionally appended to the compression name after a colon, like this: "dwa:200". (The default DWA compression value is 45.)
textureformat	string	"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
FramesPerSecond	int[2]	Frames per second playback rate (vec semantics will be marked as RATIONAL)
captureRate	int[2]	Frames per second capture rate (vec semantics will be marked as RATIONAL)
smpte:TimeCode	int[2]	SMPTE time code (vec semantics will be marked as TIMECODE)
smpte:KeyCode	int[7]	SMPTE key code (vec semantics will be marked as KEYCODE)
openexr:lineOrder	string	OpenEXR lineOrder attribute: "increasingY", "randomY", or "decreasingY".
openexr:roundingmode	int	the MIPmap rounding mode of the file.
openexr:dwaCompressionLevel	float	compression level for dwa or dwab compression (default: 45.0).
<i>other</i>		All other attributes will be added to the ImageSpec by their name and apparent type.

### Configuration settings for OpenEXR input

When opening an OpenEXR ImageInput with a *configuration* (see Section sec-inputwithconfig), the following special configuration attributes are supported:

Input Configuration Attribute	Type	Meaning
oiio:ioproxy	ptr	Pointer to a <code>Filesystem::IOProxy</code> that will handle the I/O, for example by reading from memory rather than the file system.
oiio:missingcolor	float <i>or</i> string	Either an array of float values or a string holding a comma-separated list of values, if present this is a request to use this color for pixels of any missing tiles or scanlines, rather than considering a tile/scanline read failure to be an error. This can be helpful when intentionally reading partially-written or incomplete files (such as an in-progress render).

### Configuration settings for OpenEXR output

When opening an OpenEXR ImageOutput, the following special metadata tokens control aspects of the writing itself:

Output Configuration Attribute	Type	Meaning
oio:RawColor	int	If nonzero, writing images with non-RGB color models (such as YCbCr) will keep unaltered pixel values (versus the default OIIO behavior of automatically converting from RGB to the designated color space as the pixels are written).
oio:ioproxy	ptr	Pointer to a <code>Filesystem::IOProxy</code> that will handle the I/O, for example by writing to a memory buffer.

### Custom I/O Overrides

OpenEXR input and output both support the “custom I/O” feature via the special "oio:ioproxy" attributes (see Sections `sec-imageoutput-ioproxy` and *Custom I/O proxies (and reading the file from a memory buffer)*).

### A note on channel names

The underlying OpenEXR library (`libIlmImf`) always saves channels into lexicographic order, so the channel order on disk (and thus when read!) will NOT match the order when the image was created.

But in order to adhere to OIIO’s convention that RGBAZ will always be the first channels (if they exist), OIIO’s OpenEXR reader will automatically reorder just those channels to appear at the front and in that order. All other channel names will remain in their relative order as presented to OIIO by `libIlmImf`.

### Limitations

- 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.

## 6.18 OpenVDB

OpenVDB is an open-source volume data file format. OpenVDB files commonly use the extension `.vdb`. The official OpenVDB site is: <http://www.openvdb.org/> Currently, OpenImageIO only reads OpenVDB files, and does not write them.

Volumes 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 voxel data are always `float`. OpenVDB files always report as tiled, using the leaf dimension size.

ImageSpec Attribute	Type	OpenVDB header data or explanation
ImageDescription	string	Description of image element
oio:subimagename	string	unique layer name
openvdb:indextoworld	matrix of doubles	conversion of voxel index to world space coordinates.
openvdb:worldtoindex	matrix of doubles	conversion of world space coordinates to voxel index.
worldtocamera	matrix	World-to-local coordinate mapping.

## 6.19 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`.

### Attributes

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, YResolution, ResolutionUnit		resolution and units from the PNG header.
oio:ColorSpace	string	Color space (see Section <a href="#">Color information</a> ).
oio:Gamma	float	the gamma correction value (if specified).
ICCProfile	uint8[]	The ICC color profile

### Configuration settings for PNG input

When opening an PNG ImageInput with a *configuration* (see Section [sec-inputwithconfig](#)), the following special configuration attributes are supported:

Input Configuration Attribute	Type	Meaning
oio:UnassociatedAlpha	int	If nonzero, will leave alpha unassociated (versus the default of pre-multiplying color channels by alpha if the alpha channel is unassociated).
oio:ioproxy	ptr	Pointer to a <code>Filesystem::IOProxy</code> that will handle the I/O, for example by reading from memory rather than the file system.

### Configuration settings for PNG output

When opening an PNG ImageOutput, the following special metadata tokens control aspects of the writing itself:

Output Configuration Attribute	Type	Meaning
<code>png:compressionLevel</code>	int	Compression level for zip/deflate compression, on a scale from 0 (fastest, minimal compression) to 9 (slowest, maximal compression). The default is 6. PNG compression is always lossless.
<code>png:filter</code>	int	Controls the “row filters” that prepare the image for optimal compression. The default is 0 (PNG_NO_FILTERS), but other values (which may be “or-ed” or summed to combine their effects) are 8 (PNG_FILTER_NONE), 16 (PNG_FILTER_SUB), 32 (PNG_FILTER_UP), 64 (PNG_FILTER_AVG), or 128 (PNG_FILTER_PAETH).
<code>oiio:ioproxy</code>	ptr	Pointer to a <code>Filesystem::IOProxy</code> that will handle the I/O, for example by writing to a memory buffer.
<code>oiio:dither</code>	int	If nonzero and outputting UINT8 values in the file, will add a small amount of random dither to combat the appearance of banding

### Custom I/O Overrides

PNG output supports the “custom I/O” feature via the special "`oiio:ioproxy`" attributes (see Section [sec-imageoutput-ioproxy](#)).

### 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.

## 6.20 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	PNM 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 <code>format</code> as <code>UINT8</code> ).
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.

## 6.21 PSD

PSD is the file format used for storing Adobe PhotoShop images. OpenImageIO provides limited read abilities for PSD, but not currently the ability to write PSD files.

### Configuration settings for PSD input

When opening an ImageInput with a *configuration* (see Section `sec-inputwithconfig`), the following special configuration options are supported:

Input Configuration Attribute	Type	Meaning
oio:RawColor	int	If nonzero, reading images with non-RGB color models (such as YCbCr or CMYK) will return unaltered pixel values (versus the default OIIO behavior of automatically converting to RGB).

Currently, the PSD format reader supports color modes RGB, CMYK, multichannel, grayscale, indexed, and bitmap. It does NOT currently support Lab or duotone modes.

## 6.22 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 <code>"triangle"</code> or <code>"quad"</code> .
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.

## 6.23 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/>).

### Configuration settings for RAW input

When opening an ImageInput with a *configuration* (see Section *sec-inputwithconfig*), the following special configuration options are supported:

Input Configuration Attribute	Type	Meaning
raw:auto_bright	int	If nonzero, will use libraw’s exposure correction. (Default: 0)
raw:use_camera_wb	int	If 1, use libraw’s camera white balance adjustment. (Default: 1)
raw:use_camera_matrix	int	Whether to use the embedded color profile, if it’s present: 0 = never, 1 (default) = only for DNG files, 3 = always.
raw:adjust_maximum_thr	float	If nonzero, auto-adjusting maximum value. (Default:0.0)
raw:user_sat	int	If nonzero, sets the camera maximum value that will be normalized to appear saturated. (Default: 0)
raw:aber	float[2]	Red and blue scale factors for chromatic aberration correction when decoding the raw image. The default (1,1) means to perform no correction. This is an overall spatial scale, sensible values will be very close to 1.0.
raw:half_size	int	If nonzero, outputs the image in half size. (Default: 0)
raw:user_mul	float[4]	Sets user white balance coefficients. Only applies if raw:use_camera_wb is not equal to 0.
raw:ColorSpace	string	Which color primaries to use: raw, sRGB, Adobe, Wide, ProPhoto, ACES, XYZ. (Default: sRGB)
raw:Exposure	float	Amount of exposure before de-mosaicing, from 0.25 (2 stop darken) to 8 (3 stop brighten). (Default: 0, meaning no correction.)
raw:Demosaic	string	Force a demosaicing algorithm: linear, VNG, PPG, AHD (default), DCB, AHD-Mod, AFD, VCD, Mixed, LMMSE, AMaZE, DHT, AAHD, none.
raw:HighlightMode	int	Set libraw highlight mode processing: 0 = clip, 1 = unclip, 2 = blend, 3+ = rebuild. (Default: 0.)

## 6.24 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	int	RLA “active/viewable” window.
full_width, full_height, full_x, full_y	int	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 <code>rgb</code> ).
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.
oiio:ColorSpace	string	Color space (see Section <a href="#">Color information</a> ).
oiio:Gamma	float	the gamma correction value (if specified).

### Limitations

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

## 6.25 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
<code>compression</code>	string	The compression of the SGI file ( <code>rle</code> , if RLE compression is used).
<code>ImageDescription</code>	string	Image name.

## 6.26 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
<code>compression</code>	string	The compression of the SGI file ( <code>rle</code> , if RLE compression is used).
<code>ImageDescription</code>	string	Comment
<code>oiio:BitsPerSample</code>	int	the true bits per sample of the PIC file.



## 6.27 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 <code>none</code> and <code>rle</code> 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
oiio:BitsPerSample	int	the true bits per sample of the PIC file.
oiio:ColorSpace	string	Color space (see Section <a href="#">Color information</a> ).
oiio:Gamma	float	the gamma correction value (if specified).

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.

## 6.28 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 (`tiled`) 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 "minisblack").

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

### Configuration settings for TIFF input

When opening an `ImageInput` with a *configuration* (see Section `sec-inputwithconfig`), the following special configuration options are supported:

Input Configuration Attribute	Type	Meaning
<code>oiio:UnassociatedAlpha</code>	int	If nonzero, will leave alpha unassociated (versus the default of pre-multiplying color channels by alpha if the alpha channel is unassociated).
<code>oiio:RawColor</code>	int	If nonzero, reading images with non-RGB color models (such as YCbCr) will return unaltered pixel values (versus the default OIIO behavior of automatically converting to RGB).

### Configuration settings for TIFF output

When opening an `ImageOutput`, the following special metadata tokens control aspects of the writing itself:

Output Configuration Attribute	Type	Meaning
<code>oiio:UnassociatedAlpha</code>	int	If nonzero, any alpha channel is understood to be unassociated, and the <code>EXTRASAMPLES</code> tag in the TIFF file will be set to reflect this).
<code>oiio:BitsPerSample</code>	int	Requests a rescaling to a specific bits per sample (such as writing 12-bit TIFFs).
<code>tiff:write_exif</code>	int	If zero, will not write any Exif data to the TIFF file. (The default is 1.)
<code>tiff:half</code>	int	If nonzero, allow writing TIFF files with <code>half</code> (16 bit float) pixels. The default of 0 will automatically translate to float pixels, since most non-OIIO applications will not properly read half TIFF files despite their being legal.
<code>tiff:ColorSpace</code>	string	Requests that the file be saved with a non-RGB color spaces. Choices are RGB, CMYK, %, YCbCr, CIELAB, ICCLAB, ITULAB.
<code>tiff:zipquality</code>	int	A time-vs-quality knob for zip compression, ranging from 1-9 (default is 6). Higher means compress to less space, but taking longer to do so. It is strictly a time vs space tradeoff, the quality is identical (lossless) no matter what the setting.
<code>tiff:RowsPerStrip</code>	int	Overrides TIFF scanline rows per strip with a specific request (if not supplied, OIIO will choose a reasonable default).

### TIFF compression modes

The full list of possible TIFF compression mode values are as follows (\$ ^\*\$ indicates that OpenImageIO can write that format, and is not part of the format name):

```

none $ ^*$ lzw $ ^*$ zip $ ^*$ ccitt_t4 ccitt_t6 ccittfax3 ccittfax4 ccittrle2
ccittrle $ ^*$ dcs isojbig IT8BL IT8CTPAD IT8LW IT8MP jp2000 jpeg $ ^*$
lzma next ojpeg packbits $ ^*$ pixarfilm pixarlog sgilog24 sgilog T43 T85
thunderscan

```

## 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 passed by OIIO as 8 bit images; 12 bits per pixel will be passed as 16, etc. But the `oiio:BitsPerSample` attribute in the `ImageSpec` will correctly report the original bit depth of the file. Similarly for output, you must pass 8 or 16 bit output, but `oiio:BitsPerSample` gives a hint about how you want it to be when written to the file, and it will try to accommodate the request (for signed integers, TIFF output can accommodate 2, 4, 8, 10, 12, and 16 bits).
- JPEG compression is limited to 8-bit per channel, 3-channel files.

## TIFF Attributes

ImageSpec Attribute	Type	TIFF header data or explanation
<code>ImageSpec::x</code>	int	XPosition
<code>ImageSpec::y</code>	int	YPosition
<code>ImageSpec::full_width</code>	int	PIXAR_IMAGEFULLWIDTH
<code>ImageSpec::full_length</code>	int	PIXAR_IMAGEFULLLENGTH
<code>ImageDescription</code>	string	ImageDescription
<code>DateTime</code>	string	DateTime
<code>Software</code>	string	Software
<code>Artist</code>	string	Artist
<code>Copyright</code>	string	Copyright
<code>Make</code>	string	Make
<code>Model</code>	string	Model
<code>DocumentName</code>	string	DocumentName
<code>HostComputer</code>	string	HostComputer
<code>XResolution,</code> <code>YResolution</code>	float	XResolution, YResolution
<code>ResolutionUnit</code>	string	ResolutionUnit (in or cm).
<code>Orientation</code>	int	Orientation
<code>ICCProfile</code>	uint8[]	The ICC color profile
<code>textureformat</code>	string	PIXAR_TEXTUREFORMAT
<code>wrapmodes</code>	string	PIXAR_WRAPMODES
<code>fovcot</code>	float	PIXAR_FOVCOT
<code>worldtocamera</code>	matrix	PIXAR_MATRIX_WORLDTOCAMERA
<code>worldtoscreen</code>	matrix	PIXAR_MATRIX_WORLDTOSCREEN
<code>compression</code>	string	based on TIFF Compression (one of none, lzw, zip, or others listed above).
<code>tiff:compression</code>	int	the original integer code from the TIFF Compression tag.
<code>tiff:planarconfig</code>	string	PlanarConfiguration (separate or contig). The OpenImageIO TIFF writer will honor such a request in the <code>ImageSpec</code> .
<code>tiff:PhotometricInterpretation</code>	int	Photometric
<code>tiff:PageName</code>	string	PageName
<code>tiff:PageNumber</code>	int	PageNumber
<code>tiff:RowsPerStrip</code>	int	RowsPerStrip

continues on next page

Table 2 – continued from previous page

ImageSpec Attribute	Type	TIFF header data or explanation
tiff:subfiletype	int	SubfileType
Exif:*		A wide variety of EXIF data are honored, and are all prefixed with Exif.
oiio:BitsPerSample	int	The actual bits per sample in the file (may differ from ImageSpec::format).
oiio:UnassociatedAlpha	int	Nonzero if the alpha channel contained “unassociated” alpha.

## 6.29 Webp

FIXME

## 6.30 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

## 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>
using namespace OIIO;

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

(continues on next page)

(continued from previous page)

```

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

### class ImageCache

Define an API to an abstract class that manages image files, caches of open file handles as well as tiles of pixels so that truly huge amounts of image data may be accessed by an application with low memory footprint.

### 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* \***create** (bool *shared* = true)

Create a *ImageCache* and return a raw pointer to it. This should only be freed by passing it to *ImageCache::destroy()*!

**Return** A raw pointer to an *ImageCache*, which can only be freed with *ImageCache::destroy()*.

**See** *ImageCache::destroy*

#### Parameters

- *shared*: If `true`, the pointer returned will be 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 **destroy** (*ImageCache* \**cache*, bool *teardown* = false)

Destroy an allocated *ImageCache*, including freeing all system resources that it holds.

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.

## Parameters

- `cache`: Raw pointer to the *ImageCache* to destroy.
- `teardown`: 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). This parameter has no effect if `cache` was not the single globally shared *ImageCache*.

## Setting options and limits for the image cache

These are the list of attributes that can be set or queried by `attribute/getattribute`:

- `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) used for the internal “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 filename that is not specified as an absolute path. (Default: “”)
- `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: “”)
- `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:

```
- `autotile * autotile`
    if `autoscanline` is 0
- `width * autotile`
    if `autoscanline` is nonzero.
```

In both cases, this should lead to 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 autoscanline`: `autotile` using full width tiles
- `int automip`: If 0 (the default), an untiled single-subimage file will only be able to utilize that single subimage. If 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 accept_untiled`: When nonzero, *ImageCache* accepts untiled images as usual. When 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. (default=1)
- `int accept_unmipped`: When nonzero, *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. (Default: 1)

- `int statistics:level` : verbosity of statistics auto-printed.
- `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 failure_retries` : 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. (Default: 1)
- `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.
- `int max_errors_per_file` : The maximum number of errors that will be printed for each file. The default is 100. If your output is cluttered with error messages and after the first few for each file you aren't getting any helpful additional information, this can cut down on the clutter and the runtime. (default: 100)
- `string options` This catch-all is simply a comma-separated list of `name=value` settings of named options, which will be parsed and individually set. Example:

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

Note that if an option takes a string value that must itself contain a comma, it is permissible to enclose the value in either single ( `' '` ) or double ( `" "` ) quotes.

### Read-only attributes

Additionally, there are some read-only attributes that can be queried with *getAttribute()* even though they cannot be set via *attribute()*:

- `int total_files` : The total number of unique file names referenced by calls to the *ImageCache*.
- `string[] all_filenames` : An array that will be filled with the list of the names of all files referenced by calls to the *ImageCache*. (The array is of `usttring` or `char*`.)
- `int64 stat:cache_memory_used` : Total bytes used by tile cache.
- `int stat:tiles_created`, `int stat:tiles_current`, `int stat:tiles_peak` : Total times created, still allocated (at the time of the query), and the peak number of tiles in memory at any time.
- `int stat:open_files_created` , `int stat:open_files_current` , `int stat:open_files_peak` : Total number of times a file was opened, number still opened (at the time of the query), and the peak number of files opened at any time.
- `int stat:find_tile_calls` : Number of times a filename was looked up in the file cache.
- `int64 stat:image_size` : Total size (uncompressed bytes of pixel data) of all images referenced by the *ImageCache*. (Note: Prior to 1.7, this was called `stat:files_totalsize`.)



- `int64 stat:file_size` : Total size of all files (as on disk, possibly compressed) of all images referenced by the *ImageCache*.
- `int64 stat:bytes_read` : Total size (uncompressed bytes of pixel data) read.
- `int stat:unique_files` : Number of unique files opened.
- `float stat:fileio_time` : Total I/O-related time (seconds).
- `float stat:fileopen_time` : I/O time related to opening and reading headers (but not pixel I/O).
- `float stat:file_locking_time` : Total time (across all threads) that threads blocked waiting for access to the file data structures.
- `float stat:tile_locking_time` : Total time (across all threads) that threads blocked waiting for access to the tile cache data structures.
- `float stat:find_file_time` : Total time (across all threads) that threads spent looking up files by name.
- `float stat:find_tile_time` : Total time (across all threads) that threads spent looking up individual tiles.

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:

**virtual bool attribute** (*string\_view* name, *TypeDesc* type, **const** void \*val) = 0

Set a named attribute (i.e., a property or option) of the *ImageCache*.

Example:

```
ImageCache *ic;
...
int maxfiles = 50;
ic->attribute ("max_open_files", TypeDesc::INT, &maxfiles);

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

// There are specialized versions for setting a single int,
// float, or string without needing types or pointers:
ic->attribute ("max_open_files", 50);
ic->attribute ("max_memory_MB", 4000.0f);
ic->attribute ("searchpath", "/my/path");
```

Note: 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**`).

**Return** `true` if the name and type were recognized and the attribute was set, or `false` upon failure (including it being an unrecognized attribute or not of the correct type).

#### Parameters

- name: Name of the attribute to set.
- type: *TypeDesc* describing the type of the attribute.
- val: Pointer to the value data.

**virtual bool attribute** (*string\_view* name, `int` val) = 0

Specialized *attribute()* for setting a single `int` value.

**virtual bool attribute** (*string\_view* name, float val) = 0  
Specialized *attribute()* for setting a single float value.

**virtual bool attribute** (*string\_view* name, *string\_view* val) = 0  
Specialized *attribute()* for setting a single string value.

**virtual bool getattribute** (*string\_view* name, *TypeDesc* type, void \*val) **const** = 0  
Get the named attribute, store it in \*val. All of the attributes that may be set with the *attribute()* call may also be queried with *getattribute()*.

Examples:

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

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

// There are specialized versions for retrieving a single int,
// float, or string without needing types or pointers:
int maxfiles;
ic->getattribute ("max_open_files", maxfiles);
const char *path;
ic->getattribute ("searchpath", &path);
```

Note: When retrieving 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.

**Return** `true` if the name and type were recognized and the attribute was retrieved, or `false` upon failure (including it being an unrecognized attribute or not of the correct type).

#### Parameters

- name: Name of the attribute to retrieve.
- type: *TypeDesc* describing the type of the attribute.
- val: Pointer where the attribute value should be stored.

**virtual bool getattribute** (*string\_view* name, int &val) **const** = 0  
Specialized *attribute()* for retrieving a single int value.

**virtual bool getattribute** (*string\_view* name, float &val) **const** = 0  
Specialized *attribute()* for retrieving a single float value.

**virtual bool getattribute** (*string\_view* name, char \*\*val) **const** = 0  
Specialized *attribute()* for retrieving a single string value as a `char*`.

**virtual bool getattribute** (*string\_view* name, std::string &val) **const** = 0  
Specialized *attribute()* for retrieving a single string value as a `std::string`.

## Opaque data for performance lookups

The *ImageCache* implementation needs to maintain certain per-thread state, and some methods take an opaque *Perthread* pointer to this record. There are three options for how to deal with it:

1. Don't worry about it at all: don't use the methods that want *Perthread* pointers, or always pass *nullptr* for any *Perthread\** arguments, and *ImageCache* will do thread-specific-pointer retrieval as necessary (though at some small cost).
2. If your app already stores per-thread information of its own, you may call `get_perthread_info(nullptr)` to retrieve it for that thread, and then pass it into the functions that allow it (thus sparing them the need and expense of retrieving the thread-specific pointer). However, it is crucial that this pointer not be shared between multiple threads. In this case, the *ImageCache* manages the storage, which will automatically be released when the thread terminates.
3. If your app also wants to manage the storage of the *Perthread*, it can explicitly create one with `create_perthread_info()`, pass it around, and eventually be responsible for destroying it with `destroy_perthread_info()`. When managing the storage, the app may reuse the *Perthread* for another thread after the first is terminated, but still may not use the same *Perthread* for two threads running concurrently.

**typedef pvt::ImageCachePerThreadInfo *Perthread***

Define an opaque data type that allows us to have a pointer to certain per-thread information that the *ImageCache* maintains. Any given one of these should NEVER be shared between running threads.

**typedef pvt::ImageCacheFile *ImageHandle***

Define an opaque data type that allows us to have a handle to an image (already having its name resolved) but without exposing any internals.

**virtual *Perthread* \*get\_perthread\_info(*Perthread* \*thread\_info = NULL) = 0**

Retrieve a *Perthread*, unique to the calling thread. This is a thread-specific pointer that will always return the *Perthread* for a thread, which will also be automatically destroyed when the thread terminates.

Applications that want to manage their own *Perthread* pointers (with `create_thread_info` and `destroy_thread_info`) should still call this, but passing in their managed pointer. If the passed-in *thread\_info* is not NULL, it won't create a new one or retrieve a TSP, but it will do other necessary housekeeping on the *Perthread* information.

**virtual *Perthread* \*create\_thread\_info() = 0**

Create a new *Perthread*. It is the caller's responsibility to eventually destroy it using `destroy_thread_info()`.

**virtual void destroy\_thread\_info(*Perthread* \*thread\_info) = 0**

Destroy a *Perthread* that was allocated by `create_thread_info()`.

**virtual *ImageHandle* \*get\_image\_handle(*ustring* filename, *Perthread* \*thread\_info = NULL) = 0**

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

**virtual bool good(*ImageHandle* \*file) = 0**

Return true if the image handle (previously returned by `get_image_handle()`) is a valid image that can be subsequently read.

## Getting information about images

**virtual** std::string **resolve\_filename**(const std::string &filename) const = 0

Given possibly-relative filename, resolve it and use the true path to the file, with searchpath logic applied.

**virtual** bool **get\_image\_info**(ustring filename, int subimage, int miplevel, ustring dataname, TypeDesc datatype, void \*data) = 0

Get information or metadata about the named image and store it in \*data.

Data names may include any of the following:

- "exists" : Stores the value 1 (as an int) if the file exists and is an image format that OpenImageIO can read, or 0 if the file does not exist, or could not be properly read as an image. Note that unlike all other queries, this query will "succeed" (return true) even if the file does not exist.
- "udim" : Stores the value 1 (as an int) if the file is a "virtual UDIM" or texture atlas file (as described in :ref:sec-texturesys-udim) or 0 otherwise.
- "subimages" : The number of subimages in the file, as an int.
- "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 int).
- "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).
- "worldtocamera" : The viewing matrix, which is a 4x4 matrix (an Imath::M44f, described as TypeDesc(FLOAT, MATRIX)), giving the world-to-camera 3D transformation matrix that was used when the image was created. Generally, only rendered images will have this.

- "worldtoscreen" : The projection matrix, which is a 4x4 matrix (an `Imath::M44f`, described as `TypeDesc(FLOAT, MATRIX)`), giving the matrix that projected points from world space into a 2D screen coordinate system where  $x$  and  $y$  range from -1 to +1. Generally, only rendered images will have this.
- "averagecolor" : If available in the metadata (generally only for files that have been processed by `maketx`), this will return the average color of the texture (into an array of `float`).
- "averagealpha" : If available in the metadata (generally only for files that have been processed by `maketx`), this will return the average alpha value of the texture (into a `float`).
- "constantcolor" : If the metadata (generally only for files that have been processed by `maketx`) indicates that the texture has the same values for all pixels in the texture, this will retrieve the constant color of the texture (into an array of floats). A non-constant image (or one that does not have the special metadata tag identifying it as a constant texture) will fail this query (return `false`).
- "constantalpha" : If the metadata indicates that the texture has the same values for all pixels in the texture, this will retrieve the constant alpha value of the texture (into a `float`). A non-constant image (or one that does not have the special metadata tag identifying it as a constant texture) will fail this query (return `false`).
- "stat:tilesread" : Number of tiles read from this file (`int64`).
- "stat:bytesread" : Number of bytes of uncompressed pixel data read from this file (`int64`).
- "stat:redundant\_tiles" : Number of times a tile was read, where the same tile had been read before. (`int64`).
- "stat:redundant\_bytesread" : Number of bytes (of uncompressed pixel data) in tiles that were read redundantly. (`int64`).
- "stat:redundant\_tilesread" : Number of tiles read from this file (`int`).
- "stat:image\_size" : Size of the uncompressed image pixel data of this image, in bytes (`int64`).
- "stat:file\_size" : Size of the disk file (possibly compressed) for this image, in bytes (`int64`).
- "stat:timesopened" : Number of times this file was opened (`int`).
- "stat:iotime" : Time (in seconds) spent on all I/O for this file (`float`).
- "stat:mipsused" : Stores 1 if any MIP levels beyond the highest resolution were accessed, otherwise 0. (`int`)
- "stat:is\_duplicate" : Stores 1 if this file was a duplicate of another image, otherwise 0. (`int`)
- *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.

**Return** true if `get_image_info()` is able to find the requested dataname for the image and it matched the requested datatype. If the requested data was not found or was not of the right data type, return `false`. Except for the "exists" query, a file that does not exist or could not be read properly as an image also constitutes a query failure that will return `false`.

#### Parameters

- filename: The name of the image.
- subimage/miplevel: The subimage and MIP level to query.
- dataname: The name of the metadata to retrieve.

- datatype: *TypeDesc* describing the data type.
- data: Pointer to the caller-owned memory where the values should be stored. It is the caller's responsibility to ensure that data points to a large enough storage area to accommodate the datatype requested.

**virtual bool get\_image\_info** (*ImageHandle* \*file, *Perthread* \*thread\_info, int subimage, int miplevel, *ustring* dataname, *TypeDesc* datatype, void \*data) = 0

A more efficient variety of *get\_image\_info()* for cases where you can use an *ImageHandle\** to specify the image and optionally have a *Perthread\** for the calling thread.

**virtual bool get\_imagespec** (*ustring* filename, *ImageSpec* &spec, int subimage = 0, int miplevel = 0, bool native = false) = 0

Copy the *ImageSpec* associated with the named image (the first subimage & miplevel by default, or as set by subimage and miplevel).

**Return** true upon success, false upon failure (such as being unable to find, open, or read the file, or if it does not contain the designated subimage or MIP level).

#### Parameters

- filename: The name of the image.
- spec: *ImageSpec* into which will be copied the spec for the requested image.
- subimage/miplevel: The subimage and MIP level to query.
- native: If 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 contents of the original file. These may differ due to use of certain *ImageCache* settings such as "forcefloat" or "autotile".

**virtual bool get\_imagespec** (*ImageHandle* \*file, *Perthread* \*thread\_info, *ImageSpec* &spec, int subimage = 0, int miplevel = 0, bool native = false) = 0

A more efficient variety of *get\_imagespec()* for cases where you can use an *ImageHandle\** to specify the image and optionally have a *Perthread\** for the calling thread.

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

Return a pointer to an *ImageSpec* associated with the named image (the first subimage & MIP level by default, or as set by subimage and miplevel) if the file is found and is an image format that can be read, otherwise return nullptr.

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*.

**Return** A pointer to the spec, if the image is found and able to be opened and read by an available image format plugin, and the designated subimage and MIP level exists.

#### Parameters

- filename: The name of the image.
- subimage/miplevel: The subimage and MIP level to query.
- native: If 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 contents of the original file. These may differ due to use of certain *ImageCache* settings such as "forcefloat" or "autotile".

```
virtual const ImageSpec *imagespec (ImageHandle *file, Perthread *thread_info, int subimage =
                                0, int miplevel = 0, bool native = false) = 0
```

A more efficient variety of *imagespec()* for cases where you can use an *ImageHandle\** to specify the image and optionally have a *Perthread\** for the calling thread.

## Getting Pixels

```
virtual bool get_pixels (ustring filename, int subimage, int miplevel, int xbegin, int xend, int ybe-
                        gin, 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, int cache_chbegin = 0, int cache_chend =
                        -1) = 0
```

For an image specified by name, retrieve the rectangle of pixels from the designated subimage and MIP level, storing the pixel values beginning at the address specified by *result* and with the given strides. The pixel values will be converted to the data type specified by *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.

**Return** *true* for success, *false* for failure.

### Parameters

- *filename*: The name of the image.
- *subimage/miplevel*: The subimage and MIP level to retrieve pixels from.
- *xbegin/xend/ybegin/yend/zbegin/zend*: The range of pixels to retrieve. The pixels retrieved include the *begin* value but not the end value (much like STL *begin/end* usage).
- *chbegin/chend*: Channel range to retrieve. To retrieve all channels, use *chbegin* = 0, *chend* = *nchannels*.
- *format*: *TypeDesc* describing the data type of the values you want to retrieve into *result*. The pixel values will be converted to this type regardless of how they were stored in the file.
- *result*: Pointer to the memory where the pixel values should be stored. 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).
- *xstride/ystride/zstride*: The number of bytes between the beginning of successive pixels, scanlines, and image planes, respectively. Any stride values set to *AutoStride* will be assumed to indicate a contiguous data layout in that dimension.
- *cache\_chbegin/cache\_chend*: These parameters can be used to tell the *ImageCache* to read and cache a subset of channels (if not specified or if they denote a non-positive range, all the channels of the file will be stored in the cached tile).

```
virtual bool get_pixels (ImageHandle *file, Perthread *thread_info, int subimage, int miplevel,
                        int xbegin, int xend, int ybegin, int yend, int zbegin, int zend, int chbe-
                        gin, int chend, TypeDesc format, void *result, stride_t xstride = Au-
                        toStride, stride_t ystride = AutoStride, stride_t zstride = AutoStride, int
                        cache_chbegin = 0, int cache_chend = -1) = 0
```

A more efficient variety of *get\_pixels()* for cases where you can use an *ImageHandle\** to specify the image and optionally have a *Perthread\** for the calling thread.

```
virtual bool get_pixels (ustring filename, int subimage, int miplevel, int xbegin, int xend, int ybe-
                        gin, int yend, int zbegin, int zend, TypeDesc format, void *result) = 0
```

A simplified *get\_pixels()* where all channels are retrieved, strides are assumed to be contiguous.



```
virtual bool get_pixels (ImageHandle *file, Perthread *thread_info, int subimage, int miplevel,
                        int xbegin, int xend, int ybegin, int yend, int zbegin, int zend, TypeDesc
                        format, void *result) = 0
```

A more efficient variety of `get_pixels()` for cases where you can use an `ImageHandle*` to specify the image and optionally have a `Perthread*` for the calling thread.

## Controlling the cache

```
virtual void invalidate (ustring filename, bool force = true) = 0
```

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.

If `force` is true, this invalidation will happen unconditionally; if false, the file will only be invalidated if it has been changed since it was first opened by the *ImageCache*.

```
virtual void invalidate_all (bool force = false) = 0
```

Invalidate all loaded tiles and close open file handles. 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 (if the images change) 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.

```
virtual void close (ustring filename) = 0
```

Close any open file handles associated with a named file, but do not invalidate any image spec information or pixels associated with the files. A client might do this in order to release OS file handle resources, or to make it safe for other processes to modify image files on disk.

```
virtual void close_all () = 0
```

`close()` all files known to the cache.

```
virtual Tile *get_tile (ustring filename, int subimage, int miplevel, int x, int y, int z, int chbegin =
                        0, int chend = -1) = 0
```

Find the tile specified by an image filename, subimage & miplevel, the coordinates of a pixel, and optionally a channel range. An opaque pointer to the tile will be returned, or `nullptr` 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 the same number of times that `get_tile()` was called (reference counting). This is thread-safe! If `chend < chbegin`, it will retrieve a tile containing all channels in the file.

```
virtual Tile *get_tile (ImageHandle *file, Perthread *thread_info, int subimage, int miplevel, int
                        x, int y, int z, int chbegin = 0, int chend = -1) = 0
```

A slightly more efficient variety of `get_tile()` for cases where you can use an `ImageHandle*` to specify the image and optionally have a `Perthread*` for the calling thread.

See `get_pixels()`

```
virtual void release_tile (Tile *tile) const = 0
```

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



**virtual** *TypeDesc* **tile\_format** (**const** Tile \*tile) **const** = 0

Retrieve the data type of the pixels stored in the tile, which may be different than the type of the pixels in the disk file.

**virtual** *ROI* **tile\_roi** (**const** Tile \*tile) **const** = 0

Retrieve the *ROI* describing the pixels and channels stored in the tile.

**virtual const** void \***tile\_pixels** (Tile \*tile, *TypeDesc* &format) **const** = 0

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()*.

**virtual** bool **add\_file** (*ustring* filename, *ImageInput::Creator* creator = nullptr, **const** *ImageSpec* \*config = nullptr, bool replace = false) = 0

The *add\_file()* call causes a file to be opened or added to the cache. There is no reason to use this method unless you are supplying a custom creator, or configuration, or both.

If creator is not NULL, it points to an *ImageInput::Creator* that will be used rather than the default *ImageInput::create()*, thus instead of reading from disk, creates and uses a custom *ImageInput* to generate the image. The ‘creator’ is a factory that creates the custom *ImageInput* and will be called like this:

```
std::unique_ptr<ImageInput> in (creator());
```

Once created, the *ImageCache* owns the *ImageInput* and is responsible for destroying it when done. Custom *ImageInputs* 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).

If config is not NULL, it points to an *ImageSpec* with configuration options/hints that will be passed to the underlying *ImageInput::open()* call. Thus, this can be used to ensure that the *ImageCache* opens a call with special configuration options.

This call (including any custom creator or configuration hints) will have no effect if there’s already an image by the same name in the cache. Custom creators or configurations only “work” the FIRST time a particular filename is referenced in the lifetime of the *ImageCache*. But if replace is true, any existing entry will be invalidated, closed and overwritten. So any subsequent access will see the new file. Existing texture handles will still be valid.

**virtual** bool **add\_tile** (*ustring* filename, int subimage, int miplevel, int x, int y, int z, int chbegin, int chend, *TypeDesc* format, **const** void \*buffer, stride\_t xstride = AutoStride, stride\_t ystride = AutoStride, stride\_t zstride = AutoStride, bool copy = true) = 0

Preemptively add a tile corresponding to the named image, at the given subimage, MIP level, and channel range. 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. If chend < chbegin, it will add a tile containing the full set of channels for the image. Note that if the ‘copy’ flag is false, the data is assumed to be in some kind of persistent storage and will not be copied, nor will its pixels take up additional memory in the cache.

## Errors and statistics

**virtual** std::string **geterror** () **const** = 0

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

**virtual** std::string **getstats** (int *level* = 1) **const** = 0

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.

**virtual** void **reset\_stats** () = 0

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*.

## TEXTURE ACCESS: TEXTURESYSYSTEM

### 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 4x4 transformation matrices. 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 firstchannel`: The beginning channel for the lookup. For example, to retrieve just the blue channel, you should have `firstchannel = 2` while passing `nchannels = 1` to the appropriate texture function.
- `int subimage, ustring 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 (like TIFF or multi-part OpenEXR) 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 the Section about `TextureSystem` attributes.
- `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 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`: Specifies wrap, blur, and width for the third component of 3D volume texture lookups. These are not used for 2D texture or environment lookups.

## 8.3 TextureSystem API

### **class TextureSystem**

Define an API to an abstract class that manages texture files, caches of open file handles as well as tiles of texels so that truly huge amounts of texture may be accessed by an application with low memory footprint, and ways to perform antialiased texture, shadow map, and environment map lookups.

## Creating and destroying a texture system

*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* \*create** (bool *shared* = true, *ImageCache* \**imagecache* = nullptr)

Create a *TextureSystem* and return a pointer to it. This should only be freed by passing it to *TextureSystem::destroy()*!

**Return** A raw pointer to a *TextureSystem*, which can only be freed with *TextureSystem::destroy()*.

See *TextureSystem::destroy*

### Parameters

- *shared*: If *shared* is true, the pointer returned will be a shared *TextureSystem*, (so that multiple parts of an application that request a *TextureSystem* will all end up with the same one, and the same underlying *ImageCache*). If *shared* is false, a completely unique *TextureCache* will be created and returned.
- *imagecache*: If *shared* is false and *imagecache* is not nullptr, the *TextureSystem* will use this as its underlying *ImageCache*. In that case, it is the caller who is responsible for eventually freeing the *ImageCache* after the *TextureSystem* is destroyed. If *shared* is false and *imagecache* is nullptr, then a custom *ImageCache* will be created, owned by the *TextureSystem*, and automatically freed when the TS destroys.

**static void destroy** (*TextureSystem* \**ts*, bool *teardown\_imagecache* = false)

Destroy an allocated *TextureSystem*, including freeing all system resources that it holds.

It is safe to destroy even a shared *TextureSystem*, 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 *TextureSystem*'s were created.

### Parameters

- *ts*: Raw pointer to the *TextureSystem* to destroy.
- *teardown\_imagecache*: For a shared *TextureSystem*, if the *teardown\_imagecache* 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). This parameter has no effect if *ts* was not the single globally shared *TextureSystem*.

## Setting options and limits for the texture system

These are the list of attributes that can be set or queried by *attribute/getattribute*:

All attributes ordinarily recognized by *ImageCache* are accepted and passed through to the underlying *ImageCache*. These include:

- `int max_open_files` : Maximum number of file handles held open.
- `float max_memory_MB` : Maximum tile cache size, in MB.
- `string searchpath` : Colon-separated search path for texture files.
- `string plugin_searchpath` : Colon-separated search path for plugins.
- `int autotile` : If >0, tile size to emulate for non-tiled images.

- `int autoscanline`: If nonzero, autotile using full width tiles.
- `int automip`: If nonzero, emulate mipmap on the fly.
- `int accept_untiled`: If nonzero, accept untiled images.
- `int accept_unmipped`: If nonzero, accept unmipped images.
- `int failure_retries`: How many times to retry a read failure.
- `int deduplicate`: If nonzero, detect duplicate textures (default=1).
- `string substitute_image`: If supplied, an image to substitute for all texture references.
- `int max_errors_per_file`: Limits how many errors to issue for each file. (default: 100)

#### Texture-specific settings:

- `matrix44 worldtocommon / matrix44 commontoworld`: The 4x4 matrices that provide the spatial transformation from “world” to a “common” coordinate system and back. This is mainly 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. 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.
- `int max_tile_channels`: The maximum number of color channels in a texture file for which all channels will be loaded as a single cached tile. Files with more than this number of color channels will have only the requested subset loaded, in order to save cache space (but at the possible wasted expense of separate tiles that overlap their channel ranges). The default is 5.
- `int max_mip_res`: **NEW 2.1** Sets the maximum MIP-map resolution for filtered texture lookups. The MIP levels used will be clamped to those having fewer than this number of pixels in each dimension. This can be helpful as a way to limit disk I/O when doing fast preview renders (with the tradeoff that you may see some texture more blurry than they would ideally be). The default is `1 << 30`, a value so large that no such clamping will be performed.
- `string latlong_up`: 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.”)
- `int flip_t`: If nonzero, `t` coordinates will be flipped `1-t` for all texture lookups. The default is 0.
- `string options` This catch-all is simply a comma-separated list of `name=value` settings of named options, which will be parsed and individually set.

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

Note that if an option takes a string value that must itself contain a comma, it is permissible to enclose the value in either single `'` or double `"` quotes.

#### Read-only attributes

Additionally, there are some read-only attributes that can be queried with `getattribute()` even though they cannot be set via `attribute()`:

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:

**virtual bool attribute** (*string\_view* name, *TypeDesc* type, **const** void \*val) = 0

Set a named attribute (i.e., a property or option) of the *TextureSystem*.

Example:

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

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

// There are specialized versions for retrieving a single int,
// float, or string without needing types or pointers:
ts->getattribute ("max_open_files", 50);
ic->attribute ("max_memory_MB", 4000.0f);
ic->attribute ("searchpath", "/my/path");
```

Note: 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**`).

**Return** `true` if the name and type were recognized and the attribute was set, or `false` upon failure (including it being an unrecognized attribute or not of the correct type).

#### Parameters

- name: Name of the attribute to set.
- type: *TypeDesc* describing the type of the attribute.
- val: Pointer to the value data.

**virtual bool attribute** (*string\_view* name, `int` val) = 0  
Specialized *attribute()* for setting a single `int` value.

**virtual bool attribute** (*string\_view* name, `float` val) = 0  
Specialized *attribute()* for setting a single `float` value.

**virtual bool attribute** (*string\_view* name, *string\_view* val) = 0  
Specialized *attribute()* for setting a single string value.

**virtual bool getattribute** (*string\_view* name, *TypeDesc* type, void \*val) **const** = 0

Get the named attribute of the texture system, store it in \*val. All of the attributes that may be set with the *attribute()* call may also be queried with *getattribute()*.

Examples:

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

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

// There are specialized versions for retrieving a single int,
// float, or string without needing types or pointers:
```

(continues on next page)

(continued from previous page)

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

Note: When retrieving 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.

**Return** `true` if the name and type were recognized and the attribute was retrieved, or `false` upon failure (including it being an unrecognized attribute or not of the correct type).

#### Parameters

- `name`: Name of the attribute to retrieve.
- `type`: *TypeDesc* describing the type of the attribute.
- `val`: Pointer where the attribute value should be stored.

**virtual bool getattribute** (*string\_view* name, int &val) **const** = 0

Specialized *attribute()* for retrieving a single `int` value.

**virtual bool getattribute** (*string\_view* name, float &val) **const** = 0

Specialized *attribute()* for retrieving a single `float` value.

**virtual bool getattribute** (*string\_view* name, char \*\*val) **const** = 0

Specialized *attribute()* for retrieving a single `string` value as a `char*`.

**virtual bool getattribute** (*string\_view* name, std::string &val) **const** = 0

Specialized *attribute()* for retrieving a single `string` value as a `std::string`.

## Opaque data for performance lookups

The *TextureSystem* implementation needs to maintain certain per-thread state, and some methods take an opaque `Perthread` pointer to this record. There are three options for how to deal with it:

1. Don't worry about it at all: don't use the methods that want `Perthread` pointers, or always pass `nullptr` for any ``Perthread*1` arguments, and *ImageCache* will do thread-specific-pointer retrieval as necessary (though at some small cost).
2. If your app already stores per-thread information of its own, you may call `get_perthread_info(nullptr)` to retrieve it for that thread, and then pass it into the functions that allow it (thus sparing them the need and expense of retrieving the thread-specific pointer). However, it is crucial that this pointer not be shared between multiple threads. In this case, the *ImageCache* manages the storage, which will automatically be released when the thread terminates.
3. If your app also wants to manage the storage of the `Perthread`, it can explicitly create one with `create_perthread_info()`, pass it around, and eventually be responsible for destroying it with `destroy_perthread_info()`. When managing the storage, the app may reuse the `Perthread` for another thread after the first is terminated, but still may not use the same `Perthread` for two threads running concurrently.

**virtual Perthread \*get\_perthread\_info** (Perthread \*thread\_info = nullptr) = 0

Retrieve a `Perthread`, unique to the calling thread. This is a thread-specific pointer that will always return the `Perthread` for a thread, which will also be automatically destroyed when the thread terminates.



Applications that want to manage their own Perthread pointers (with `create_thread_info` and `destroy_thread_info`) should still call this, but passing in their managed pointer. If the passed-in `thread_info` is not `nullptr`, it won't create a new one or retrieve a TSP, but it will do other necessary housekeeping on the Perthread information.

**virtual** Perthread \***create\_thread\_info**() = 0

Create a new Perthread. It is the caller's responsibility to eventually destroy it using `destroy_thread_info()`.

**virtual** void **destroy\_thread\_info**(Perthread \**threadinfo*) = 0

Destroy a Perthread that was allocated by `create_thread_info()`.

**virtual** TextureHandle \***get\_texture\_handle**(*ustring filename*, Perthread \**thread\_info* = `nullptr`) = 0

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

**virtual** bool **good**(TextureHandle \**texture\_handle*) = 0

Return true if the texture handle (previously returned by `get_image_handle()`) is a valid texture that can be subsequently read.

## Texture lookups

**virtual** bool **texture**(*ustring filename*, TextureOpt &*options*, float *s*, float *t*, float *dsdx*, float *dt dx*, float *dsdy*, float *dt dy*, int *nchannels*, float \**result*, float \**dresultds* = `nullptr`, float \**dresultdt* = `nullptr`) = 0

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 *nchannels* parameter determines the number of channels to retrieve (e.g., 1 for a single value, 3 for an RGB triple, etc.). The filtered results will be stored in `result[0..nchannels-1]`.

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 *dt dx* that define the change in *s* and *t* per unit of *x*, and *dsdy* and *dt dy* 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.

**Return** true upon success, or false if the file was not found or could not be opened by any available ImageIO plugin.

### Parameters

- *filename*: The name of the texture.
- *options*: Fields within *options* that are honored for 2D texture lookups include the following:
  - int *firstchannel*: The index of the first channel to look up from the texture.
  - int *subimage* / *ustring subimagename*: The subimage or face within the file, specified by either by name (if non-empty) or index. 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" *TextureSystem* attribute.
- `const float *missingcolor`: If not `nullptr`, specifies the color that will be returned for missing or broken textures (rather than being an error).
- `s/t`: The 2D texture coordinates.
- `dsdxdt dxdsdy dtdy`: The differentials of `s` and `t` relative to canonical directions `x` and `y`. The choice of `x` and `y` are not important to the implementation; it can be any imposed 2D coordinates, such as pixels in screen space, adjacent samples in parameter space on a surface, etc. The `st` derivatives determine the size and shape of the ellipsoid over which the texture lookup is filtered.
- `nchannels`: The number of channels of data to retrieve into `result` (e.g., 1 for a single value, 3 for an RGB triple, etc.).
- `result[]`: The result of the filtered texture lookup will be placed into `result[0..nchannels-1]`.
- `dresultds/dresultdt`: If non-null, these designate storage locations for the derivatives of `result`, i.e., the rate of change per unit `s` and `t`, respectively, of the filtered texture. If supplied, they must allow for `nchannels` of storage.

**virtual bool texture** (`TextureHandle *texture_handle`, `PerThread *thread_info`, `TextureOpt &options`, `float s`, `float t`, `float dsdx`, `float dtdx`, `float dsdy`, `float dtdy`, `int nchannels`, `float *result`, `float *dresultds = nullptr`, `float *dresultdt = nullptr`) = 0

Slightly faster version of *texture()* lookup if the app already has a texture handle and per-thread info.

**virtual bool texture3d** (`ustring filename`, `TextureOpt &options`, `const Imath::V3f &P`, `const Imath::V3f &dPdx`, `const Imath::V3f &dPdy`, `const Imath::V3f &dPdZ`, `int nchannels`, `float *result`, `float *dresultds = nullptr`, `float *dresultdt = nullptr`, `float *dresultdr = nullptr`) = 0

Perform a filtered 3D volumetric texture lookup on a position centered at 3D position `P` (with given differentials) from the texture identified by `filename`, and using relevant texture `options`. The filtered results will be stored in `result[0..nchannels-1]`.

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.

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

#### Parameters

- `filename`: The name of the texture.
- `options`: Fields within `options` that are honored for 3D texture lookups include the following:
  - `int firstchannel`: The index of the first channel to look up from the texture.
  - `int subimage / ustring subimagename`: The subimage or field within the volume, specified by either by name (if non-empty) or index. This will be ignored if the file does not have multiple subimages or separate per-face textures.

- `Wrap swrap, twrap, rwrap` : Specify the *wrap mode* 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`" *TextureSystem* attribute).
- `const float *missingcolor` : If not `nullptr`, 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).
- `P` : The 2D texture coordinates.
- `dPdx/dPdy/dPdz` : The differentials of `P`. 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.
- `nchannels` : The number of channels of data to retrieve into `result` (e.g., 1 for a single value, 3 for an RGB triple, etc.).
- `result[]` : The result of the filtered texture lookup will be placed into `result[0..nchannels-1]`.
- `dresultds/dresultdt/dresultdr` : If non-null, these designate storage locations for the derivatives of `result`, i.e., the rate of change per unit `s`, `t`, and `r`, respectively, of the filtered texture. If supplied, they must allow for `nchannels` of storage.

```
virtual bool texture3d (TextureHandle *texture_handle, Perthread *thread_info, TextureOpt &options,
    const Imath::V3f &P, const Imath::V3f &dPdx, const Imath::V3f &dPdy, const Imath::V3f &dPdz, int nchannels, float *result,
    float *dresultds = nullptr, float *dresultdt = nullptr, float *dresultdr = nullptr) = 0
```

Slightly faster version of *texture3d()* lookup if the app already has a texture handle and per-thread info.

```
virtual bool environment (ustring filename, TextureOpt &options, const Imath::V3f &R, const Imath::V3f &dRdx,
    const Imath::V3f &dRdy, int nchannels, float *result, float *dresultds = nullptr, float *dresultdt = nullptr) = 0
```

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[]`.

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

#### Parameters

- `filename` : The name of the texture.

- **options:** Fields within `options` that are honored for environment lookups include the following:
  - `int firstchannel`: The index of the first channel to look up from the texture.
  - `int subimage / ustring subimagename`: The subimage or face within the file, specified by either by name (if non-empty) or index. This will be ignored if the file does not have multiple subimages or separate per-face textures.
  - `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" *TextureSystem* attribute.
  - `const float *missingcolor`: If not `nullptr`, specifies the color that will be returned for missing or broken textures (rather than being an error).
- **R:** The direction vector to look up.
- **dRdx/dRdy:** The differentials of R with respect to image coordinates x and y.
- **nchannels:** The number of channels of data to retrieve into `result` (e.g., 1 for a single value, 3 for an RGB triple, etc.).
- **result[]:** The result of the filtered texture lookup will be placed into `result[0..nchannels-1]`.
- **dresultds/dresultdt:** If non-null, these designate storage locations for the derivatives of result, i.e., the rate of change per unit s and t, respectively, of the filtered texture. If supplied, they must allow for `nchannels` of storage.

**virtual bool environment** (`TextureHandle *texture_handle`, `Perthread *thread_info`, `TextureOpt &options`, `const Imath::V3f &R`, `const Imath::V3f &dRdx`, `const Imath::V3f &dRdy`, `int nchannels`, `float *result`, `float *dresultds` = `nullptr`, `float *dresultdt` = `nullptr`) = 0

Slightly faster version of *environment()* if the app already has a texture handle and per-thread info.

## Batched texture lookups

**virtual bool texture** (`ustring filename`, `TextureOptBatch &options`, `Tex::RunMask mask`, `const float *s`, `const float *t`, `const float *dsdx`, `const float *dtdx`, `const float *dsdy`, `const float *dtdy`, `int nchannels`, `float *result`, `float *dresultds` = `nullptr`, `float *dresultdt` = `nullptr`) = 0

Perform filtered 2D texture lookups on a batch of positions from the same texture, all at once. The parameters `s`, `t`, `dsdx`, `dtdx`, and `dsdy`, `dtdy` are each a pointer to `[BatchWidth]` values. The mask determines which of those array elements to actually compute.

The `float*` results act like `float[nchannels][BatchWidth]`, so that effectively `result[0..BatchWidth-1]` are the “red” result for each lane, `result[BatchWidth..2*BatchWidth-1]` are the “green” results, etc. The `dresultds` and `dresultdt` should either both be provided, or else both be `nullptr` (meaning no derivative results are required).

**Return** true upon success, or false if the file was not found or could not be opened by any available ImageIO plugin.

#### Parameters

- **filename:** The name of the texture.
- **options:** A TextureOptBatch containing texture lookup options. This is conceptually the same as a TextureOpt, but the following fields are arrays of [BatchWidth] elements: sblur, tblur, swidth, twidth. The other fields are, as with TextureOpt, ordinary scalar values.
- **mask:** A bit-field designating which “lanes” should be computed: if mask & (1<<i) is nonzero, then results should be computed and stored for result[...] [i].
- **s/t:** Pointers to the 2D texture coordinates, each as a float [BatchWidth].
- **dwdx/dtdx/dsdy/dtdy:** The differentials of s and t relative to canonical directions x and y, each as a float [BatchWidth].
- **nchannels:** The number of channels of data to retrieve into result (e.g., 1 for a single value, 3 for an RGB triple, etc.).
- **result[]:** The result of the filtered texture lookup will be placed here, as float [nchannels] [BatchWidth]. (Note the “SOA” data layout.)
- **dresultds/dresultdt:** If non-null, these designate storage locations for the derivatives of result, and like result are in SOA layout: float [nchannels] [BatchWidth]

**virtual bool texture** (TextureHandle \*texture\_handle, Perthread \*thread\_info, TextureOptBatch &options, Tex::RunMask mask, **const** float \*s, **const** float \*t, **const** float \*dwdx, **const** float \*dtdx, **const** float \*dsdy, **const** float \*dtdy, int nchannels, float \*result, float \*dresultds = nullptr, float \*dresultdt = nullptr) = 0

Slightly faster version of [texture\(\)](#) lookup if the app already has a texture handle and per-thread info.

**virtual bool texture3d** (ustring filename, TextureOptBatch &options, Tex::RunMask mask, **const** float \*P, **const** float \*dPdx, **const** float \*dPdy, **const** float \*dPdz, int nchannels, float \*result, float \*dresultds = nullptr, float \*dresultdt = nullptr, float \*dresultdr = nullptr) = 0

Perform filtered 3D volumetric texture lookups on a batch of positions from the same texture, all at once. The “point-like” parameters P, dPdx, dPdy, and dPdz are each a pointers to arrays of float value[3] [BatchWidth] (or alternately like Imath::Vec3<FloatWide>). That is, each one points to all the x values for the batch, immediately followed by all the y values, followed by the z values. The mask determines which of those array elements to actually compute.

The various results arrays are also arranged as arrays that behave as if they were declared float result[channels] [BatchWidth], where all the batch values for channel 0 are adjacent, followed by all the batch values for channel 1, etc.

**Return** true upon success, or false if the file was not found or could not be opened by any available ImageIO plugin.

#### Parameters

- **filename:** The name of the texture.
- **options:** A TextureOptBatch containing texture lookup options. This is conceptually the same as a TextureOpt, but the following fields are arrays of [BatchWidth] elements: sblur, tblur, swidth, twidth. The other fields are, as with TextureOpt, ordinary scalar values.
- **mask:** A bit-field designating which “lanes” should be computed: if mask & (1<<i) is nonzero, then results should be computed and stored for result[...] [i].

- `P`: Pointers to the 3D texture coordinates, each as a `float [3] [BatchWidth]`.
- `dPdx/dPdy/dPdz`: The differentials of `P` relative to canonical directions `x`, `y`, and `z`, each as a `float [3] [BatchWidth]`.
- `nchannels`: The number of channels of data to retrieve into `result` (e.g., 1 for a single value, 3 for an RGB triple, etc.).
- `result[]`: The result of the filtered texture lookup will be placed here, as `float [nchannels] [BatchWidth]`. (Note the “SOA” data layout.)
- `dresultds/dresultdt/dresultdr`: If non-null, these designate storage locations for the derivatives of `result`, and like `result` are in SOA layout: `float [nchannels] [BatchWidth]`

```
virtual bool texture3d (TextureHandle *texture_handle, Perthread *thread_info, TextureOptBatch
                        &options, Tex::RunMask mask, const float *P, const float *dPdx,
                        const float *dPdy, const float *dPdz, int nchannels, float *result, float
                        *dresultds = nullptr, float *dresultdt = nullptr, float *dresultdr = nullptr) =
0
```

Slightly faster version of `texture3d()` lookup if the app already has a texture handle and per-thread info.

```
virtual bool environment (ustring filename, TextureOptBatch &options, Tex::RunMask mask,
                        const float *R, const float *dRdx, const float *dRdy, int nchan-
                        nels, float *result, float *dresultds = nullptr, float *dresultdt = nullptr)
= 0
```

Perform filtered directional environment map lookups on a batch of directions from the same texture, all at once. The “point-like” parameters `R`, `dRdx`, and `dRdy` are each a pointers to arrays of `float value [3] [BatchWidth]` (or alternately like `Imath::Vec3<FloatWide>`). That is, each one points to all the `x` values for the batch, immediately followed by all the `y` values, followed by the `z` values. The mask determines which of those array elements to actually compute.

The various results arrays are also arranged as arrays that behave as if they were declared `float result [channels] [BatchWidth]`, where all the batch values for channel 0 are adjacent, followed by all the batch values for channel 1, etc.

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

#### Parameters

- `filename`: The name of the texture.
- `options`: A `TextureOptBatch` containing texture lookup options. This is conceptually the same as a `TextureOpt`, but the following fields are arrays of `[BatchWidth]` elements: `sblur`, `tblur`, `swidth`, `twidht`. The other fields are, as with `TextureOpt`, ordinary scalar values.
- `mask`: A bit-field designating which “lanes” should be computed: if `mask & (1<<i>)` is nonzero, then results should be computed and stored for `result[...] [i]`.
- `R`: Pointers to the 3D texture coordinates, each as a `float [3] [BatchWidth]`.
- `dRdx/dRdy`: The differentials of `R` relative to canonical directions `x` and `y`, each as a `float [3] [BatchWidth]`.
- `nchannels`: The number of channels of data to retrieve into `result` (e.g., 1 for a single value, 3 for an RGB triple, etc.).
- `result[]`: The result of the filtered texture lookup will be placed here, as `float [nchannels] [BatchWidth]`. (Note the “SOA” data layout.)

- `dresultds/dresultdt`: If non-null, these designate storage locations for the derivatives of `result`, and like `result` are in SOA layout: `float [nchannels][BatchWidth]`

**virtual bool environment** (TextureHandle \*texture\_handle, Perthread \*thread\_info, TextureOptBatch &options, Tex::RunMask mask, **const** float \*R, **const** float \*dRdx, **const** float \*dRdy, int nchannels, float \*result, float \*dresultds = nullptr, float \*dresultdt = nullptr) = 0

Slightly faster version of `environment()` if the app already has a texture handle and per-thread info.

### Texture metadata and raw texels

**virtual std::string resolve\_filename** (**const** std::string &filename) **const** = 0

Given possibly-relative 'filename', resolve it using the search path rules and return the full resolved filename.

**virtual bool get\_texture\_info** (ustring filename, int subimage, ustring dataname, TypeDesc datatype, void \*data) = 0

Get information or metadata about the named texture and store it in \*data.

Data names may include any of the following:

- `exists` (int): Stores the value 1 if the file exists and is an image format that OpenImageIO can read, or 0 if the file does not exist, or could not be properly read as a texture. Note that unlike all other queries, this query will “succeed” (return `true`) even if the file does not exist.
- `udim` (int) : Stores the value 1 if the file is a “virtual UDIM” or texture atlas file (as described in Section :ref:sec-texturesys-udim) or 0 otherwise.
- `subimages` (int) : The number of subimages/faces in the file, as an integer.
- `resolution` (int[2] or int[3]): The resolution of the texture file, if an array of 2 integers (described as `TypeDesc (INT, 2)`), or 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` (int) : The number of MIPmap levels for the specified subimage (an integer).
- `texturetype` (string) : 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` (string) : 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` (int) : The number of color channels in the file.
- `format` (int) : 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` (int) : 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` (int[4] or int[6]): 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 (matrix)` : 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).
- `worldtocamera (matrix)` : The viewing matrix, which is a 4x4 matrix (an `Imath::M44f`, described as `TypeMatrix44`) giving the world-to-camera 3D transformation matrix that was used when the image was created. Generally, only rendered images will have this.
- `worldtoscreen (matrix)` : The projection matrix, which is a 4x4 matrix (an `Imath::M44f`, described as `TypeMatrix44`) giving the matrix that projected points from world space into a 2D screen coordinate system where *x* and *y* range from  $-1$  to  $+1$ . Generally, only rendered images will have this.
- `averagecolor (float[nchannels])` : If available in the metadata (generally only for files that have been processed by `maketx`), this will return the average color of the texture (into an array of floats).
- `averagealpha (float)` : If available in the metadata (generally only for files that have been processed by `maketx`), this will return the average alpha value of the texture (into a float).
- `constantcolor (float[nchannels])` : If the metadata (generally only for files that have been processed by `maketx`) indicates that the texture has the same values for all pixels in the texture, this will retrieve the constant color of the texture (into an array of floats). A non-constant image (or one that does not have the special metadata tag identifying it as a constant texture) will fail this query (return false).
- `constantalpha (float)` : If the metadata indicates that the texture has the same values for all pixels in the texture, this will retrieve the constant alpha value of the texture. A non-constant image (or one that does not have the special metadata tag identifying it as a constant texture) will fail this query (return false).
- `stat:tilesread (int64)` : Number of tiles read from this file.
- `stat:bytesread (int64)` : Number of bytes of uncompressed pixel data read
- `stat:redundant_tiles (int64)` : Number of times a tile was read, where the same tile had been read before.
- `stat:redundant_bytesread (int64)` : Number of bytes (of uncompressed pixel data) in tiles that were read redundantly.
- `stat:redundant_bytesread (int)` : Number of tiles read from this file.
- `stat:timesopened (int)` : Number of times this file was opened.
- `stat:iotime (float)` : Time (in seconds) spent on all I/O for this file.
- `stat:mipsused (int)` : Stores 1 if any MIP levels beyond the highest resolution were accessed, otherwise 0.
- `stat:is_duplicate (int)` : Stores 1 if this file was a duplicate of another image, otherwise 0.
- *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.

**Return** true if `get_textureinfo()` is able to find the requested `dataname` for the texture and it matched the requested `datatype`. If the requested data was not found or was not of the right data type, return false. Except for the "exists" query, a file that does not exist or could not be read properly as an image also constitutes a query failure that will return false.



**Parameters**

- `filename`: The name of the texture.
- `subimage`: The subimage to query. (The metadata retrieved is for the highest-resolution MIP level of that subimage.)
- `dataname`: The name of the metadata to retrieve.
- `datatype`: *TypeDesc* describing the data type.
- `data`: Pointer to the caller-owned memory where the values should be stored. It is the caller's responsibility to ensure that `data` points to a large enough storage area to accommodate the `datatype` requested.

**virtual bool** `get_texture_info` (TextureHandle \*texture\_handle, Perthread \*thread\_info, int subimage, *ustring* dataname, *TypeDesc* datatype, void \*data) = 0

A more efficient variety of `get_texture_info()` for cases where you can use a `TextureHandle*` to specify the image and optionally have a `Perthread*` for the calling thread.

**virtual bool** `get_imagespec` (*ustring* filename, int subimage, *ImageSpec* &spec) = 0

Copy the *ImageSpec* associated with the named texture (the first subimage by default, or as set by subimage).

**Return** `true` upon success, `false` upon failure (such as being unable to find, open, or read the file, or if it does not contain the designated subimage or MIP level).

**Parameters**

- `filename`: The name of the image.
- `subimage`: The subimage to query. (The spec retrieved is for the highest-resolution MIP level of that subimage.)
- `spec`: *ImageSpec* into which will be copied the spec for the requested image.

**virtual bool** `get_imagespec` (TextureHandle \*texture\_handle, Perthread \*thread\_info, int subimage, *ImageSpec* &spec) = 0

A more efficient variety of `get_imagespec()` for cases where you can use a `TextureHandle*` to specify the image and optionally have a `Perthread*` for the calling thread.

**virtual const *ImageSpec* \***`imagespec` (*ustring* filename, int subimage = 0) = 0

Return a pointer to an *ImageSpec* associated with the named texture if the file is found and is an image format that can be read, otherwise return `nullptr`.

This method is much more efficient than `get_imagespec()`, since it just returns a pointer to the spec held internally by the *TextureSystem* (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* and its underlying *ImageCache*.

**Return** A pointer to the spec, if the image is found and able to be opened and read by an available image format plugin, and the designated subimage exists.

**Parameters**

- `filename`: The name of the image.
- `subimage`: The subimage to query. (The spec retrieved is for the highest-resolution MIP level of that subimage.)

```
virtual const ImageSpec *imagespec (TextureHandle *texture_handle, Perthread *thread_info =  
                                     nullptr, int subimage = 0) = 0
```

A more efficient variety of *imagespec* () for cases where you can use a TextureHandle\* to specify the image and optionally have a Perthread\* for the calling thread.

```
virtual bool get_texels (ustring filename, TextureOpt &options, int miplevel, int xbegin, int xend,  
                        int ybegin, int yend, int zbegin, int zend, int chbegin, int chend, TypeDesc  
                        format, void *result) = 0
```

For a texture specified by name, retrieve the rectangle of raw unfiltered texels from the subimage specified in options and at the designated miplevel, storing the pixel values beginning at the address specified by result. The pixel values will be converted to the data type specified by 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. Channels requested but not present in the file will get the options.fill value.

**Return** true for success, false for failure.

#### Parameters

- filename: The name of the image.
- options: A TextureOpt describing access options, including wrap modes, fill value, and subimage, that will be used when retrieving pixels.
- miplevel: The MIP level to retrieve pixels from (0 is the highest resolution level).
- xbegin/xend/ybegin/yend/zbegin/zend: The range of pixels to retrieve. The pixels retrieved include the begin value but not the end value (much like STL begin/end usage).
- chbegin/chend: Channel range to retrieve. To retrieve all channels, use chbegin = 0, chend = nchannels.
- format: *TypeDesc* describing the data type of the values you want to retrieve into result. The pixel values will be converted to this type regardless of how they were stored in the file or in the cache.
- result: Pointer to the memory where the pixel values should be stored. 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).

```
virtual bool get_texels (TextureHandle *texture_handle, Perthread *thread_info, TextureOpt  
                        &options, int miplevel, int xbegin, int xend, int ybegin, int yend, int zbe-  
                        gin, int zend, int chbegin, int chend, TypeDesc format, void *result) =  
                        0
```

A more efficient variety of *get\_texels* () for cases where you can use a TextureHandle\* to specify the image and optionally have a Perthread\* for the calling thread.

#### Controlling the cache

```
virtual void invalidate (ustring filename, bool force = true) = 0
```

Invalidate any cached information about the named file, including loaded texture tiles from that texture, and close any open file handle associated with the file. This calls ImageCache::invalidate(filename, force) on the underlying *ImageCache*.

```
virtual void invalidate_all (bool force = false) = 0
```

Invalidate all cached data for all textures. This calls ImageCache::invalidate\_all(force) on the underlying *ImageCache*.

**virtual void close** (*ustring* filename) = 0

Close any open file handles associated with a named file, but do not invalidate any image spec information or pixels associated with the files. A client might do this in order to release OS file handle resources, or to make it safe for other processes to modify textures on disk. This calls `ImageCache::close(force)` on the underlying *ImageCache*.

**virtual void close\_all** () = 0

*close()* all files known to the cache.

## Errors and statistics

**virtual std::string geterror** () const = 0

If any of the API routines returned false indicating an error, this routine will return the error string (and clear any error flags). If no error has occurred since the last time *geterror()* was called, it will return an empty string.

**virtual std::string getstats** (int level = 1, bool icstats = true) const = 0

Returns a big string containing useful statistics about the *TextureSystem* 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.

**virtual void reset\_stats** () = 0

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*.

## Public Functions

**virtual ImageCache\* imagecache** () const = 0

Return an opaque, non-owning pointer to the underlying *ImageCache* (if there is one).

## 8.4 Batched Texture Lookups

On CPU architectures with SIMD processing, texturing entire batches of samples at once may provide a large speedup compared to texturing each sample point individually. The batch size is fixed (for any build of OpenImageIO) and may be accessed with the following constant:

**constexpr int OIIO::Tex::BatchWidth** = OIIO\_TEXTURE\_SIMD\_BATCH\_WIDTH

The SIMD width for batched texturing operations. This is fixed within any release of OpenImageIO, but may change from release to release and also may be overridden at build time. A typical batch size is 16.

**typedef simd::VecType<float, OIIO\_TEXTURE\_SIMD\_BATCH\_WIDTH>::type OIIO::Tex::FloatWide**

A type alias for a SIMD vector of floats with the batch width.

**typedef simd::VecType<int, OIIO\_TEXTURE\_SIMD\_BATCH\_WIDTH>::type OIIO::Tex::IntWide**

A type alias for a SIMD vector of ints with the batch width.

All of the batched calls take a *run mask*, which describes which subset of “lanes” should be computed by the batched lookup:

**typedef** uint64\_t OIIO::Tex::RunMask

RunMask is defined to be an integer large enough to hold at least BatchWidth bits. The least significant bit corresponds to the first (i.e., [0]) position of all batch arrays. For each position *i* in the batch, the bit identified by  $(1 \ll i)$  controls whether that position will be computed.

**enumerator** RunMaskOn

The defined constant RunMaskOn contains the value with all bits  $0 \dots \text{BatchWidth}-1$  set to 1.

### 8.4.1 Batched Options

TextureOptBatch is a structure that holds the options for doing an entire batch of lookups from the same texture at once. The members of TextureOptBatch correspond to the similarly named members of the single-point TextureOpt, so we refer you to Section [TextureOpt](#) for detailed explanations, and this section will only explain the differences between batched and single-point options.

int **firstchannel**

int **subimage**

ustring **subimagename**

Tex::Wrap **s**wrap, **t**wrap, **r**wrap

Tex::MipMode **mipmode**

Tex::InterpMode **interpmode**

int **anisotropic**

bool **conservative\_filter**

float **fill**

**const** float **\*missingcolor**

These fields are all scalars — a single value for each TextureOptBatch — which means that the value of these options must be the same for every texture sample point within a batch. If you have a number of texture lookups to perform for the same texture, but they have (for example) differing wrap modes or subimages from point to point, then you must split them into separate batch calls.

float **s**blur[Tex::BatchWidth]

float **t**blur[Tex::BatchWidth]

float **r**blur[Tex::BatchWidth]

These arrays hold the *s*, and *t* blur amounts, for each sample in the batch, respectively. (And the *r* blur amount, used only for volumetric texture3d() lookups.)

float **s**width[Tex::BatchWidth]

float **t**width[Tex::BatchWidth]

float **r**width[Tex::BatchWidth]

These arrays hold the *s*, and *t* filtering width multiplier for derivatives, for each sample in the batch, respectively. (And the *r* multiplier, used only for volumetric texture3d() lookups.)

### 8.4.2 Batched Texture Lookup Calls

bool TextureSystem::texture (ustring filename, TextureOptBatch &options, Tex::RunMask mask, **const** float \*s, **const** float \*t, **const** float \*dsdx, **const** float \*dtdx, **const** float \*dsdy, **const** float \*dtdy, int nchannels, float \*result, float \*dresultds = nullptr, float \*dresultdt = nullptr)

bool TextureSystem::texture (TextureHandle \*texture\_handle, Perthread \*thread\_info, TextureOptBatch &options, Tex::RunMask mask, **const** float \*s, **const** float \*t, **const** float \*dsdx, **const** float \*dtdx, **const** float \*dsdy, **const** float \*dtdy, int nchannels, float \*result, float \*dresultds = nullptr, float \*dresultdt = nullptr)

Perform filtered 2D texture lookups on a batch of positions from the same texture, all at once. The parameters *s*, *t*, *dstdx*, *stdx*, and *dstdy*, *stdy* are each a pointer to `[BatchWidth]` values. The mask determines which of those array elements to actually compute.

The various results are arranged as arrays that behave as if they were declared:

```
float result[channels][BatchWidth]
```

In other words, all the batch values for channel 0 are adjacent, followed by all the batch values for channel 1, etc. (This is “SOA” order.)

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 (ustring filename, TextureOptBatch &options, Tex::RunMask mask, const float *P, const
               float *dPdx, const float *dPdy, const float *dPdz, int nchannels, float *result, float
               *dresultds = nullptr, float *dresultdt = nullptr, float *dresultdr = nullptr)
```

```
bool texture3d (TextureHandle *texture_handle, Perthread *thread_info, TextureOptBatch &options,
               Tex::RunMask mask, const float *P, const float *dPdx, const float *dPdy, const
               float *dPdz, int nchannels, float *result, float *dresultds = nullptr, float *dresultdt = nullptr,
               float *dresultdr = nullptr)
```

Perform filtered 3D volumetric texture lookups on a batch of positions from the same texture, all at once. The “point-like” parameters *P*, *dPdx*, *dPdy*, and *dPdz* are each a pointers to arrays of `float value[3][BatchWidth]`. That is, each one points to all the *x* values for the batch, immediately followed by all the *y* values, followed by the *z* values.

The various results arrays are also arranged as arrays that behave as if they were declared `float result[channels][BatchWidth]`, where all the batch values for channel 0 are adjacent, followed by all the batch values for channel 1, etc.

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

```
bool environment (ustring filename, TextureOptBatch &options, Tex::RunMask mask, const float *R,
                  const float *dRdx, const float *dRdy, int nchannels, float *result, float *dresultds
                  = nullptr, float *dresultdt = nullptr)
```

```
bool environment (TextureHandle *texture_handle, Perthread *thread_info, TextureOptBatch &options,
                  Tex::RunMask mask, const float *R, const float *dRdx, const float *dRdy, int
                  nchannels, float *result, float *dresultds = nullptr, float *dresultdt = nullptr)
```

Perform filtered directional environment map lookups on a batch of positions from the same texture, all at once. The “point-like” parameters *R*, *dRdx*, and *dRdy* are each a pointers to arrays of `float value[3][BatchWidth]`. That is, each one points to all the *x* values for the batch, immediately followed by all the *y* values, followed by the *z* values.

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.

The various results arrays are also arranged as arrays that behave as if they were declared `float result[channels][BatchWidth]`, where all the batch values for channel 0 are adjacent, followed by all the batch values for channel 1, etc.

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

## 8.5 UDIM and texture atlases

### Texture lookups

The `texture()` call supports virtual filenames that expand per lookup for UDIM and other tiled texture atlas techniques. The substitutions will occur if the texture filename initially passed to `texture()` does not exist as a concrete file and contains one or more of the following substrings:

<UDIM>	$1001 + \text{utile} + \text{vtile} * 10$
<u>	utile
<v>	vtile
<U>	utile + 1
<V>	vtile + 1

where the tile numbers are derived from the input `u,v` texture coordinates as follows:

```
// Each unit square of texture is a different tile
utile = max (0, int(u));
vtile = max (0, int(v));
// Re-adjust the texture coordinates to the offsets within the tile
u = u - utile;
v = v - vtile;
```

Example:

```
ustring filename ("paint.<UDIM>.tif");
float s = 1.4, t = 3.8;
texsys->texture (filename, s, t, ...);
```

will retrieve from file `paint.1032.tif` at coordinates (0.4,0.8).

### Retrieving metadata

Calls to `get_texture_info()` on UDIM files will retrieve the metadata of the first file it finds matching the name template. The call will fail (return `nullptr` and not retrieve data) if no concrete texture file can be found that matches the udim naming template.

### Handles of udim files

Calls to `get_texture_handle()` will always succeed. Withing knowing a specific `u` and `v`, it has no way to know that the concrete file you will eventually ask for would not succeed.

## IMAGEBUF: 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.

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

The ImageBuf class definition requires that you:

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

**enum** OIIO::ImageBuf::IBStorage

An ImageBuf can store its pixels in one of several ways (each identified by an IBStorage enumerated value):

*Values:*

**UNINITIALIZED**

An ImageBuf that doesn't represent any image at all (either because it is newly constructed with the default constructor, or had an error during construction).

**LOCALBUFFER**

"Local storage" is allocated to hold the image pixels internal to the ImageBuf. This memory will be freed when the ImageBuf is destroyed.

**APPBUFFER**

The ImageBuf "wraps" pixel memory already allocated and owned by the calling application. The caller will continue to own that memory and be responsible for freeing it after the ImageBuf is destroyed.

**IMAGECACHE**

The ImageBuf is "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.

## 9.2 Constructing, destructing, resetting an ImageBuf

There are several ways to construct an ImageBuf. Each constructor has a corresponding `reset` method that takes the same arguments. Calling `reset` on an existing ImageBuf is equivalent to constructing a new ImageBuf from scratch (even if the ImageBuf, prior to reset, previously held an image).

### 9.2.1 Making an empty or uninitialized ImageBuf

`OIIO::ImageBuf::ImageBuf()`

Default constructor makes an empty/uninitialized ImageBuf. There isn't much you can do with an uninitialized buffer until you call `reset()`. The storage type of a default-constructed ImageBuf is `IBStorage::UNINITIALIZED`.

`void OIIO::ImageBuf::reset()`

Destroy any previous contents of the ImageBuf and re-initialize it to resemble a freshly constructed ImageBuf using the default constructor (holding no image, with storage `IBStorage::UNINITIALIZED`).

### 9.2.2 Constructing a readable ImageBuf

`OIIO::ImageBuf::ImageBuf(string_view name, int subimage = 0, int miplevel = 0, ImageCache *imagecache = nullptr, const ImageSpec *config = nullptr)`

Construct a read-only 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 lazily, only 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.

The implementation may end up either reading the entire image internally owned memory (if so, the storage will be `LOCALBUFFER`), or it may rely on being backed by an *ImageCache* (in this case, the storage will be `IMAGECACHE`) depending on the image size and other factors.

#### Parameters

- `name`: The image to read.
- `subimage/miplevel`: The subimage and MIP level to read (defaults to the first subimage of the file, highest-res MIP level).
- `imagecache`: Optionally, a particular *ImageCache* to use. If `nullptr`, the default global/shared image cache will be used.
- `config`: Optionally, a pointer to an *ImageSpec* whose metadata contains configuration hints that set options related to the opening and reading of the file.

`void OIIO::ImageBuf::reset(string_view name, int subimage, int miplevel, ImageCache *imagecache = nullptr, const ImageSpec *config = nullptr)`

Destroy any previous contents of the ImageBuf and re-initialize it as if newly constructed with the same arguments, as a read-only representation of an existing image file.



### 9.2.3 Constructing a writable ImageBuf

`OIIO::ImageBuf::ImageBuf (const ImageSpec &spec, InitializePixels zero = InitializePixels::Yes)`

Construct a writable ImageBuf with the given specification (including resolution, data type, metadata, etc.). The ImageBuf will allocate and own its own pixel memory and will free that memory automatically upon destruction, `clear()`, or `reset()`. Upon successful initialization, the storage will be reported as `LOCALBUFFER`.

#### Parameters

- `spec`: An *ImageSpec* describing the image and its metadata. If not enough information is given to know how much memory to allocate (width, height, depth, channels, and data format), the ImageBuf will remain in an `UNINITIALIZED` state and will have no local pixel storage.
- `zero`: After a successful allocation of the local pixel storage, this parameter controls whether the pixels will be initialized to hold zero (black) values (`InitializePixels::Yes`) or if the pixel memory will remain uninitialized (`InitializePixels::No`) and thus may hold nonsensical values. Choosing `No` may save the time of writing to the pixel memory if you know for sure that you are about to overwrite it completely before you will need to read any pixel values.

`void OIIO::ImageBuf::reset (const ImageSpec &spec, InitializePixels zero = InitializePixels::Yes)`

Destroy any previous contents of the ImageBuf and re-initialize it as if newly constructed with the same arguments, as a read/write image with locally allocated storage that can hold an image as described by `spec`. The optional `zero` parameter controls whether the pixel values are filled with black/empty, or are left uninitialized after being allocated.

Note that if the *ImageSpec* does not contain enough information to specify how much memory to allocate (width, height, channels, and data format), the ImageBuf will remain uninitialized (regardless of how `zero` is set).

`bool OIIO::ImageBuf::make_writeable (bool keep_cache_type = false)`

Make the ImageBuf be writable. That means that if it was previously backed by an *ImageCache* (storage was `IMAGECACHE`), it will force a full read so that the whole image is in local memory. This will invalidate any current iterators on the image. It has no effect if the image storage is not `IMAGECACHE`.

**Return** Return `true` if it works (including if no read was necessary), `false` if something went horribly wrong.

#### Parameters

- `keep_cache_type`: If true, preserve any ImageCache-forced data types (you might want to do this if it is critical that the apparent data type doesn't change, for example if you are calling `make_writeable()` from within a type-specialized function).

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

`OIIO::ImageBuf::ImageBuf (const ImageSpec &spec, void *buffer)`

Construct a writable ImageBuf that “wraps” existing pixel memory owned by the calling application. The ImageBuf does not own the pixel storage and will not free/delete that memory, even when the ImageBuf is destroyed. Upon successful initialization, the storage will be reported as `APPBUFFER`.

#### Parameters

- `spec`: An *ImageSpec* describing the image and its metadata. If not enough information is given to know the “shape” of the image (width, height, depth, channels, and data format), the ImageBuf will remain in an `UNINITIALIZED` state.
- `buffer`: A pointer to the caller-owned memory containing the storage for the pixels. It must be already allocated with enough space to hold a full image as described by `spec`.

void OIIO::ImageBuf::reset (const *ImageSpec* &spec, void \*buffer)

Destroy any previous contents of the ImageBuf and re-initialize it as if newly constructed with the same arguments, to “wrap” existing pixel memory owned by the calling application.

## 9.2.5 Reading and Writing disk images

bool OIIO::ImageBuf::read (int subimage = 0, int miplevel = 0, bool force = false, *TypeDesc* convert = *TypeDesc::UNKNOWN*, ProgressCallback progress\_callback = nullptr, void \*progress\_callback\_data = nullptr)

Read 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.

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.

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.

**Return** true upon success, or false if the read failed (in which case, you should be able to retrieve an error message via *geterror()*).

### Parameters

- subimage/miplevel: The subimage and MIP level to read.
- force: If true, will force an immediate full read into ImageBuf-owned local pixel memory (yielding a LOCALPIXELS storage buffer). Otherwise, it is up to the implementation whether to immediately read or have the image backed by an *ImageCache* (storage IMAGECACHE.)
- convert: If set to a specific type (notUNKNOWN), the ImageBuf memory will be allocated for that type specifically and converted upon read.
- progress\_callback/progress\_callback\_data: 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).

bool OIIO::ImageBuf::read (int subimage, int miplevel, int chbegin, int chend, bool force, *TypeDesc* convert, ProgressCallback progress\_callback = nullptr, void \*progress\_callback\_data = nullptr)

Read the file, if possible only allocating and reading a subset of channels, [chbegin..chend-1]. This can be a performance and memory improvement for some image file formats, if you know that any use of the ImageBuf will only access a subset of channels from a many-channel file.

Additional parameters:

### Parameters

- `chbegin/chend`: The subset (a range with “exclusive end”) of channels to read, if the implementation is able to read only a subset of channels and have a performance advantage by doing so. If `chbegin` is 0 and `chend` is either negative or greater than the number of channels in the file, all channels will be read. Please note that it is “advisory” and not guaranteed to be honored by the underlying implementation.

`bool OIIO::ImageBuf::init_spec (string_view filename, int subimage, int miplevel)`

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.

**Return** `true` upon success, or `false` if the read failed (in which case, you should be able to retrieve an error message via `geterror()`).

### Parameters

- `filename`: The filename to read from (should be the same as the filename used when the ImageBuf was constructed or reset.)
- `subimage/miplevel`: The subimage and MIP level to read.

`bool OIIO::ImageBuf::write (string_view filename, TypeDesc dtype = TypeUnknown, string_view fileformat = string_view(), ProgressCallback progress_callback = nullptr, void *progress_callback_data = nullptr) const`

Write the image to the named file, converted to the specified pixel data type `dtype` (TypeUnknown signifies to use the data type of the buffer), and file format (an empty `fileformat` means to infer the type from the filename extension).

By default, it will always try to write a scanline-oriented file, unless the `set_write_tiles()` method has been used to override this.

**Return** `true` upon success, or `false` if the write failed (in which case, you should be able to retrieve an error message via `geterror()`).

### Parameters

- `filename`: The filename to write to.
- `dtype`: Optional override of the pixel data format to use in the file being written. The default (UNKNOWN) means to try writing the same data format that as pixels are stored within the ImageBuf memory (or whatever type was specified by a prior call to `set_write_format()`). In either case, if the file format does not support that data type, another will be automatically chosen that is supported by the file type and loses as little precision as possible.
- `fileformat`: Optional override of the file format to write. The default (empty string) means to infer the file format from the extension of the `filename` (for example, “foo.tif” will write a TIFF file).

- `progress_callback/progress_callback_data`: If `progress_callback` is non-NULL, the underlying write, if expensive, may make several calls to `progress_callback(progress_callback_data, portion_done)` which allows you to implement some sort of progress meter.

`bool OIIO::ImageBuf::write(ImageOutput *out, ProgressCallback progress_callback = nullptr, void *progress_callback_data = nullptr) const`

Write the pixels of the ImageBuf to an open *ImageOutput*. The *ImageOutput* must have already been opened with a spec that indicates a resolution identical to that of this ImageBuf (but it may have specified a different pixel data type, in which case data conversions will happen automatically). This method does NOT close the file when it's done (and so may be called in a loop to write a multi-image file).

Note that since this uses an already-opened *ImageOutput*, which is too late to change how it was opened, it does not honor any prior calls to `set_write_format` or `set_write_tiles`.

The main application of this method is to allow an ImageBuf (which by design may hold only a *single* image) to be used for the output of one image of a multi-subimage and/or MIP-mapped image file.

**Return** `true` if all went ok, `false` if there were errors writing.

#### Parameters

- `out`: A pointer to an already-opened *ImageOutput* to which the pixels of the ImageBuf will be written.
- `progress_callback/progress_callback_data`: If `progress_callback` is non-NULL, the underlying write, if expensive, may make several calls to `progress_callback(progress_callback_data, portion_done)` which allows you to implement some sort of progress meter.

`void OIIO::ImageBuf::set_write_format(TypeDesc format)`

Set the pixel data format that will be used for subsequent `write()` calls that do not themselves request a specific data type request.

Note that this does not affect the variety of `write()` that takes an open *ImageOutput\** as a parameter.

#### Parameters

- `format`: The data type to be used for all channels.

`void OIIO::ImageBuf::set_write_format(cspan<TypeDesc> format)`

Set the per-channel pixel data format that will be used for subsequent `write()` calls that do not themselves request a specific data type request.

#### Parameters

- `format`: The type of each channel (in order). Any channel's format specified as `TypeUnknown` will default to be whatever type is described in the *ImageSpec* of the buffer.

`void OIIO::ImageBuf::set_write_tiles(int width = 0, int height = 0, int depth = 0)`

Override the tile sizing for subsequent calls to the `write()` method (the variety that does not take an open *ImageOutput\**). Setting all three dimensions to 0 indicates that the output should be a scanline-oriented file.

This lets you write a tiled file from an ImageBuf that may have been read originally from a scanline file, or change the dimensions of a tiled file, or to force the file written to be scanline even if it was originally read from a tiled file.

In all cases, if the file format ultimately written does not support tiling, or the tile dimensions requested, a suitable supported tiling choice will be made automatically.

## 9.3 Getting and setting information about an ImageBuf

`bool OIIO::ImageBuf::initialized() const`

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

*IBStorage* `OIIO::ImageBuf::storage() const`

Which type of storage is being used for the pixels? 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 &OIIO::ImageBuf::spec() const`

Return a read-only (const) reference to the image spec that describes the buffer.

`const ImageSpec &OIIO::ImageBuf::nativespec() const`

Return a read-only (const) reference to the “native” image spec (that describes the file, which may be slightly different than the spec of the ImageBuf, particularly if the IB is backed by an *ImageCache* that is imposing some particular data format or tile size).

This may differ from *spec()* 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).

*ImageSpec* & `OIIO::ImageBuf::specmod()`

Return a writable reference to the *ImageSpec* that describes 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.

*string\_view* `OIIO::ImageBuf::name(void) const`

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

*string\_view* `OIIO::ImageBuf::file_format_name(void) const`

Return the name of the image file format of the disk file we read into this image, for example “openexr”. Returns an empty string if this image was not the result of a *read()*.

`int OIIO::ImageBuf::subimage() const`

Return the index of the subimage the ImageBuf is currently holding.

`int OIIO::ImageBuf::nsubimages() const`

Return the number of subimages in the file.

`int OIIO::ImageBuf::miplevel() const`

Return the index of the miplevel the ImageBuf is currently holding.

`int OIIO::ImageBuf::nmiplevels() const`

Return the number of miplevels of the current subimage.

`int OIIO::ImageBuf::nchannels() const`

Return the number of color channels in the image. 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 OIIO::ImageBuf::orientation() const**

Return the current "Orientation" metadata for the image, per the table in `sec-metadata-orientation_`

**void OIIO::ImageBuf::set\_orientation(int orient)**

Set the "Orientation" metadata for the image.

**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 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.

*ROI* **OIIO::ImageBuf::roi() const**

Return pixel data window for this ImageBuf as a *ROI*.

*ROI* **OIIO::ImageBuf::roi\_full() const**

Return full/display window for this ImageBuf as a *ROI*.

**void OIIO::ImageBuf::set\_origin(int x, int y, int z = 0)**

Alters the metadata of the spec in the ImageBuf to reset the “origin” of the pixel data window to be the specified coordinates. This does not affect the size of the pixel data window, only its position.

**void OIIO::ImageBuf::set\_full(int xbegin, int xend, int ybegin, int yend, int zbegin, int zend)**

Set the “full” (a.k.a. display) window to Alters the metadata of the spec in the ImageBuf to reset the “full” image size (a.k.a. “display window”) to

`[xbegin,xend) x [ybegin,yend) x [zbegin,zend) ``

This does not affect the size of the pixel data window.

**void OIIO::ImageBuf::set\_roi\_full(const *ROI* &newroi)**

Set full/display window for this ImageBuf to a *ROI*. Does NOT change the channels of the spec, regardless of `newroi`.

**bool OIIO::ImageBuf::contains\_roi(*ROI* roi) const**

Is the specified roi completely contained in the data window of this ImageBuf?

*TypeDesc* **OIIO::ImageBuf::pixeltype() const**

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

**int OIIO::ImageBuf::threads() const**

Retrieve the current thread-spawning policy of this ImageBuf.

**void OIIO::ImageBuf::threads(int n) const**

Set the threading policy for this ImageBuf, controlling the maximum amount of parallelizing thread “fan-out” that might occur during expensive operations. The default of 0 means that the global attribute (“threads”) value should be used (which itself defaults to using as many threads as cores).



The main reason to change this value is to set it to 1 to indicate that the calling thread should do all the work rather than spawning new threads. That is probably the desired behavior in situations where the calling application has already spawned multiple worker threads.

## 9.4 Copying ImageBuf's and blocks of pixels

**const** ImageBuf &OIIO::ImageBuf::operator=(const ImageBuf &src)

Copy assignment.

**const** ImageBuf &OIIO::ImageBuf::operator=(ImageBuf &&src)

Move assignment.

bool OIIO::ImageBuf::copy(const ImageBuf &src, TypeDesc format = TypeUnknown)

Try to copy the pixels and metadata from *src* to *\*this* (optionally with an explicit data format conversion).

If the previous state of *\*this* was uninitialized, owning its own local pixel memory, or referring to a read-only image backed by *ImageCache*, then local pixel memory will be allocated to hold the new pixels and the call always succeeds unless the memory cannot be allocated. In this case, the *format* parameter may request a pixel data type that is different from that of the source buffer.

If *\*this* previously referred to an app-owned memory buffer, the memory cannot be re-allocated, so the call will only succeed if the app-owned buffer is already the correct resolution and number of channels. The data type of the pixels will be converted automatically to the data type of the app buffer.

**Return** true upon success or false upon error/failure.

### Parameters

- *src*: Another ImageBuf from which to copy the pixels and metadata.
- *format*: Optionally request the pixel data type to be used. The default of *TypeUnknown* means to use whatever data type is used by the *src*. If *\*this* is already initialized and has APPBUFFER storage ("wrapping" an application buffer), this parameter is ignored.

ImageBuf OIIO::ImageBuf::copy(TypeDesc format) const

Return a full copy of *this* ImageBuf (optionally with an explicit data format conversion).

void OIIO::ImageBuf::copy\_metadata(const ImageBuf &src)

Copy all the metadata from *src* to *\*this* (except for pixel data resolution, channel types and names, and data format).

bool OIIO::ImageBuf::copy\_pixels(const ImageBuf &src)

Copy the pixel data from *src* to *\*this*, automatically converting to the existing data format of *\*this*. It only copies pixels in the overlap regions (and channels) of the two images; pixel data in *\*this* that do exist in *src* will be set to 0, and pixel data in *src* that do not exist in *\*this* will not be copied.

void OIIO::ImageBuf::swap(ImageBuf &other)

Swap the entire contents with another ImageBuf.

## 9.5 Getting and setting pixel values

### Getting and setting individual pixels – slow

`float OIIO::ImageBuf::getchannel` (int *x*, int *y*, int *z*, int *c*, WrapMode *wrap* = WrapBlack) **const**  
Retrieve a single channel of one pixel.

**Return** The data value, converted to a `float`.

#### Parameters

- *x/y/z*: The pixel coordinates.
- *c*: The channel index to retrieve.
- *wrap*: WrapMode that determines the behavior if the pixel coordinates are outside the data window: WrapBlack, WrapClamp, WrapPeriodic, WrapMirror.

`void OIIO::ImageBuf::getpixel` (int *x*, int *y*, int *z*, float *\*pixel*, int *maxchannels* = 1000, WrapMode *wrap* = WrapBlack) **const**  
Retrieve the pixel value by *x*, *y*, *z* pixel indices, placing its contents in `pixel[0..n-1]` where *n* is the smaller of *maxchannels* the actual number of channels stored in the buffer.

#### Parameters

- *x/y/z*: The pixel coordinates.
- *pixel*: The results are stored in `pixel[0..nchannels-1]`. It is up to the caller to ensure that *pixel* points to enough memory to hold the required number of channels.
- *maxchannels*: Optional clamp to the number of channels retrieved.
- *wrap*: WrapMode that determines the behavior if the pixel coordinates are outside the data window: WrapBlack, WrapClamp, WrapPeriodic, WrapMirror.

`void OIIO::ImageBuf::interpixel` (float *x*, float *y*, float *\*pixel*, WrapMode *wrap* = WrapBlack) **const**  
Sample the image plane at pixel coordinates (*x*,*y*), using linear interpolation between pixels, placing the result in `pixel[]`.

#### Parameters

- *x/y*: The pixel coordinates. Note that pixel data values themselves are at the pixel centers, so pixel (*i*,*j*) is at image plane coordinate (*i*+0.5, *j*+0.5).
- *pixel*: The results are stored in `pixel[0..nchannels-1]`. It is up to the caller to ensure that *pixel* points to enough memory to hold the number of channels in the image.
- *wrap*: WrapMode that determines the behavior if the pixel coordinates are outside the data window: WrapBlack, WrapClamp, WrapPeriodic, WrapMirror.

`void OIIO::ImageBuf::interpixel_bicubic` (float *x*, float *y*, float *\*pixel*, WrapMode *wrap* = WrapBlack) **const**  
Bicubic interpolation at pixel coordinates (*x*,*y*).

`void OIIO::ImageBuf::interpixel_NDC` (float *s*, float *t*, float *\*pixel*, WrapMode *wrap* = WrapBlack) **const**  
Linearly interpolate at NDC coordinates (*s*,*t*), where (0,0) is the upper left corner of the display window, (1,1) the lower right corner of the display window.



**Note** `interppixel()` uses pixel coordinates (ranging 0..resolution) whereas `interppixel_NDC()` uses NDC coordinates (ranging 0..1).

```
void OIIO::ImageBuf::interppixel_bicubic_NDC (float s, float t, float *pixel, WrapMode wrap =
                                             WrapBlack) const
    Bicubic interpolation at NDC space coordinates (s,t), where (0,0) is the upper left corner of the display (a.k.a.
    “full”) window, (1,1) the lower right corner of the display window.
```

```
void OIIO::ImageBuf::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 in pixel[0..n-1]. The number of channels copied,
    n, is the minimum of maxchannels and the actual number of channels in the image.
```

```
void OIIO::ImageBuf::setpixel (int i, const float *pixel, int maxchannels = 1000)
    Set the i-th pixel value of the image (out of width*height*depth), from floating-point values in pixel[]. Set
    at most maxchannels (will be clamped to the actual number of channels).
```

### Getting and setting regions of pixels – fast

```
bool OIIO::ImageBuf::get_pixels (ROI roi, TypeDesc format, void *result, stride_t xstride = Au-
                                toStride, stride_t ystride = AutoStride, stride_t zstride = Au-
                                toStride) const
    Retrieve the rectangle of pixels spanning the ROI (including channels) 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 de-
    fault, 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.
```

```
bool OIIO::ImageBuf::set_pixels (ROI roi, TypeDesc format, const void *data, stride_t xstride =
                                AutoStride, stride_t ystride = AutoStride, stride_t zstride = Au-
                                toStride)
    Copy the data into the given ROI of the ImageBuf. The data points to values specified by format, with layout
    detailed by the stride values (in bytes, with AutoStride indicating “contiguous” layout). It is up to the caller to
    ensure that data points to an area of memory big enough to account for the ROI. Return true if the operation
    could be completed, otherwise return false.
```

## 9.6 Deep data in an ImageBuf

```
bool OIIO::ImageBuf::deep() const
    Does this ImageBuf store deep data? Returns true if the ImageBuf holds a “deep” image, false if the
    ImageBuf holds an ordinary pixel-based image.
```

```
int OIIO::ImageBuf::deep_samples (int x, int y, int z = 0) const
    Retrieve the number of deep data samples corresponding to pixel (x,y,z). Return 0 if not a deep image, or if the
    pixel is outside of the data window, or if the designated pixel has no deep samples.
```

```
void OIIO::ImageBuf::set_deep_samples (int x, int y, int z, int nsamples)
    Set the number of deep samples for pixel (x,y,z). If data has already been allocated, this is equivalent to inserting
    or erasing samples.
```

```
void OIIO::ImageBuf::deep_insert_samples (int x, int y, int z, int samplepos, int nsamples)
    Insert nsamples new samples, starting at position samplepos of pixel (x,y,z).
```

`void OIIO::ImageBuf::deep_erase_samples (int x, int y, int z, int samplepos, int nsamples)`  
Remove `nsamples` samples, starting at position `samplepos` of pixel `(x,y,z)`.

`float OIIO::ImageBuf::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 if not a deep image or if the pixel coordinates or channel number are out of range or if that pixel has no deep samples.

`uint32_t OIIO::ImageBuf::deep_value_uint (int x, int y, int z, int c, int s) const`  
Return the value (as a `uint32_t`) of sample `s` of channel `c` of pixel `(x, y, z)`. Return 0 if not a deep image or if the pixel coordinates or channel number are out of range or if that pixel has no deep samples.

`void OIIO::ImageBuf::set_deep_value (int x, int y, int z, int c, int s, float value)`  
Set the value of sample `s` of channel `c` of pixel `(x, y, z)` to a `float` value (it is expected that channel `c` is a floating point type).

`void OIIO::ImageBuf::set_deep_value (int x, int y, int z, int c, int s, uint32_t value)`  
Set the value of sample `s` of channel `c` of pixel `(x, y, z)` to a `uint32_t` value (it is expected that channel `c` is an integer type).

`const void *OIIO::ImageBuf::deep_pixel_ptr (int x, int y, int z, int c, int s = 0) const`  
Return a pointer to the raw data of pixel `(x, y, z)`, channel `c`, sample `s`. Return `nullptr` 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. Use with caution these pointers may be invalidated by calls that adjust the number of samples in any pixel.

*DeepData* &OIIO::ImageBuf::deepdata ()

`const DeepData &OIIO::ImageBuf::deepdata () const`  
Returns a reference to the underlying *DeepData* for a deep image.

## 9.7 Error Handling

`template<typename ...Args>`  
`void OIIO::ImageBuf::errorf (const char *fmt, const Args&... args) const`  
Error reporting for ImageBuf: call this with printf-like arguments to report an error.

`bool OIIO::ImageBuf::has_error (void) const`  
Returns `true` if the ImageBuf has had an error and has an error message ready to retrieve via *geterror()*.

`std::string OIIO::ImageBuf::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.

## 9.8 Miscellaneous

`void *localpixels ()`  
`const void *localpixels () const`  
Return a raw pointer to the “local” pixel memory, if they are fully in RAM and not backed by an ImageCache, or `nullptr` otherwise. You can also test it like a `bool` to find out if pixels are local.

`void *pixeladdr (int x, int y, int z = 0, int ch = 0)`  
`const void *pixeladdr (int x, int y, int z = 0, int ch = 0) const`  
Return the address where pixel `(x, y, z)`, channel `ch`, is stored in the image buffer. Use with extreme caution! Will return `nullptr` if the pixel values aren’t local in RAM.

`int OIIO::ImageBuf::pixelindex (int x, int y, int z, bool check_range = false) const`  
 Return the index of pixel (x,y,z). If *check\_range* is true, return -1 for an invalid coordinate that is not within the data window.

`static WrapMode OIIO::ImageBuf::WrapMode_from_string (string_view name)`  
 Return the `WrapMode` corresponding to the name ("default", "black", "clamp", "periodic", "mirror"). For an unknown name, this will return `WrapDefault`.

## 9.9 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.

---

**Note:** `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`.

---

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 `Iterator<BUFT, USERT>` is templated based on two types: `BUFT` the type of the data stored in the `ImageBuf`, and `USERT` type 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);
```

template<>

**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.

template<>

**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.

template<>

**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.

**void Iterator::set\_deep\_samples()**  
For deep images only (and non-const ImageBuf), set the number of deep samples for the current pixel. This only is useful if the ImageBuf has not yet had the `deep_alloc()` method called.

**USERT Iterator::deep\_value(int c, int s) const**  
**uint32\_t Iterator::deep\_value\_int(int c, int s) const**  
For deep images only, returns the value of channel *c*, sample number *s*, at the current pixel.

**void Iterator::set\_deep\_value(int c, int s, float value)**  
**void Iterator::set\_deep\_value(int c, int s, uint32\_t value)**  
For deep images only (and non-cconst ImageBuf, sets the value of channel *c*, sample number *s*, at the current pixel. This only is useful if the ImageBuf has already had the `deep_alloc()` method called.

### 9.9.1 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";
}

```

### 9.9.2 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;
}

```

## 9.10 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.

### 9.10.1 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 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 */);
```

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 */);
```

### 9.10.2 Strategy 2: Template your iterating functions based on buffer type

Consider the following alternate version of the `make_black` function from Section *Example: Set all pixels in a region to black*

```
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
    }
```

(continues on next page)

<sup>1</sup> `ImageCache` only supports a limited set of types internally, currently `float`, `half`, `uint8`, `uint16`, and all other data types are converted to these automatically as they are read into the cache.

(continued from previous page)

```

    }
    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)
{
    bool ok;
    OIIO_DISPATCH_COMMON_TYPES (ok, "make_black", make_black_impl,
                                buf.spec().format, buf, region);
    return ok;
}

```

This other type-dispatching helper macros will be discussed in more detail in Chapter *ImageBufAlgo: Image Processing*.





## IMAGEBUFALGO: IMAGE PROCESSING

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`.

### 10.1 ImageBufAlgo common principles

This section explains the general rules common to all ImageBufAlgo functions. Only exceptions to these rules will be explained in the subsequent listings of all the individual ImageBufAlgo functions.

#### 10.1.1 Return values and error messages

Most ImageBufAlgo functions that produce image data come in two forms:

1. Return an ImageBuf.

The return value is a new ImageBuf containing the result image. In this case, an entirely new image will be created to hold the result. In case of error, the result image returned can have any error conditions checked with `has_error()` and `geterror()`.

```
// Method 1: Return an image result
ImageBuf fg ("fg.exr"), bg ("bg.exr");
ImageBuf dst = ImageBufAlgo::over (fg, bg);
if (dst.has_error())
    std::cout << "error: " << dst.geterror() << "\n";
```

2. Pass a destination ImageBuf reference as the first parameter.

The function is passed a *destination* ImageBuf where the results will be stored, and the return value is a `bool` that is true if the function succeeds or false if the function fails. Upon failure, the destination ImageBuf (the one that is being altered) will have an error message set.

```
// Method 2: Write into an existing image
ImageBuf fg ("fg.exr"), bg ("bg.exr");
ImageBuf dst; // will be the output image
bool ok = ImageBufAlgo::over (dst, fg, bg);
if (! ok)
    std::cout << "error: " << dst.geterror() << "\n";
```

The first option (return an ImageBuf directly) is a more compact and intuitive notation that is natural for most simple uses. But the second option (pass an ImageBuf& referring to an existing destination) offers additional flexibility, including more careful control over allocations, the ability to partially overwrite regions of an existing image, and the

ability for the destination image to also be one of the input images (for example, `add (A, A, B)` adds B into existing image A, with no third image allocated at all).

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.

### 10.1.2 Region of interest

Most `ImageBufAlgo` functions take an optional ROI parameter that restricts the operation to a range in x, y, z, and channels. The default-constructed ROI (also known as `ROI : All()`) means no region restriction – the whole image will be copied or altered.

For `ImageBufAlgo` functions that write into a destination `ImageBuf` parameter and it is already initialized (i.e. allocated with a particular size and data type), 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.

For `ImageBufAlgo` functions that return an `ImageBuf` directly, or if their `dst` parameter is an uninitialized `ImageBuf`, the ROI (if set) determines the size of the result image. If the ROI is the default `All`, the result image size will be the union of the pixel data windows of the input images and have a data type determined by the data types of the input images.

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).

### 10.1.3 Constant and per-channel values

Many `ImageBufAlgo` functions take per-channel constant-valued arguments (for example, a fill color). These parameters are passed as `cspan<float>`. These are generally expected to have length equal to the number of channels. But you may also pass a single float which will be used as the value for all channels. (More generally, what is happening is that the last value supplied is replicated for any missing channel.)

Some `ImageBufAlgo` functions have parameters of type `Image_or_Const`, which may take either an `ImageBuf` reference, or a per-channel constant, or a single constant to be used for all channels.

### 10.1.4 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~ref{sec:attribute:threads}](#)), 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

For the ImageBufAlgo functions in this section, there is no “source” image. Therefore, either an initialized `dst` must be supplied (to give a pre-allocated size and data type of the image), or else it is strictly necessary to supply an `ROI` parameter to specify the size of the new image (the data type in this case will always be `float`). It is an error if one of the pattern generation ImageBufAlgo functions is neither supplied a pre-allocated `dst` nor a non-default `ROI`.

### 10.2.1 zero() – create a black image

`ImageBuf OIIO::ImageBufAlgo::zero (ROI roi, int nthreads = 0)`

Create an all-black `float` image of size and channels as described by the `ROI`.

Examples:

```
// Create a new 3-channel, 512x512 float image filled with 0.0 values.
ImageBuf zero = ImageBufAlgo::zero (ROI(0,512,0,512,0,1,0,3));

// Zero out an existing buffer, keeping it the same size and data type
ImageBuf A = ...;
...
ImageBufAlgo::zero (A);

// Zero out a rectangular region of an existing buffer
ImageBufAlgo::zero (A, ROI (0, 100, 0, 100));

// Zero out just the green channel, leave everything else the same
ROI roi = A.roi ();
roi.chbegin = 1; // green
roi.chend = 2;   // one past the end of the channel region
ImageBufAlgo::zero (A, roi);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::zero (ImageBuf &dst, ROI roi = {}, int nthreads =
                                0)
    Write to an existing image dst (allocating if it is uninitialized).
```

### 10.2.2 fill() – fill a region with a solid color or gradient

*group* **fill**

Fill an image region with given channel values, either returning a new image or altering the existing `dst` image within the `ROI`. Note that the values arrays start with channel 0, even if the `ROI` indicates that a later channel is the first to be changed.

Three varieties of `fill()` exist: (a) a single set of channel values that will apply to the whole `ROI`, (b) two sets of values that will create a linearly interpolated gradient from top to bottom of the `ROI`, (c) four sets of values that will be bilinearly interpolated across all four corners of the `ROI`.

## Functions

ImageBuf **fill** (*cspan*<float> *values*, *ROI* *roi*, int *nthreads* = 0)

ImageBuf **fill** (*cspan*<float> *top*, *cspan*<float> *bottom*, *ROI* *roi*, int *nthreads* = 0)

ImageBuf **fill** (*cspan*<float> *topleft*, *cspan*<float> *topright*, *cspan*<float> *bottomleft*, *cspan*<float> *bottomright*, *ROI* *roi*, int *nthreads* = 0)

bool **fill** (ImageBuf &*dst*, *cspan*<float> *values*, *ROI* *roi* = {}, int *nthreads* = 0)

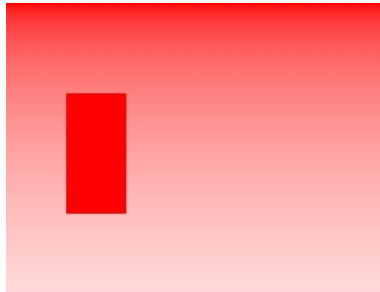
bool **fill** (ImageBuf &*dst*, *cspan*<float> *top*, *cspan*<float> *bottom*, *ROI* *roi* = {}, int *nthreads* = 0)

bool **fill** (ImageBuf &*dst*, *cspan*<float> *topleft*, *cspan*<float> *topright*, *cspan*<float> *bottomleft*, *cspan*<float> *bottomright*, *ROI* *roi* = {}, int *nthreads* = 0)

Examples:

```
// Create a new 640x480 RGB image, with a top-to-bottom gradient
// from red to pink
float pink[3] = { 1, 0.7, 0.7 };
float red[3] = { 1, 0, 0 };
ImageBuf A = ImageBufAlgo::fill (red, pink, ROI(0, 640, 0, 480, 0, 1, 0, 3));

// Draw a filled red rectangle overtop existing image A.
ImageBufAlgo::fill (A, red, ROI(50,100, 75, 175));
```



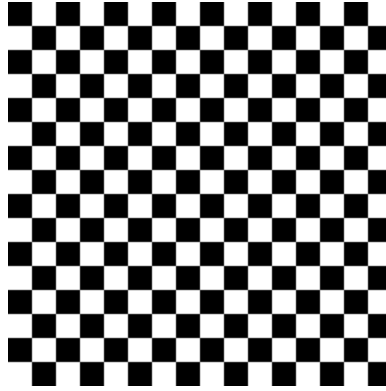
### 10.2.3 checker() – make a checker pattern

ImageBuf OIIO::ImageBufAlgo::**checker** (int *width*, int *height*, int *depth*, *cspan*<float> *color1*, *cspan*<float> *color2*, int *xoffset*, int *yoffset*, int *zoffset*, *ROI* *roi*, int *nthreads* = 0)

Create a checkerboard pattern of size given by *roi*, with origin given by the *offset* values, checker size given by the *width*, *height*, *depth* values, and alternating between *color1*[] and *color2*[]. The pattern is defined in abstract “image space” independently of the pixel data window of *dst* or the *ROI*.

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);
```



Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::checker (ImageBuf &dst, int width, int height, int
                                depth, cspan<float> color1, cspan<float>
                                color2, int xoffset = 0, int yoffset = 0, int
                                zoffset = 0, ROI roi = {}, int nthreads = 0)
    Write to an exsisting image dst (allocating if it is uninitialized).
```

## 10.2.4 noise() – make a noise pattern

ImageBuf OIIO::ImageBufAlgo::noise (*string\_view* noisetype, float A = 0.0f, float B = 0.1f, bool mono = false, int seed = 0, ROI roi = {}, int nthreads = 0)

Return an image of “noise” in every pixel and channel specified by the roi. There are several noise types to choose from, and each behaves differently and has a different interpretation of the A and B parameters:

- “gaussian” adds Gaussian (normal distribution) noise values with mean value A and standard deviation B.
- “uniform” adds noise values uniformly distributed on range [A,B).
- “salt” changes to value A a portion of pixels given by B.

If the `mono` flag is true, a single noise value will be applied to all channels specified by `roi`, but if `mono` is false, a separate noise value will be computed for each channel in the region.

The random number generator is actually driven by a hash on the “image

space” coordinates and channel, independently of the pixel data window of `dst` or the `ROI`. Choosing different seed values will result in a different pattern, but for the same seed value, the noise at a given pixel coordinate (x,y,z) channel c will be completely deterministic and repeatable.

Examples:

```
// Create a new 256x256 field of grayscale uniformly distributed noise on [0,
↪1)
ImageBuf A = ImageBufAlgo::noise ("uniform", 0.0f /*min*/, 1.0f /*max*/,
                                true /*mono*/, 1 /*seed*/, ROI(0,256,0,256,0,1,0,3));

// Add color Gaussian noise to an existing image
ImageBuf B ("tahoe.jpg");
ImageBufAlgo::noise (B, "gaussian", 0.0f /*mean*/, 0.1f /*stddev*/,
```

(continues on next page)

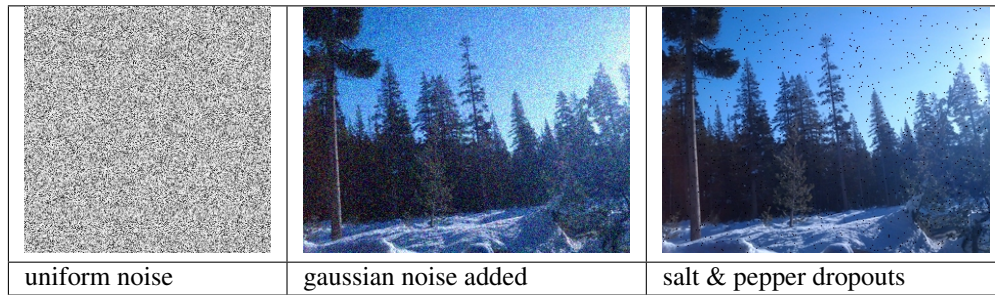
(continued from previous page)

```

        false /*mono*/, 1 /*seed*/);

// Use salt and pepper noise to make occasional random dropouts
ImageBuf C ("tahoe.jpg");
ImageBufAlgo::noise (C, "salt", 0.0f /*value*/, 0.01f /*portion*/,
        true /*mono*/, 1 /*seed*/);

```



Result-as-parameter version:

```

bool OIIO::ImageBufAlgo::noise (ImageBuf &dst, string_view noisetype, float
                                A = 0.0f, float B = 0.1f, bool mono = false, int
                                seed = 0, ROI roi = {}, int nthreads = 0)
    Write to an existing image dst (allocating if it is uninitialized).

```

## 10.2.5 Drawing shapes: points, lines, boxes

```

bool OIIO::ImageBufAlgo::render_point (ImageBuf &dst, int x, int y, cspan<float> color = 1.0f,
                                        ROI roi = {}, int nthreads = 0)

```

Render a single point at (x,y) of the given color “over” the existing image dst. If there is no alpha channel, the color will be written unconditionally (as if the alpha is 1.0).

Examples:

```

ImageBuf A (ImageSpec (640, 480, 4, TypeDesc::FLOAT));
float red[4] = { 1, 0, 0, 1 };
ImageBufAlgo::render_point (A, 50, 100, red);

```

```

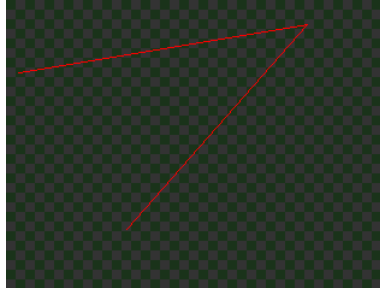
bool OIIO::ImageBufAlgo::render_line (ImageBuf &dst, int x1, int y1, int x2, int y2, cspan<float>
                                      color = 1.0f, bool skip_first_point = false, ROI roi = {}, int
                                      nthreads = 0)

```

Render a line from pixel (x1,y1) to (x2,y2) into dst, doing an “over” of the color (if it includes an alpha channel) onto the existing data in dst. The color should include as many values as `roi.chend-1`. The *ROI* can be used to limit the pixel area or channels that are modified, and default to the entirety of dst. If `skip_first_point` is true, the first point (x1, y1) will not be drawn (this can be helpful when drawing poly-lines, to avoid double-rendering of the vertex positions).

Examples:

```
ImageBuf A (ImageSpec (640, 480, 4, TypeDesc::FLOAT));
float red[4] = { 1, 0, 0, 1 };
ImageBufAlgo::render_line (A, 10, 60, 250, 20, red);
ImageBufAlgo::render_line (A, 250, 20, 100, 190, red, true);
```

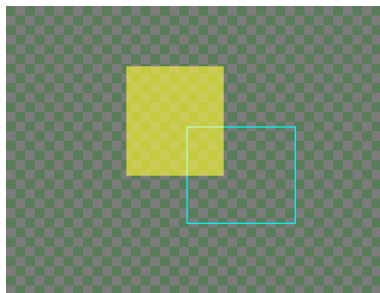


```
bool OIIO::ImageBufAlgo::render_box (ImageBuf &dst, int x1, int y1, int x2, int y2, cspan<float>
                                     color = 1.0f, bool fill = false, ROI roi = {}, int nthreads =
                                     0)
```

Render a filled or unfilled box with corners at pixels (x1,y1) and (x2,y2) into dst, doing an “over” of the color (if it includes an alpha channel) onto the existing data in dst. The color must include as many values as roi.chend-1. The ROI can be used to limit the pixel area or channels that are modified, and default to the entirety of dst. If fill is true, the box will be completely filled in, otherwise only its outline will be drawn.

Examples:

```
ImageBuf A (ImageSpec (640, 480, 4, TypeDesc::FLOAT));
float cyan[4] = { 1, 0, 0, 1 };
ImageBufAlgo::render_box (A, 150, 100, 240, 180, cyan);
float yellow_transparent[4] = { 0.5, 0.5, 0, 0.5 };
ImageBufAlgo::render_box (A, 100, 50, 180, 140, yellow_transparent, true);
```



## 10.2.6 Drawing text

```
bool OIIO::ImageBufAlgo::render_text (ImageBuf &dst, int x, int y, string_view text, int fontsize
                                     = 16, string_view fontname = "", cspan<float> textcolor =
                                     1.0f, TextAlignX alignx = TextAlignX::Left, TextAlignY
                                     aligny = TextAlignY::Baseline, int shadow = 0, ROI roi =
                                     {}, int nthreads = 0)
```

Render a text string (encoded as UTF-8) into image `dst`. If the `dst` image is not yet initialized, it will be initialized to be a black background exactly large enough to contain the rasterized text. If `dst` is already initialized, the text will be rendered into the existing image by essentially doing an “over” of the character into the existing pixel data.

### Parameters

- `dst`: Destination ImageBuf text is rendered into this image.
- `x/y`: The position to place the text.
- `text`: The text to draw.
- `fontsize/fontname`: Size and name of the font. If the name is not a full pathname to a font file, it will search for a matching font, defaulting to some reasonable system font if not supplied at all), and with a nominal height of `fontsize` (in pixels).
- `textcolor`: Color for drawing the text, defaulting to opaque white (1.0,1.0,...) in all channels if not supplied. If provided, it is expected to point to a float array of length at least equal to `R.spec().nchannels`, or defaults will be chosen for you).
- `alignx/aligny`: The default behavior is to align the left edge of the character baseline to (x,y). Optionally, `alignx` and `aligny` can override the alignment behavior, with horizontal alignment choices of `TextAlignX::Left`, `Right`, and `Center`, and vertical alignment choices of `Baseline`, `Top`, `Bottom`, or `Center`.
- `shadow`: If nonzero, a “drop shadow” of this radius will be used to make the text look more clear by dilating the alpha channel of the composite (makes a black halo around the characters).

Examples:

```
ImageBufAlgo::render_text (ImgA, 50, 100, "Hello, world");

float red[] = { 1, 0, 0, 1 };
ImageBufAlgo::render_text (ImgA, 100, 200, "Go Big Red!",
                           60, "Arial Bold", red);

float white[] = { 1, 1, 1, 1 };
ImageBufAlgo::render_text (ImgB, 320, 240, "Centered",
                           60, "Arial Bold", white,
                           TextAlignX::Center, TextAlignY::Center);
```





*ROI* `OIOO::ImageBufAlgo::text_size (string_view text, int fontsize = 16, string_view fontname = "")`

The helper function `text_size()` merely computes the dimensions of the text, returning it as an *ROI* relative to the left side of the baseline of the first character. Only the *x* and *y* dimensions of the *ROI* will be used. The *x* dimension runs from left to right, and *y* runs from top to bottom (image coordinates). For a failure (such as an invalid font name), the *ROI* will return `false` if you call its `defined()` method.

Example:

```
// Render text centered in the image, using text_size to find out
// the size we will need and adjusting the coordinates.
ImageBuf A (ImageSpec (640, 480, 4, TypeDesc::FLOAT));
ROI Aroi = A.roi();
ROI size = ImageBufAlgo::text_size ("Centered", 48, "Courier New");
if (size.defined()) {
    int x = Aroi.xbegin + Aroi.width()/2 - (size.xbegin + size.width()/2);
    int y = Aroi.ybegin + Aroi.height()/2 - (size.ybegin + size.height()/2);
    ImageBufAlgo::render_text (A, x, y, "Centered", 48, "Courier New");
}
```

## 10.3 Image transformations and data movement

### 10.3.1 Shuffling channels

`ImageBuf OIOO::ImageBufAlgo::channels (const ImageBuf &src, int nchannels, cspan<int> channelorder, cspan<float> channelvalues = {}, cspan<std::string> newchannelnames = {}, bool shuffle_channel_names = false, int nthreads = 0)`

Generic channel shuffling: return (or store in `dst`) a copy of `src`, but with channels in the order `channelorder[0..nchannels-1]` (or set to a constant value, designated by `channelorder[0] = -1` and having the fill value in `channelvalues[i]`). In-place operation is allowed (i.e., `dst` and `src` the same image, but an extra copy will occur).

#### Parameters

- `nchannels`: The total number of channels that will be set up in the `dst` image.

- **channelorder**: For each channel in `dst`, the index of the `src` channel from which to copy. Any `channelorder[i] < 0` indicates that the channel `i` should be filled with constant value `channelvalues[i]` rather than copy any channel from `src`. If `channelorder` itself is empty, the implied channel order will be `{0, 1, ..., nchannels-1}`, meaning that it's only renaming channels, not reordering them.
- **channelvalues**: Fill values for color channels in which `channelorder[i] < 0`.
- **newchannelnames**: An array of new channel names. Channels for which this specifies an empty string will have their name taken from the `src` channel that was copied. If `newchannelnames` is entirely empty, all channel names will simply be copied from `src`.
- **shuffle\_channel\_names**: If true, the channel 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. If false (the default), If false, the resulting `dst` image will have default channel names in the usual order ("R", "G", etc.), but i

Examples:

```
// Copy the first 3 channels of an RGBA, drop the alpha
ImageBuf RGBA (...); // assume it's initialized, 4 chans
ImageBuf RGB = ImageBufAlgo::channels (RGBA, 3, {} /*default ordering*/);

// Copy just the alpha channel, making a 1-channel image
ImageBuf Alpha = ImageBufAlgo::channels (RGBA, 1, 3 /*alpha_channel*/);

// Swap the R and B channels into an existing image
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" };
ImageBuf RGBA = ImageBufAlgo::channels (RGB, 4, channelorder,
                                         channelvalues, channelnames);

// Simple copying of channels from dst to src but fixing the number
// of channels dropping those in excess, or adding 0.0-filled channels
// if there is a shortfall)
ImageBuf out = ImageBufAlgo::channels (RGBA, nchannels, {}, {}, {}, true);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::channels (ImageBuf &dst, const ImageBuf &src,
                                   int nchannels, cspan<int> channelorder,
                                   cspan<float> channelvalues = {},
                                   cspan<std::string> newchannelnames =
                                   {}, bool shuffle_channel_names = false,
                                   int nthreads = 0)
```

Write to an existing image `dst` (allocating if it is uninitialized).

`ImageBuf OIIO::ImageBufAlgo::channel_append(const ImageBuf &A, const ImageBuf &B, ROI roi = {}, 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 (RGBA, Z);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::channel_append (ImageBuf &dst, const Image-
                                         Buf &A, const ImageBuf &B,
                                         ROI roi = {}, int nthreads = 0)
    Write to an existing image dst (allocating if it is uninitialized).
```

`ImageBuf OIIO::ImageBufAlgo::copy(const ImageBuf &src, TypeDesc convert = TypeUnknown, ROI roi = {}, int nthreads = 0)`

Return the specified region of pixels of `src` as specified by `roi` (which will default to the whole of `src`, optionally with the pixel type overridden by `convert` (if it is not `TypeUnknown`)).

Examples:

```
// Set B to be A, but converted to float
ImageBuf A (...); // Assume initialized
ImageBuf B = ImageBufAlgo::copy (A, TypeDesc::FLOAT);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::copy (ImageBuf &dst, const ImageBuf &src, Type-
                               Desc convert = TypeUnknown, ROI roi = {},
                               int nthreads = 0)
    Write to an existing image dst (allocating if it is uninitialized). If dst is not already
    initialized, it will be set to the same size as roi (defaulting to all of src)
```

`ImageBuf OIIO::ImageBufAlgo::crop(const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

Return the specified region of `src` as an image, without altering its position in the image plane.

Pixels from `src` which are outside `roi` will not be copied, and new black pixels will be added for regions of `roi` which were outside the data window 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 (A, ROI(0,200,0,100));
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::crop (ImageBuf &dst, const ImageBuf &src, ROI
                               roi = {}, int nthreads = 0)
    Write to an existing image dst (allocating if it is uninitialized).
```

```
ImageBuf OIIO::ImageBufAlgo::cut (const ImageBuf &src, ROI roi = {}, int nthreads = 0)
```

Return the designated region of `src`, but repositioned to the image origin and with the full/display window set to exactly cover the new pixel data 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 (A, ROI(50,250,200,300));
// Note: B will have origin 0,0, NOT (50,200).
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::cut (ImageBuf &dst, const ImageBuf &src, ROI
                              roi = {}, int nthreads = 0)
    Write to an existing image dst (allocating if it is uninitialized).
```

```
bool OIIO::ImageBufAlgo::paste (ImageBuf &dst, int xbegin, int ybegin, int zbegin, int chbegin,
                                const ImageBuf &src, ROI srcroi = {}, int nthreads = 0)
```

Copy `src` pixels within `srcroi` into the `dst` image, offset so that source location (0,0,0) will be copied to destination location (xbegin,ybegin,zbegin). If the `srcroi` is `ROI::All()`, the entirety of the data window of `src` will be used. It will copy into channels[chbegin...], as many channels as are described by `srcroi`. Pixels or channels of `src` inside `srcroi` will replace the corresponding destination pixels entirely, whereas `src` pixels outside of `srcroi` will not be copied and the corresponding offset pixels of `dst` will not be altered.

Examples:

```
// Paste small.exr on top of big.exr, offset by (100,100)
ImageBuf Big ("big.exr");
ImageBuf Small ("small.exr");
ImageBufAlgo::paste (Big, 100, 100, 0, 0, Small);
```

group **rotateN**

Return (or copy into `dst`) a rotated copy of the image pixels of `src`, in 90 degree increments. Pictorially:

rotate90	rotate180	rotate270
-----	-----	-----

(continues on next page)

(continued from previous page)

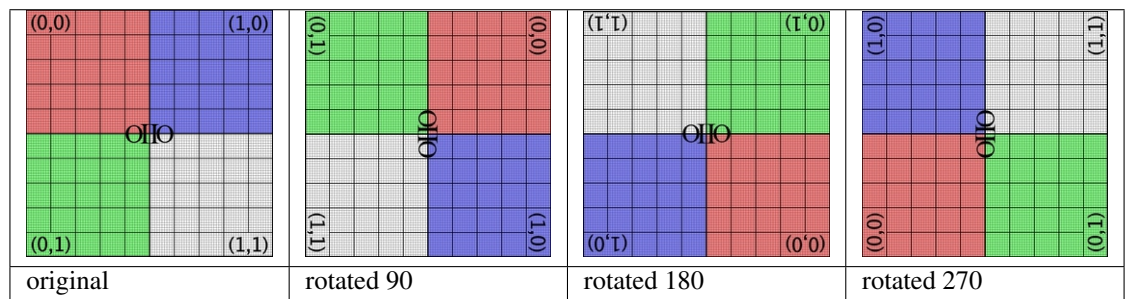
AB	-->	CA	AB	-->	DC	AB	-->	BD
CD		DB	CD		BA	CD		AC

Functions

```
ImageBuf rotate90 (const ImageBuf &src, ROI roi = {}, int nthreads = 0)
ImageBuf rotate180 (const ImageBuf &src, ROI roi = {}, int nthreads = 0)
ImageBuf rotate270 (const ImageBuf &src, ROI roi = {}, int nthreads = 0)
bool rotate90 (ImageBuf &dst, const ImageBuf &src, ROI roi = {}, int nthreads = 0)
bool rotate180 (ImageBuf &dst, const ImageBuf &src, ROI roi = {}, int nthreads = 0)
bool rotate270 (ImageBuf &dst, const ImageBuf &src, ROI roi = {}, int nthreads = 0)
```

Examples:

```
ImageBuf A ("grid.jpg");
ImageBuf R90 = ImageBufAlgo::rotate90 (A);
ImageBuf R170 = ImageBufAlgo::rotate180 (A);
ImageBuf R270 = ImageBufAlgo::rotate270 (A);
```



group flip-flop-transpose

Return (or copy into dst) a subregion of src, but with the scanlines exchanged vertically (flip), or columns exchanged horizontally (flop), or transposed across the diagonal by swapping rows for columns (transpose) within the display/full window. In other words,

flip			flop			transpose		
-----			-----			-----		
AB	-->	CD	AB	-->	BA	AB	-->	AC
CD		AB	CD		DC	CD		BD

## Functions

`ImageBuf flip (const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

`ImageBuf flop (const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

`ImageBuf transpose (const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

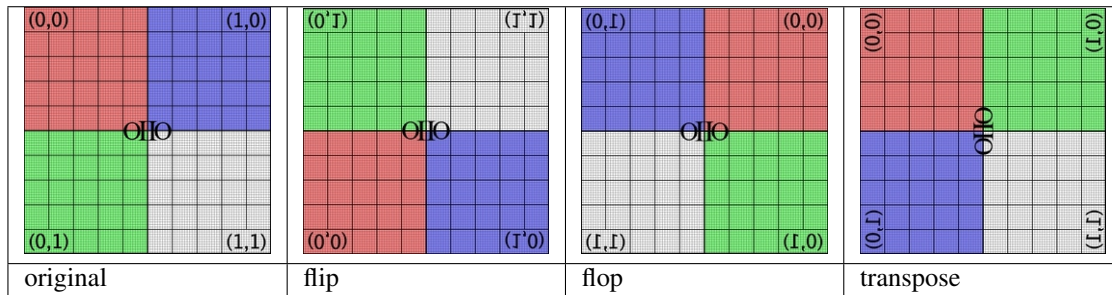
`bool flip (ImageBuf &dst, const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

`bool flop (ImageBuf &dst, const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

`bool transpose (ImageBuf &dst, const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

Examples:

```
ImageBuf A ("grid.jpg");
ImageBuf B;
B = ImageBufAlgo::flip (A);
B = ImageBufAlgo::flop (A);
B = ImageBufAlgo::transpose (A);
```



`ImageBuf OIIO::ImageBufAlgo::reorient (const ImageBuf &src, int nthreads = 0)`

Return (or store into `dst`) a copy of `src`, but with whatever series of rotations, flips, or flops are necessary to transform the pixels into the configuration suggested by the “Orientation” metadata of the image (and the “Orientation” metadata is then set to 1, ordinary orientation).

Examples:

```
ImageBuf A ("tahoe.jpg");
A = ImageBufAlgo::reorient (A);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::reorient (ImageBuf &dst, const ImageBuf &src,
                                     int nthreads = 0)
    Write to an existing image dst (allocating if it is uninitialized).
```

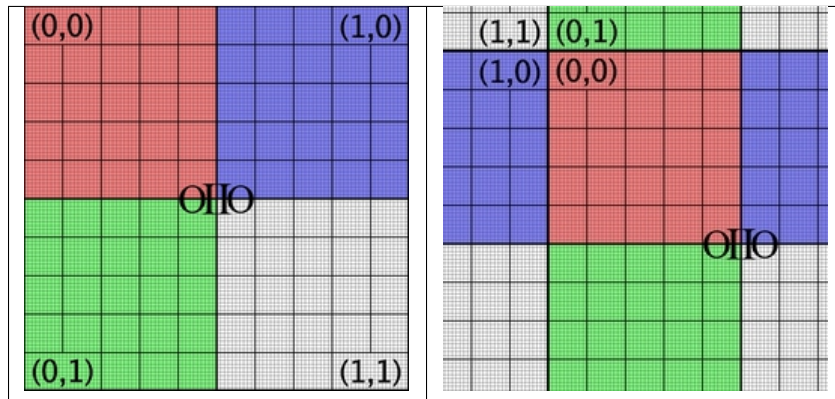
`ImageBuf OIIO::ImageBufAlgo::circular_shift (const ImageBuf &src, int xshift, int yshift, int zshift = 0, ROI roi = {}, int nthreads = 0)`

Return a subregion of `src`, 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 ("grid.jpg");
ImageBuf B = ImageBufAlgo::circular_shift (A, 70, 30);
```



Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::circular_shift (ImageBuf &dst, const ImageBuf &src,
                                          int xshift, int yshift, int zshift = 0, ROI roi
                                          = {}, int nthreads = 0)
```

Write to an existing image dst (allocating if it is uninitialized).

#### group **rotate**

Rotate the src image by the angle (in radians, with positive angles clockwise). When center\_x and center\_y are supplied, they denote the center of rotation; in their absence, the rotation will be about the center of the image's display window.

Only the pixels (and channels) of dst that are specified by roi will be copied from the rotated src; the default roi is to alter all the pixels in dst. If dst is uninitialized, it will be resized to be an ImageBuf large enough to hold the rotated image if recompute\_roi is true, or will have the same ROI as src if recompute\_roi is false. It is an error to pass both an uninitialized dst and an undefined roi.

The filter is used to weight the src pixels falling underneath it for each dst pixel. The caller may specify a reconstruction filter by name and width (expressed in pixels units of the dst image), or rotate() will choose a reasonable default high-quality default filter (lanczos3) if the empty string is passed, and a reasonable filter width if filterwidth is 0. (Note that some filter choices only make sense with particular width, in which case this filterwidth parameter may be ignored.)

### Functions

```
ImageBuf rotate (const ImageBuf &src, float angle, string_view filtername = string_view(), float filterwidth = 0.0f, bool recompute_roi = false, ROI roi = {}, int nthreads = 0)
```

```
ImageBuf rotate (const ImageBuf &src, float angle, Filter2D *filter, bool recompute_roi = false, ROI roi = {}, int nthreads = 0)
```

```
ImageBuf rotate (const ImageBuf &src, float angle, float center_x, float center_y, string_view filtername = string_view(), float filterwidth = 0.0f, bool recompute_roi = false, ROI roi = {}, int nthreads = 0)
```

```
ImageBuf rotate (const ImageBuf &src, float angle, float center_x, float center_y, Filter2D *filter,
                 bool recompute_roi = false, ROI roi = {}, int nthreads = 0)
```

```
bool rotate (ImageBuf &dst, const ImageBuf &src, float angle, string_view filename =
             string_view(), float filterwidth = 0.0f, bool recompute_roi = false, ROI roi = {}, int nthreads
             = 0)
```

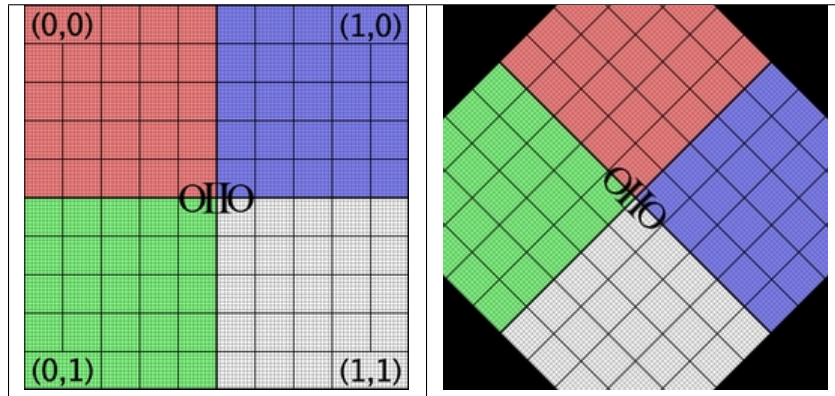
```
bool rotate (ImageBuf &dst, const ImageBuf &src, float angle, Filter2D *filter, bool recompute_roi
             = false, ROI roi = {}, int nthreads = 0)
```

```
bool rotate (ImageBuf &dst, const ImageBuf &src, float angle, float center_x, float center_y,
             string_view filename = string_view(), float filterwidth = 0.0f, bool recompute_roi = false,
             ROI roi = {}, int nthreads = 0)
```

```
bool rotate (ImageBuf &dst, const ImageBuf &src, float angle, float center_x, float center_y, Fil-
             ter2D *filter, bool recompute_roi = false, ROI roi = {}, int nthreads = 0)
```

Examples:

```
ImageBuf Src ("tahoe.exr");
ImageBuf Dst = ImageBufAlgo::rotate (Src, 45.0);
```



### group **resize**

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 either (a) explicitly pass a reconstruction `filter`, or (b) specify one by `filename` and `filterwidth`. If `filter` is `nullptr` or if `filename` is the empty string `resize()` will choose a reasonable high-quality default (blackman-harris when upsizing, lanczos3 when downsizing). 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.



## Functions

`ImageBuf` **resize** (`const` `ImageBuf` &*src*, *string\_view* *filtername* = "", `float` *filterwidth* = 0.0f, *ROI* *roi* = {}, `int` *nthreads* = 0)

`ImageBuf` **resize** (`const` `ImageBuf` &*src*, `Filter2D` \**filter*, *ROI* *roi* = {}, `int` *nthreads* = 0)

`bool` **resize** (`ImageBuf` &*dst*, `const` `ImageBuf` &*src*, *string\_view* *filtername* = "", `float` *filterwidth* = 0.0f, *ROI* *roi* = {}, `int` *nthreads* = 0)

`bool` **resize** (`ImageBuf` &*dst*, `const` `ImageBuf` &*src*, `Filter2D` \**filter*, *ROI* *roi* = {}, `int` *nthreads* = 0)

Examples:

```
// Resize the image to 640x480, using the default filter
ImageBuf Src ("tahoe.exr");
ROI roi (0, 640, 0, 480, 0, 1, /*chans:*/ 0, Src.nchannels());
ImageBuf Dst = ImageBufAlgo::resize (Src, "", 0, roi);
```

`ImageBuf` `OIIO::ImageBufAlgo::resample` (`const` `ImageBuf` &*src*, `bool` *interpolate* = true, *ROI* *roi* = {}, `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).

For “deep” images, this function returns copies the closest source pixel needed, rather than attempting to interpolate deep pixels (regardless of the value of *interpolate*).

Examples:

```
// Resample quickly to 320x240, using the default filter
ImageBuf Src ("tahoe.exr");
ROI roi (0, 320, 0, 240, 0, 1, /*chans:*/ 0, Src.nchannels());
ImageBuf Dst = ImageBufAlgo::resample (Src, false, roi);
```

Result-as-parameter version:

`bool` `OIIO::ImageBufAlgo::resample` (`ImageBuf` &*dst*, `const` `ImageBuf` &*src*, `bool` *interpolate* = true, *ROI* *roi* = {}, `int` *nthreads* = 0)

Write to an existing image *dst* (allocating if it is uninitialized).

### *group* **fit**

Fit *src* into *dst* (to a size specified by *roi*, if *dst* is not initialized), resizing but preserving its original aspect ratio. Thus, it will resize so be the largest size with the same aspect ratio that can fit inside the region, but will not stretch to completely fill it in both dimensions.

If `exact` is true, will result in an exact match on aspect ratio and centering (partial pixel shift if necessary), whereas `exact=false` will only preserve aspect ratio and centering to the precision of a whole pixel.

The filter is used to weight the `src` pixels falling underneath it for each `dst` pixel. The caller may specify a reconstruction filter by name and width (expressed in pixels units of the `dst` image), or `rotate()` will choose a reasonable default high-quality default filter (lanczos3) if the empty string is passed, and a reasonable filter width if `filterwidth` is 0. (Note that some filter choices only make sense with particular width, in which case this `filterwidth` parameter may be ignored.)

## Functions

`ImageBuf fit (const ImageBuf &src, string_view filename = "", float filterwidth = 0.0f, bool exact = false, ROI roi = {}, int nthreads = 0)`

`ImageBuf fit (const ImageBuf &src, Filter2D *filter, bool exact = false, ROI roi = {}, int nthreads = 0)`

`bool fit (ImageBuf &dst, const ImageBuf &src, string_view filename = "", float filterwidth = 0.0f, bool exact = false, ROI roi = {}, int nthreads = 0)`

`bool fit (ImageBuf &dst, const ImageBuf &src, Filter2D *filter, bool exact = false, ROI roi = {}, int nthreads = 0)`

Examples:

```
// Resize to fit into a max of 640x480, preserving the aspect ratio
ImageBuf Src ("tahoe.exr");
ROI roi (0, 640, 0, 480, 0, 1, /*chans:*/ 0, Src.nchannels());
ImageBuf Dst = ImageBufAlgo::fit (Src, "", 0, true, roi);
```

## group warp

Warp the `src` image using the supplied 3x3 transformation matrix.

Only the pixels (and channels) of `dst` that are specified by `roi` will be copied from the warped `src`; the default `roi` is to alter all the pixels in `dst`. If `dst` is uninitialized, it will be sized to be an `ImageBuf` large enough to hold the warped image if `recompute_roi` is true, or will have the same `ROI` as `src` if `recompute_roi` is false. It is an error to pass both an uninitialised `dst` and an undefined `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 `nullptr` 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.

## Functions

`ImageBuf warp (const ImageBuf &src, const Imath::M33f &M, string_view filename = string_view(), float filterwidth = 0.0f, bool recompute_roi = false, ImageBuf::WrapMode wrap = ImageBuf::WrapDefault, ROI roi = {}, int nthreads = 0)`

`ImageBuf warp (const ImageBuf &src, const Imath::M33f &M, const Filter2D *filter, bool recompute_roi = false, ImageBuf::WrapMode wrap = ImageBuf::WrapDefault, ROI roi = {}, int nthreads = 0)`

```
bool warp (ImageBuf &dst, const ImageBuf &src, const Imath::M33f &M, string_view filtername
= string_view(), float filterwidth = 0.0f, bool recompute_roi = false, ImageBuf::WrapMode
wrap = ImageBuf::WrapDefault, ROI roi = {}, int nthreads = 0)
```

```
bool warp (ImageBuf &dst, const ImageBuf &src, const Imath::M33f &M, const Filter2D *filter,
bool recompute_roi = false, ImageBuf::WrapMode wrap = ImageBuf::WrapDefault, ROI roi
= {}, int nthreads = 0)
```

Examples:

```
Imath::M33f M ( 0.7071068, 0.7071068, 0,
               -0.7071068, 0.7071068, 0,
               20,        -8.284271, 1);
ImageBuf Src ("tahoe.exr");
ImageBuf Dst = ImageBufAlgo::warp (dst, src, M, "lanczos3");
```

## 10.4 Image arithmetic

```
ImageBuf OIIO::ImageBufAlgo::add (Image_or_Const A, Image_or_Const B, ROI roi = {}, int
nthreads = 0)
```

Compute per-pixel sum  $A + B$ , returning the result image.

A and B may each either be an ImageBuf&, or a `cspan<float>` giving a per-channel constant, or a single constant used for all channels. (But at least one must be an image.)

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");
ROI roi = get_roi (A.spec());
roi.chbegin = 0; roi.chend = 3;
ImageBuf Sum = ImageBufAlgo::add (Sum, A, 0.2f, roi);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::add (ImageBuf &dst, Image_or_Const A, Image_or_Const B, ROI roi = {}, int nthreads = 0)
```

Write to an existing image `dst` (allocating if it is uninitialized).

```
ImageBuf OIIO::ImageBufAlgo::sub (Image_or_Const A, Image_or_Const B, ROI roi = {}, int
nthreads = 0)
```

Compute per-pixel signed difference  $A - B$ , returning the result image.

A and B may each either be an `ImageBuf&`, or a `cspan<float>` giving a per-channel constant, or a single constant used for all channels. (But at least one must be an image.)

Examples:

```
ImageBuf A ("a.exr");
ImageBuf B ("b.exr");
ImageBuf Diff = ImageBufAlgo::sub (A, B);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::sub (ImageBuf &dst, Image_or_Const A, Im-
                             age_or_Const B, ROI roi = {}, int nthreads =
                             0)
    Write to an exsisting image dst (allocating if it is uninitialized).
```

`ImageBuf OIIO::ImageBufAlgo::absdiff (Image_or_Const A, Image_or_Const B, ROI roi = {}, int nthreads = 0)`

Compute per-pixel absolute difference  $\text{abs}(A - B)$ , returning the result image.

A and B may each either be an `ImageBuf&`, or a `cspan<float>` giving a per-channel constant, or a single constant used for all channels. (But at least one must be an image.)

Examples:

```
ImageBuf A ("a.exr");
ImageBuf B ("b.exr");
ImageBuf Diff = ImageBufAlgo::absdiff (A, B);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::absdiff (ImageBuf &dst, Image_or_Const A, Im-
                                  age_or_Const B, ROI roi = {}, int nthreads
                                  = 0)
    Write to an exsisting image dst (allocating if it is uninitialized).
```

`ImageBuf OIIO::ImageBufAlgo::abs (const ImageBuf &A, ROI roi = {}, int nthreads = 0)`

Compute per-pixel absolute value  $\text{abs}(A)$ , returning the result image.

Examples:

```
ImageBuf A ("a.exr");
ImageBuf Abs = ImageBufAlgo::abs (A);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::abs (ImageBuf &dst, const ImageBuf &A, ROI roi
                              = {}, int nthreads = 0)
    Write to an exsisting image dst (allocating if it is uninitialized).
```

`ImageBuf OIIO::ImageBufAlgo::mul (Image_or_Const A, Image_or_Const B, ROI roi = {}, int  
nthreads = 0)`

Compute per-pixel product  $A * B$ , returning the result image.

Either both A and B are images, or one is an image and the other is a `cspan<float>` giving a per-channel constant or a single constant used for all 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, 0.5f, roi);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::mul (ImageBuf &dst, Image_or_Const A, Im-
                             age_or_Const B, ROI roi = {}, int nthreads =
                             0)
    Write to an exsisting image dst (allocating if it is uninitialized).
```

`ImageBuf OIIO::ImageBufAlgo::div (Image_or_Const A, Image_or_Const B, ROI roi = {}, int  
nthreads = 0)`

Compute per-pixel division  $A / B$ , returning the result image. Division by zero is defined to result in zero.

A is always an image, and B is either an image or a `cspan<float>` giving a per-channel constant or a single constant used for all channels.

Examples:

```
ImageBuf A ("a.exr");
ImageBuf B ("b.exr");
ImageBuf Result = ImageBufAlgo::div (Result, 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::div (A, A, 2.0f, roi);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::div (ImageBuf &dst, Image_or_Const A, Im-
                             age_or_Const B, ROI roi = {}, int nthreads =
                             0)
    Write to an exsisting image dst (allocating if it is uninitialized).
```

ImageBuf OIIO::ImageBufAlgo::**mad**(Image\_or\_Const A, Image\_or\_Const B, Image\_or\_Const C, *ROI* roi = {}, int nthreads = 0)

Compute per-pixel multiply-and-add  $A * B + C$ , returning the result image.

A, B, and C are each either an image, or a `cspan<float>` giving a per-channel constant or a single constant used for all channels. (Note: at least one must be an image.)

Examples:

```
ImageBuf A ("a.exr");
ImageBuf B ("b.exr");
ImageBuf C ("c.exr");
ImageBuf Result = ImageBufAlgo::mad (A, B, C);

// Compute the "inverse" A, which is 1.0-A, as A*(-1) + 1
// Do this in-place, and only for the first 3 channels (leave any
// alpha channel, if present, as it is).
ROI roi = get_roi (A.spec());
roi.chbegin = 0;  roi.chend = 3;
ImageBufAlgo::mad (A, A, -1.0, 1.0, roi);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::mad(ImageBuf &dst, Image_or_Const A, Image_or_Const B, Image_or_Const C, ROI roi = {}, int nthreads = 0)
```

Write to an existing image dst (allocating if it is uninitialized).

ImageBuf OIIO::ImageBufAlgo::**over**(const ImageBuf &A, const ImageBuf &B, *ROI* roi = {}, int nthreads = 0)

Return the composite of A over B using the Porter/Duff definition of “over”, returning true upon success and false for any of a variety of failures (as described below).

A and B (and dst, if already defined/allocated) must have valid alpha channels identified by their *ImageSpec* `alpha_channel` field. If A or B do not have alpha channels (as determined by those rules) or if the number of non-alpha channels do not match between A and B, *over()* will fail, returning false.

If dst is not already an initialized ImageBuf, it will be sized to encompass the minimal rectangular pixel region containing the union of the defined pixels of A and B, and with a number of channels equal to the number of non-alpha channels of A and B, plus an alpha channel. However, if dst is already initialized, it will not be resized, and the “over” operation will apply to its existing pixel data window. In this case, dst must have an alpha channel designated and must have the same number of non-alpha channels as A and B, otherwise it will fail, returning false.

A, B, and dst need not perfectly overlap in their pixel data windows; pixel values of A or B that are outside their respective pixel data window will be treated as having “zero” (0,0,0...) value.

Examples:

```
ImageBuf A ("fg.exr");
ImageBuf B ("bg.exr");
ImageBuf Composite = ImageBufAlgo::over (A, B);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::over(ImageBuf &dst, const ImageBuf &A,
                             const ImageBuf &B, ROI roi = {}, int
                             nthreads = 0)
    Write to an exsisting image dst (allocating if it is uninitialized).
```

```
ImageBuf OIIO::ImageBufAlgo::zover(const ImageBuf &A, const ImageBuf &B, bool
                                   z_zeroisinf = false, ROI roi = {}, int nthreads = 0)
```

Just like `ImageBufAlgo::over()`, but inputs A and B must have designated 'z' channels, and on a pixel-by-pixel basis, the z values will determine which of A or B will be considered the foreground or background (lower z is foreground). 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);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::zover(ImageBuf &dst, const ImageBuf &A,
                             const ImageBuf &B, bool z_zeroisinf =
                             false, ROI roi = {}, int nthreads = 0)
    Write to an exsisting image dst (allocating if it is uninitialized).
```

```
ImageBuf OIIO::ImageBufAlgo::invert(const ImageBuf &A, ROI roi = {}, int nthreads = 0)
```

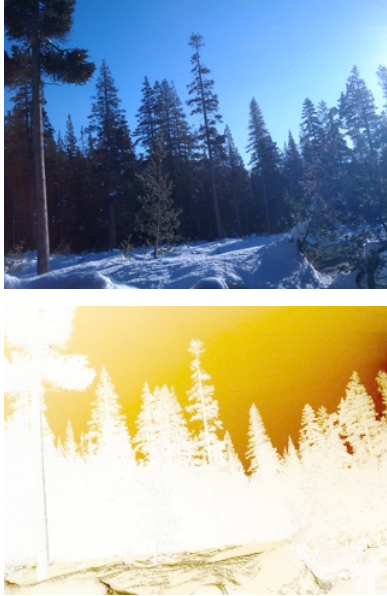
Compute per-pixel value inverse  $1.0 - A$  (which you can think of as roughly meaning switching white and black), returning the result image.

Tips for callers: (1) You probably want to set `roi` to restrict the operation to only the color channels, and not accidentally include alpha, z, or others. (2) There may be situations where you want to `unpremult()` before the invert, then `premult()` the result, so that you are computing the inverse of the unmasked color.

Examples:

```
ImageBuf A ("a.exr");
ImageBuf Inverse = ImageBufAlgo::invert (Inverse, A);

// In this example, we are careful to deal with alpha in an RGBA image.
// First we copy A to Inverse, un-premultiply the color values by alpha,
// invert just the color channels in-place, and then re-premultiply the
// colors by alpha.
roi = A.roi();
roi.chend = 3; // Restrict roi to only R,G,B
ImageBuf Inverse = ImageBufAlgo::unpremult (A);
ImageBufAlgo::invert (Inverse, Inverse, roi);
ImageBufAlgo::premult (Inverse, Inverse);
```



Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::invert (ImageBuf &dst, const ImageBuf &A, ROI
                                roi = {}, int nthreads = 0)
    Write to an exsisting image dst (allocating if it is uninitialized).
```

```
ImageBuf OIIO::ImageBufAlgo::pow (const ImageBuf &A, cspan<float> B, ROI roi = {}, int nthreads
                                   = 0)
```

Compute per-pixel raise-to-power  $A \wedge B$ . returning the result image. It is permitted for `dst` and `A` to be the same image.

`A` is always an image, and `B` is either an image or a `cspan<float>` giving a per-channel constant or a single constant used for all channels.

Examples:

```
// Gamma-correct by 2.2 channels 0-2 of the image, in-place
ROI roi = get_roi (A.spec());
roi.chbegin = 0;  roi.chend = 3;
ImageBufAlgo::pow (A, A, 1.0f/2.2f, roi);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::pow (ImageBuf &dst, const ImageBuf &A,
                              cspan<float> B, ROI roi = {}, int nthreads = 0)
    Write to an exsisting image dst (allocating if it is uninitialized).
```



ImageBuf OIIO::ImageBufAlgo::**channel\_sum**(const ImageBuf &src, *cspan*<float> weights = 1.0f, *ROI* roi = {}, int *nthreads* = 0)

Converts a multi-channel image into a one-channel image via a weighted sum of channels:

```
(channel[0]*weight[0] + channel[1]*weight[1] + ...)
```

returning the resulting one-channel image. The weights, if not supplied, default to { 1, 1, 1, ... }.

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, 0.0 };
ImageBuf A ("a.exr");
ImageBuf lum = ImageBufAlgo::channel_sum (A, luma_weights);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::channel_sum(ImageBuf &dst, const ImageBuf
                                     &src, cspan<float> weights = 1.0f,
                                     ROI roi = {}, int nthreads = 0)
```

Write to an existing image dst (allocating if it is uninitialized).

ImageBuf OIIO::ImageBufAlgo::**clamp**(const ImageBuf &src, *cspan*<float> min = -std::numeric\_limits<float>::max(), *cspan*<float> max = std::numeric\_limits<float>::max(), bool *clampalpha01* = false, *ROI* roi = {}, int *nthreads* = 0)

Return pixels of src with pixel values clamped as follows:

- min specifies the minimum clamp value for each channel (if min is empty, no minimum clamping is performed).
- max specifies the maximum clamp value for each channel (if max is empty, no maximum clamping is performed).
- If clampalpha01 is true, then additionally any alpha channel is clamped to the 0-1 range.

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] in-place
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);
```

Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::clamp (ImageBuf &dst, const Image-
    Buf &src, cspan<float> min = -
    std::numeric_limits<float>::max(),
    cspan<float> max =
    std::numeric_limits<float>::max(), bool
    clampalpha01 = false, ROI roi = {}, int
    nthreads = 0)
```

Write to an exsisting image dst (allocating if it is uninitialized).

```
ImageBuf OIIO::ImageBufAlgo::contrast_remap (const ImageBuf &src, cspan<float> black =
    0.0f, cspan<float> white = 1.0f, cspan<float>
    min = 0.0f, cspan<float> max = 1.0f,
    cspan<float> scontrast = 1.0f, cspan<float>
    sthresh = 0.5f, ROI = {}, int nthreads = 0)
```

Return pixel values that are a contrast-remap of the corresponding values of the `src` image, transforming pixel value domain [black, white] to range [min, max], either linearly or with optional application of a smooth sigmoidal remapping (if `scontrast` != 1.0).

The following steps are performed, in order:

1. Linearly rescale values [black, white] to [0, 1].
2. If `scontrast` != 1, apply a sigmoidal remapping where a larger `scontrast` value makes a steeper slope, and the steepest part is at value `sthresh` (relative to the new remapped value after steps 1 & 2; the default is 0.5).
3. Rescale the range of that result: 0.0 -> min and 1.0 -> max.

Values outside of the [black,white] range will be extrapolated to outside [min,max], so it may be prudent to apply a `clamp()` to the results.

The black, white, min, max, scontrast, sthresh parameters may each either be a single float value for all channels, or a span giving per-channel values.

You can use this function for a simple linear contrast remapping of [black, white] to [min, max] if you use the default values for sthresh. Or just a simple sigmoidal contrast stretch within the [0,1] range if you leave all other parameters at their defaults, or a combination of these effects. Note that if `black == white`, the result will be a simple binary thresholding where values < black map to min and values >= black map to max.

Examples:

```
ImageBuf A ("tahoe.tif");

// Simple linear remap that stretches input 0.1 to black, and input
// 0.75 to white.
ImageBuf linstretch = ImageBufAlgo::contrast_remap (A, 0.1f, 0.75f);

// Remapping 0->1 and 1->0 inverts the colors of the image,
// equivalent to ImageBufAlgo::invert().
ImageBuf inverse = ImageBufAlgo::contrast_remap (A, 1.0f, 0.0f);

// Use a sigmoid curve to add contrast but without any hard cutoffs.
// Use a contrast parameter of 5.0.
```

(continues on next page)

(continued from previous page)

```
ImageBuf sigmoid = ImageBufAlgo::contrast_remap (a, 0.0f, 1.0f,
                                                0.0f, 1.0f, 5.0f);
```



Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::contrast_remap (ImageBuf &dst, const ImageBuf &src,
                                          cspan<float> black = 0.0f, cspan<float>
                                          white = 1.0f, cspan<float> min = 0.0f, cspan<float>
                                          max = 1.0f, cspan<float> scontrast =
                                          1.0f, cspan<float> sthresh = 0.5f, ROI = {},
                                          int nthreads = 0)
```

Write to an existing image *dst* (allocating if it is uninitialized).

#### group **color\_map**

Remap value range by spline or name

Return (or copy into *dst*) pixel values determined by looking up a color map using values of the source image, using either the channel specified by *srcchannel*, or the luminance of *src*'s RGB if *srcchannel* is -1. This happens for all pixels within the *ROI* (which defaults to all of *src*), and if *dst* is not already initialized, it will be initialized to the *ROI* and with color channels equal to *channels*.

In the variant that takes a *knots* parameter, this specifies the values of a linearly-interpolated color map given by *knots*[*nknots*\**channels*]. An input value of 0.0 is mapped to *knots*[0..*channels*-1] (one value for each color channel), and an input value of 1.0 is mapped to *knots*[(*nknots*-1)\**channels*..*knots*.size()-1].

In the variant that takes a *mapname* parameter, this is the name of a color map. Recognized map names include: “inferno”, “viridis”, “magma”, “plasma”, all of which are perceptually uniform, strictly increasing in luminance, look good when converted to grayscale, and work for people with all types of colorblindness. Also “turbo” has most of these properties (except for being strictly increasing in luminance) and is a nice rainbow-like pattern. Also supported are the following color maps that do not have those desirable qualities (and are thus not recommended, but are present for back-compatibility or for use by clueless people): “blue-red”, “spectrum”, and “heat”. In all cases, the implied *channels* is 3.

## Functions

`ImageBuf color_map (const ImageBuf &src, int srcchannel, int nknots, int channels, cspan<float> knots, ROI roi = {}, int nthreads = 0)`

`ImageBuf color_map (const ImageBuf &src, int srcchannel, string_view mapname, ROI roi = {}, int nthreads = 0)`

`bool color_map (ImageBuf &dst, const ImageBuf &src, int srcchannel, int nknots, int channels, cspan<float> knots, ROI roi = {}, int nthreads = 0)`

`bool color_map (ImageBuf &dst, const ImageBuf &src, int srcchannel, string_view mapname, ROI roi = {}, int nthreads = 0)`

Examples:

```
// Use luminance of a.exr (assuming Rec709 primaries and a linear
// scale) and map to a spectrum-like palette.:
ImageBuf A ("a.exr");
ImageBuf B = ImageBufAlgo::color_map (A, -1, "turbo");

float mymap[] = { 0.25, 0.25, 0.25, 0, 0.5, 0, 1, 0, 0 };
B = ImageBufAlgo::color_map (A, -1 /* use luminance */,
                             3 /* num knots */, 3 /* channels */,
                             mymap);
```



### group range

Nonlinear range remapping for contrast preservation

`rangecompress()` returns (or copy into `dst`) all pixels and color channels of `src` within region `roi` (defaulting to all the defined pixels of `dst`), rescaling their range with a logarithmic transformation. Alpha and z channels are not transformed.

`rangeexpand()` performs the inverse transformation (logarithmic back into linear).

If `useluma` is true, the luma of channels [`roi.chbegin..roi.chbegin+2`] (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 color shift).

The purpose of these function is as follows: 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). By compressing the range pixel values, then performing the operation, then expanding the range of the result again, the result can be much more pleasing (even if not exactly correct).

## Functions

ImageBuf **rangecompress** (**const** ImageBuf &src, bool *useluma* = false, *ROI* roi = {}, int *nthreads* = 0)

ImageBuf **rangeexpand** (**const** ImageBuf &src, bool *useluma* = false, *ROI* roi = {}, int *nthreads* = 0)

bool **rangecompress** (ImageBuf &dst, **const** ImageBuf &src, bool *useluma* = false, *ROI* roi = {}, int *nthreads* = 0)

bool **rangeexpand** (ImageBuf &dst, **const** ImageBuf &src, bool *useluma* = false, *ROI* roi = {}, int *nthreads* = 0)

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 (Src);

// 3. Now do the resize
ImageBuf Dst = ImageBufAlgo::resize (Compressed, "lanczos3", 6.0,
                                     ROI(0, 640, 0, 480));

// 4. Expand range to be linear again (operate in-place)
ImageBufAlgo::rangeexpand (Dst, Dst);
```

## 10.5 Image comparison and statistics

PixelStats OIIO::ImageBufAlgo::computePixelStats (**const** ImageBuf &src, *ROI* roi = {}, int *nthreads* = 0)

Compute statistics about the *ROI* of the *src* image, returning a PixelStats structure. Upon success, the returned vectors in the result structure will have size == *src*.nchannels(). If there is a failure, the vector sizes will be 0 and an error will be set in *src*.

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;
};
```

Examples:

```
ImageBuf A ("a.exr");
ImageBufAlgo::PixelStats stats;
```

(continues on next page)

(continued from previous page)

```

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";
}

```

CompareResults OIIO::ImageBufAlgo::compare (const ImageBuf &A, const ImageBuf &B, float failthresh, float warnthresh, ROI roi = {}, 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;
    bool error;
};

```

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";
}

```

```
int OIIO::ImageBufAlgo::compare_Yee (const ImageBuf &A, const ImageBuf &B, CompareRe-
                                     sults &result, float luminance = 100, float fov = 45, ROI roi
                                     = {}, int nthreads = 0)
```

Compare two images using Hector Yee's perceptual metric, returning the number of pixels that fail the comparison. Only the first three channels (or first three channels specified by *roi*) are compared. Free parameters are the ambient luminance in the room and the field of view of the image display; our defaults are probably reasonable guesses for an office environment. The 'result' structure will store the maxerror, and the maxx, maxy, maxx of the pixel that failed most severely. (The other fields of the CompareResults are not used for Yee comparison.)

Works for all pixel types. But it's basically meaningless if the first three channels aren't RGB in a linear color space that sort of resembles AdobeRGB.

Return true on success, false on error.

```
bool OIIO::ImageBufAlgo::isConstantColor (const ImageBuf &src, float threshold = 0.0f,
                                          span<float> color = {}, ROI roi = {}, int nthreads
                                          = 0)
```

Do all pixels within the *ROI* have the same values for channels [*roi.chbegin*..*roi.chend*-1], within a tolerance of +/- *threshold*? If so, return true and store that color in *color[chbegin...chend-1]* (if *color* is not empty); otherwise return false. 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");
std::vector<float> color (A.nchannels());
if (ImageBufAlgo::isConstantColor (A, color)) {
    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 OIIO::ImageBufAlgo::isConstantChannel (const ImageBuf &src, int channel, float val,
                                             float threshold = 0.0f, ROI roi = {}, int nthreads
                                             = 0)
```

Does the requested channel have a given value (within a tolerance of +/- *threshold*) for every channel within the *ROI*? (For this function, the *ROI*'s *chbegin*/*chend* are ignored.) Return true if so, otherwise return false. 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");
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))
```

(continues on next page)

(continued from previous page)

```
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 OIIO::ImageBufAlgo::isMonochrome (const ImageBuf &src, float threshold = 0.0f, ROI roi =
                                     {}, int nthreads = 0)
```

Is the image monochrome within the *ROI*, i.e., for every pixel within the region, do all channels [roi.chbegin, roi.chend) have the same value (within a tolerance of +/- threshold)? 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 OIIO::ImageBufAlgo::color_count (const ImageBuf &src, imagesize_t *count, int ncolors,
                                     cspan<float> color, cspan<float> eps = 0.001f, ROI roi =
                                     {}, int nthreads = 0)
```

Count how many pixels in 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).

*eps*[0..*nchans*-1] are the error tolerances for a match, for each channel. Setting *eps*[*c*] = *numeric\_limits*<float>::max() will effectively make it ignore the channel. The default *eps* is 0.001 for all channels (this value is chosen because it requires exact matches for 8 bit images, but allows a wee bit of imprecision for float images).

Upon success, return *true* and store the number of pixels that matched each color *count*[..*ncolors*-1]. If there is an error, returns *false* and sets an appropriate error message set in *src*.

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;
```

(continues on next page)



(continued from previous page)

```

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 OIIO::ImageBufAlgo::color\_range\_check** (**const** ImageBuf &src, imagesize\_t \*lowcount, imagesize\_t \*highcount, imagesize\_t \*inrange-count, *cspan*<float> low, *cspan*<float> high, *ROI* roi = {}, 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 \*inrange-count. 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, inrange-count;
ImageBufAlgo::color_range_check (A, &lowcount, &highcount, &inrange-count,
                                low, high, roi);
std::cout << lowcount << " pixels had components < 0\n";
std::cout << highcount << " pixels had components > 1\n";
std::cout << inrange-count << " pixels were fully within [0,1] range\n";

```

***ROI* OIIO::ImageBufAlgo::nonzero\_region** (**const** ImageBuf &src, *ROI* roi = {}, int nthreads = 0)

Find the minimal rectangular region within roi (which defaults to the entire pixel data window of src) that consists of nonzero pixel values. In other words, gives the region that is a “shrink-wraps” of src to exclude black border pixels. Note that if the entire image was black, the *ROI* returned will contain no pixels.

For “deep” images, this function returns the smallest *ROI* that contains all pixels that contain depth samples, and excludes the border pixels that contain no depth samples at all.

Examples:

```

ImageBuf A ("a.exr");
ROI shrunk = ImageBufAlgo::nonzero_region (A);
if (shrunk.undefined())
    std::cout << "All pixels were empty\n";
else
    std::cout << "Non-empty region was " << shrunk << "\n";

```

```

std::string OIIO::ImageBufAlgo::computePixelHashSHA1 (const ImageBuf &src, string_view
                                                         extrainfo = "", ROI roi = {}, int block-
                                                         size = 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` (if `nthreads` is 0, it will use the global OIIO thread count). 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);

```

```

std::vector<imagesize_t> OIIO::ImageBufAlgo::histogram (const ImageBuf &src, int channel = 0,
                                                         int bins = 256, float min = 0.0f, float max
                                                         = 1.0f, bool ignore_empty = false, ROI
                                                         roi = {}, int nthreads = 0)

```

Compute a histogram of `src`, for the given channel and `ROI`. Return a vector of length `bins` that contains the counts of how many pixel values were in each of `bins` equally spaced bins covering the range of values `[min,max]`. Values `< min` count for bin 0, values `> max` count for bin `nbins-1`. If `ignore_empty` is `true`, no counts will be incremented for any pixels whose value is 0 in all channels.

Examples:

```

ImageBuf Src ("tahoe.exr");
const int bins = 4;
std::vector<imagesize_t> hist =
    ImageBufAlgo::histogram (Src, 0, bins, 0.0f, 1.0f);
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";

```

## 10.6 Convolutions and frequency-space algorithms

`ImageBuf OIIO::ImageBufAlgo::make_kernel` (*string\_view* name, float width, float height, float depth = 1.0f, bool normalize = true)

Make a 1-channel float image of the named kernel. The size of the image will be big enough to contain the kernel given its size (width x 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 `depth > 1`, a volumetric kernel will be created. Use with caution!

Kernel names can be: “gaussian”, “sharp-gaussian”, “box”, “triangle”, “blackman-harris”, “mitchell”, “b-spline”, “catmull-rom”, “lanczos3”, “disk”, “binomial”, “laplacian”.

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:

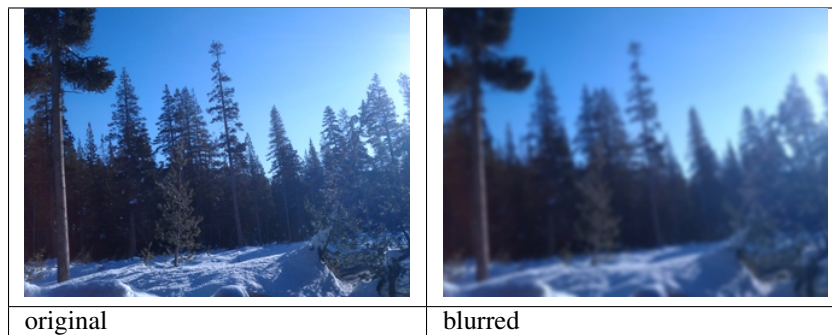
```
ImageBuf K = ImageBufAlgo::make_kernel ("gaussian", 5.0f, 5.0f);
```

`ImageBuf OIIO::ImageBufAlgo::convolve` (const ImageBuf &src, const ImageBuf &kernel, bool normalize = true, *ROI* roi = {}, int nthreads = 0)

Return the convolution of `src` and a kernel. If `roi` is not defined, it defaults to the full size `src`. If `normalized` 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 ("gaussian", 5.0f, 5.0f);
ImageBuf Blurred = ImageBufAlgo::convolve (Src, K);
```



Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::convolve (ImageBuf &dst, const ImageBuf &src,
                                   const ImageBuf &kernel, bool normalize = true,
                                   ROI roi = {}, int nthreads = 0)
```

Write to an existing image `dst` (allocating if it is uninitialized). 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`.

`ImageBuf OIIO::ImageBufAlgo::laplacian (const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

Return the Laplacian of the corresponding region of `src`. The Laplacian is the generalized second derivative of the image

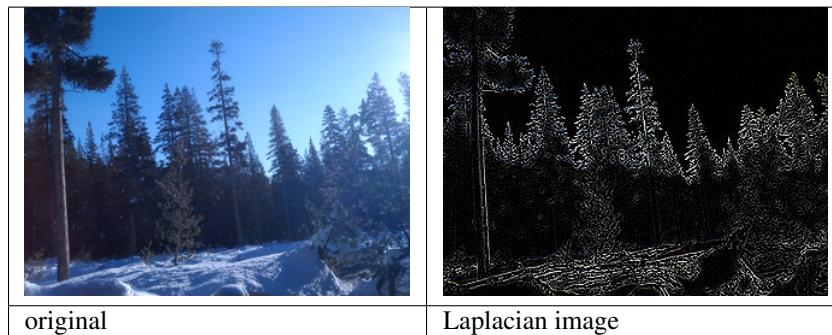
$$\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2}$$

which is approximated by convolving the image with a discrete 3x3 Laplacian kernel,

```
[ 0  1  0 ]
[ 1 -4  1 ]
[ 0  1  0 ]
```

Examples:

```
ImageBuf src ("tahoe.exr");
ImageBuf lap = ImageBufAlgo::laplacian (src);
```



`bool OIIO::ImageBufAlgo::laplacian (ImageBuf &dst, const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

Write to an existing image `dst` (allocating if it is uninitialized).

### group **fft-iff**

Fast Fourier Transform and inverse

Return (or copy into `dst`) the discrete Fourier transform (DFT), or its inverse, of the section of `src` denoted by `roi`. If `roi` is not defined, it will be all of `src`'s pixels.

`fft()` takes the discrete Fourier transform (DFT) of the section of `src` denoted by `roi`, returning it or 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()` x `roi.height`, with channel 0 being the “real” part and channel 1 being the the “imaginary” part. The values returned are actually the unitary DFT, meaning that it is scaled by  $1/\sqrt{\text{npixels}}$ .

`ifft()` takes the inverse discrete Fourier transform, transforming a 2-channel complex (real and imaginary) frequency domain image and into a single-channel spatial domain image. `src` must be a 2-channel float image, and is assumed to be a complex frequency-domain signal with the “real” component in channel 0 and the “imaginary” component in channel 1. `dst` will end up being a float image of one channel (the real component is kept, the imaginary component of the spatial-domain will be discarded). Just as with `fft()`, the `ifft()` function is dealing with the unitary DFT, so it is scaled by  $1/\sqrt{\text{npixels}}$ .

## Functions

ImageBuf **fft** (**const** ImageBuf &*src*, *ROI* *roi* = {}, int *nthreads* = 0)

ImageBuf **ifft** (**const** ImageBuf &*src*, *ROI* *roi* = {}, int *nthreads* = 0)

bool **fft** (ImageBuf &*dst*, **const** ImageBuf &*src*, *ROI* *roi* = {}, int *nthreads* = 0)

bool **ifft** (ImageBuf &*dst*, **const** ImageBuf &*src*, *ROI* *roi* = {}, int *nthreads* = 0)

Examples:

```
ImageBuf Src ("tahoe.exr");

// Take the DFT of the first channel of Src
ImageBuf Freq = ImageBufAlgo::fft (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 (Freq);
```

### group **complex-polar**

Converting complex to polar and back

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.

The transformation between the two representations are:

```
real = amplitude * cos(phase);
imag = amplitude * sin(phase);

amplitude = hypot (real, imag);
phase = atan2 (imag, real);
```

## Functions

ImageBuf **complex\_to\_polar** (const ImageBuf &src, *ROI* roi = {}, int nthreads = 0)

bool **complex\_to\_polar** (ImageBuf &dst, const ImageBuf &src, *ROI* roi = {}, int nthreads = 0)

ImageBuf **polar\_to\_complex** (const ImageBuf &src, *ROI* roi = {}, int nthreads = 0)

bool **polar\_to\_complex** (ImageBuf &dst, const ImageBuf &src, *ROI* roi = {}, int nthreads = 0)

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 (Polar);

// 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 (Complex);
```

## 10.7 Image Enhancement / Restoration

ImageBuf OIIO::ImageBufAlgo::**fixNonFinite** (const ImageBuf &src, NonFiniteFixMode mode = NONFINITE\_BOX3, int \*pixelsFixed = nullptr, *ROI* roi = {}, int nthreads = 0)

*fixNonFinite* () returns in image containing the values of src (within the *ROI*), while repairing any non-finite (NaN/Inf) pixels. If pixelsFixed is not nullptr, store in it the number of pixels that contained non-finite value. It is permissible to operate in-place (with src and dst referring to the same image).

How the non-finite values are repaired is specified by one of the mode parameter, which is an enum of NonFiniteFixMode.

This function 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";
```

**Result-as-parameter version:**

```
bool OIIO::ImageBufAlgo::fixNonFinite (ImageBuf &dst, const ImageBuf
                                         &src, NonFiniteFixMode mode =
                                         NONFINITE_BOX3, int *pixelsFixed =
                                         nullptr, ROI roi = {}, int nthreads = 0)
```

Write to an existing image dst (allocating if it is uninitialized).

`ImageBuf OIIO::ImageBufAlgo::fillholes_pushpull (const ImageBuf &src, ROI roi = {}, int  
nthreads = 0)`

Copy the specified *ROI* of `src` 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 within `roi`, and any place where the alpha of `src` was < 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 (Src);
```

**Result-as-parameter version:**

`bool OIIO::ImageBufAlgo::fillholes_pushpull (ImageBuf &dst, const ImageBuf &src, ROI roi = {}, int  
nthreads = 0)`

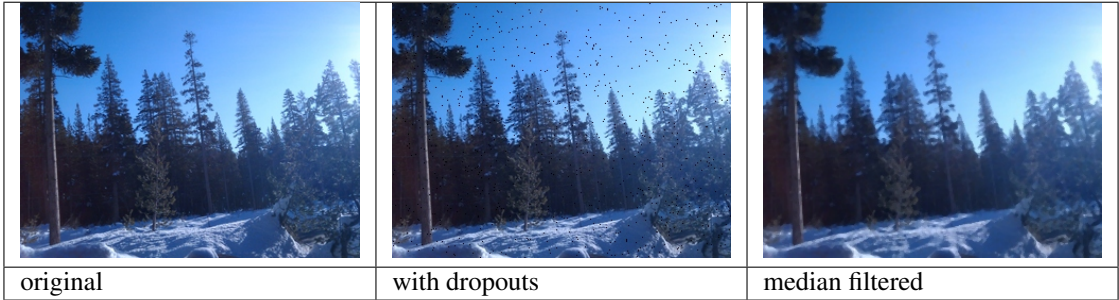
Write to an existing image `dst` (allocating if it is uninitialized).

`bool OIIO::ImageBufAlgo::median_filter (ImageBuf &dst, const ImageBuf &src, int width = 3,  
int height = -1, ROI roi = {}, int nthreads = 0)`

Write to an existing image `dst` (allocating if it is uninitialized).

Examples:

```
ImageBuf Noisy ("tahoe.exr");  
ImageBuf Clean = ImageBufAlgo::median_filter (Noisy, 3, 3);
```



**Result-as-parameter version:**

`bool OIIO::ImageBufAlgo::median_filter (ImageBuf &dst, const ImageBuf  
&src, int width = 3, int height = -1, ROI  
roi = {}, int nthreads = 0)`

Write to an existing image `dst` (allocating if it is uninitialized).



```
ImageBuf OIIO::ImageBufAlgo::unsharp_mask(const ImageBuf &src, string_view kernel =  
    "gaussian", float width = 3.0f, float contrast = 1.0f,  
    float threshold = 0.0f, ROI roi = {}, int nthreads =  
    0)
```

Return 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,

```
dst = src + contrast * thresh(src - blur(src))
```

The specific blur can be selected by kernel name and width (for example, “gaussian” is typical). As a special case, “median” is also accepted as the kernel name, in which case a median filter is performed rather than a blurring convolution (Gaussian and other blurs sometimes over-sharpen edges, whereas using the median filter will sharpen compact high-frequency details while not over-sharpening long edges).

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 (Blurry, "gaussian", 5.0f);
```

**Result-as-parameter version:**

```
bool OIIO::ImageBufAlgo::unsharp_mask(ImageBuf &dst, const ImageBuf  
    &src, string_view kernel = "gaussian",  
    float width = 3.0f, float contrast = 1.0f,  
    float threshold = 0.0f, ROI roi = {}, int  
    nthreads = 0)
```

Write to an exsisting image `dst` (allocating if it is uninitialized).

## 10.8 Morphological filters

```
ImageBuf OIIO::ImageBufAlgo::dilate(const ImageBuf &src, int width = 3, int height = -1, ROI  
    roi = {}, int nthreads = 0)
```

Return a dilated version of the corresponding region of `src`. Dilation is defined as the maximum value of all pixels under nonzero values of the structuring element (which is taken to be a width x height square). If height is not set, it will default to be the same as width. Dilation makes bright features wider and more prominent, dark features thinner, and removes small isolated dark spots.

**Result-as-parameter version:**

```
bool OIIO::ImageBufAlgo::dilate(ImageBuf &dst, const ImageBuf &src, int  
    width = 3, int height = -1, ROI roi = {}, int  
    nthreads = 0)
```

Write to an exsisting image `dst` (allocating if it is uninitialized).



```
ImageBuf OIIO::ImageBufAlgo::erode (const ImageBuf &src, int width = 3, int height = -1, ROI roi
                                     = {}, int nthreads = 0)
```

Return an eroded version of the corresponding region of `src`. Erosion is defined as the minimum value of all pixels under nonzero values of the structuring element (which is taken to be a width x height square). If height is not set, it will default to be the same as width. Erosion makes dark features wider, bright features thinner, and removes small isolated bright spots.

#### Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::erode (ImageBuf &dst, const ImageBuf &src, int width
                                = 3, int height = -1, ROI roi = {}, int nthreads = 0)
```

Write to an existing image `dst` (allocating if it is uninitialized).

Dilation and erosion are basic morphological filters, and more complex ones are often constructed from them:

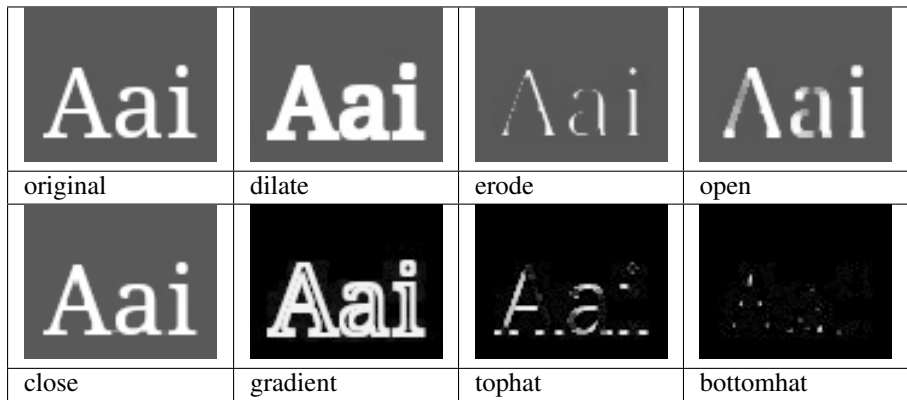
- “open” is erode followed by dilate, and it keeps the overall shape while removing small bright regions;
- “close” is dilate followed by erode, and it keeps the overall shape while removing small dark regions;
- “morphological gradient” is dilate minus erode, which gives a bright perimeter edge;
- “tophat” is the original source minus the “open”, which isolates local peaks;
- “bottomhat” is the “close” minus the original source, which isolates dark holes.

Examples:

```
ImageBuf Source ("source.tif");

ImageBuf Dilated = ImageBufAlgo::dilate (Source, 3, 3);
ImageBuf Eroded  = ImageBufAlgo::erode  (Source, 3, 3);

// Morphological "open" is dilate(erode((source)))
ImageBuf Opened  = ImageBufAlgo::dilate (Eroded, 3, 3);
// Morphological "close" is erode(dilate(source))
ImageBuf Closed  = ImageBufAlgo::erode  (Dilated, 3, 3);
// Morphological "gradient" is dilate minus erode
ImageBuf Gradient = ImageBufAlgo::sub   (Dilated, Eroded);
// Tophat filter is source minus open
ImageBuf Tophat   = ImageBufAlgo::sub   (Source, Opened);
// Bottomhat filter is close minus source
ImageBuf Bottomhat = ImageBufAlgo::sub   (Close, Source);
```



## 10.9 Color space conversion

### *group* **colorconvert**

Convert between color spaces

Return (or copy into *dst*) the pixels of *src* within the *ROI*, applying a color space transformation. In-place operations (*dst* == *src*) are supported.

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.

### Parameters

- *fromspace/tospace*: For the varieties of `colorconvert()` that use named color spaces, these specify the color spaces by name.
- *context\_key/context\_value*: For the varieties of `colorconvert()` that use named color spaces, these optionally specify a “key” name/value pair to establish a context (for example, a shot-specific transform).
- *processor*: For the varieties of `colorconvert()` that have a *processor* parameter, it is a raw `ColorProcessor*` object that implements the color transformation. This is a special object created by a `ColorConfig` (see `OpenImageIO/color.h` for details).
- *unpremult*: If true, divide the RGB channels by alpha (if it exists and is nonzero) before color conversion, then re-multiply by alpha after the color conversion. Passing *unpremult*=false skips this step, which may be desirable if you know that the image is “unassociated alpha” (a.k.a., “not pre-multiplied colors”).
- *colorconfig*: An optional `ColorConfig*` specifying an OpenColorIO configuration. If not supplied, the default OpenColorIO color configuration found by examining the `$OCIO` environment variable will be used instead.

### Functions

```
ImageBuf colorconvert (const ImageBuf &src, string_view fromspace, string_view tospace, bool
    unpremult = true, string_view context_key = "", string_view context_value =
    "", ColorConfig *colorconfig = nullptr, ROI roi = {}, int nthreads = 0)
    Transform between named color spaces, returning an ImageBuf result.
```

```
ImageBuf colorconvert (const ImageBuf &src, const ColorProcessor *processor, bool unpre-
    mult, ROI roi = {}, int nthreads = 0)
    Transform using a ColorProcessor, returning an ImageBuf result.
```

```
bool colorconvert (ImageBuf &dst, const ImageBuf &src, string_view fromspace, string_view
    tospace, bool unpremult = true, string_view context_key = "", string_view con-
    text_value = "", ColorConfig *colorconfig = nullptr, ROI roi = {}, int nthreads =
    0)
    Transform between named color spaces, storing results into an existing ImageBuf.
```

```
bool colorconvert (ImageBuf &dst, const ImageBuf &src, const ColorProcessor *processor,
    bool unpremult, ROI roi = {}, int nthreads = 0)
    Transform using a ColorProcessor, storing results into an existing ImageBuf.
```

bool **colorconvert** (span<float> *color*, const ColorProcessor \**processor*, bool *unpremult*)

Apply a color transform in-place to just one color: *color*[0..*nchannels*-1]. *nchannels* should either be 3 or 4 (if 4, the last channel is alpha).

bool **colorconvert** (float \**color*, int *nchannels*, const ColorProcessor \**processor*, bool *unpremult*)

Examples:

```
#include <OpenImageIO/imagebufalgo.h>
#include <OpenImageIO/color.h>
using namespace OIIO;

ImageBuf Src ("tahoe.jpg");
ColorConfig cc;
ColorProcessor *processor = cc.createColorProcessor ("vd8", "lnf");
ImageBuf dst = ImageBufAlgo::colorconvert (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 (Src, "vd8", "lnf", true);
```

ImageBuf OIIO::ImageBufAlgo::**colormatrixtransform**(const ImageBuf &*src*, const Imath::M44f &*M*, bool *unpremult* = true, *ROI* *roi* = {}, int *nthreads* = 0)

Return a copy of the pixels of *src* within the *ROI*, applying a color transform specified by a 4x4 matrix. In-place operations (*dst* == *src*) are supported.

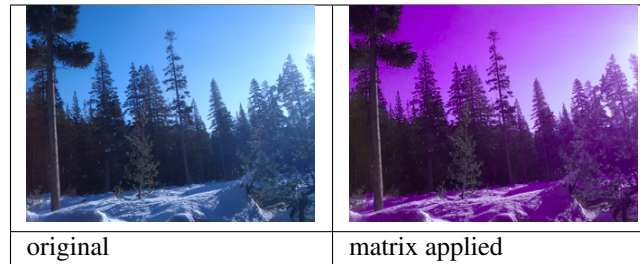
**Version** 2.1+

#### Parameters

- *M*: A 4x4 matrix. Following Imath conventions, the color is a row vector and the matrix has the “translation” part in elements [12..14] (matching the memory layout of OpenGL or RenderMan), so the math is *color* \* *Matrix* (NOT *M*\**c*).
- *unpremult*: If true, divide the RGB channels by alpha (if it exists and is nonzero) before color conversion, then re-multiply by alpha after the after the color conversion. Passing *unpremult*=false skips this step, which may be desirable if you know that the image is “unassociated alpha” (a.k.a., “not pre-multiplied colors”).

Examples:

```
ImageBuf Src ("tahoe.jpg");
Imath::M44f M ( .8047379, .5058794, -.3106172, 0,
               -.3106172, .8047379, .5058794, 0,
               .5058794, -.3106172, .8047379, 0,
               0, 0, 0, 1);
ImageBuf dst = ImageBufAlgo::colormatrixtransform (Src, M);
```

**Result-as-parameter version:**

```
bool OIIO::ImageBufAlgo::colormatrixtransform(ImageBuf &dst, const
                                              ImageBuf &src, const
                                              Imath::M44f &M, bool
                                              unpremult = true, ROI roi =
                                              {}, int nthreads = 0)
```

Write to an exsisting image `dst` (allocating if it is uninitialized).

```
ImageBuf OIIO::ImageBufAlgo::ociolook(const ImageBuf &src, string_view looks, string_view
                                       fromspace, string_view tospace, bool unpremult = true,
                                       bool inverse = false, string_view context_key = "",
                                       string_view context_value = "", ColorConfig *colorcon-
                                       fig = nullptr, ROI roi = {}, int nthreads = 0)
```

Return a copy of the pixels of `src` within the *ROI*, applying an OpenColorIO “look” transform to the pixel values. In-place operations (`dst == src`) are supported.

**Parameters**

- `looks`: The looks to apply (comma-separated).
- `fromspace/tospace`: For the varieties of `colorconvert()` that use named color spaces, these specify the color spaces by name.
- `unpremult`: If `true`, divide the RGB channels by alpha (if it exists and is nonzero) before color conversion, then re-multiply by alpha after the after the color conversion. Passing `unpremult=false` skips this step, which may be desirable if you know that the image is “unassociated alpha” (a.k.a., “not pre-multiplied colors”).
- `inverse`: If `true`, it will reverse the color transformation and look application.
- `context_key/context_value`: Optional key/value to establish a context (for example, a shot-specific transform).
- `colorconfig`: An optional `ColorConfig*` specifying an OpenColorIO configuration. If not supplied, the default OpenColorIO color configuration found by examining the `$OCIO` environment variable will be used instead.

**Examples:**

```
ImageBuf Src ("tahoe.jpg");
ImageBuf Dst = ImageBufAlgo::ociolook (Src, "look", "vd8", "lnf",
                                       true, false, "SHOT", "pe0012");
```

**Result-as-parameter version:**

```
bool OIIO::ImageBufAlgo::ociolook(ImageBuf &dst, const ImageBuf &src,
                                   string_view looks, string_view fromspace,
                                   string_view tospace, bool unpremult = true,
                                   bool inverse = false, string_view context_key
                                   = "", string_view context_value = "", Color-
                                   Config *colorconfig = nullptr, ROI roi = {},
                                   int nthreads = 0)
    Write to an exsisting image dst (allocating if it is uninitialized).
```

```
ImageBuf OIIO::ImageBufAlgo::ociodisplay(const ImageBuf &src, string_view display,
                                           string_view view, string_view fromspace = "",
                                           string_view looks = "", bool unpremult = true,
                                           string_view context_key = "", string_view con-
                                           text_value = "", ColorConfig *colorconfig = nullptr,
                                           ROI roi = {}, int nthreads = 0)
```

Return the pixels of `src` within the `ROI`, applying an OpenColorIO “display” transform to the pixel values. In-place operations (`dst == src`) are supported.

#### Parameters

- `display`: The OCIO “display” to apply. If this is the empty string, the default display will be used.
- `view`: The OCIO “view” to use. If this is the empty string, the default view for this display will be used.
- `fromspace`: If `fromspace` is not supplied, it will assume that the source color space is whatever is indicated by the source image’s metadata or filename, and if that cannot be deduced, it will be assumed to be scene linear.
- `looks`: The looks to apply (comma-separated). This may be empty, in which case no “look” is used. Note: this parameter value is not used when building against OpenColorIO 2.x.
- `unpremult`: If true, divide the RGB channels by alpha (if it exists and is nonzero) before color conversion, then re-multiply by alpha after the after the color conversion. Passing `unpremult=false` skips this step, which may be desirable if you know that the image is “unassociated alpha” (a.k.a., “not pre-multiplied colors”).
- `inverse`: If true, it will reverse the color transformation and display application.
- `context_key/context_value`: Optional key/value to establish a context (for example, a shot-specific transform).
- `colorconfig`: An optional `ColorConfig*` specifying an OpenColorIO configuration. If not supplied, the default OpenColorIO color configuration found by examining the `$OCIO` environment variable will be used instead.

Examples:

```
ImageBuf Src ("tahoe.exr");
ImageBuf Dst = ImageBufAlgo::ociodisplay (Src, "sRGB", "Film", "lnf",
                                           "", true, "SHOT", "pe0012");
```

**Result-as-parameter version:**

```
bool OIIO::ImageBufAlgo::ociodisplay (ImageBuf &dst, const ImageBuf &src,
                                     string_view display, string_view view,
                                     string_view fromspace = "", string_view
                                     looks = "", bool unpremult = true,
                                     string_view context_key = "", string_view
                                     context_value = "", ColorConfig *color-
                                     config = nullptr, ROI roi = {}, int nthreads
                                     = 0)
```

Write to an exsisting image dst (allocating if it is uninitialized).

```
ImageBuf OIIO::ImageBufAlgo::ociofiletransform (const ImageBuf &src, string_view name,
                                                bool unpremult = true, bool inverse = false,
                                                ColorConfig *colorconfig = nullptr, ROI roi
                                                = {}, int nthreads = 0)
```

Return the pixels of src within the ROI, applying an OpenColorIO “file” transform. In-place operations (dst == src) are supported.

#### Parameters

- name: The name of the file containing the transform information.
- unpremult: If true, divide the RGB channels by alpha (if it exists and is nonzero) before color conversion, then re-multiply by alpha after the after the color conversion. Passing unpremult=false skips this step, which may be desirable if you know that the image is “unassociated alpha” (a.k.a., “not pre-multiplied colors”).
- inverse: If true, it will reverse the color transformation.
- colorconfig: An optional ColorConfig\* specifying an OpenColorIO configuration. If not supplied, the default OpenColorIO color configuration found by examining the \$OCIO environment variable will be used instead.

Examples:

```
ImageBuf Src ("tahoe.jpg");
ImageBuf Dst = ImageBufAlgo::ociofiletransform (Src, "footransform.csp");
```

#### Result-as-parameter version:

```
bool OIIO::ImageBufAlgo::ociofiletransform (ImageBuf &dst, const Image-
                                             Buf &src, string_view name,
                                             bool unpremult = true, bool in-
                                             verse = false, ColorConfig *col-
                                             orconfig = nullptr, ROI roi = {},
                                             int nthreads = 0)
```

Write to an exsisting image dst (allocating if it is uninitialized).

*group* **premult**

Premultiply or un-premultiply color by alpha

The `unpremult` operation returns (or copies into `dst`) the pixels of `src` within the *ROI*, and in the process divides 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 `premult` operation returns (or copies into `dst`) the pixels of `src` within the *ROI*, and in the process multiplies 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.”

Both operations are simply a copy if there is no identified alpha channel (and a no-op if `dst` and `src` are the same image).

## Functions

`ImageBuf unpremult (const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

`bool unpremult (ImageBuf &dst, const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

`ImageBuf premult (const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

`bool premult (ImageBuf &dst, const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

Examples:

```
// Convert in-place from associated alpha to unassociated alpha
ImageBuf A ("a.exr");
ImageBufAlgo::unpremult (A, A);

// Convert in-place from unassociated alpha to associated alpha
ImageBufAlgo::premult (A, A);
```

## 10.10 Import / export

### *group* `make_texture`

The `make_texture()` function turns an image into a tiled, MIP-mapped, texture file and write it to disk (outputfilename).

Named fields in config:

- `format` : Data format of the texture file (default: UNKNOWN = same format as the input)
- `tile_width/tile_height/tile_depth` : Preferred tile size (default: 64x64x1)

Metadata in `config.extra_attribs`

- `compression (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:runstats (int)` : If nonzero, print run 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, or was created with different command-line arguments. (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:opaque_detect (int)` : If nonzero, drop the alpha channel if alpha is 1.0 in all pixels (default: 0).
- `maketx:compute_average (int)` : If nonzero, compute and store the average color of the texture (default: 1).
- `maketx:unpremult (int)` : If nonzero, unpremultiply color by alpha before color conversion, then multiply by alpha after color conversion (default: 0).
- `maketx:incolospace, maketx:outcolospace (string)` : These two together will apply a color conversion (with `OpenColorIO`, if compiled). Default: ""
- `maketx:colorconfig (string)` : Specifies a custom `OpenColorIO` color config file. Default: ""
- `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 on HDR images.
- `maketx:sharpen (float)` : If nonzero, sharpens details when creating MIPmap levels. The amount is the contrast matrix. The default is 0, meaning no sharpening.
- `maketx:nchannels (int)` : 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)
- `maketx:channelnames (string)` : If set, overrides the channel names of the output image (comma-separated).
- `maketx:fileformatname (string)` : If set, will specify the output file format. (default: "", meaning infer the format from the output filename)
- `maketx:prman_metadata (int)` : If set, output some metadata that PRMan will need for its textures. (0)
- `maketx:oiio_options (int)` : (Deprecated; all are handled by default)
- `maketx:prman_options (int)` : If nonzero, override a whole bunch of settings as needed to make textures that are compatible with PRMan. (0)



- `maketx:mipimages` (string) : Semicolon-separated list of alternate images to be used for individual MIPmap levels, rather than simply downsizing. (default: “”)
- `maketx:full_command_line` (string) : The command or program used to generate this call, will be embedded in the metadata. (default: “”)
- `maketx:ignore_unassoc` (int) : If nonzero, will disbelieve any evidence that the input image is unassociated alpha. (0)
- `maketx:read_local_MB` (int) : 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)
- `maketx:forcefloat` (int) : Forces a conversion through float data for the sake of ImageBuf math. (1)
- `maketx:hash` (int) : Compute the sha1 hash of the file in parallel. (1)
- `maketx:allow_pixel_shift` (int) : 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)
- `maketx:bumpformat` (string) : For the MakeTxBumpWithSlopes mode, chooses whether to assume the map is a height map (“height”), a normal map (“normal”), or automatically determine it from the number of channels (“auto”, the default).

### Parameters

- `mode`: Describes what type of texture file we are creating and may be one of:
  - `MakeTxTexture` : Ordinary 2D texture.
  - `MakeTxEnvLat1` : Latitude-longitude environment map
  - `MakeTxEnvLat1FromLightProbe` : Latitude-longitude environment map constructed from a “light probe” image.
  - `MakeTxBumpWithSlopes` : Bump/displacement map with extra slope data channels (6 channels total, containing both the height and 1st and 2nd moments of slope distributions) for bump-to-roughness conversion in shaders.
- `outputfilename`: Name of the file in which to save the resulting texture.
- `config`: 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 listed below.
- `ostream`: If not `nullptr`, it should point to a stream (for example, `&std::cout`, or a pointer to a local `std::stringstream` to capture output), which is where console output and error messages will be deposited.

### Functions

bool **make\_texture** (MakeTextureMode *mode*, const ImageBuf &*input*, string\_view *outputfilename*, const ImageSpec &*config*, std::ostream \**ostream* = nullptr)  
Version of `make_texture` that starts with an ImageBuf.

bool **make\_texture** (MakeTextureMode *mode*, string\_view *filename*, string\_view *outputfilename*, const ImageSpec &*config*, std::ostream \**ostream* = nullptr)  
Version of `make_texture` that starts with a filename and reads the input from that file, rather than being given an ImageBuf directly.

```
bool make_texture (MakeTextureMode mode, const std::vector<std::string> &filenames,
                  string_view outputfilename, const ImageSpec &config, std::ostream *out-
                  stream = nullptr)
```

Version of make\_texture that takes multiple filenames (reserved for future expansion, such as assembling several faces into a cube map).

Examples:

```
// This command line:
//   maketx in.exr --hcomp --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:opaque_detect", 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";
```

OpenCV interoperability is performed by the from\_OpenCV() and to\_OpenCV() functions:

```
ImageBuf OIIO::ImageBufAlgo::from_OpenCV (const cv::Mat &mat, TypeDesc convert = TypeUn-
                                         known, ROI roi = {}, int nthreads = 0)
```

Convert an OpenCV cv::Mat into an ImageBuf, copying the pixels (optionally converting to the pixel data type specified by convert, if not UNKNOWN, which means to preserve the original data type if possible). Return true if ok, false if it couldn't figure out how to make the conversion from Mat to ImageBuf. If OpenImageIO was compiled without OpenCV support, this function will return an empty image with error message set.

```
bool OIIO::ImageBufAlgo::to_OpenCV (cv::Mat &dst, const ImageBuf &src, ROI roi = {}, int
                                     nthreads = 0)
```

Construct an OpenCV cv::Mat containing the contents of ImageBuf src, and return true. If it is not possible, or if OpenImageIO was compiled without OpenCV support, then return false. Note that OpenCV only supports up to 4 channels, so >4 channel images will be truncated in the conversion.

```
ImageBuf OIIO::ImageBufAlgo::capture_image (int cameranum = 0, TypeDesc convert = TypeUn-
                                             known)
```

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 (0, TypeDesc::UINT8);
WebcamImage.write ("webcam.jpg");
```

## 10.11 Deep images

A number of ImageBufAlgo functions are designed to work with “deep” images. These are detailed below. In general, ImageBufAlgo functions not listed in this section should not be expected to work with deep images.

### 10.11.1 Functions specific to deep images

ImageBuf OIIO::ImageBufAlgo::deepen (const ImageBuf &src, float zvalue = 1.0f, ROI roi = {}, int nthreads = 0)

Return the “deep” equivalent of the “flat” input src. Turning a flat image into a deep one means:

- If the src image has a “Z” channel: if the source pixel’s Z channel value is not infinite, the corresponding pixel of dst will get a single depth sample that copies the data from the source pixel; otherwise, dst will get an empty pixel. In other words, infinitely far pixels will not turn into deep samples.
- If the src image lacks a “Z” channel: if any of the source pixel’s channel values are nonzero, the corresponding pixel of dst will get a single depth sample that copies the data from the source pixel and uses the zvalue parameter for the depth; otherwise, if all source channels in that pixel are zero, the destination pixel will get no depth samples.

If src is already a deep image, it will just copy pixel values from src.

Examples:

```
ImageBuf Flat ("RGBAZ.exr");
ImageBuf Deep = ImageBufAlgo::deepen (Flat);
```

**Result-as-parameter version:**

bool OIIO::ImageBufAlgo::deepen (ImageBuf &dst, const ImageBuf &src, float zvalue = 1.0f, ROI roi = {}, int nthreads = 0)  
Write to an existing image dst (allocating if it is uninitialized).

ImageBuf OIIO::ImageBufAlgo::flatten (const ImageBuf &src, ROI roi = {}, int nthreads = 0)

Return the “flattened” composite of deep image src. That is, it converts a deep image to a simple flat image by front-to-back compositing the samples within each pixel. If src is already a non-deep/flat image, it will just copy pixel values from src to dst. If dst is not already an initialized ImageBuf, it will be sized to match src (but made non-deep).

Examples:

```
ImageBuf Deep ("deepalpha.exr");
ImageBuf Flat = ImageBufAlgo::flatten (Deep);
```

**Result-as-parameter version:**

bool OIIO::ImageBufAlgo::flatten (ImageBuf &dst, const ImageBuf &src, ROI roi = {}, int nthreads = 0)  
Write to an existing image dst (allocating if it is uninitialized).

`ImageBuf OIIO::ImageBufAlgo::deep_merge(const ImageBuf &A, const ImageBuf &B, bool  
occlusion_cull = true, ROI roi = {}, int nthreads = 0)`

Return the deep merge of the samples of deep images A and B, overwriting any existing samples of `dst` in the *ROI*. If `occlusion_cull` is true, any samples occluded by an opaque sample will be deleted.

Examples:

```
ImageBuf DeepA ("hardsurf.exr");  
ImageBuf DeepB ("volume.exr");  
ImageBuf Merged = ImageBufAlgo::deep_merge (DeepA, DeepB);
```

**Result-as-parameter version:**

```
bool OIIO::ImageBufAlgo::deep_merge(ImageBuf &dst, const ImageBuf &A,  
const ImageBuf &B, bool occlusion_cull  
= true, ROI roi = {}, int nthreads = 0)
```

Write to an existing image `dst` (allocating if it is uninitialized).

`ImageBuf OIIO::ImageBufAlgo::deep_holdout(const ImageBuf &src, const ImageBuf &hold-  
out, ROI roi = {}, int nthreads = 0)`

Return the samples of deep image `src` that are closer than the opaque frontier of deep image holdout, returning true upon success and false for any failures. Samples of `src` that are farther than the first opaque sample of holdout (for the corresponding pixel) will not be copied to `dst`. Image holdout is only used as the depth threshold; no sample values from holdout are themselves copied to `dst`.

Examples:

```
ImageBuf Src ("image.exr");  
ImageBuf Holdout ("holdout.exr");  
ImageBuf Merged = ImageBufAlgo::deep_holdout (Src, Holdout);
```

**Result-as-parameter version:**

```
bool OIIO::ImageBufAlgo::deep_holdout(ImageBuf &dst, const ImageBuf  
&src, const ImageBuf &holdout, ROI  
roi = {}, int nthreads = 0)
```

Write to an existing image `dst` (allocating if it is uninitialized).

### 10.11.2 General functions that also work for deep images

`ImageBuf channels (const ImageBuf &src, int nchannels, cspan<int> channelorder, cspan<float> channelvalues = NULL, cspan<std::string> newchannelnames = {}, bool shuffle_channel_names = false)`

`bool channels (ImageBuf &dst, const ImageBuf &src, int nchannels, cspan<int> channelorder, cspan<float> channelvalues = NULL, cspan<std::string> newchannelnames = {}, bool shuffle_channel_names = false)`

Reorder, rename, remove, or add channels to a deep image.

`bool compare (const ImageBuf &A, const ImageBuf &B, float failthresh, float warnthresh, CompareResults &result, ROI roi = {}, int nthreads = 0)`  
Numerically compare two images.

`bool computePixelStats (PixelStats &stats, const ImageBuf &src, ROI roi = {}, int nthreads = 0)`  
Compute per-channel statistics about the image.

`ImageBuf crop (const ImageBuf &src, ROI roi = {}, int nthreads = 0)`

`bool crop (ImageBuf &dst, const ImageBuf &src, ROI roi = {}, int nthreads = 0)`  
Crop the specified region of `src`, discarding samples outside the ROI.

`ROI nonzero_region (const ImageBuf &src, ROI roi = {}, int nthreads = 0)`  
For “deep” images, this function returns the smallest ROI that contains all pixels that contain depth samples, and excludes the border pixels that contain no depth samples at all.

`ImageBuf add (const ImageBuf &A, cspan<float> B, ROI roi = {}, int nthreads = 0)`

`bool add (ImageBuf &dst, const ImageBuf &A, cspan<float> B, ROI roi = {}, int nthreads = 0)`

`ImageBuf sub (const ImageBuf &A, cspan<float> B, ROI roi = {}, int nthreads = 0)`

`bool sub (ImageBuf &dst, const ImageBuf &A, cspan<float> B, ROI roi = {}, int nthreads = 0)`

`ImageBuf mul (const ImageBuf &A, cspan<float> B, ROI roi = {}, int nthreads = 0)`

`bool mul (ImageBuf &dst, const ImageBuf &A, cspan<float> B, ROI roi = {}, int nthreads = 0)`

`ImageBuf div (const ImageBuf &A, cspan<float> B, ROI roi = {}, int nthreads = 0)`

`bool div (ImageBuf &dst, const ImageBuf &A, cspan<float> B, ROI roi = {}, int nthreads = 0)`  
Add, subtract, multiply, or divide all the samples in a deep image A by per-channel values B [ ].

`ImageBuf fixNonFinite (const ImageBuf &src, NonFiniteFixMode mode = NONFINITE_BOX3, int *pixelsFixed = nullptr, ROI roi = {}, int nthreads = 0)`

`bool fixNonFinite (ImageBuf &dst, const ImageBuf &src, NonFiniteFixMode mode = NONFINITE_BOX3, int *pixelsFixed = nullptr, ROI roi = {}, int nthreads = 0)`  
Repair nonfinite (NaN or Inf) values, setting them to 0.0.

`ImageBuf resample (const ImageBuf &src, bool interpolate = true, ROI roi = {}, int nthreads = 0)`

`bool resample (ImageBuf &dst, const ImageBuf &src, bool interpolate = true, ROI roi = {}, int nthreads = 0)`

Compute 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), for each pixel merely copying the closest deep pixel of the source image (no true interpolation is done for deep images).



## 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 oio
from OIIO import ImageInput, ImageOutput
from OIIO import ImageBuf, ImageSpec, ImageBufAlgo
```

### 11.2 TypeDesc

The `TypeDesc` class that describes data types of pixels and metadata, described in detail in Section *Data Type Descriptions: TypeDesc*, is replicated for Python.

#### **class 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.

#### **class AGGREGATE**

The `AGGREGATE` enum corresponds to the C++ `TypeDesc::AGGREGATE` and contains the following values:

```
SCALAR VEC2 VEC3 VEC4 MATRIX33 MATRIX44
```

These names are also exported to the OpenImageIO namespace.

#### **class VECSEMANTICS**

The `VECSEMANTICS` enum corresponds to the C++ `TypeDesc::VECSEMANTICS` and contains the following values:

```
NOSEMANTICS COLOR POINT VECTOR NORMAL TIMECODE KEYCODE RATIONAL
```

These names are also exported to the OpenImageIO namespace.

```
TypeUnknown TypeString TypeFloat TypeHalf
TypeInt TypeUInt TypeInt16 TypeUInt16
TypeColor TypePoint TypeVector TypeNormal
TypeFloat2 TypeVector2 TypeFloat4 TypeVector2i
TypeMatrix TypeMatrix33
TypeTimeCode TypeKeyCode TypeRational TypePointer
```

Pre-constructed TypeDesc objects for some common types, available in the outer OpenImageIO scope.

Example:

```
t = TypeFloat
```

**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())
```

(continues on next page)



(continued from previous page)

```
> size = 24
> elementtype = point
> elementsize = 12
```

**typedesc == typedesc****typedesc != typedesc****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 *Rectangular region of interest: ROI*, is replicated for Python.

**ROI()****ROI** (*xbegin, xend, ybegin, yend, zbegin=0, zend=1, chbegin=0, chend=1000*)

Construct an ROI with the given bounds. The constructor with no arguments makes an ROI that is “undefined.”

Example:

```
roi = ROI (0, 640, 0, 480, 0, 1, 0, 4) # video res RGBA
```

**ROI.xbegin****ROI.xend****ROI.ybegin****ROI.yend****ROI.zbegin****ROI.zend****ROI.chbegin****ROI.chend**

The basic fields of the ROI (all of type `int`).

**ROI.All**

A pre-constructed undefined ROI understood to mean unlimited ROI on an image.

**ROI.defined**

True if the ROI is defined, False if the ROI is undefined.

**ROI.width**

**ROI.height**

**ROI.depth**

**ROI.nchannels**

The number of pixels in each dimension, and the number of channels, as described by the ROI. (All of type `int`.)

**int ROI.npixels**

The total number of pixels in the region described by the ROI (as an `int`).

**ROI.contains** (*x, y, z=0, ch=0*)

Returns `True` if the ROI contains the coordinate.

**ROI.contains** (*other*)

Returns `True` if the ROI *other* is entirely contained within this ROI.

**ROI.get\_roi** (*imagespec*)

**ROI.get\_roi\_full** (*imagespec*)

Returns an 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 *Image Specification: ImageSpec*, is replicated for Python.

**ImageSpec** ()

**ImageSpec** (*typedesc*)

**ImageSpec** (*xres, yres, nchannels, typedesc*)

**ImageSpec** (*roi, typedesc*)

Constructors of an `ImageSpec`. These correspond directly to the constructors in the C++ bindings.

Example:

```
import OpenImageIO as oiio
...
# default ctr
```

(continues on next page)

(continued from previous page)

```

s = ImageSpec()

# construct with known pixel type, unknown resolution
s = ImageSpec(oioioi.UINT8)

# construct with known resolution, channels, pixel data type
s = ImageSpec(640, 480, 4, "half")

# construct from an ROI
s = ImageSpec (ROI(0,640,0,480,0,1,0,3), TypeFloat)

```

**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.

**ImageSpec.format**

A TypeDesc describing the pixel data.

**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 `TypeDesc` objects.

Example:

```

if spec.channelformats == None:
    print ("All color channels are", str(spec.format))
else:
    print ("Channel formats: ")
    for t in spec.channelformats:
        print ("\t", t)

```

**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**

True if the image is a *deep* (multiple samples per pixel) image, of False if it is an ordinary image.

**ImageSpec.extra\_attribs**

Direct access to the `extra_attribs` named metadata, appropriate for iterating over the entire list rather than searching for a particular named value.

- `len(extra_attribs)` : Returns the number of extra attributes.
- `extra_attribs[i].name` : The name of the indexed attribute.
- `extra_attribs[i].type` : The type of the indexed attribute, as a `TypeDesc`.
- `extra_attribs[i].value` : The value of the indexed attribute.

Example:

```
s = ImageSpec(...)
...
print ("extra_attribs size is", len(s.extra_attribs))
for i in range(len(s.extra_attribs)) :
    print (i, s.extra_attribs[i].name, str(s.extra_attribs[i].type), " :")
    print ("\t", s.extra_attribs[i].value)
print
```

**ImageSpec.roi**

The ROI describing the pixel data window.

**ImageSpec.roi\_full**

The ROI describing the “display window” (or “full size”).

**ImageSpec.set\_format (typedesc)**

Given a `TypeDesc`, sets the `format` field and clear any per-channel formats in `channelformats`.

Example:

```
s = ImageSpec ()
s.set_format (TypeDesc("uint8"))
```

**ImageSpec.default\_channel\_names ()**

Sets `channel_names` to the default names given the value of the `nchannels` field.

**ImageSpec.channelindex (name)**

Return (as an `int`) the index of the channel with the given name, or -1 if it does not exist.

**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 *Image Specification: ImageSpec* for details.)

**ImageSpec.pixel\_bytes ()****ImageSpec.pixel\_bytes (native=False)****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 *Image Specification: ImageSpec* 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 *Image Specification: ImageSpec* 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 *Image Specification: ImageSpec* for details.)

`ImageSpec.erase_attribute(name, searchtype=TypeUnknown, casesensitive=False)`

Remove any specified attributes matching the regular expression name from the list of extra\_attrs.

`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.getattribute(name)`

`ImageSpec.getattribute(name, typedesc)`

Retrieves a named metadata value from extra\_attrs. The generic getattribute() 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.getattribute("foo")    # None if not found
foo = s.getattribute("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[name]

*NEW in 2.1*

Retrieve or set metadata using a dictionary-like syntax, rather than attribute() and getattribute(). This is best illustrated by example:

```
comp = spec["Compression"]
# Same as: comp = spec.getattribute("Compression")

spec["Compression"] = comp
# Same as: spec.attribute("Compression", comp)
```

`ImageSpec.metadata_val (paramval, human=False)`

For a ParamValue, format its value as a string.

`ImageSpec.serialize (format='text', verbose='Detailed')`

Return a string containing the serialization of the ImageSpec. The format may be either “text” or “XML”. The verbose may be one of “brief”, “detailed”, or “detailedhuman”.

`ImageSpec.to_xml ()`

Equivalent to `serialize ("xml", "detailedhuman")`.

`ImageSpec.from_xml (xml)`

Initializes the ImageSpec from the information in the string xml containing an XML-serialized ImageSpec.

`ImageSpec.channel_name (chan)`

Returns a string containing the name of the channel with index chan.

`ImageSpec.channelindex (name)`

Return the integer index of the channel with the given name, or -1 if the name is not a name of one of the channels.

`ImageSpec.channelformat (chan)`

Returns a TypeDesc of the channel with index chan.

`ImageSpec.get_channelformats ()`

Returns a tuple containing all the channel formats.

`ImageSpec.valid_tile_range (xbegin, xend, ybegin, yend, zbegin, zend)`

Returns True if the given tile range exactly covers a set of tiles, or False if it isn't (or if the image is not tiled).

`ImageSpec.copy_dimensions (other)`

Copies from ImageSpec other only the fields describing the size and data types, but not the arbitrary named metadata or channel names.

`ImageSpec.undefined ()`

Returns True for a newly initialized (undefined) ImageSpec.

### 11.4.1 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 oioo

# 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" %
              (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 :
```

(continues on next page)

(continued from previous page)

```

        print ("  full res   %dx%d%+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%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 len(spec.channelformats) > 0 :
    print "  channelformats = ", spec.channelformats
print "  alpha channel = ", spec.alpha_channel
print "  z channel = ", spec.z_channel
print "  deep = ", spec.deep
for i in spec.extra_attribs :
    if type(i.value) == str :
        print "  ", i.name, "= \"\" + i.value + "\""
    else :
        print "  ", i.name, "=", 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 DeepData

The DeepData class describing “deep” image data (multiple depth sample per pixel), which is explained in detail in Section [Reading “deep” data](#), is replicated for Python.

### DeepData()

Constructs a DeepData object. It needs to have its `init()` and `alloc()` methods called before it can hold any meaningful data.

### DeepData.**init**(*npixels, nchannels, channeltypes, channelnames*)

Initializes this DeepData to hold *npixels* total pixels, with *nchannels* color channels. The data types of the channels are described by *channeltypes*, a tuple of `TypeDesc` values (one per channel), and the names are provided in a tuple of `string`s` *channelnames*. After calling `init`, you still need to set the number of samples for each pixel (using `set_nsamples`) and then call `alloc()` to actually allocate the sample memory.

### DeepData.**initialized**()

Returns True if the DeepData is initialized at all.

### DeepData.**allocated**()

Returns True if the DeepData has already had pixel memory allocated.

### DeepData.**pixels**

This `int` field contains the total number of pixels in this collection of deep data.

### DeepData.**channels**

This `int` field contains the number of channels.

### DeepData.**A\_channel**

### DeepData.**AR\_channel**

### DeepData.**AG\_channel**

### DeepData.**AB\_channel**

### DeepData.**Z\_channel**

### DeepData.**Zback\_channel**

The channel index of certain named channels, or -1 if they don’t exist. For `AR_channel`, `AG_channel`, `AB_channel`, if they don’t exist, they will contain the value of `A_channel`, and `Zback_channel` will contain the value of `z_channel` if there is no actual Zback.

### DeepData.**channelname**(*c*)

Retrieve the name of channel *C*, as a `string`.

### DeepData.**channeltype**(*c*)

Retrieve the data type of channel *C*, as a `TypeDesc`.

### DeepData.**channelsize**(*c*)

Retrieve the size (in bytes) of one datum of channel *C*.

### DeepData.**samplesize**()

Retrieve the packed size (in bytes) of all channels of one sample.

### DeepData.**set\_samples**(*pixel, nsamples*)

Set the number of samples for a given pixel (specified by integer index).

### DeepData.**samples**(*pixel*)

Get the number of samples for a given pixel (specified by integer index).

### DeepData.**insert\_samples**(*pixel, samplepos, n*)

Insert *n* samples starting at the given position of an indexed pixel.

### DeepData.**erase\_samples**(*pixel, samplepos, n*)

Erase *n* samples starting at the given position of an indexed pixel.



`DeepData.set_deep_value` (*pixel, channel, sample, value*)  
Set specific float value of a given pixel, channel, and sample index.

`DeepData.set_deep_value_uint` (*pixel, channel, sample, value*)  
Set specific unsigned int value of a given pixel, channel, and sample index.

`DeepData.deep_value` (*pixel, channel, sample, value*)  
Retrieve the specific value of a given pixel, channel, and sample index (for float channels).

`DeepData.deep_value_uint` (*pixel, channel, sample*)  
Retrieve the specific value of a given pixel, channel, and sample index (for uint channels).

`DeepData.copy_deep_sample` (*pixel, sample, src, srcpixel, srcsample*)  
Copy a deep sample from `DeepData src` into this `DeepData`.

`DeepData.copy_deep_pixel` (*pixel, src, srcpixel*)  
Copy a deep pixel from `DeepData src` into this `DeepData`.

`DeepData.split` (*pixel, depth*)  
Split any samples of the pixel that cross depth. Return `True` if any splits occurred, `False` if the pixel was unmodified.

`DeepData.sort` (*pixel*)  
Sort the samples of the pixel by their Z depth.

`DeepData.merge_overlaps` (*pixel*)  
Merge any adjacent samples in the pixel that exactly overlap in z range. This is only useful if the pixel has previously been split at all sample starts and ends, and sorted by depth.

`DeepData.merge_deep_pixels` (*pixel, src, srcpixel*)  
Merge the samples of `src`'s pixel into this `DeepData`'s pixel.

`DeepData.occlusion_cull` (*pixel*)  
Eliminate any samples beyond an opaque sample.

`DeepData.opaque_z` (*pixel*)  
For the given pixel index. return the z value at which the pixel reaches full opacity.

## 11.6 ImageInput

See Chapter *ImageInput: Reading Images* 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* [, *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
```

**ImageInput.close()**

Closes an open image file, returning `True` if successful, `False` otherwise.

Example:

```
input = ImageInput.open (filename)
...
input.close ()
```

**ImageInput.format\_name()**

Returns the format name of the open file, as a string.

Example:

```
input = ImageInput.open (filename)
if input :
    print filename, "was a", input.format_name(), "file."
input.close ()
```

**ImageInput.spec()**

Returns an `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
```

**ImageInput.spec(subimage, miplevel=0)**

Returns a full copy of the `ImageSpec` corresponding to the designated subimage and MIP level.

**ImageSpec ImageInput.spec\_dimensions(subimage, miplevel=0)**

Returns a partial copy of the `ImageSpec` corresponding to the designated subimage and MIP level, only copying the dimension fields and not any of the arbitrary named metadata (and is thus much less expensive).

**ImageInput.current\_subimage()**

Returns the current subimage of the file.

**ImageInput.current\_miplevel()**

Returns the current MIP level of the file.

**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
```

**ImageInput.read\_image(format='float')**

**ImageInput.read\_image(chbegin, chend, format='float')**

`ImageInput.read_image (subimage, miplevel, chbegin, chend, format='float')`

Read the entire image and return the pixels as a NumPy array of values of the given format (described by a `TypeDesc` or a string, float by default). If the format is unknown, the pixels will be returned in the native format of the file. If an error occurs, `None` will be returned.

For a normal (2D) image, the array returned will be 3D indexed as `[y][x][channel]`. For 3D volumetric images, the array returned will be 4D with shape indexed as `[z][y][x][channel]`.

Example:

```
input = ImageInput.open (filename)
spec = input.spec ()
pixels = input.read_image ()
print "The first pixel is", pixels[0][0]
print "The second pixel is", pixels[0][1]
input.close ()
```

`ImageInput.read_scanline (y, z, format='float')`

Read scanline number `y` from depth plane `z` from the open file, returning the pixels as a NumPy array of values of the given type (described by a `TypeDesc` or a string, float by default). If the type is `TypeUnknown`, the pixels will be returned in the native format of the file. If an error occurs, `None` will be returned.

The pixel array returned will be a 2D ndarray, indexed as `[x][channel]`.

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, "float")
        # process the scanline
else :
    print "It's a tiled file"
input.close ()
```

`ImageInput.read_tile (x, y, z, format='float')`

Read the tile whose upper left corner is pixel `(x,y,z)` from the open file, returning the pixels as a NumPy array of values of the given type (described by a `TypeDesc` or a string, float by default). If the type is `TypeUnknown`, the pixels will be returned in the native format of the file. If an error occurs, `None` will be returned.

For a normal (2D) image, the array of tile pixels returned will be a 3D ndarray indexed as `[y][x][channel]`. For 3D volumetric images, the array returned will be 4D with shape indexed as `[z][y][x][channel]`.

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, oio.FLOAT)
                # process the tile
else :
    print "It's a scanline file"
input.close ()
```

```
ImageInput.read_scanlines (subimage, miplevel, ybegin, yend, z, chbegin, chend, format='float')
ImageInput.read_scanlines (ybegin, yend, z, chbegin, chend, format='float')
ImageInput.read_tiles (xbegin, xend, ybegin, yend, zbegin, zend, chbegin, chend, format='float')
ImageInput.read_tiles (subimage, miplevel, xbegin, xend, ybegin, yend, zbegin, zend, format='float')
```

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.

For normal 2D images, both `read_scanlines` and `read_tiles` will return a 3D array indexed as `[z][y][x][channel]`.

For 3D volumetric images, both `read_scanlines` will return a 3D array indexed as `[y][x][channel]`, and `read_tiles` will return a 4D array indexed as `[z][y][x][channel]`,

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)
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)
```

```
ImageInput.read_native_deep_scanlines (subimage, miplevel, ybegin, yend, z, chbegin, chend)
ImageInput.read_native_deep_tiles (subimage, miplevel, xbegin, xend, ybegin, yend, zbegin, zend,
                                   chbegin, chend)
ImageInput.read_native_deep_image (subimage=0, miplevel=0)
```

Read a collection of scanlines, tiles, or an entire image of “deep” pixel data from the specified subimage and MIP level. The begin/end coordinates are all integer values. The value returned will be a `DeepData` if the read succeeds, or `None` if the read fails.

These methods are guaranteed to be thread-safe against simultaneous calls to any of the other other `read_native` calls that take an explicit subimage/miplevel.

```
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 ()
```

### 11.6.1 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()
    if not pixels :
        print "Could not read:", input.geterror()
        return
    input.close()      # we're done with the file at this point
    minval = pixels[0][0]  # initialize to the first pixel value
    maxval = pixels[0][0]
    for y in range(spec.height) :
        for x in range(spec.width) :
            p = pixels[y][x]
            for c in range(nchans) :
                if p[c] < minval[c] :
                    minval[c] = p[c]
                if p[c] > maxval[c] :
                    maxval[c] = p[c]
    print "Min values per channel were", minval
    print "Max values per channel were", maxval
```

## 11.7 ImageOutput

See Chapter *ImageOutput: Writing Images* for detailed explanations of the C++ ImageOutput class APIs. The Python APIs are very similar.

`ImageOutput.create (name, 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()
```

`ImageOutput.format_name()`

The file format name of a created `ImageOutput`, as a string.

Example:

```
output = ImageOutput.create (filename)
if output :
    print "Created output", filename, "as a", output.format_name()
```

`ImageOutput.supports (feature)`

For a created `ImageOutput`, returns `True` if the file format supports the named feature (such as “tiles”, “mipmap”, etc., see Section *ImageOutput Class Reference* 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")
```

`ImageOutput.open (filename, spec, mode='Create')`

Opens the named output file, with an `ImageSpec` describing the image to be output. The mode may be one of “Create”, “AppendSubimage”, or “AppendMIPLevel”. See Section *ImageOutput Class Reference* for details. Returns `True` upon success, `False` upon failure (error messages retrieved via `ImageOutput.geterror()`).

**Returns** `True` for success, `False` for failure.

Example:

```
output = ImageOutput.create (filename)
if not output :
    print "Error:", oiio.geterror()
spec = ImageSpec (640, 480, 3, "uint8")
ok = output.open (filename, spec)
if not ok :
    print "Could not open", filename, ":", output.geterror()
```

`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.

**Returns** `True` for success, `False` for failure.

`ImageOutput.close()`

Closes an open output.

**Returns** `True` for success, `False` for failure.

`ImageOutput.spec()`

Returns the `ImageSpec` of the currently-open output image.

`ImageOutput.write_image (pixels)`

Write the currently opened image all at once. The `pixels` parameter should be a Numpy `ndarray` containing data elements indexed as `[y][x][channel]` for normal 2D images, or for 3D volumetric images, as

[z][y][x][channel], in other words, exactly matching the shape of array returned by `ImageInput.read_image()`. (It will also work fine if the array is 1D “flattened” version, as long as it contains the correct total number of values.) The data type is deduced from the contents of the array itself. 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 ()
input.close ()

output = ImageOutput.create (filename)
if output.supports("tiles") :
    spec.tile_width = 64
    spec.tile_height = 64
    output.open (filename, spec)
    output.write_image (pixels)
    output.close ()
```

`ImageOutput.write_scanline` (*y*, *z*, *pixels*)

`ImageOutput.write_scanlines` (*ybegin*, *yend*, *z*, *pixels*)

Write one or many scanlines to the currently open file. Returns `True` upon success, `False` upon failure.

The `pixels` parameter should be a Numpy ndarray containing data elements indexed as [*x*][*channel*] for `write_scanline` or as [*y*][*x*][*channels*] for `write_scanlines`, exactly matching the shape returned by `ImageInput.read_scanline` or `ImageInput.read_scanlines`. (It will also work fine if the array is 1D “flattened” version, as long as it contains the correct total number of values.)

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)
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)
        output.write_scanline (y, z, 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)
    output.write_scanlines (spec.y, spec.y+spec.height, z, pixels)
...
```

`ImageOutput.write_tile` (*x*, *y*, *z*, *pixels*)

`ImageOutput.write_tiles` (*xbegin*, *xend*, *ybegin*, *yend*, *zbegin*, *zend*, *pixels*)

Write one or many tiles to the currently open file. Returns `True` upon success, `False` upon failure.

The `pixels` parameter should be a Numpy ndarray containing data elements indexed as [*y*][*x*][*channel*] for normal 2D images, or as [*z*][*y*][*x*][*channels*] for 3D volumetric images, exactly

matching the shape returned by `ImageInput.read_tile` or `ImageInput.read_tiles`. (It will also work fine if the array is 1D “flattened” version, as long as it contains the correct total number of values.)

Example:

```
input = ImageInput.open (in_filename)
spec = input.spec ()
output = ImageOutput.create (out_filename)
output.open (out_filename, spec)
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)
            output.write_tile (x, y, z, 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)
        output.write_tiles (spec.x, spec.x+spec.width,
                             y, y+tile_width, z, z+tile_width, pixels)
...
```

`ImageOutput.write_deep_scanlinesa` (*ybegin, yend, z, deepdata*)

`ImageOutput.write_deep_tiles` (*xbegin, xend, ybegin, yend, zbegin, zend, deepdata*)

`ImageOutput.write_deep_image` (*deepdata*)

Write a collection of scanlines, tiles, or an entire image of “deep” pixel data. The begin/end coordinates are all integer values, and *deepdata* should be a `DeepData`.

`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)
output.copy_image (input)
output.close ()
input.close ()
```

`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(),
```

(continues on next page)



(continued from previous page)

```

    # since there is no ImageOutput object!
else :
    ok = output.open (filename, spec)
    if not ok :
        print "Open error:", output.geterror()
    ok = output.write_image (pixels)
    if not ok :
        print "Write error:", output.geterror()
    output.close ()

```

## 11.8 ImageBuf

See Chapter *ImageBuf: Image Buffers* for detailed explanations of the C++ ImageBuf class APIs. The Python APIs are very similar.

### ImageBuf ()

Construct a new, empty ImageBuf. The ImageBuf is uninitialized and is awaiting a call to `reset ()` or `copy ()` before it is useful.

### 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 (filename, subimage, miplevel, config)

Construct a read-only ImageBuf that will read from the named file, with an ImageSpec `config` giving configuration hints.

Example:

```

import OpenImageIO as oiio
...
config = ImageSpec()
config.attribute("oiio:RawColor", 1)
buf = ImageBuf ("grid.tif", 0, 0, config)

```

### ImageBuf (imagespec, zero=True)

Construct a writeable ImageBuf of the dimensions and data format specified by an ImageSpec. The pixels will be initialized to black/empty values if `zero` is True, otherwise the pixel values will remain uninitialized.

Example:

```

spec = ImageSpec (640, 480, 3, "float")
buf = ImageBuf (spec)

```

### ImageBuf.clear ()

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

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*, *subimage*=0, *miplevel*=0, *config*=*ImageSpec*())

Restore the ImageBuf to a newly-constructed state, to read from a filename (optionally specifying a subimage, MIP level, and/or a “configuration” ImageSpec).

**ImageBuf.reset** (*imagespec*, *zero*=*True*)

Restore the ImageBuf to the newly-constructed state of a writeable ImageBuf specified by an ImageSpec. The pixels will be initialized to black/empty if *zero* is *True*, otherwise the pixel values will remain uninitialized.

**ImageBuf.read** (*subimage*=0, *miplevel*=0, *force*=*False*, *convert*=*oiio.UNKNOWN*)

**ImageBuf.read** (*subimage*, *miplevel*, *chbegin*, *chend*, *force*, *convert*)

Explicitly read the image from the file (of a file-reading ImageBuf), optionally specifying a particular subimage, MIP level, and channel range. 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. An explicit *read()* is generally only needed to change the subimage or *miplevel*, or to force an in-buffer read or format conversion.

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
```

**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.

**ImageBuf.write** (*filename*, *dtype*="", *fileformat*="")

Write the contents of the ImageBuf to the named file. Optionally, *dtype* can override the pixel data type (by default, the pixel data type of the buffer), and *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")  
  
# Convert to uint16 TIFF  
buf = ImageBuf ("in.exr")  
buf.write ("out.tif", "uint16")
```

`ImageBuf.write (imageoutput)`

Write the contents of the ImageBuf as the next subimage to an open ImageOutput.

Example:

```
buf = ImageBuf (...) # Existing ImageBuf

out = ImageOutput.create("out.exr")
out.open ("out.exr", buf.spec())

buf.write (out)
out.close()
```

`ImageBuf.make_writable (keep_cache_type=False)`

Force the ImageBuf to be writable. That means that if it was previously backed by an ImageCache (storage was IMAGECACHE), it will force a full read so that the whole image is in local memory.

`ImageBuf.set_write_format (format=oiio.UNKNOWN)`

`ImageBuf.set_write_tiles (width=0, height=0, depth=0)`

Override the data format or tile size in a subsequent call to `write()`. The `format` argument to `set_write_format` may be either a single data type description for all channels, or a tuple giving the data type for each channel in order.

Example:

```
# Conversion to a tiled unsigned 16 bit integer file
buf = ImageBuf ("in.tif")
buf.set_write_format ("uint16")
buf.set_write_tiles (64, 64)
buf.write ("out.tif")
```

`ImageBuf.spec()`

`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
```

`ImageBuf.specmod()`

`ImageBuf.specmod()` provides a reference to the writeable `ImageSpec` inside 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")
```

`ImageBuf.name()`

The file name of the image (as a string).

`ImageBuf.file_format_name()`

The file format of the image (as a string).

`ImageBuf.subimage`

`ImageBuf.miplevel`

`ImageBuf.nsubimages`

`ImageBuf.nmiplevels`

Several fields giving information about the current subimage and MIP level, and the total numbers thereof in the file.

`ImageBuf.xbegin`

`ImageBuf.xend`

`ImageBuf.ybegin`

`ImageBuf.yend`

`ImageBuf.zbegin`

`ImageBuf.zend`

The range of valid pixel data window. Remember that the end is *one past* the last pixel.

`ImageBuf.xmin`

`ImageBuf.xmax`

`ImageBuf.ymin`

`ImageBuf.ymax`

`ImageBuf.zmin`

`ImageBuf.zmax`

The minimum and maximum (inclusive) coordinates of the pixel data window.

`ImageBuf.orientation`

`ImageBuf.oriented_width`

`ImageBuf.oriented_height`

`ImageBuf.oriented_x`

`ImageBuf.oriented_y`

`ImageBuf.oriented_full_width`

`ImageBuf.oriented_full_height`

`ImageBuf.oriented_full_x`

`ImageBuf.oriented_full_y`

The Orientation field gives the suggested display orientation of the image (see Section [Display hints](#)).

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.

`ImageBuf.roi`

`ImageBuf.roi_full`

These fields hold 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_origin(x, y, z=0)`

Changes the “origin” of the data pixel data window to the specified coordinates.

Example:

```
# Shift the pixel data so the upper left is at pixel (10, 10)
buf.set_origin (10, 10)
```

`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)
```

`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.

`ImageBuf.pixeltype ()`

Returns a `TypeDesc` describing the data type of the pixels stored within the `ImageBuf`.

`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 ImageBuf.copy (format=TypeUnknown)`**

Return a full copy of this `ImageBuf` (with optional data format conversion, if `format` is supplied).

Example:

```
A = ImageBuf ("A.tif")

# Make a separate, duplicate copy of A
B = A.copy ()

# Make another copy of A, but converting to float pixels
C = A.copy ("float")
```

`ImageBuf.copy (other_imagebuf, format=TypeUnknown)`

Make this `ImageBuf` a complete copy of the other `ImageBuf`. If a `format` is provided, this will get the specified pixel data type rather than using the same pixel format as the source `ImageBuf`.

Example:

```
A = ImageBuf ("A.tif")

# Make a separate, duplicate copy of A
B = ImageBuf ()
B.copy (A)

# Make another copy of A, but converting to float pixels
C = ImageBuf ()
C.copy (A, oio.FLOAT)
```

`ImageBuf.swap (other_imagebuf)`

Swaps the content of this `ImageBuf` and the other `ImageBuf`.

Example:

```
A = ImageBuf ("A.tif")
B = ImageBuf ("B.tif")
```

(continues on next page)

(continued from previous page)

```
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="black")**

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: "black", "clamp", "periodic", "mirror").

Example:

```
buf = ImageBuf ("tahoe.jpg")
p = buf.getpixel (50, 50)
print p
> (0.37, 0.615, 0.97)
```

**magetBuf.getchannel (x, y, z, channel, wrap='black')**

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: "black", "clamp", "periodic", "mirror").

Example:

```
buf = ImageBuf ("tahoe.jpg")
green = buf.getchannel (50, 50, 0, 1)
```

**ImageBuf.interppixel (x, y, wrap='black')**

**ImageBuf.interppixel\_bicubic (x, y, wrap='black')**

Interpolates the image value (bilinearly or bicubically) 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: "black", "clamp", "periodic", "mirror").

Example:

```
buf = ImageBuf ("tahoe.jpg")
midx = float(buf.xbegin + buf.xend) / 2.0
midy = float(buf.ybegin + buf.yend) / 2.0
p = buf.interppixel (midx, midy)
# Now p is the interpolated value from right in the center of
# the data window
```

**ImageBuf.interppixel\_NDC (x, y, wrap='black')**

**ImageBuf.interppixel\_bicubic\_NDC (x, y, wrap='black')**

Interpolates the image value (bilinearly or bicubically) 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: "black", "clamp", "periodic", "mirror").

Example:

```
buf = ImageBuf ("tahoe.jpg")
p = buf.interppixel_NDC (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 float (one for each color channel).

Example:

```
buf = ImageBuf (ImageSpec (640, 480, 3, oio.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))
```

`ImageBuf.get_pixels (format=TypeFloat, roi=ROI.All)`

Retrieves the rectangle of pixels (and channels) specified by `roi` from the image and returns them as an array of values with type specified by `format`.

As with the `ImageInput` read functions, the return value is a NumPy ndarray containing data elements indexed as `[y][x][channel]` for normal 2D images, or for 3D volumetric images, as `[z][y][x][channel]`. Returns True upon success, False upon failure.

Example:

```
buf = ImageBuf ("tahoe.jpg")
pixels = buf.get_pixels (oio.FLOAT) # no ROI means the whole image
```

`ImageBuf.set_pixels (roi, data)`

Sets the rectangle of pixels (and channels) specified by `roi` with values in the `data`, which is a NumPy ndarray of values indexed as `[y][x][channel]` for normal 2D images, or for 3D volumetric images, as `[z][y][x][channel]`. (It will also work fine if the array is 1D “flattened” version, as long as it contains the correct total number of values.) The data type is deduced from the contents of the array itself.

Example:

```
buf = ImageBuf (...)
pixels = (...)
buf.set_pixels (ROI(), pixels)
```

`ImageBuf.has_error`

This field will be True if an error has occurred in the `ImageBuf`.

`ImageBuf.geterror()`

Retrieve the error message (and clear the `has_error` flag).

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()
```

`ImageBuf.pixelindex(x, y, z, check_range=False)`

Return the index of pixel (x,y,z).

`ImageBuf.deep`

Will be True if the file contains “deep” pixel data, or False for an ordinary images.

`ImageBuf.deep_samples` (*x, y, z=0*)  
Return the number of deep samples for pixel (*x,y,z*).

`ImageBuf.set_deep_samples` (*x, y, z, nsamples*)  
Set the number of deep samples for pixel (*x,y,z*).

`ImageBuf.deep_insert_samples` (*x, y, z, samplepos, nsamples*)  
`ImageBuf.deep_erase_samples` (*x, y, z, samplepos, nsamples*)  
Insert or erase *nsamples* samples starting at the given position of pixel (*x,y,z*).

`ImageBuf.deep_value` (*x, y, z, channel, sample*)  
`ImageBuf.deep_value_uint` (*x, y, z, channel, sample*)  
Return the value of the given deep sample (particular pixel, channel, and sample number) for a channel that is a float or an unsigned integer type, respectively.

`ImageBuf.set_deep_value` (*x, y, z, channel, sample, value*)  
`ImageBuf.set_deep_value_uint` (*x, y, z, channel, sample, value*)  
Set the value of the given deep sample (particular pixel, channel, and sample number) for a channel that is a float or an unsigned integer type, respectively.

**DeepData** `ImageBuf.deepdata`  
A reference to the underlying `DeepData` of the image.

## 11.9 ImageBufAlgo

The C++ `ImageBufAlgo` functions are described in detail in Chapter *ImageBufAlgo: Image Processing*. 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 *ImageBufAlgo: Image Processing*, 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).



## 11.9.1 Pattern generation

**ImageBuf ImageBufAlgo.zero (roi, nthreads=0)**

**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 of 0 values
buf = ImageBufAlgo.zero (ROI(0, 640, 0, 480, 0, 1, 0, 3))
```

**ImageBuf ImageBufAlgo.fill (values, roi=ROI.All, nthreads=0)**

**ImageBuf ImageBufAlgo.fill (top, bottom, roi=ROI.All, nthreads=0)**

**ImageBuf ImageBufAlgo.fill (topleft, topright, bottomleft, bottomright, roi=ROI.All, nthreads=0)**

**bool ImageBufAlgo.fill (dst, values, roi=ROI.All, nthreads=0)**

**bool ImageBufAlgo.fill (dst, top, bottom, roi=ROI.All, nthreads=0)**

**bool ImageBufAlgo.fill (dst, topleft, topright, bottomleft, bottomright, roi=ROI.All, nthreads=0)**

Return a filled float image of size roi, or set the the pixels of image dst within the ROI to a color or gradient.

Three fill options are available: (a) if one color tuple is supplied, the whole ROI will be filled with that constant value, (b) if two color tuples are supplied, a linear gradient will be applied from top to bottom, (c) if four color tuples are supplied, the ROI will be filled with values bilinearly interpolated from the four corner colors supplied.

Example:

```
# 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))
```

**ImageBuf ImageBufAlgo.checker(width, height, depth, color1, color2, xoffset=0, yoffset=0, nthreads=0)**

**bool ImageBufAlgo.checker(dst, width, height, depth, color1, color2, xoffset=0, yoffset=0, nthreads=0)**

Return (or copy into dst) a checkerboard pattern. The colors are specified as tuples giving the values for each color channel.

Example:

```
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))
```

**ImageBuf ImageBufAlgo.noise (noisetype, A=0.0, B=0.1, mono=False, seed=0, roi=ROI.All, nthreads=0)**

**bool ImageBufAlgo.noise (dst, noisetype, A=0.0, B=0.1, mono=False, seed=0, roi=ROI.All, nthreads=0)**

Return an image of pseudorandom noise, or add pseudorandom noise to the specified region of existing region dst.

For noise type “uniform”, the noise is uniformly distributed on the range [A,B). For noise “gaussian”, the noise will have a normal distribution with mean A and standard deviation B. For noise “salt”, the value A will be stored in a random set of pixels whose proportion (of the overall image) is B. For all noise types, choosing different seed values will result in a different pattern. If the mono flag is True, a single noise value will be applied to all channels specified by roi, but if mono is False, a separate noise value will be computed for each channel in the region.

Example:

```
buf = ImageBuf(ImageSpec(640, 480, 3, oio.UINT8))
ImageBufAlgo.zero (buf)
ImageBufAlgo.noise (buf, 'uniform', 0.25, 0.75)
```

`ImageBufAlgo.render_point (dst, x, y, color=1, 1, 1, 1)`

Render a point at pixel (x,y) of dst. The color (if supplied) is a tuple giving the per-channel colors. Return True for success, False for failure.

Example:

```
buf = ImageBuf(ImageSpec (640, 480, 4, oiio.FLOAT))
ImageBufAlgo.render_point (buf, 10, 20, (1,0,0,1))
```

**bool ImageBufAlgo.render\_line (dst, x1, y1, x2, y2, color=(1,1,1,1), skip\_first\_point=False)**

Render a line from pixel \$(x\_1,y\_1)\$ to \$(x\_2,y\_2)\$ into dst. The color (if supplied) is a tuple giving the per-channel colors.

Example:

```
buf = ImageBuf(ImageSpec (640, 480, 4, oiio.FLOAT))
ImageBufAlgo.render_line (buf, 10, 10, 500, 20, (1,0,0,1))
```

**bool ImageBufAlgo.render\_box (dst, x1, y1, x2, y2, color=(1,1,1,1), filled=False)**

Render a filled or unfilled box with corners at pixels \$(x\_1,y\_1)\$ and \$(x\_2,y\_2)\$ into dst. The color (if supplied) is a tuple giving the per-channel colors.

Example:

```
buf = ImageBuf(ImageSpec (640, 480, 4, oiio.FLOAT))
ImageBufAlgo.render_box (buf, 150, 100, 240, 180, (0,1,1,1))
ImageBufAlgo.render_box (buf, 100, 50, 180, 140, (0.5, 0.5, 0, 0.5), True)
```

**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. Choices for alignx are “left”, “right”, and “center”, and choices for aligny are “baseline”, “top”, “bottom”, and “center”.

Example:

```
buf = ImageBuf(ImageSpec (640, 480, 4, oiio.FLOAT))
ImageBufAlgo.render_text (buf, 50, 100, "Hello, world")
ImageBufAlgo.render_text (buf, 100, 200, "Go Big Red!",
                           60, "Arial Bold", (1,0,0,1))
```

**ROI ImageBufAlgo.text\_size (text, fontsize=16, fontname="")**

Compute the size that will be needed for the text as an ROI and return it. The size will not be defined if an error occurred (such as not being a valid font name).

Example:

```
A = ImageBuf(ImageSpec (640, 480, 4, oiio.FLOAT))
Aroi = A.roi
size = ImageBufAlgo.text_size ("Centered", 40, "Courier New")
if size.defined :
    x = Aroi.xbegin + Aroi.width/2 - (size.xbegin + size.width/2)
    y = Aroi.ybegin + Aroi.height/2 - (size.ybegin + size.height/2)
    ImageBufAlgo.render_text (A, x, y, "Centered", 40, "Courier New")

# Note: this was for illustration. An easier way to do this is:
# render_text (A, x, y, "Centered", 40, "Courier New", alignx="center")
```

## 11.9.2 Image transformations and data movement

**ImageBuf ImageBufAlgo.channels** (src, channelorder, newchannelnames=(), shuffle\_channel\_names=0)  
**bool ImageBufAlgo.channels** (dst, src, channelorder, newchannelnames=(), shuffle\_channel\_names=0)

Return (or store in dst) shuffled channels of src, 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.

If newchannelnames is supplied, it is a tuple of new channel names. (See the C++ version for more full explanation.)

Example:

```
# Copy the first 3 channels of an RGBA, drop the alpha
RGBA = ImageBuf("rgba.tif")
RGB = ImageBufAlgo.channels (RGBA, (0,1,2))

# Copy just the alpha channel, making a 1-channel image
Alpha = ImageBufAlgo.channels (RGBA, ("A",))

# Swap the R and B channels
BGRA = ImageBufAlgo.channels (RGBA, (2, 1, 0, 3))

# Add an alpha channel with value 1.0 everywhere to an RGB image
RGBA = ImageBufAlgo.channels (RGB, ("R", "G", "B", 1.0),
                               ("R", "G", "B", "A"))
```

**ImageBuf ImageBufAlgo.channel\_append** (A, B, roi=ROI.All, nthreads=0) **bool ImageBufAlgo.channel\_append** (dst, A, B, roi=ROI.All, nthreads=0)

Append the channels of images A and B together into one image.

Example:

```
RGBA = ImageBuf ("rgba.exr")
Z = ImageBuf ("z.exr")
RGBAZ = ImageBufAlgo.channel_append (RGBA, Z)
```

**ImageBuf ImageBufAlgo.copy** (src, convert=TypeDesc.UNKNOWN, roi=ROI.All, nthreads=0)  
**bool ImageBufAlgo.copy** (dst, src, convert=TypeDesc.UNKNOWN, roi=ROI.All, nthreads=0)

Copy the specified region of pixels of src at the same locations, optionally with the pixel type overridden by convert (if it is not UNKNOWN).

Example:

```
# Copy A's upper left 200x100 region into B
B = ImageBufAlgo.copy (A, ROI(0,200,0,100))
```

**ImageBuf ImageBufAlgo.crop** (src, roi=ROI.All, nthreads=0)  
**bool ImageBufAlgo.crop** (dst, src, roi=ROI.All, nthreads=0)

Reset dst to be the specified region of src.

Example:

```
# Set B to be the upper left 200x100 region of A
A = ImageBuf ("a.tif")
B = ImageBufAlgo.crop (A, ROI(0,200,0,100))
```

```
ImageBuf ImageBufAlgo.cut (src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.cut (dst, src, roi=ROI.All, nthreads=0)
```

Reset dst to be the specified region of src, but moved so that the resulting new image has its pixel data at the image plane origin.

Example:

```
# Set B to be the lower left 200x100 region of A, moved to the origin
A = ImageBuf ("a.tif")
B = ImageBufAlgo.cut (A, ROI(0,200,380,480))
```

```
bool ImageBufAlgo.paste (dst, xbegin, ybegin, zbegin, chbegin, src, ROI srcroi=ROI.All, nthreads=0)
Copy the specified region of src into dst with the given offset (xbegin, ybegin, zbegin).
```

Example:

```
# 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)
```

```
ImageBuf ImageBufAlgo.rotate90 (src, roi=ROI.All, nthreads=0)
ImageBuf ImageBufAlgo.rotate180 (src, roi=ROI.All, nthreads=0)
ImageBuf ImageBufAlgo.rotate270 (src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.rotate90 (dst, src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.rotate180 (dst, src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.rotate270 (dst, src, roi=ROI.All, nthreads=0)
```

Copy while rotating the image by a multiple of 90 degrees.

Example:

```
A = ImageBuf ("tahoe.exr")
B = ImageBufAlgo.rotate90 (A)
```

```
ImageBuf ImageBufAlgo.flip (src, roi=ROI.All, nthreads=0)
ImageBuf ImageBufAlgo.flop (src, roi=ROI.All, nthreads=0)
ImageBuf ImageBufAlgo.transpose (src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.flip (dst, src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.flop (dst, src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.transpose (dst, src, roi=ROI.All, nthreads=0)
```

Copy while reversing orientation vertically (flip) or horizontally (flop), or diagonally (transpose).

Example:

```
A = ImageBuf ("tahoe.exr")
B = ImageBufAlgo.flip (A)
```

```
ImageBuf ImageBufAlgo.reorient (src, nthreads=0)
bool ImageBufAlgo.reorient (dst, src, nthreads=0)
```

Copy src, applying whatever series of rotations, flips, or flops are necessary to transform the pixels into the configuration suggested by the "Orientation" metadata of the image (and the "Orientation" metadata is then set to 1, ordinary orientation).

Example:

```
A = ImageBuf ("tahoe.jpg")
ImageBufAlgo.reorient (A, A)
```

**ImageBuf ImageBufAlgo.circular\_shift (src, xshift, yshift, zshift=0, roi=ROI.All, nthreads=0)**  
**bool ImageBufAlgo.circular\_shift (dst, src, xshift, yshift, zshift=0, roi=ROI.All, nthreads=0)**  
 Copy while circularly shifting by the given amount.

Example:

```
A = ImageBuf ("tahoe.exr")
B = ImageBufAlgo.circular_shift (A, 200, 100)
```

**ImageBuf ImageBufAlgo.rotate (src, angle, filtername="", filtersize=0.0, recompute\_roi=False)**  
**ImageBuf ImageBufAlgo.rotate (src, angle, center\_x, center\_y, filtername="", filtersize=0.0, recompute\_roi=False)**  
**bool ImageBufAlgo.rotate (dst, src, angle, filtername="", filtersize=0.0, recompute\_roi=False)**  
**bool ImageBufAlgo.rotate (dst, src, angle, center\_x, center\_y, filtername="", filtersize=0.0, recompute\_roi=False)**  
 Copy arotated version of the corresponding portion of *src*. The angle is in radians, with positive values indicating clockwise rotation. If the filter and size are not specified, an appropriate default will be chosen.

Example:

```
Src = ImageBuf ("tahoe.exr")
Dst = ImageBufAlgo.rotate (Src, math.radians(45.0))
```

**ImageBuf ImageBufAlgo.warp (src, M, filtername="", filtersize=0.0, wrap="default", recompute\_roi=False)**  
**bool ImageBufAlgo.warp (dst, src, M, filtername="", filtersize=0.0, wrap="default", recompute\_roi=False)**  
 Compute a warped (transformed) copy of *src*, with the warp specified by *M* consisting of 9 floating-point numbers representing a 3x3 transformation matrix. If the filter and size are not specified, an appropriate default will be chosen.

Example:

```
M = (0.7071068, 0.7071068, 0, -0.7071068, 0.7071068, 0, 20, -8.284271, 1)
Src = ImageBuf ("tahoe.exr")
Dst = ImageBufAlgo.warp (Src, M)
```

**ImageBuf ImageBufAlgo.resize (src, filtername="", filtersize=0.0, roi=ROI.All, nthreads=0)**  
**bool ImageBufAlgo.resize (dst, src, filtername="", filtersize=0.0, roi=ROI.All, nthreads=0)**  
 Compute 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.

Example:

```
# Resize the image to 640x480, using the default filter
Src = ImageBuf ("tahoe.exr")
Dst = ImageBufAlgo.resize (Src, roi=ROI(0,640,0,480,0,1,0,3))
```

**ImageBuf ImageBufAlgo.resample (src, interpolate=True, roi=ROI.All, nthreads=0)**  
**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*).

Example:

```
# Resample quickly to 320x240 to make a low-quality thumbnail
Src = ImageBuf ("tahoe.exr")
Dst = ImageBufAlgo.resample (Src, roi=ROI(0,640,0,480,0,1,0,3))
```

**ImageBuf ImageBufAlgo.fit** (src, filtername="", filtersize=0.0, exact=false, roi=ROI.All, nt  
**bool ImageBufAlgo.fit** (dst, src, filtername="", filtersize=0.0, exact=false, roi=ROI.All, nt  
Fit src into the roi while preserving the original aspect ratio, without stretching. If the filter and size are not specified, an appropriate default will be chosen.

Example:

```
# Resize to fit into a max of 640x480, preserving the aspect ratio
Src = ImageBuf ("tahoe.exr")
Dst = ImageBufAlgo.fit (Src, roi=ROI(0,640,0,480,0,1,0,3))
```

### 11.9.3 Image arithmetic

**ImageBuf ImageBufAlgo.add** (A, B, roi=ROI.All, nthreads=0)  
**bool ImageBufAlgo.add** (dst, A, B, roi=ROI.All, nthreads=0)

Compute  $A + B$ . A and B each may be an ImageBuf, a float value (for all channels) or a tuple giving a float for each color channel.

Example:

```
# Add two images
buf = ImageBufAlgo.add (ImageBuf("a.exr"), ImageBuf("b.exr"))

# Add 0.2 to channels 0-2
ImageBufAlgo.add (buf, buf, (0.2,0.2,0.2,0))
```

**ImageBuf ImageBufAlgo.sub** (A, B, roi=ROI.All, nthreads=0)  
**bool ImageBufAlgo.sub** (dst, A, B, roi=ROI.All, nthreads=0)

Compute  $A - B$ . A and B each may be an ImageBuf, a float value (for all channels) or a tuple giving a float for each color channel.

Example:

```
buf = ImageBufAlgo.sub (ImageBuf("a.exr"), ImageBuf("b.exr"))
```

**ImageBuf ImageBufAlgo.absdiff** (A, B, roi=ROI.All, nthreads=0)  
**bool ImageBufAlgo.absdiff** (dst, A, B, roi=ROI.All, nthreads=0)

Compute  $\text{abs}(A - B)$ . A and B each may be an ImageBuf, a float value (for all channels) or a tuple giving a float for each color channel.

Example:

```
buf = ImageBufAlgo.absdiff (ImageBuf("a.exr"), ImageBuf("b.exr"))
```

**ImageBuf ImageBufAlgo.abs** (A, roi=ROI.All, nthreads=0)  
**bool ImageBufAlgo.abs** (dst, A, roi=ROI.All, nthreads=0)

Compute  $\text{abs}(A)$ . A is an ImageBuf.

Example:

```
buf = ImageBufAlgo.abs (ImageBuf("a.exr"))
```

**ImageBuf ImageBufAlgo.mul** (A, B, roi=ROI.All, nthreads=0)  
**bool ImageBufAlgo.mul** (dst, A, B, roi=ROI.All, nthreads=0)

Compute  $A * B$  (channel-by-channel multiplication). A and B each may be an ImageBuf, a float value (for all channels) or a tuple giving a float for each color channel.

Example:

```
# Multiply the two images
buf = ImageBufAlgo.mul (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, 1.0))
```

**ImageBuf ImageBufAlgo.div (A, B, roi=ROI.All, nthreads=0)**

**bool ImageBufAlgo.div (dst, A, B, roi=ROI.All, nthreads=0)**

Compute  $A / B$  (channel-by-channel division), where  $x/0$  is defined to be 0. A and B each may be an ImageBuf, a float value (for all channels) or a tuple giving a float for each color channel.

Example:

```
# Divide a.exr by b.exr
buf = ImageBufAlgo.div (ImageBuf("a.exr"), ImageBuf("b.exr"))

# Reduce intensity of buf's channels 0-2 by 50%, in place
ImageBufAlgo.div (buf, buf, (2.0, 2.0, 2.0, 1.0))
```

**ImageBuf ImageBufAlgo.mad (A, B, C, roi=ROI.All, nthreads=0)**

**bool ImageBufAlgo.mad (dst, A, B, C, roi=ROI.All, nthreads=0)**

Compute  $A * B + C$  (channel-by-channel multiplication and addition). A, B, and C each may be an ImageBuf, a float value (for all channels) or a tuple giving a float for each color channel.

Example:

```
# Multiply a and b, then add c
buf = ImageBufAlgo.mad (ImageBuf("a.exr"),
                        (1.0f, 0.5f, 0.25f), ImageBuf("c.exr"))
```

**ImageBuf ImageBufAlgo.invert (A, roi=ROI.All, nthreads=0)**

**bool ImageBufAlgo.invert (dst, A, roi=ROI.All, nthreads=0)**

Compute  $1-A$  (channel by channel color inverse). A is an ImageBuf.

Example:

```
buf = ImageBufAlgo.invert (ImageBuf("a.exr"))
```

**ImageBuf ImageBufAlgo.pow (A, B, roi=ROI.All, nthreads=0)**

**bool ImageBufAlgo.pow (dst, A, B, roi=ROI.All, nthreads=0)**

Compute  $\text{pow}(A, B)$  (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.

Example:

```
# Linearize a 2.2 gamma-corrected image (channels 0-2 only)
img = ImageBuf ("a.exr")
buf = ImageBufAlgo.pow (img, (2.2, 2.2, 2.2, 1.0))
```

**ImageBuf ImageBufAlgo.channel\_sum (src, weights=(), roi=ROI.All, nthreads=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).

Example:

```
# Compute luminance via a weighted sum of R,G,B
# (assuming Rec709 primaries and a linear scale)
```

(continues on next page)

(continued from previous page)

```

ImageBuf()
weights = (.2126, .7152, .0722)
luma = ImageBufAlgo.channel_sum (ImageBuf("a.exr"), weights)

```

**ImageBuf ImageBufAlgo.contrast\_remap (src, black=0.0, white=1.0, min=0.0, max=1.0, sthresh=1.0)**

**bool ImageBufAlgo.contrast\_remap (ImageBuf &dst, src, black=0.0, white=1.0, min=0.0, max=1.0, sthresh=1.0)**

Return (or copy into dst) pixel values that are a contrast-remap of the corresponding values of the src image, transforming pixel value domain [black, white] to range [min, max], either linearly or with optional application of a smooth sigmoidal remapping (if scontrast != 1.0).

Example:

```

A = ImageBuf('tahoe.tif');

# Simple linear remap that stretches input 0.1 to black, and input
# 0.75 to white.
linstretch = ImageBufAlgo.contrast_remap (A, black=0.1, white=0.75)

# Remapping 0->1 and 1->0 inverts the colors of the image,
# equivalent to ImageBufAlgo.invert().
inverse = ImageBufAlgo.contrast_remap (A, black=1.0, white=0.0)

# Use a sigmoid curve to add contrast but without any hard cutoffs.
# Use a contrast parameter of 5.0.
sigmoid = ImageBufAlgo.contrast_remap (a, contrast=5.0)

```

**ImageBuf ImageBufAlgo.color\_map (src, srcchannel, nknots, channels, knots, roi=ROI.All, nthreads=0)**

**ImageBuf ImageBufAlgo.color\_map (src, srcchannel, mapname, roi=ROI.All, nthreads=0)**

**bool ImageBufAlgo.color\_map (dst, src, srcchannel, nknots, channels, knots, roi=ROI.All, nthreads=0)**

**bool ImageBufAlgo.color\_map (dst, src, srcchannel, mapname, roi=ROI.All, nthreads=0)**

Return an image (or copy into dst) pixel values determined by applying the color map to the values of src, using either the channel specified by srcchannel, or the luminance of src's RGB if srcchannel is -1.

In the first variant, the values linearly-interpolated color map are given by the tuple knots[nknots\*channels].

In the second variant, just the name of a color map is specified. Recognized map names include: “inferno”, “viridis”, “magma”, “plasma”, all of which are perceptually uniform, strictly increasing in luminance, look good when converted to grayscale, and work for people with all types of colorblindness. The “turbo” color map is also nice in most of these ways (except for being strictly increasing in luminance). Also supported are the following color maps that do not have those desirable qualities (and are thus not recommended): “blue-red”, “spectrum”, and “heat”. In all cases, the implied channels is 3.

Example:

```

heatmap = ImageBufAlgo.color_map (ImageBuf("a.jpg"), -1, "inferno")

heatmap = ImageBufAlgo.color_map (ImageBuf("a.jpg"), -1, 3, 3,
                                   (0.25, 0.25, 0.25, 0, 0.5, 0, 1, 0, 0))

```

**ImageBuf ImageBufAlgo.clamp (src, min, max, bool clampalpha01=False, roi=ROI.All, nthreads=0)**

**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.

Example:



```
# Clamp image buffer A in-place to the [0,1] range for all channels.
ImageBufAlgo.clamp (A, A, 0.0, 1.0)
```

```
ImageBuf ImageBufAlgo.rangecompress (src, useluma=False, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.rangecompress (dst, src, useluma=False, roi=ROI.All, nthreads=0)
ImageBuf ImageBufAlgo.rangeexpand (src, useluma=False, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.rangeexpand (dst, src, useluma=False, roi=ROI.All, nthreads=0)
```

Copy from `src`, 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`).

Example:

```
# 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 = ImageBufAlgo.rangecompress (Src)

# 3. Now do the resize
roi = ROI (0, 640, 0, 480, 0, 1, 0, Compressed.nchannels)
Dst = ImageBufAlgo.resize (Compressed, "lanczos3", 6.0, roi)

# 4. Expand range to be linear again (operate in-place)
ImageBufAlgo.rangeexpand (Dst, Dst)
```

```
ImageBuf ImageBufAlgo.over (A, B, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.over (dst, A, B, roi=ROI.All, nthreads=0)
```

Composite ImageBuf A over ImageBuf B.

Example:

```
Comp = ImageBufAlgo.over (ImageBuf("fg.exr"), ImageBuf("bg.exr"))
```

```
ImageBuf ImageBufAlgo.zover (A, B, bool z_zeroisinf=False, roi=ROI.All, nthreads=0)
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.

Example:

```
Comp = ImageBufAlgo.zover (ImageBuf("fg.exr"), ImageBuf("bg.exr"))
```

### 11.9.4 Image comparison and statistics

**PixelStats** `ImageBufAlgo.computePixelStats (src, roi=ROI.All, nthreads=0)`

Compute statistics about the ROI of the image `src`. The `PixelStats` structure is defined as containing the following data fields: `min`, `max`, `avg`, `stddev`, `nancount`, `infcount`, `finitecount`, `sum`, `sum2`, each of which is a tuple with one value for each channel of the image.

Example:

```
A = ImageBuf("a.exr")
stats = ImageBufAlgo.computePixelStats (A)
print "    min = ", stats.min
print "    max = ", stats.max
print "    average = ", stats.avg
print "    standard deviation = ", stats.stddev
print "    # NaN values = ", stats.nancount
print "    # Inf values = ", stats.infcount
print "    # finite values = ", stats.finitecount
```

**CompareResults** `ImageBufAlgo.compare (A, B, failthresh, warnthresh, roi=ROI.All, nthreads=0)`

Numerically compare two `ImageBuf`'s, `A` and `B`. The `failthresh` and `warnthresh` supply failure and warning difference thresholds. The return value is 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
error                                 # True if there was an error
```

Example:

```
A = ImageBuf ("a.exr")
B = ImageBuf ("b.exr")
comp = ImageBufAlgo.compare (A, B, 1.0/255.0, 0.0)
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, threshold=0.0, 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`.

Example:

```
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, threshold=0.0, roi=ROI.All, nthreads=0)**  
Returns True if all pixels of src within the ROI have the given channel value val.

Example:

```
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, threshold=0.0, roi=ROI.All, nthreads=0)**  
Returns True if the image is monochrome within the ROI.

Example:

```
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"
```

**ROI ImageBufAlgo.nonzero\_region (src, roi=ROI.All, nthreads=0)**  
Returns an ROI that tightly encloses the minimal region within roi that contains all pixels with nonzero values.

Example:

```
A = ImageBuf ("a.exr")
nonzero_roi = ImageBufAlgo.nonzero_region(A)
```

**std::string ImageBufAlgo.computePixelHashSHA1 (src, extrainfo = "", roi=ROI.All, blocksize=64)**  
Compute the SHA-1 byte hash for all the pixels in the ROI of src.

Example:

```
A = ImageBuf ("a.exr")
hash = ImageBufAlgo.computePixelHashSHA1 (A, blocksize=64)
```

**tuple histogram (src, channel=0, bins=256, min=0.0, max=1.0, ignore\_empty=False, roi=ROI.All, nthreads=0)**  
Computes a histogram of the given channel of image src, within the ROI, returning a tuple of length bins containing count of pixels whose value was in each of the equally-sized range bins between min and max. If ignore\_empty is True, pixels that are empty (all channels 0 including alpha) will not be counted in the total.

## 11.9.5 Convolutions

**ImageBuf ImageBufAlgo.make\_kernel (name, width, height, depth=1.0, normalize=True)**  
Create 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", "cubic", "keys", "simon", "rifman", "disk", "binomial", "laplacian". 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.

Example:

```
K = ImageBufAlgo.make_kernel ("gaussian", 5.0, 5.0)
```

```
ImageBuf ImageBufAlgo.convolve (src, kernel, normalize=True, roi=ROI.All, nthreads=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).
```

Example:

```
# Blur an image with a 5x5 Gaussian kernel
Src = ImageBuf ("tahoe.exr")
K = ImageBufAlgo.make_kernel (K, "gaussian", 5.0, 5.0)
Blurred = ImageBufAlgo.convolve (Src, K)
```

```
ImageBuf ImageBufAlgo.laplacian (src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.laplacian (dst, src, roi=ROI.All, nthreads=0)
Replace the given ROI of dst with the Laplacian of the corresponding part of src.
```

Example:

```
Src = ImageBuf ("tahoe.exr")
L = ImageBufAlgo.laplacian (Src)
```

```
ImageBuf ImageBufAlgo.fft (src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.fft (dst, src, roi=ROI.All, nthreads=0)
ImageBuf ImageBufAlgo.ifft (src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.ifft (dst, src, roi=ROI.All, nthreads=0)
Compute the forward or inverse discrete Fourier Transform.
```

Example:

```
Src = ImageBuf ("tahoe.exr")

# Take the DFT of the first channel of Src
Freq = ImageBufAlgo.fft (Src)

# At this point, Freq is a 2-channel float image (real, imag)
# Convert it back from frequency domain to a spatial iamge
Spatial = ImageBufAlgo.ifft (Freq)
```

```
ImageBuf ImageBufAlgo.complex_to_polar (src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.complex_to_polar (dst, src, roi=ROI.All, nthreads=0)
ImageBuf ImageBufAlgo.polar_to_complex (src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.polar_to_complex (dst, src, roi=ROI.All, nthreads=0)
Transform a 2-channel image from complex (real, imaginary) representation to polar (amplitude, phase), or vice versa.
```

Example:

```
Polar = ImageBuf ("polar.exr")

Complex = ImageBufAlgo.polar_to_complex (Polar)

# At this point, Complex is a 2-channel complex image (real, imag)
```

(continues on next page)

(continued from previous page)

```
# Convert it back from frequency domain to a spatial iamge
Spatial = ImageBufAlgo.ifft (Complex)
```

## 11.9.6 Image Enhancement / Restoration

**ImageBuf ImageBufAlgo.fixNonFinite (src, mode=NONFINITE\_BOX3, roi=ROI.All, nthreads=0)**  
**bool ImageBufAlgo.fixNonFinite (dst, src, mode=NONFINITE\_BOX3, roi=ROI.All, nthreads=0)**  
 Copy pixel values from `src` 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
```

Example:

```
Src = ImageBuf ("tahoe.exr")
ImageBufAlgo.fixNonFinite (Src, Src, OpenImageIO.NONFINITE_BOX3)
```

**ImageBuf ImageBufAlgo.fillholes\_pushpull (src, roi=ROI.All, nthreads=0)**  
**bool ImageBufAlgo.fillholes\_pushpull (dst, src, roi=ROI.All, nthreads=0)**

Copy the specified ROI of `src` 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  $< 1$ , `dst` will have a pixel color that is a plausible “filling” of the original alpha hole.

Example:

```
Src = ImageBuf ("holes.exr")
Filled = ImageBufAlgo.fillholes_pushpull (Src)
```

**bool ImageBufAlgo.median\_filter (dst, src, width=3, height=-1, roi=ROI.All, nthreads=0)**  
 Replace the given ROI of `dst` with the `width` x `height` median filter of the corresponding region of `src` using the “unsharp mask” technique.

Example:

```
Noisy = ImageBuf ("tahoe.exr")
Clean = ImageBuf ()
ImageBufAlgo.median_filter (Clean, Noisy, 3, 3)
```

**ImageBuf ImageBufAlgo.dilate (src, width=3, height=-1, roi=ROI.All, nthreads=0)**  
**bool ImageBufAlgo.dilate (dst, src, width=3, height=-1, roi=ROI.All, nthreads=0)**  
**ImageBuf ImageBufAlgo.erode (src, width=3, height=-1, roi=ROI.All, nthreads=0)**  
**bool ImageBufAlgo.erode (dst, src, width=3, height=-1, roi=ROI.All, nthreads=0) }**  
 Compute a dilated or eroded version of the corresponding region of `src`.

Example:

```
Source = ImageBuf ("source.tif")
Dilated = ImageBufAlgo.dilate (Source, 3, 3)
```

**ImageBuf ImageBufAlgo.unsharp\_mask (src, kernel="gaussian", width=3.0, contrast=1.0, thresh=0.0)**

**bool ImageBufAlgo.unsharp\_mask (dst, src, kernel="gaussian", width=3.0, contrast=1.0, threshold=0.0)**  
Compute a sharpened version of the corresponding region of `src` using the “unsharp mask” technique.

Example:

```
Blurry = ImageBuf ("tahoe.exr")
Sharp = ImageBufAlgo.unsharp_mask (Blurry, "gaussian", 5.0)
```

### 11.9.7 Color manipulation

**ImageBuf ImageBufAlgo.colorconvert (src, fromspace, tospace, unpremult=True, context\_key="")**  
**bool ImageBufAlgo.colorconvert (dst, src, fromspace, tospace, unpremult=True, context\_key="")**  
Apply a color transform to the pixel values.

Example:

```
Src = ImageBuf ("tahoe.jpg")
Dst = ImageBufAlgo.colorconvert (Src, "sRGB", "linear")
```

**ImageBuf ImageBufAlgo.colormatrixtransform (src, M, unpremult=True, roi=ROI.All, nthreads=1)**  
**bool ImageBufAlgo.colormatrixtransform (dst, src, M, unpremult=True, roi=ROI.All, nthreads=1)**  
*NEW in 2.1*

Apply a 4x4 matrix color transform to the pixel values. The matrix can be any tuple of 16 float values.

Example:

```
Src = ImageBuf ("tahoe.jpg")
M = ( .8047379, .5058794, -.3106172, 0,
      -.3106172, .8047379, .5058794, 0,
      .5058794, -.3106172, .8047379, 0,
      0, 0, 0, 1)
Dst = ImageBufAlgo.colormatrixtransform (Src, M)
```

**ImageBuf ImageBufAlgo.ociolook (src, looks, fromspace, tospace, unpremult=True, inverse=False)**  
**bool ImageBufAlgo.ociolook (dst, src, looks, fromspace, tospace, unpremult=True, inverse=False)**  
Apply an OpenColorIO “look” transform to the pixel values.

Example:

```
Src = ImageBuf ("tahoe.jpg")
Dst = ImageBufAlgo.ociolook (Src, "look", "vd8", "lnf",
                             context_key="SHOT", context_value="pe0012")
```

**ImageBuf ImageBufAlgo.ociodisplay (src, display, view, fromspace="", looks="", unpremult=True)**  
**bool ImageBufAlgo.ociodisplay (dst, src, display, view, fromspace="", looks="", unpremult=True)**  
Apply an OpenColorIO “display” transform to the pixel values.

Example:

```
Src = ImageBuf ("tahoe.exr")
Dst = ImageBufAlgo.ociodisplay (Src, "sRGB", "Film", "lnf",
                                context_key="SHOT", context_value="pe0012")
```

**ImageBuf ImageBufAlgo.ociofiletransform (src, name, unpremult=True, inverse=False, colorcorrection="")**  
**bool ImageBufAlgo.ociofiletransform (dst, src, name, unpremult=True, inverse=False, colorcorrection="")**  
Apply an OpenColorIO “file” transform to the pixel values. In-place operations (`dst` and `src` being the same image) are supported.

Example:

```
Src = ImageBuf ("tahoe.exr")
Dst = ImageBufAlgo.ociofiletransform (Src, "foottransform.csp")
```

```
ImageBuf ImageBufAlgo.unpremult (src, roi=ROI.All, nthreads=0)
bool ImageBufAlgo.unpremult (dst, src, roi=ROI.All, nthreads=0)
ImageBuf ImageBufAlgo.premult (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.
```

Example:

```
# Convert in-place from associated alpha to unassociated alpha
A = ImageBuf ("a.exr")
ImageBufAlgo.unpremult (A, A)
```

### 11.9.8 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 *Import / export*.

Example:

```
# This command line:
#   maketx in.exr --hcomp --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:opaque_detect", 1)
ImageBufAlgo.make_texture (oiio.MakeTxTexture, Input,
                           "texture.exr", config)
```

**ImageBuf ImageBufAlgo::capture\_image (camer anum, convert = OpenImageIO.UNKNOWN)**  
Capture a still image from a designated camera.

Example:

```
WebcamImage = ImageBufAlgo.capture_image (0, OpenImageIO.UINT8)
WebcamImage.write ("webcam.jpg")
```

### 11.9.9 Functions specific to deep images

```
ImageBuf ImageBufAlgo.deepen (src, zvalue=1.0, roi=ROI.All, nthreads=0)  
bool ImageBufAlgo.deepen (dst, src, zvalue=1.0, roi=ROI.All, nthreads=0)
```

Convert a flat image to a deep one that has one depth sample per pixel (but no depth samples for the pixels corresponding to those in the source image that have infinite “Z” or that had 0 for all color channels and no “Z” channel).

Example:

```
Deep = ImageBufAlgo.deepen (ImageBuf("az.exr"))
```

```
ImageBuf ImageBufAlgo.flatten (src, roi=ROI.All, nthreads=0)  
bool ImageBufAlgo.flatten (dst, src, roi=ROI.All, nthreads=0)
```

Composite the depth samples within each pixel of “deep” ImageBuf `src` to produce a “flat” ImageBuf.

Example:

```
Flat = ImageBufAlgo.flatten (ImageBuf("deepalpha.exr"))
```

```
ImageBuf ImageBufAlgo.deep_merge (A, B, occlusion_cull, roi=ROI.All, nthreads=0)  
bool ImageBufAlgo.deep_merge (dst, A, B, occlusion_cull, roi=ROI.All, nthreads=0)
```

Merge the samples of two deep images A and B into a deep result. If `occlusion_cull` is `True`, samples beyond the first opaque sample will be discarded, otherwise they will be kept.

Example:

```
DeepA = ImageBuf("hardsurf.exr")  
DeepB = ImageBuf("volume.exr")  
Merged = ImageBufAlgo.deep_merge (DeepA, DeepB)
```

```
ImageBuf ImageBufAlgo.deep_holdout (src, holdout, roi=ROI.All, nthreads=0)  
bool ImageBufAlgo.deep_holdout (dst, src, holdout, roi=ROI.All, nthreads=0)
```

Return the pixels of `src`, but only copying the samples that are closer than the opaque frontier of image `holdout`. That is, `holdout` will serve as a depth holdout mask, but no samples from `holdout` will actually be copied to `dst`.

Example:

```
Img = ImageBuf("image.exr")  
Mask = ImageBuf("mask.exr")  
Thresholded = ImageBufAlgo.deep_holdout (Img, Mask)
```

### 11.9.10 Other ImageBufAlgo methods that understand deep images

In addition to the previously described methods that are specific to deep images, the following ImageBufAlgo methods (described in their respective sections) work with deep inputs:

```
ImageBufAlgo.add  
ImageBufAlgo.channels  
ImageBufAlgo.compare  
ImageBufAlgo.computePixelStats  
ImageBufAlgo.crop  
ImageBufAlgo.div  
ImageBufAlgo.fixNonFinite  
ImageBufAlgo.mul
```

(continues on next page)



(continued from previous page)

```
ImageBufAlgo.nonzero_region
ImageBufAlgo.resample
ImageBufAlgo.sub
```

## 11.10 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 [Global Attributes](#), please refer there for details.

### **openimageio\_version**

The OpenImageIO version number is an `int`, 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.

### **geterror()**

Retrieves the latest global error, as a string.

**attribute** (*name*, *typedesc*, *value*)

**attribute** (*name*, *int\_value*)

**attribute** (*name*, *float\_value*)

**attribute** (*name*, *str\_value*)

Sets a global attribute (see Section [Global Attributes](#) for details), returning `True` upon success, or `False` if it was not a recognized attribute.

Example:

```
oio.attribute ("threads", 0)
```

**getattribute** (*name*, *typedesc*)

**get\_int\_attribute** (*name*, *defaultval*=0)

**get\_float\_attribute** (*name*, *defaultval*=0.0)

**get\_string\_attribute** (*name*, *defaultval*="")

Retrieves an attribute value from the named set of global OIO options. (See Section [Global Attributes](#).) The `getattribute()` function returns the value 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. Type variety with the type in the name also takes a default value.

Example:

```
formats = oio.get_string_attribute ("format_list")
```

## 11.11 Python Recipes

This section illustrates the Python syntax for doing many common image operations from Python scripts, but that aren't already given as examples in the earlier function descriptions. All example code fragments assume the following boilerplate:

```
#!/usr/bin/env python

import OpenImageIO as oiio
from OpenImageIO import ImageBuf, ImageSpec, ImageBufAlgo
```

### Subroutine to create a constant-colored image

```
# Create an ImageBuf holding a n image of constant color, given the
# resolution, data format (defaulting to UINT8), fill value, and image
# origin.
def make_constimage (xres, yres, chans=3, format=oiio.UINT8, value=(0,0,0),
                    xoffset=0, yoffset=0) :
    spec = ImageSpec (xres,yres,chans,format)
    spec.x = xoffset
    spec.y = yoffset
    b = ImageBuf (spec)
    oiio.ImageBufAlgo.fill (b, value)
    return b
```

The image is returned as an ImageBuf, then up to the caller what to do with it next.

### Subroutine to save an image to disk, printing errors

```
# Save an ImageBuf to a given file name, with optional forced image format
# and error handling.
def write_image (image, filename, format=oiio.UNKNOWN) :
    if not image.has_error :
        image.write (filename, format)
    if image.has_error :
        print "Error writing", filename, ":", image.geterror()
```

### Converting between file formats

```
img = ImageBuf ("input.png")
write_image (img, "output.tif")
```

### Comparing two images and writing a difference image

```
A = ImageBuf ("A.tif")
B = ImageBuf ("B.tif")
compresults = ImageBufAlgo.compare (A, B, 1.0e-6, 1.0e-6)
if compresults.nfail > 0 :
    print "Images did not match, writing difference image diff.tif"
    diff = ImageBufAlgo.absdiff (A, B)
    image_write (diff, "diff.tif")
```

### Changing the data format or bit depth

```
img = ImageBuf ("input.exr")
# presume that it's a "half" OpenEXR file
# write it back out as a "float" file:
write_image (img, "output.exr", oio.FLOAT)
```

### Changing the compression

The following command converts writes a TIFF file, specifically using LZW compression:

```
img = ImageBuf ("in.tif")
img.specmod().attribute ("compression", "lzw")
write_image (img, "compressed.tif")
```

The following command writes its results as a JPEG file at a compression quality of 50 (pretty severe compression):

```
img = ImageBuf ("big.jpg")
img.specmod().attribute ("quality", 50)
write_image (img, "small.jpg")
```

### Converting between scanline and tiled images

```
img = ImageBuf ("scan.tif")
img.set_write_tiles (16, 16)
write_image (img, "tile.tif")

img = ImageBuf ("tile.tif")
img.set_write_tiles (0, 0)
write_image (img, "scan.tif")
```

### Adding captions or metadata

```
img = ImageBuf ("foo.jpg")
# Add a caption:
img.specmod().attribute ("ImageDescription", "Hawaii vacation")
# Add keywords:
img.specmod().attribute ("keywords", "volcano,lava")
write_image (img, "foo.jpg")
```

### Changing image boundaries

Change the origin of the pixel data window:

```
img = ImageBuf ("in.exr")
img.set_origin (256, 80)
write_image (img, "offset.exr")
```

Change the display window:

```
img = ImageBuf ("in.exr")
img.set_full (16, 1040, 16, 784)
write_image (img, "out.exr")
```

Change the display window to match the data window:

```
img = ImageBuf ("in.exr")
img.set_full (img.roi())
write_image (img, "out.exr")
```

Cut (trim and extract) a 128x128 region whose upper left corner is at location (900,300), moving the result to the origin (0,0) of the image plane and setting the display window to the new pixel data window:

```
img = ImageBuf ("in.exr")
b = ImageBufAlgo.cut (img, oiio.ROI(900,1028,300,428))
write_image (b, "out.exr")
```

**Extract just the named channels from a complicated many-channel image, and add an alpha channel that is 1 everywhere**

```
img = ImageBuf ("allmyaovs.exr")
b = ImageBufAlgo.channels (img, ("spec.R", "spec.G", "spec.B", 1.0))
write_image (b, "spec.tif")
```

**Fade 30% of the way between two images**

```
a = ImageBufAlgo.mul (ImageBuf("A.exr"), 0.7)
b = ImageBufAlgo.mul (ImageBuf("B.exr"), 0.3)
fade = ImageBufAlgo.add (a, b)
write_image (fade, "fade.exr")
```

**Composite of small foreground over background, with offset**

```
fg = ImageBuf ("fg.exr")
fg.set_origin (512, 89)
bg = ImageBuf ("bg.exr")
comp = ImageBufAlgo.over (fg, bg)
write_image (comp, "composite.exr")
```

**Write multiple ImageBufs into one multi-subimage file**

```
bufs = (...)    # Suppose that bufs is a tuple of ImageBuf
specs = (...)    # specs is a tuple of the specs that describe them

# Open with intent to write the subimages
out = ImageOutput.create ("multipart.exr")
out.open ("multipart.exr", specs)
for s in range(len(bufs)) :
    if s > 0 :
        out.open ("multipart.exr", specs[s], "AppendSubimage")
    bufs[s].write (out)
out.close ()
```



## OIIOTOOL: THE OIIO 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 result image onto the stack.

#### 12.1.1 Argument order matters!

**oiiotool** processes operations *in order*. Thus, the order of operations on the command line is extremely important. For example,

```
oiiotool in.tif -resize 640x480 -o out.tif
```

has the effect of reading `in.tif` (thus making it the *current image*), resizing it (taking the original off the stack, and placing the resized result back on the stack), and then writing the new current image to the file `out.tif`. Contrast that with the following subtly-incorrect command:

```
oiiotool in.tif -o out.tif -resize 640x480
```

has the effect of reading `in.tif` (thus making it the *current image*), saving the current image to the file `out.tif` (note that it will be an exact copy of `in.tif`), resizing the current image, and then... exiting. Thus, the resized image is never saved, and `out.tif` will be an unaltered copy of `in.tif`.

### 12.1.2 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. Multiple optional modifiers can be chained together, with colon separators. As an example:

```

oiiotool in.tif --text:x=400:y=600:color=1,0,0 "Hello" -o out.tif
      \_____/ \_____/ \_____/ \_____/ \_____/
      |         |         |         |         |
command -----+         |         |         |         +----- required argument
                  |         |         |
optional modifiers -----+-----+-----+
(separated by ':')
```

### 12.1.3 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
```

Two special command line arguments can be used to disable numeric wildcard expansion: `--wildcardoff` disables numeric wildcard expansion for subsequent command line arguments, until `--wildcardon` re-enables it for subsequent command line arguments. Turning wildcard expansion off for selected arguments can be helpful if you have arguments that must contain the wildcard characters themselves. For example:

```
oiiotool input.@@@.tif --wildcardoff --sattrib Caption "lg@openimageio.org" \
--wildcardon -o output.@@@.tif
```

In this example, the `@` characters in the filenames should be expanded into numeric file sequence wildcards, but the `@` in the caption (denoting an email address) should not.

### 12.1.4 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`.

### 12.1.5 Expression evaluation and substitution

**oiiotool** can perform *expression evaluation and substitution* on command-line arguments. As command-line arguments are needed, they are scanned for containing braces `{ }`. If found, the braces and any text they enclose will be evaluated as an expression and replaced by its result. The contents of an expression may be any of:

- *number*

A numerical value (e.g., 1 or 3.14159).

- *imagename.metadata*

The named metadata of an image.

The *imagename* may be one of: `TOP` (the top or current image), `IMG[i]` describing the *i*-th image on the stack (thus `TOP` is a synonym for `IMG[0]`, the next image on the stack is `IMG[1]`, etc.), or `IMG[name]` to denote an image named by filename or by label name. Remember that the positions on the stack (including `TOP`) refer to *at that moment*, with successive commands changing the contents of the top image.

The *metadata* may be any of:

- the name of any standard metadata of the specified image (e.g., ImageDescription, or width)
- filename : the name of the file (e.g., foo.tif)
- file\_extension : the extension of the file (e.g., tif)
- geom : the pixel data size in the form 640x480+0+0)
- full\_geom : the “full” or “display” size)
- MINCOLOR : the minimum value in each channel(channels are comma-separated)
- MAXCOLOR : the maximum value in each channel(channels are comma-separated)
- AVGCOLOR : the average pixel value of the image (channels are comma-separated)

- *imagename.'metadata'*

If the metadata name is not a “C identifier” (initial letter followed by any number of letter, number, or underscore), it is permissible to use single or double quotes to enclose the metadata name. For example, suppose you want to retrieve metadata named “foo/bar”, you could say

```
{TOP.'foo/bar'}
```

Without the quotes, it might try to retrieve TOP.foo (which doesn’t exist) and divide it by bar.

- Arithmetic

Sub-expressions may be joined by +, -, \*, /, //, and % for arithmetic operations. (Note that like in Python 3, / is floating point division, while // signifies integer division.) Parentheses are supported, and standard operator precedence applies.

- Special variables

- FRAME\_NUMBER : the number of the frame in this iteration of wildcard expansion.
- FRAME\_NUMBER\_PAD : like FRAME\_NUMBER, but 0-padded based on the value set on the command line by --framepadding.

To illustrate how this works, consider the following command, which trims a four-pixel border from all sides and outputs a new image prefixed with “**cropped\_**”, without needing to know the resolution or filename of the original image:

```
oiiotool input.exr -cut "{TOP.width-2*4}x{TOP.height-2*4}+{TOP.x+4}+{TOP.y+4}" \
-o cropped_{TOP.filename}
```

If you should come across filenames that contain braces (these are very rare, but have been known to happen), you temporarily disable expression evaluation with the --evaloff end --evalon flags. For example:

```
$ oiiotool --info "{weird}.exr"
> oiiotool ERROR: expression : syntax error at char 1 of `weird'

$ oiiotool --info --evaloff "{weird}.exr"
> weird.exr : 2048 x 1536, 3 channel, half openexr
```

## 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.

### 12.2.1 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
```

or:

```
oiiotool --info:verbose=1 *.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.

### 12.2.2 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).”

### 12.2.3 Comparing two images

To print a report of the differences between two images of the same resolution:

```
oiiotool old.tif new.tif --diff
```

If you also want to save an image showing just the differences:

```
oiiotool old.tif new.tif --diff --absdiff -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`.

Sometimes you want to compare images but allow a certain number of small difference, say allowing the comparison to pass as long as no more than 1% of pixels differs by more than 1/255, and as long as no single pixel differs by more than 2/255. You can do this with thresholds:

```
oiiotool old.tif new.tif --fail 0.004 -failpercent 1 --hardfail 0.008 --diff
```

### 12.2.4 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:

```
oiiotool 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:

```
oiiotool rgbaz.tif -d half -d Z=float -o rgbaz2.exr
```

### 12.2.5 Changing the compression

The following command converts writes a TIFF file, specifically using LZW compression:

```
oiiotool 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), illustrating how some compression methods allow a quality metric to be optionally appended to the name:

```
iconvert --compression jpeg:50 50 big.jpg small.jpg
```

## 12.2.6 Converting between scanline and tiled images

Convert a scanline file to a tiled file with 16x16 tiles:

```
oiiotool s.tif --tile 16 16 -o t.tif
```

Convert a tiled file to scanline:

```
oiiotool t.tif --scanline -o s.tif
```

## 12.2.7 Adding captions or metadata

Add a caption to the metadata:

```
oiiotool foo.jpg --caption "Hawaii vacation" -o bar.jpg
```

Add keywords to the metadata:

```
oiiotool foo.jpg --keyword "volcano,lava" -o bar.jpg
```

Add other arbitrary metadata:

```
oiiotool in.exr --attrib "FStop" 22.0 \
    --attrib "IPTC:City" "Berkeley" -o out.exr

oiiotool in.exr --attrib:type=timecode smpte:TimeCode "11:34:04:00" \
    -o out.exr

oiiotool in.exr --attrib:type=int[4] FaceBBox "140,300,219,460" \
    -o out.exr
```

## 12.2.8 Changing image boundaries

Change the origin of the pixel data window:

```
oiiotool in.exr --origin +256+80 -o offset.exr
```

Change the display window:

```
oiiotool in.exr --fullsize 1024x768+16+16 -o out.exr
```

Change the display window to match the data window:

```
oiiotool in.exr --fullpixels -o out.exr
```

Crop (trim) an image to a 128x128 region whose upper left corner is at location (900,300), leaving the remaining pixels in their original positions on the image plane (i.e., the resulting image will have origin at 900,300), and retaining its original display window:

```
oiiotool in.exr --crop 128x128+900+300 -o out.exr
```

Cut (trim and extract) a 128x128 region whose upper left corner is at location (900,300), moving the result to the origin (0,0) of the image plane and setting the display window to the new pixel data window:

```
oiiotool in.exr --cut 128x128+900+300 -o out.exr
```

## 12.2.9 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 --mulc 0.9,0.9,0.9,1.0 -o out.exr
```

## 12.2.10 Remove gamma-correction from an image

Convert a gamma-corrected image (with gamma = 2.2) to linear values:

```
oiiotool corrected.exr --powc 2.2,2.2,2.2,1.0 -o linear.exr
```

## 12.2.11 Resize an image

Resize to a specific resolution:

```
oiiotool original.tif --resize 1024x768 -o specific.tif
```

Resize both dimensions by a known scale factor:

```
oiiotool original.tif --resize 200% -o big.tif  
oiiotool original.tif --resize 25% -o small.tif
```

Resize each dimension, independently, by known scale factors:

```
oiiotool original.tif --resize 300%x200% -o big.tif  
oiiotool original.tif --resize 100%x25% -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
```

## 12.2.12 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 lgl0 lnf -o out.exr
```

### 12.2.13 Grayscale and RGB

Turn a single channel image into a 3-channel gray RGB:

```
oiiotool gray.tif --ch 0,0,0 -o rgb.tif
```

Convert a color image to luminance grayscale:

```
oiiotool RGB.tif --chsum:weight=.2126,.7152,.0722 -o luma.tif
```

### 12.2.14 Channel reordering and padding

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 R=0,G=0,B -o justblue.tif
```

Swap the red and blue channels from an RGBA image:

```
oiiotool rgba.tif --ch R=B,G,B=R,A -o bgra.tif
```

Extract just the named channels from a many-channel image, as efficiently as possible (avoiding memory and I/O for the unused channels):

```
oiiotool -i:ch=R,G,B manychannels.exr -o rgb.exr
```

Add an alpha channel to an RGB image, setting it to 1.0 everywhere, and naming it "A" so it will be recognized as an alpha channel:

```
oiiotool rgb.tif --ch R,G,B,A=1.0 -o rgba.tif
```

Add an alpha channel to an RGB image, setting it to be the same as the R channel and naming it "A" so it will be recognized as an alpha channel:

```
oiiotool rgb.tif --ch R,G,B,A=R -o rgba.tif
```

Add a z channel to an RGBA image, setting it to 3.0 everywhere, and naming it "Z" so it will be recognized as a depth channel:

```
oiiotool rgba.exr --ch R,G,B,A,Z=3.0 -o rgbaz.exr
```

### 12.2.15 Fade between two images

Fade 30% of the way from A to B:

```
oiiotool A.exr --mulc 0.7 B.exr --mulc 0.3 --add -o fade.exr
```

### 12.2.16 Simple compositing

Simple “over” composite of aligned foreground and background:

```
oiiotool fg.exr bg.exr --over -o composite.exr
```

Composite of small foreground over background, with offset:

```
oiiotool fg.exr --origin +512+89 bg.exr --over -o composite.exr
```

### 12.2.17 Creating an animated GIF from still images

Combine several separate JPEG images into an animated GIF with a frame rate of 8 frames per second:

```
oiiotool foo??.jpg --siappendall --attrib FramesPerSecond 10.0 -o anim.gif
```

### 12.2.18 Frame sequences: 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
```

### 12.2.19 Expression example: annotate the image with its caption

This command reads a file, and draws any text in the “ImageDescription” metadata, 30 pixels from the bottom of the image:

```
oiiotool input.exr --text:x=30:y={TOP.height-30} {TOP.ImageDescription} -o out.exr
```

Note that this works without needing to know the caption ahead of time, and will always put the text 30 pixels from the bottom of the image without requiring you to know the resolution.



### 12.2.20 Contrast enhancement: stretch pixel value range to exactly fit [0-1]

This command reads a file, subtracts the minimum pixel value and then divides by the (new) maximum value, per channel, thus expanding its pixel values to the full [0-1] range:

```
oiiotool input.tif -subc {TOP.MINCOLOR} -divc {TOP.MAXCOLOR} -o out.tif
```

Note that this is a naive way to improve contrast and because each channel is handled independently, it may result in color hue shifts.

### 12.2.21 Split a multi-image file into separate files

Take a multi-image TIFF file, split into its constituent subimages and output each one to a different file, with names sub0001.tif, sub0002.tif, etc.:

```
oiiotool multi.tif -sisplit -o:all=1 sub%04d.tif
```

## 12.3 oiiotool commands: general and image information

### --help

Prints full usage information to the terminal, as well as information about image formats supported, known color spaces, filters, OIIO build options and library dependencies.

### -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).

### -n

No saved output — do not save any image files. This is helpful for certain kinds of tests, or in combination with `--runstats` or `--debug`, for getting detailed information about what a command sequence will do and what it costs, but without producing any saved output files.

### --debug

Debug mode — print lots of information about what operations are being performed.

### --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.

Optional appended modifiers include:

- `format=name`: The format name may be one of: `text` (default) for readable text, or `xml` for an XML description of the image metadata.

- `verbose=1` : If nonzero, the information will contain all metadata, not just the minimal amount.

**--echo** <message>

Prints the message to the console, at that point in the left-to-right execution of command line arguments. The message may contain expressions for substitution.

Optional appended modifiers include:

- `newline=n` : The number of newlines to print after the message (default is 1, but 0 will suppress the newline, and a larger number will make more vertical space).

Examples:

```
oiiotool test.tif --resize 256x0 --echo "result is {TOP.width}x{TOP.height}"
```

This will resize the input to be 256 pixels wide and automatically size it vertically to preserve the original aspect ratio, and then print a message to the console revealing the resolution of the resulting image.

**--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 modifiers include:

- `empty=verbose` : If 0, will cause deep images to skip printing of information about pixels with no samples.

**--diff****--fail** <A> **--failpercent** <B> **--hardfail** <C>**--warn** <A> **--warnpercent** <B> **--hardwarn** <C>

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.

The `--fail`, `--failpercent`, and `--hardfail` options set thresholds 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.

The `--warn`, `--warnpercent`, and `hardwarn` options set thresholds 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.

**--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 modifiers 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:

```
oiiotool 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 (100–150×4 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,

```
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`.

**--views** <name1,name2,...>

Supplies a comma-separated list of view names (substituted for %V and %v). If not supplied, the view list will be `left`, `right`.

**--wildcardoff**, **--wildcardon**

Turns off (or on) numeric wildcard expansion for subsequent command line arguments. This can be useful in selectively disabling numeric wildcard expansion for a subset of the command line.

**--evaloff**, **--evalon**

Turns off (or on) expression evaluation (things with { }) for subsequent command line arguments. This can be useful in selectively disabling expression evaluation expansion for a subset of the command line, for example if you actually have filenames containing curly braces.

## 12.4 oiiotool 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.

### 12.4.1 Reading images

<filename>

**-i** <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.

The **-i** command may be used, which allows additional options that control the reading of just that one file.

Optional appended modifiers include:

**:now= int** If 1, read the image now, before proceeding to the next command.

**:autocc= int** Enable or disable **--autocc** for this input image.

**:info= int** Print info about this file (even if the global **--info** was not used) if nonzero. If the value is 2, print full verbose info (like **--info -v**).

**:infoformat= name** When printing info, the format may be one of: `text` (default) for readable text, or `xml` for an XML description of the image metadata.

**:type= name** Upon reading, convert the pixel data to the named type. This can override the default behavior of internally storing whatever type is the most precise one found in the file.

**:ch= name...** Causes the input to read only the specified channels. This is equivalent to following the input with a **--ch** command, except that by integrating into the **-i**, it potentially can avoid the I/O of the unneeded channels.

**--no-autopremult, --autopremult**

By default, OpenImageIO's format readers convert any "unassociated alpha" (color values that are not "premultiplied" by alpha) to the usual associated/premultiplied convention. If the `--no-autopremult` flag is found, subsequent inputs will not do this premultiplication. It can be turned on again via `--autopremult`.

**--autoorient**

Automatically do the equivalent of `--reorient` on every image as it is read in, if it has a nonstandard orientation. This is generally a good idea to use if you are using `oiiootool` to combine images that may have different orientations.

**--autocc**

Turns on automatic color space conversion: Every input image file will be immediately converted to a scene-referred linear color space, and every file written will be first transformed to an appropriate output color space based on the filename or type. Additionally, if the name of an output file contains a color space and that color space is associated with a particular data format, it will output that data format (akin to `-d`).

The rules for deducing color spaces are as follows, in order of priority:

1. If the filename (input or output) contains as a substring the name of a color space from the current OpenColorIO configuration, that will be assumed to be the color space of input data (or be the requested color space for output).
2. For input files, if the ImageInput set the "oiio:ColorSpace" metadata, it will be honored if the filename did not override it.
3. When outputting to JPEG files, assume that sRGB is the desired output color space (since JPEG requires sRGB), but still this only occurs if the filename does not specify something different.

If the implied color transformation is unknown (for example, involving a color space that is not recognized), a warning will be printed, but it the rest of `oiiootool` processing will proceed (but without having transformed the colors of the image).

Example:

If the input file `in_lg10.dpx` is in the `lg10` color space, and you want to read it in, brighten the RGB uniformly by 10% (in a linear space, of course), and then save it as a 16 bit integer TIFF file encoded in the `vd16` color space, you could specify the conversions explicitly:

```
oiiootool in_lg10.dpx --colorconvert lg10 linear \
--mulc 1.1,1.1,1.1,1.0 -colorconvert linear vd16 \
-d uint16 -o out_vd16.tif
```

or rely on the naming convention matching the OCIO color space names and use automatic conversion:

```
oiiootool --autocc in_lg10.dpx --mulc 1.1 -o out_vd16.tif
```

**--native**

Normally, all images read by `oiiootool` are read into an `ImageBuf` backed by an underlying `ImageCache`, and are automatically converted to `float` pixels for internal storage (because any subsequent image processing is usually much faster and more accurate when done on floating-point values).

This option causes (1) input images to be stored internally in their native pixel data type rather than converted to float, and (2) to bypass the `ImageCache` (reading directly into an `ImageBuf`) if the pixel data type is not one of the types that is supported internally to `ImageCache` (`UINT8`, `uint16`, `half`, and `float`).

images whose pixels are comprised of data types that are not natively representable exactly in the `ImageCache` to bypass the `ImageCache` and be read directly into an `ImageBuf`.

The typical use case for this is when you know you are dealing with unusual pixel data types that might lose precision if converted to `float` (for example, if you have images with `uint32` or `double` pixels). Another

use case is if you are using `oiiotool` merely for file format or data format conversion, with no actual image processing math performed on the pixel values – in that case, you might save time and memory by bypassing the conversion to `float`.

**--cache** <size>

Set the underlying ImageCache size (in MB). See Section *ImageCache API*.

**--autotile** <tilesize>

For the underlying ImageCache, turn on auto-tiling with the given tile size. Setting *tilesize* to 0 turns off auto-tiling (the default is off). If auto-tile is turned on, The ImageCache “autoscanline” feature will also be enabled. See Section *ImageCache API* for details.

**--iconfig** <name> <value>

Sets configuration metadata that will apply to the next input file read.

Optional appended modifiers include:

- `type= typename` : Specify the metadata type.

If the optional `type=` specifier is used, that provides an explicit type for the metadata. If not provided, 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.

Examples:

```
oiiotool --iconfig "oio:UnassociatedAlpha" 1 in.png -o out.tif
```

## 12.4.2 Writing images

**-o** <filename>

Outputs the current image to the named file. This does not remove the current image from the image stack, it merely saves a copy of it.

Optional appended modifiers include:

**:type= name** Set the pixel data type (like `-d`) for this output image (e.g., `:uint8`, `uint16`, `half`, `float`, etc.).

**:bits= int** Set the bits per pixel (if nonstandard for the datatype) for this output image.

**:dither= int** Turn dither on or off for this output. (default: 0)

**:autocc= int** Enable or disable `--autocc` for this output image.

**:autocrop= int** Enable or disable autocrop for this output image.

**:autotrim= int** Enable or disable `--autotrim` for this output image.

**:separate= int, contig= int** Set separate or contiguous planar configuration for this output.

**:fileformatname= string** Specify the desired output file format, overriding any guess based on file name extension.

**:scanline= int** If nonzero, force scanline output.

**:tile= int x int** Force tiling with given size.

**:all= n** Output all images currently on the stack using a pattern. See further explanation below.

The `all=n` option causes *all* images on the image stack to be output, with the filename argument used as a pattern assumed to contain a `%d`, which will be substituted with the index of the image (beginning with *n*). For example, to take a multi-image TIFF and extract all the subimages and save them as separate files:

```
oiiotool multi.tif -sisplit -o:all=1 sub%04d.tif
```

This will output the subimages as separate files `sub0001.tif`, `sub0002.tif`, and so on.

**-otex** <filename>  
**-oenv** <filename>  
**-obump** <filename>

Outputs the current image to the named file, as a MIP-mapped texture or environment map, identical to that which would be output by `maketx` (Chapter *Making Tiled MIP-Map Texture Files With `maketx` or `oiiotool`*). The advantage of using `oiiotool` rather than `maketx` is simply that you can have a complex `oiiotool` command line sequence of image operations, culminating in a direct saving of the results as a texture map, rather than saving to a temporary file and then separately invoking `maketx`.

In addition to all the optional arguments of `-o`, optional appended arguments for `-otex`, `-oenv`, and `-obump` also include:

- :wrap= string** Set the default `$s$` and `$t$` wrap modes of the texture, to one of: `:black`, `clamp`, `periodic`, `mirror`.
- :swrap= string** Set the default `$s$` wrap mode of the texture.
- :twrap= string** Set the default `$t$` wrap mode of the texture.
- :resize= int** If nonzero, resize to a power of 2 before starting to create the MIPmap levels. (default: 0)
- :nomipmap= int** If nonzero, do not create MIP-map levels at all. (default: 0)
- :updatemode= int** If nonzero, do not create and overwrite the existing texture if it appears to already match the source pixels. (default: 0)
- :monochrome\_detect= int** Detect monochrome (R=G=B) images and turn them into 1-channel textures. (default: 0)
- :opaque\_detect= int** Detect opaque (A=1) images and drop the alpha channel from the texture. (default: 0)
- :unpremult= int** Unpremultiply colors before any per-MIP-level color conversions, and re-premultiply after. (default: 0)
- :incolospace= string** Specify color space conversion.
- :outcolospace= string** ...
- :highlightcomp= int** Use highlight compensation for HDR images when resizing for MIP-map levels. (default: 0)
- :sharpen= float** Additional sharpening factor when resizing for MIP-map levels. (default: 0.0)
- :filter= string** Specify the filter for MIP-map level resizing. (default: box)
- :prman\_metadata= int** Turn all all options required to make the resulting texture file compatible with PRMan (particular tile sizes, formats, options, and metadata). (default: 0)
- :prman\_options= int** Include the metadata that PRMan's texture system wants. (default: 0)
- :bumpformat= string** For `-obump` only, specifies the interpretation of 3-channel source images as one of: `height`, `normal`, `auto` (default).

Examples:

```
oiiotool in.tif -otex out.tx

oiiotool in.jpg --colorconvert sRGB linear -d uint16 -otex out.tx

oiiotool --pattern:checker 512x512 3 -d uint8 -otex:wrap=periodic checker.tx

oiiotool in.exr -otex:highlightcomp=1:sharpen=0.5 out.exr
```

**-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>

**--compression** <method:quality>

Sets the compression method, and optionally a quality setting, for the output image. Each ImageOutput plugin will have its own set of methods that it supports.

Sets the compression method, and optionally a quality setting, for the output image. Each ImageOutput plugin will have its own set of methods that it supports.

**--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).

This is considered deprecated, and in general we now recommend just appending the quality metric to the `--compression name:qual`.

**--dither**

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 `oiiotool` 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.

#### **--metamerge**

When this flag is used, most image operations will try to merge the metadata found in all of their source input images into the output. The default (if this is not used) is that image operations with multiple input images will just take metadata from the first source image.

(This was added for OpenImageIO 2.1.)

## 12.5 `oiiotool` commands that change the current image metadata

This section describes `oiiotool` 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>

**--sattrib** <name> <value>

Adds or replaces metadata with the given *name* to have the specified *value*.

Optional appended modifiers include:

- `type=typename` : Specify the metadata type.

If the optional `type=` specifier is used, that provides an explicit type for the metadata. If not provided, 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.

The `--sattrib` command is equivalent to `--attrib:type=string`.

Examples:

```
oiiotool in.jpg --attrib "IPTC:City" "Berkeley" -o out.jpg

oiiotool in.jpg --attrib:type=string "Name" "0" -o out.jpg

oiiotool in.exr --attrib:type=matrix worldtocam \
    "1,0,0,0,0,1,0,0,0,0,1,0,2.3,2.1,0,1" -o out.exr

oiiotool in.exr --attrib:type=timecode smpte:TimeCode "11:34:04:00" \
    -o out.exr
```

**--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.

**--nosoftwareattrib**

When set, this prevents the normal adjustment of "Software" and "ImageHistory" metadata to reflect what **oiiotool** is doing.

**--sansattrib**

When set, this edits the command line inserted in the "Software" and "ImageHistory" metadata to omit any verbose **--attrib** and **--sattrib** commands.

**--eraseattrib** <pattern>

Removes any metadata whose name matches the regular expression *pattern*. The pattern will be case insensitive.

Examples:

```
# Remove one item only
oiiotool in.jpg --eraseattrib "smpte:TimeCode" -o no_timecode.jpg

# Remove all GPS tags
oiiotool in.jpg --eraseattrib "GPS:.*" -o no_gps_metadata.jpg

# Remove all metadata
oiiotool in.exr --eraseattrib ".*" -o no_metadata.exr
```

**--orientation** <orient>

Explicitly sets the image's "Orientation" metadata to a numeric value (see Section [Display hints](#) 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.

**--orientcw**

**--orientccw**

**--orient180**

Adjusts the image's "Orientation" metadata by rotating the suggested viewing orientation 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.

See the **--rotate90**, **--rotate180**, **--rotate270**, and **--reorient** commands for true rotation of the pixels (not just the metadata).

**--origin** <neworigin>

Set the pixel data window origin, essentially translating the existing pixel data window to a different position on the image plane. The new data origin is in the form:

```
[+-]x[+-]y
```

Examples:

```
--origin +20+10      x=20, y=10
--origin +0-40       x=0, y=-40
```

### **--originoffset** <offset>

Alter the data window origin, translating the existing pixel data window by this relative amount. The offset is in the form:

```
[+-]x[+-]y
```

Examples:

```
# Assuming the old origin was +100+20...
--originoffset +20+10      # new x=120, y=30
--originoffset +0-40       # new x=100, y=-20
```

### **--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, extract the specified subimage. The subimage specifier *n* is either an integer giving the index of the subimage to extract (starting with 0), or the *name* of the subimage to extract (comparing to the "oio:subimagenam" metadata).

**--sisplit**

Remove the top image from the stack, split it into its constituent subimages, and push them all onto the stack (first to last).

**--siappend**

Replaces the top two images on the stack with a single new image comprised of the subimages of both images appended together.

**--siappendall**

Replace *all* of the individual images on the stack with a single new image comprised of the subimages of all original images appended together.

**--ch** <channellist>

Replaces the top image with a new image whose channels have been reordered as given by the *channellist*. The *channellist* is a comma-separated list of channel designations, each of which may be

- an integer channel index of the channel to copy,
- the name of a channel to copy,
- *newname* = *oldname*, which copies a named channel and also renames it,
- = *float*, which will set the channel to a constant value, or
- *newname* = *float*, which sets the channel to a constant value as well as names the new channel. Examples include: *R, G, B, R=0.0, G, B, A=1.0, R=B, G, B=R, 4, 5, 6, A*.

Channel numbers outside the valid range of input channels, or unknown names, will be replaced by black channels. 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.

## 12.7 oiiotool 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.

## 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. Optional appended arguments include:

- *type= name* : Create the image in memory with the named data type (default: float).

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
--create:type=uint8 1920x1080 3 # RGB, store internally as uint8
```

### --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. Optional appended arguments include:

- *type= name* : Create the image in memory with the named data type (default: float).

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 *:color=r,g,b,...* arguments giving a numerical value for each channel.
- **checker** : A black and white checkerboard pattern. The optional argument *:width=* sets with width of the checkers (defaulting to 8 pixels).
- **fill** : A constant color or gradient, depending on the optional colors. Argument *:color=r,g,b,...* results in a constant color. Argument *:top=r,g,b,...:bottom=...* results in a top-to-bottom gradient. Argument *:left=r,g,b,...:right=...* results in a left-to-right gradient. Argument *:topleft=r,g,b,...:topright=...:bottomleft=...:bottomright=...* results in a 4-corner bilinear gradient.
- **noise** : Create a noise image, with the option *:type=* specifying the kind of noise: (1) **gaussian** (default) for normal distribution noise with mean and standard deviation given by *:mean=* and *:stddev=*, respectively (defaulting to 0 and 0.1); (2) **uniform** for uniformly-distributed noise over the range of values given by options *:min=* and *:max=* (defaults: 0 and 0.1); (3) **salt** for "salt and pepper" noise where a portion of pixels given by option *portion=* (default: 0.1) is replaced with value given by option *value=* (default: 0). For any of these noise types, the option *seed=* can be used to change the random number seed and *mono=1* can be used to make monochromatic noise (same value in all channels).

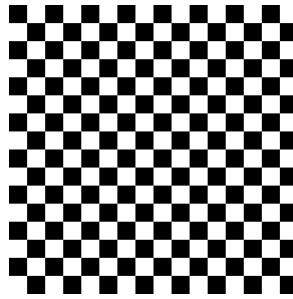
Examples:

A constant 4-channel, 640x480 image with all pixels (0.5, 0.5, 0.1, 1):

```
--pattern constant:color=0.3,0.5,0.1,1.0 640x480 4
```

A 256x256 RGB image with a 16-pixel-wide checker pattern:

```
--pattern checker:width=16:height=16 256x256 3
```



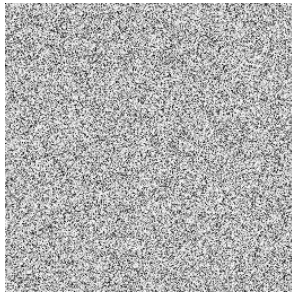
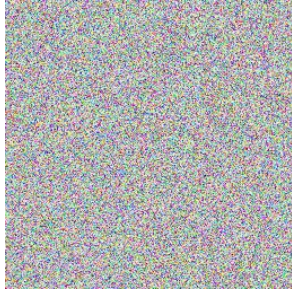
Horizontal, vertical, or 4-corner gradients:

```
--pattern fill:top=0.1,0.1,0.1:bottom=0,0,0.5 640x480 3  
--pattern fill:left=0.1,0.1,0.1:right=0,0.75,0 640x480 3  
--pattern fill:topleft=.1,.1,.1:topright=1,0,0:bottomleft=0,1,  
↪0:bottomright=0,0,1 640x480 3
```



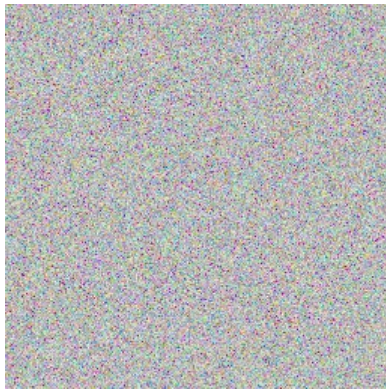
The first example puts uniform noise independently in 3 channels, while the second generates a single greyscale noise and replicates it in all channels.

```
oiiotool --pattern noise:type=uniform:min=1:max=1 256x256 3 -o colornoise.  
↪jpg  
oiiotool --pattern noise:type=uniform:min=0:max=1:mono=1 256x256 3 -o   
↪greynoise.jpg}
```



Generate Gaussian noise with mean 0.5 and standard deviation 0.2 for each channel:

```
oiiotool --pattern noise:type=gaussian:mean=0.5:stddev=0.2 256x256 3 -o   
↪gaussnoise.jpg
```



#### **--kernel** <name> <size>

Create new 1-channel float image big enough to hold the named kernel and size (size is expressed as *width* x *height*, e.g. 5x5). 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, cubic, keys, simon, rifman, disk. There are also catmull-rom and

lanczos3 (and its synonym, nuke-lanczos6), but they are fixed-size kernels that don't scale with the width, and are therefore probably less useful in most cases.

Examples:

```
oiiotool --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 #1
```

## 12.9 oiiotool commands that do image processing

### **--add**

**--addc** <value>

**--addc** <value0,value1,value2...>

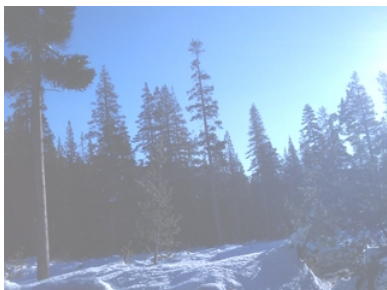
Replace the *two* top images with a new image that is the pixel-by-pixel sum of those images (**--add**), or add a constant color value to all pixels in the top image (**--addc**).

For **--addc**, 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.

Examples:

```
oiiotool imageA.tif imageB.tif --add -o sum.jpg
```

```
oiiotool tahoe.jpg --addc 0.5 -o addc.jpg
```





**--sub**

```
-- subc <value>
```

```
-- subc <value0,value1,value2...>
```

Replace the *two* top images with a new image that is the pixel-by-pixel difference between the first and second images (`--sub`), or subtract a constant color value from all pixels in the top image (`--subc`).

For `--subc`, if a single constant value is given, it will be subtracted from all color channels. Alternatively, a series of comma-separated constant values (with no spaces!) may be used to specify a different value to subtract from each channel in the image.

**--mul**

```
-- mulc <value>
```

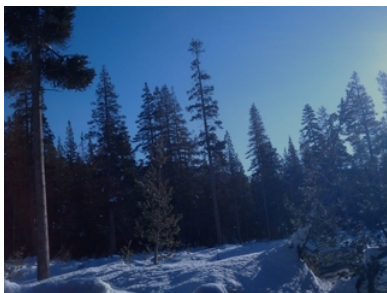
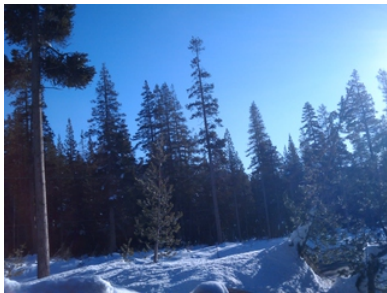
```
-- mulc <value0,value1,value2...>
```

Replace the *two* top images with a new image that is the pixel-by-pixel multiplicative product of those images (`--mul`), or multiply all pixels in the top image by a constant value (`--mulc`).

For `--mulc`, if a single constant value is given, it will be multiplied to all color channels. Alternatively, a series of comma-separated constant values (with no spaces!) may be used to specify a different value to multiply with each channel in the image.

Example:

```
# Scale image brightness to 20% of its original
oiiotool tahoe.jpg --mulc 0.2 -o mulc.jpg
```

**--div**

```
-- divc <value>
```

```
-- divc <value0,value1,value2...>
```

Replace the *two* top images with a new image that is the pixel-by-pixel, channel-by-channel result of the first

image divided by the second image (`--div`), or divide all pixels in the top image by a constant value (`--divc`). Division by zero is defined as resulting in 0.

For `--divc`, if a single constant value is given, all color channels will have their values divided 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.

#### **--mad**

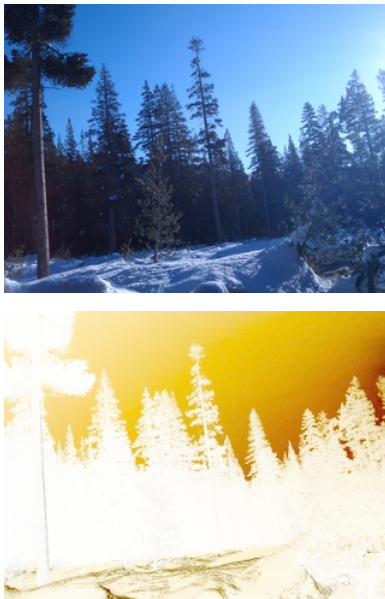
Replace the *three* top images A, B, and C (C being the top of stack, B below it, and A below B), and compute  $A*B+C$ , placing the result on the stack. Note that `A B C --mad` is equivalent to `A B --mul C --add`, though using `--mad` may be somewhat faster and preserve more precision.

#### **--invert**

Replace the top images with its color inverse. It only inverts the first three channels, in order to preserve alpha.

Example:

```
oiiotool tahoe.jpg --inverse -o inverse.jpg
```



#### **--absdiff**

**--absdiffc** <value>

**--absdiffc** <value0,value1,value2...>

Replace the *two* top images with a new image that is the absolute value of the difference between the first and second images (`--absdiff`), or replace the top image by the absolute value of the difference between each pixel and a constant color (`--absdiffc`).

#### **--abs**

Replace the current image with a new image that has each pixel consisting of the *absolute value* of the old pixel value.

**--powc** <value>

**--powc** <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.

**--noise**

Alter the top image to introduce noise, with the option `:type=` specifying the kind of noise: (1) `gaussian` (default) for normal distribution noise with mean and standard deviation given by `:mean=` and `:stddev=`, respectively (defaulting to 0 and 0.1); (2) `uniform` for uniformly-distributed noise over the range of values given by options `:min=` and `:max=` (defaults: 0 and 0.1); (3) `salt` for “salt and pepper” noise where a portion of pixels given by option `portion=` (default: 0.1) is replaced with value given by option `value=` (default: 0).

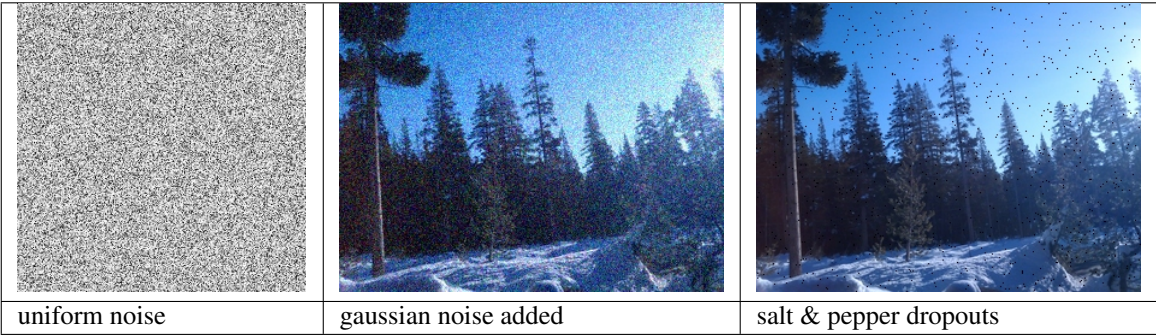
Optional appended modifiers include:

- `:seed= int` Can be used to change the random number seed.
- `:mono=1` Make monochromatic noise (same value in all channels).
- `:nchannels= int` Limit which channels are affected by the noise.

Example:

```
# Add color gaussian noise to an image
oiiotool tahoe.jpg --noise:type=gaussian:stddev=0.1 -o noisy.jpg

# Simulate bad pixels by turning 1% of pixels black, but only in RGB
# channels (leave A alone)
oiiotool tahoe-rgba.tif --noise:type=salt:value=0:portion=0.01:mono=1:nchannels=3
↪ \
-o dropouts.tif
```



**--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
```

**--contrast**

Remap pixel values from [black, white] to [min, max], with an optional smooth sigmoidal contrast stretch as well.

Optional appended modifiers include:

**black= *vals*** Specify black value(s), default 0.0.

**white= *vals*** Specify white value(s), default 1.0.

**min= *vals*** Specify the minimum range value(s), default 0.0.

**max= *vals*** Specify the maximum range value(s), default 1.0.

**scontrast= *vals*** Specify sigmoidal contrast slope value(s),  
default 1.0. **sthresh= *vals***

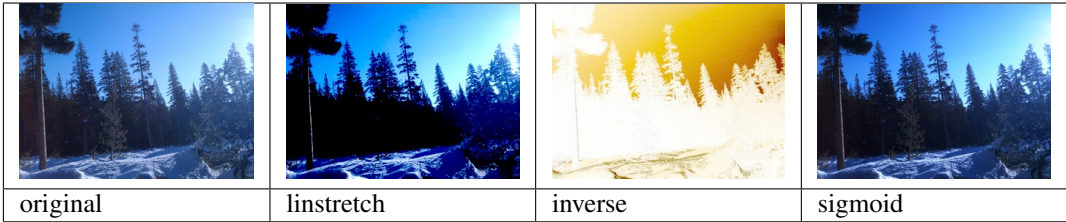
Specify sigmoidal threshold value(s) giving the position of maximum slope, default 0.5.

**clamp= *on*** If *on* is nonzero, will optionally clamp all result channels to [min,max].

Each *vals* may be either a single floating point value for all channels, or a comma-separated list of per-channel values.

Examples:

```
oiiotool tahoe.tif --contrast:black=0.1:white=0.75 -o linstretch.tif
oiiotool tahoe.tif --contrast:black=1.0:white=0.0:clamp=0 -o inverse.tif
oiiotool tahoe.tif --contrast:scontrast=5 -o sigmoid.tif
```



**--colormap** <mapname>  
Creates an RGB color map based on the luminance of the input image. The `mapname` may be one of: “magma”, “inferno”, “plasma”, “viridis”, “turbo”, “blue-red”, “spectrum”, and “heat”. Or, `mapname` may also be a comma-separated list of RGB triples, to form a custom color map curve.

Note that “magma”, “inferno”, “plasma”, “viridis” are perceptually uniform, strictly increasing in luminance, look good when converted to grayscale, and work for people with all types of colorblindness. The “turbo” color map also shares all of these qualities except for being strictly increasing in luminance. These are all desirable qualities that are lacking in the other, older, crappier maps (blue-red, spectrum, and heat). Don’t be fooled by the flashy “spectrum” colors — it is an empirically bad color map compared to the preferred ones.

Example:

```
oiiotool tahoe.jpg --colormap inferno -o inferno.jpg
oiiotool tahoe.jpg --colormap viridis -o viridis.jpg
oiiotool tahoe.jpg --colormap turbo -o turbo.jpg
oiiotool tahoe.jpg --colormap .25,.25,.25,0,.5,0,1,0,0 -o custom.jpg
```



**--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, with foreground pixel (0,0) being pasted to the background at the *location* specified (expressed as `+xpos+ypos`, e.g., `+100+50`, or of course using `-` for negative offsets). Only pixels within the actual data region of the foreground image are pasted in this manner.

Note that if `location` is `+0+0`, the foreground image’s data region will be copied to its same position in the background image (this is useful if you are pasting an image that already knows its correct data window offset).

Optional appended modifiers include:

- `mergeroi=1` : If the value is nonzero, the result image will be sized to be the *union* of the input images (versus being the same data window as the background image). (The `mergeroi` modifier was added in OIIO 2.1.)
- `all=1` : If the value is nonzero, will paste *all* images on the image stack, not just the top two images. This can be useful to paste-merge many images at once, for example, if you have rendered a large image in abutting tiles and wish to re-assemble them into a single image. (The `all` modifier was added in OIIO 2.1.)

Examples:

```
# Result will be the size of bg, but with fg on top and with an
# offset of (100,100).
oiiotool fg.exr bg.exr -paste +100+100 -o out.exr

# Use "merge" mode, so result will be sized to contain both fg
# and bg. Also, paste fg into its natural position given by its
# data window.
oiiotool fg.exr bg.exr -paste:mergeroi=1 +0+0 -o out.exr

# Merge many non-overlapping "tiles" into one combined image
oiiotool img*.exr -paste:mergeroi=1:all=1 +0+0 -o combined.exr
```

**--mosaic <size>**

Removes  $w \times h$  images, dictated by the *size*, and turns them into a single image mosaic. Optional appended modifiers include:

- `pad= num` : Select the number of pixels of black padding to add between images (default: 0).

Examples:

```
oiiotool left.tif right.tif --mosaic:pad=16 2x1 -o out.tif

oiiotool 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 modifiers include:

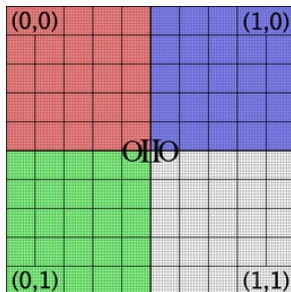
**zeroisinf= num** If nonzero, indicates that  $z=0$  pixels should be treated as if they were infinitely far away. (The default is 0, meaning that “zero means zero.”).

**--rotate90**

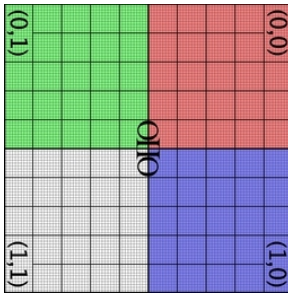
Replace the current image with a new image that is rotated 90 degrees clockwise.

Example:

```
oiiotool grid.jpg --rotate90 -o rotate90.jpg
```



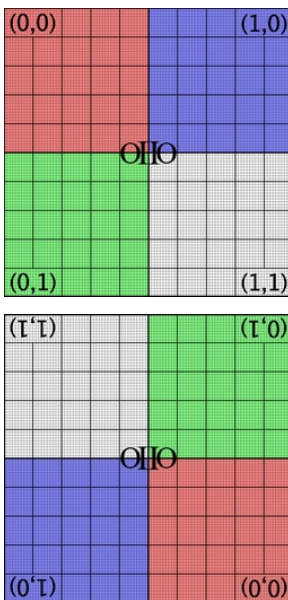


**--rotate180**

Replace the current image with a new image that is rotated by 180 degrees.

Example:

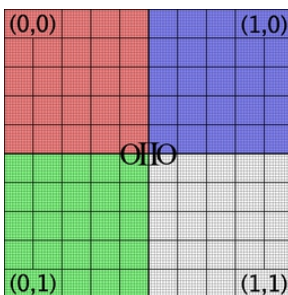
```
oiiotool grid.jpg --rotate180 -o rotate180.jpg
```

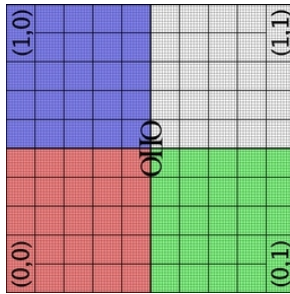
**--rotate270**

Replace the current image with a new image that is rotated 270 degrees clockwise (or 90 degrees counter-clockwise).

Example:

```
oiiotool grid.jpg --rotate270 -o rotate270.jpg
```

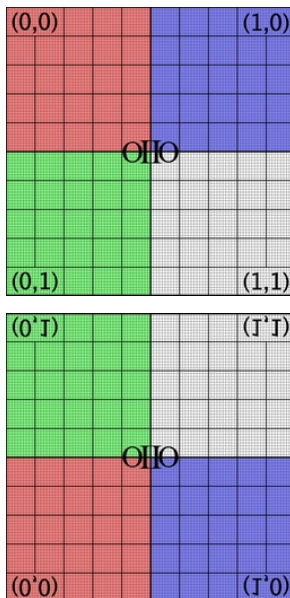


**--flip**

Replace the current image with a new image that is flipped vertically, with the top scanline becoming the bottom, and vice versa.

Example:

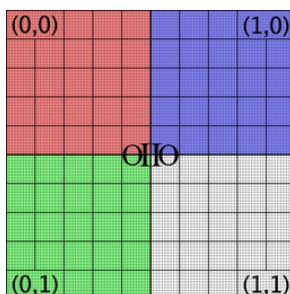
```
oiiotool grid.jpg --flip -o flip.jpg
```

**--flop**

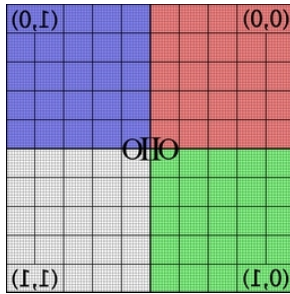
Replace the current image with a new image that is flopped horizontally, with the leftmost column becoming the rightmost, and vice versa.

Example:

```
oiiotool grid.jpg --flop -o flop.jpg
```





**--reorient**

Replace the current image with a new image that is rotated and/or flipped as necessary to move the pixels to match the Orientation metadata that describes the desired display orientation.

Example:

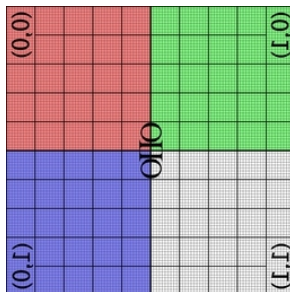
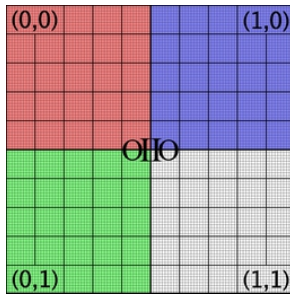
```
oiiotool tahoe.jpg --reorient -o oriented.jpg
```

**--transpose**

Replace the current image with a new image that is reflected about the x-y axis (x and y coordinates and sizes are swapped).

Example:

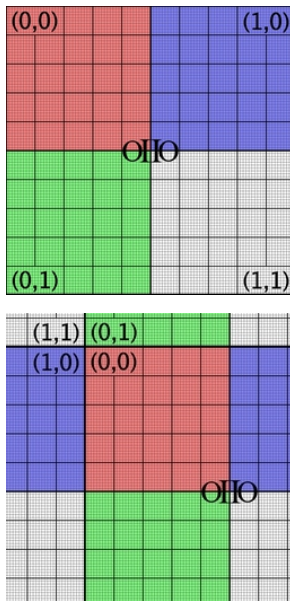
```
oiiotool grid.jpg --transpose -o transpose.jpg
```

**--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.

Example:

```
oiiotool grid.jpg --cshift +70+30 -o cshift.jpg
```

**--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

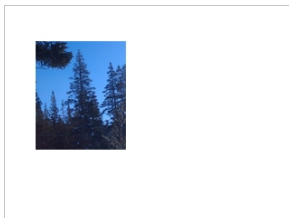
*xmin,ymin,xmax,ymax*

Note that `crop` does not *reposition* pixels, it only trims or pads to reset the image's pixel data window to the specified region.

If `oiio:tool`'s global `-a` flag is used (**all** subimages), or if the optional `--crop:allsubimages=1` is employed, the crop will be applied identically to all subimages.

Examples:

```
# Both of these crop to a 100x120 region that begins at x=35,y=40
oiio:tool tahoe.exr --crop 100x120+35+40 -o crop.exr
oiio:tool tahoe.exr --crop 35,40,134,159 -o crop.exr
```



**--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.

**--trim**

Replace the current image with a new image that is cropped to contain the minimal rectangular ROI that contains all of the nonzero-valued pixels of the original image.

Examples:

```
oiiotool greenrect.exr -trim -o trimmed.jpg

.. image:: figures/pretrim.jpg
   :width: 1.5 in
.. image:: figures/trim.jpg
   :width: 1.5 in
```

**--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

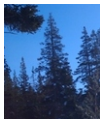
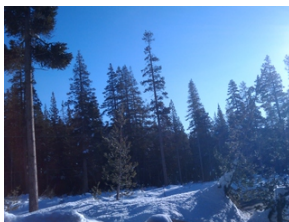
*width x height [+/-] xoffset [+/-] yoffset*

or

*xmin,ymin,xmax,ymax*

Examples:

```
# Both of these crop to a 100x120 region that begins at x=35,y=40
oiiotool tahoe.exr --cut 100x120+35+40 -o cut.exr
oiiotool tahoe.exr --cut 35,40,134,159 -o cut.exr
```

**--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, either by simple bilinear interpolation or by just copying the “closest” pixel. The size is in the form of any of these:

*width x height*

*width x height [+/-] xoffset [+/-] yoffset*

*xmin,ymin,xmax,ymax*

*wscale% x hscale%*

if width or height is 0, that dimension will be automatically computed so as to preserve the original aspect ratio.

Optional appended modifiers include:

**interp= bool** If set to zero, it will just copy the “closest” pixel; if nonzero, bilinear interpolation of the surrounding 4 pixels will be used.

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 whose display (full) size is the given pixel data resolution and offset. The *size* is in the form

*width x height*

*width x height [+/-] xoffset [+/-] yoffset*

*xmin,ymin,xmax,ymax*

*wscale% x hscale%*

if width or height is 0, that dimension will be automatically computed so as to preserve the original aspect ratio.

Optional appended modifiers 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*

*width x height [+/-] xoffset [+/-] yoffset*

Optional appended modifiers include:

- **filter= name** : Filter name. The default is blackman-harris when increasing resolution, lanczos3 when decreasing resolution.

- `pad= p` : 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.
- `exact= e` : If the argument is nonzero, will result in an exact match on aspect ratio and centering (partial pixel shift if necessary), whereas the default (0) will only preserve aspect ratio and centering to the precision of a whole pixel.
- `wrap= w` : For “exact” aspect ratio fitting, this determines the wrap mode used for the resizing kernel (default: `black`, other choices include `clamp`, `periodic`, `mirror`).

Examples:

```
oiiotool in.exr --fit:pad=1:exact=1 640x480 -o out.exr
oiiotool in.exr --fit 1024x1024 -o out.exr
```

### **--pixelaspect** <aspect>

Replace the current image with a new image that scales up the width or height in order to match the requested pixel aspect ratio. If displayed in a manner that honors the `PixelAspectRatio`, it should look the same, but it will have different pixel dimensions than the original. It will always be the same or higher resolution, so it does not lose any detail present in the original.

As an example, if you have a 512x512 image with pixel aspect ratio 1.0, `--pixelaspect 2.0` will result in a 512x1024 image that has “`PixelAspectRatio`” metadata set to 2.0.

Optional appended modifiers include:

- `filter= name` : Filter name. The default is `lanczos3`.

Examples:

```
oiiotool mandrill.tif --pixelaspect 2.0 -o widepixels.tif
```

### **--rotate** <angle>

Replace the current image with a new image that is rotated by the given angle (in degrees). Positive angles mean to rotate counter-clockwise, negative angles mean clockwise. By default, the center of rotation is at the exact center of the display window (a.k.a. “full” image), but can be explicitly set with the optional `center=x,y` option.

Optional appended modifiers include:

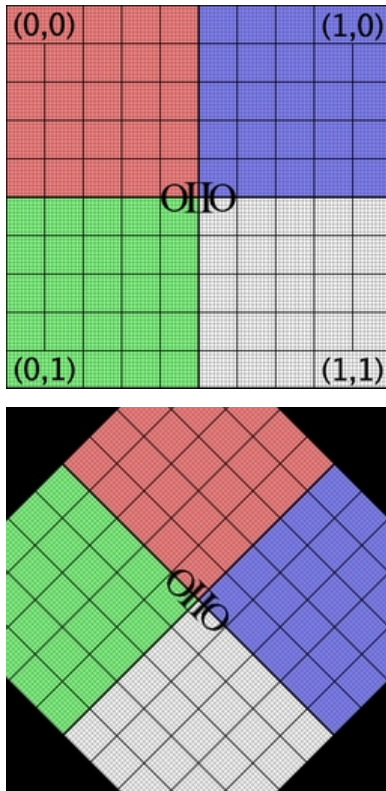
**center= *x,y*** Alternate center of rotation.

**filter= *name*** Filter name. The default is `lanczos3`.

**recompute\_roi= *val*** If nonzero, recompute the pixel data window to exactly hold the transformed image (default=0).

Examples:

```
oiiotool mandrill.tif --rotate 45 -o rotated.tif
oiiotool mandrill.tif --rotate:center=80,91.5:filter=lanczos3 45 -o rotated.tif
```

**--warp** <M33>

Replace the current image with a new image that is warped by the given 3x3 matrix (presented as a comma-separated list of values, without any spaces).

Optional appended modifiers include:

**filter=** *name* Filter name. The default is lanczos3.

**recompute\_roi=** *val* If nonzero, recompute the pixel data window to exactly hold the transformed image (default=0).

Examples:

```
oiiotool mandrill.tif --warp "0.707,0.707,0,-0.707,0.707,0,128,-53.02,1" -o
↳warped.tif
```

**--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 *width* x *height*. (The sizes may be floating point numbers.)

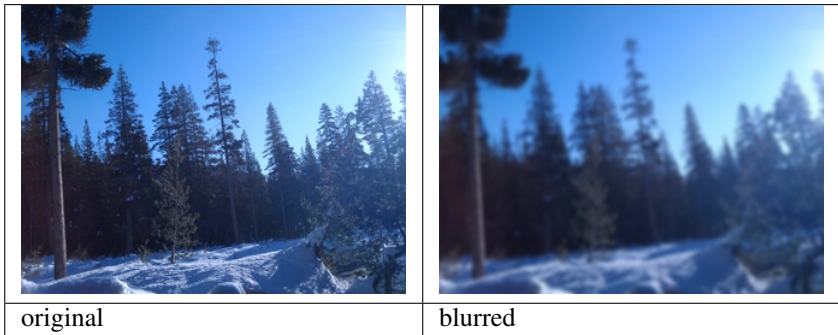
Optional appended modifiers include:



**kernel=** *name* Kernel name. The default is gaussian.

Examples:

```
oiiotool image.jpg --blur 5x5 -o blurred.jpg
oiiotool image.jpg --blur:kernel=bspline 7x7 -o blurred.jpg
```



**--median** <size>

Perform a median filter on the top image with a window of the given size expressed as *width x height*. (The sizes are integers.) This helps to eliminate noise and other unwanted high-frequency detail, but without blurring long edges the way a `--blur` command would.

Examples:

```
oiiotool noisy.jpg --median 3x3 -o smoothed.jpg
```



**--dilate** <size>

**--erode** <size>

Perform dilation or erosion on the top image with a window of the given size expressed as *width x height*. (The sizes are integers.) Dilation takes the maximum of pixel values inside the window, and makes bright features wider and more prominent, dark features thinner, and removes small isolated dark spots. Erosion takes the minimum of pixel values inside the window, and makes dark features wider, bright features thinner, and removes small isolated bright spots.

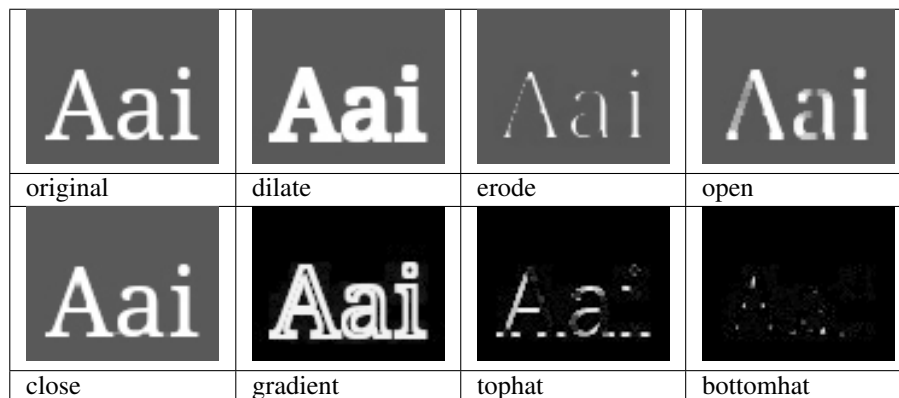
Examples:

```
oiiotool orig.tif --dilate 3x3 -o dilate.tif
oiiotool orig.tif --erode 3x3 -o erode.tif
oiiotool orig.tif --erode 3x3 --dilate 3x3 -o open.tif
oiiotool orig.tif --dilate 3x3 --erode 3x3 -o close.tif
```

(continues on next page)

(continued from previous page)

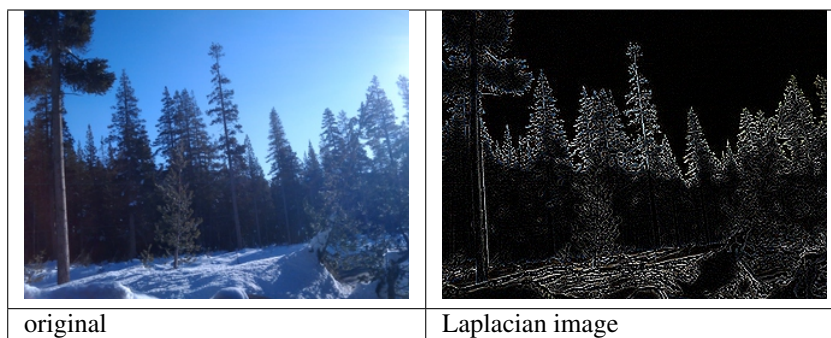
```
oiiotool orig.tif --dilate 3x3 --erode 3x3 -sub -o gradient.tif
oiiotool orig.tif open.tif -o tophat.tif
oiiotool close.tif orig.tif -o bottomhat.tif
```

**--laplacian**

Calculates the Laplacian of the top image.

Examples:

```
oiiotool tahoe.jpg --laplacian tahoe-laplacian.exr
```

**--unsharp**

Unblur the top image using an “unsharp mask.”

Optional appended modifiers include:

**kernel= *name*** Name of the blur kernel (default: *gaussian*). If the kernel name is *median*, the unsharp mask algorithm will use a median filter rather than a blurring filter in order to compute the low-frequency image.

**width= *w*** Width of the blur kernel (default: 3).

**contrast= *c*** Contrast scale (default: 1.0)

**threshold= *t*** 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

oiiotool image.jpg --unsharp:kernel=median -o sharper.jpg
# Note: median filter helps emphasize compact high-frequency details
# without over-sharpening long edges as the default unsharp filter
# sometimes does.

```

**--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
oiiotool image.jpg --ch 2 --fft -o fft.exr

# Reconstruct from the FFT
oiiotool fft.exr --ifft -o reconstructed.exr

# Output the power spectrum: real^2 + imag^2
oiiotool fft.exr --dup --mul --chsum -o powerspectrum.exr

```

**--polar****--unpolar**

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:

```

oiiotool complex.exr --polar -o polar.exr
oiiotool 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 3x3 region surrounding the pixel. If *strategy* is `error`, nonfinite values will be left alone, but it will result in an error that will terminate **oiiotool**.

**--clamp**

Replace the top image with a copy in which pixel values have been clamped. Optional arguments include:

Optional appended modifiers include:

- `min= val` : Specify a minimum value for all channels.
- `min= val0, val1, ...` : Specify minimum value for each channel individually.
- `max= val` : Specify a maximum value for all channels.

- `max= val0,val1,...` : Specify maximum value for each channel individually.
- `clampalpha= val` : If `val` 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:

- `--clamp:min=0` : Clamp all channels to a minimum of 0 (all negative values are changed to 0).
- `--clamp:min=0:max=1` : Clamp all channels to [0,1].
- `--clamp:clampalpha=1` : Clamp the designated alpha channel to [0,1].
- `--clamp:min=,,0:max=,,3.0` : 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 modifiers include:

**luma= val** `val` 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.

### **--line** <x1,y1,x2,y2,...>

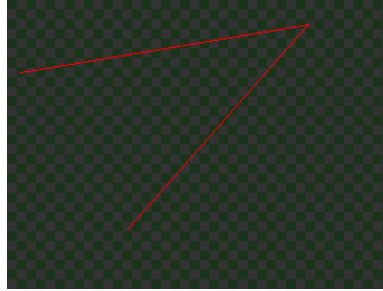
Draw (rasterize) an open polyline connecting the list of pixel positions, as a comma-separated list of alternating *x* and *y* values. Additional optional arguments include:

**color= r,g,b,...** specify the color of the line

The default color, if not supplied, is opaque white.

Examples:

```
oiiotool checker.exr --line:color=1,0,0 10,60,250,20,100,190 -o out.exr
```



**--box** <x1, y1, x2, y2>

Draw (rasterize) a filled or unfilled a box with opposite corners (x1, y1) and (x2, y2). Additional optional arguments include:

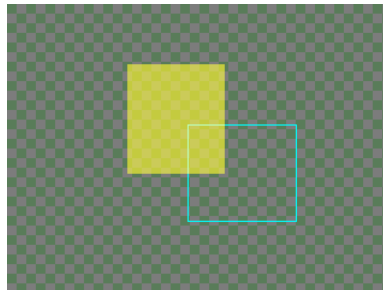
**color**=*r,g,b,...* specify the color of the lines

**fill**=*bool* if nonzero, fill in the box

The default color, if not supplied, is opaque white.

Examples:

```
oiiotool checker.exr --box:color=0,1,1,1 150,100,240,180 \
--box:color=0.5,0.5,0,0.5:fill=1 100,50,180,140 -o out.exr
```



**--fill** <size>

Alter the top image by filling the ROI specified by *size*. The fill can be a constant color, vertical gradient, horizontal gradient, or four-corner gradient.

Optional modifiers for constant color:

**color**=*r,g,b,...* the color of the constant

Optional modifiers for vertical gradient:

**top**=*r,g,b,...* the color for the top edge of the region

**bottom**=*r,g,b,...* the color for the bottom edge of the region

Optional modifiers for horizontal gradient:

**left**=*r,g,b,...* the color for the left edge of the region

**right**=*r,g,b,...* the color for the right edge of the region

Optional modifiers for 4-corner gradient:

**topleft**=*r,g,b,...* the color for the top left corner of the region

**topright**=*r,g,b,...* the color for the top right corner of the region

**bottomleft**=*r,g,b,...* the color for the bottom left corner of the region

**bottomright**=*r,g,b,...* the color for the bottom right corner of the region

Examples:

```
# make a grey-to-blue vertical gradient
oiiotool --create 640x480 3 \
  --fill:top=0.1,0.1,0.1:bottom=0,0,0.5 640x480 -o gradient.tif

# make a grey-to-green horizontal gradient
oiiotool --create 640x480 3 \
  --fill:left=0.1,0.1,0.1:right=0,0.75,0 640x480 -o gradient.tif

# four-corner interpolated gradient
oiiotool --create 640x480 3 \
  --fill:topleft=.1,.1,.1:topright=1,0,0:bottomleft=0,1,0:bottomright=0,0,1 \
  640x480 -o gradient.tif
```

**--text** <words>

Draw (rasterize) text overtop of the current image.

Optional appended modifiers include:

**x=***xpos* *x* position (in pixel coordinates) of the text

**y=***ypos* *y* position (in pixel coordinates) of the text

**size=***size* font size (height, in pixels)

**font=***name* font name, full path to the font file on disk (use double quotes "name" if the path name includes spaces)

**color=***r,g,b,...* specify the color of the text

**xalign=***val* controls horizontal text alignment: left (default), right, center.

**yalign=***val* controls vertical text alignment: base (default), top, bottom, center.

**shadow=***size* if nonzero, will make a dark shadow halo to make the text more clear on bright backgrounds.

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 --create 320x240 3 --text:x=10:y=400:size=40 "Hello world" \
  --text:x=100:y=200:font="Arial Bold":color=1,0,0:size=60 "Go Big Red!" \
  --tocolorspace sRGB -o text.jpg

oiiotool --create 320x240 3 --text:x=160:y=120:xalign=center "Centered" \
  --tocolorspace sRGB -o textcentered.jpg

oiiotool tahoe-small.jpg \
  --text:x=160:y=40:xalign=center:size=40:shadow=0 "shadow = 0" \
  --text:x=160:y=80:xalign=center:size=40:shadow=1 "shadow = 1" \
  --text:x=160:y=120:xalign=center:size=40:shadow=2 "shadow = 2" \
  --tocolorspace sRGB -o textshadowed.jpg
```

Note that because of slightly differing fonts and versions of Freetype available, we do not expect drawn text to be pixel-for-pixel identical on different platforms supported by OpenImageIO.

## 12.10 oiiotool commands for color management

Many of the color management commands depend on an installation of OpenColorIO (<http://opencolorio.org>).

If OIIO has been compiled with OpenColorIO support and the environment variable `$OCIO` is set to point to a valid OpenColorIO configuration file, you will have access to all the color spaces that are known by that OCIO configuration. Alternately, you can use the `--colorconfig` option to explicitly point to a configuration file. If no valid configuration file is found (either in `$OCIO` or specified by `--colorconfig`) or OIIO was not compiled with OCIO support, then the only color space transforms 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, looks, and displays that `oiiotool` knows about.

### `--iscolorspace <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.

### `--colorconfig <filename>`

Instruct `oiiotool` to read an OCIO configuration from a custom location. Without this, the default is to use the `$OCIO` environment variable as a guide for the location of the configuration file.

### `--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). Optional appended modifiers include:

- `key= name, value= str :`

Adds a key/value pair to the “context” that OpenColorIO will use when applying the look. Multiple key/value pairs may be specified by making each one a comma-separated list.

- `unpremult= val :`

If the numeric *val* is nonzero, the pixel values will be “un-premultiplied” (divided by alpha) prior to the actual color conversion, and then re-multiplied by alpha afterwards. The default is 0, meaning the color transformation will not be automatically bracketed by divide-by-alpha / mult-by-alpha operations.

- `strict= val :`

When nonzero (the default), an inability to perform the color transform will be a hard error. If strict is 0, inability to find the transformation will just print a warning and simply copy the image without changing colors.

### `--tocolorspace <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.

### `--ccmatrix <m00,m01,...>`

#### NEW 2.1

Replace the current image with a new image whose colors are transformed according to the 3x3 or 4x4 matrix formed from the 9 or 16 comma-separated floating-point values in the subsequent argument (spaces are allowed only if the whole collection of values is enclosed in quotes so that they are a single command-line argument).

The values fill in the matrix left to right, first row, then second row, etc. This means that colors are treated as “row vectors” that are post-multiplied by the matrix ( $C \cdot M$ ).

Optional appended modifiers include:

- `unpremult= val` :

If the numeric *val* is nonzero, the pixel values will be “un-premultiplied” (divided by alpha) prior to the actual color conversion, and then re-multiplied by alpha afterwards. The default is 0, meaning the color transformation not will be automatically bracketed by divide-by-alpha / mult-by-alpha operations.

- `invert= val` :

If nonzero, this will cause the matrix to be inverted before being applied.

- `transpose= val` :

If nonzero, this will cause the matrix to be transposed (this allowing you to more easily specify it as if the color values were column vectors and the transformation as  $M \times C$ ).

Example:

```
# Convert ACES to ACEScg using a matrix
oiio tool aces.exr --ccmatrix:transpose=1 \
    "1.454,-0.237,-0.215,-0.077,1.176,-0.010,0.008,-0.006, 0.998" \
    -o acescg.exr
```

#### **--ociolook** <lookname>

Replace the current image with a new image whose pixels are transformed using the named OpenColorIO look description. Optional appended arguments include:

- `from= val`

Assume the image is in the named color space. If no `from=` is supplied, it will try to deduce it from the image’s metadata or previous `--iscolospace` directives.

- `to= val`

Convert to the named space after applying the look.

- `inverse= val`

If *val* is nonzero, inverts the color transformation and look application.

- `key= name, value= str`

Adds a key/value pair to the “context” that OpenColorIO will used when applying the look. Multiple key/value pairs may be specified by making each one a comma-separated list.

- `unpremult= val` :

If the numeric *val* is nonzero, the pixel values will be “un-premultiplied” (divided by alpha) prior to the actual color conversion, and then re-multiplied by alpha afterwards. The default is 0, meaning the color transformation not will be automatically bracketed by divide-by-alpha / mult-by-alpha operations.

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 **oiio tool** help (**oiio tool** `--help`), at the very bottom you will see the list of all looks that **oiio tool** knows about.

Examples:

```
oiio tool 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*. An empty string for *displayname* means to use the default display, and an empty string for *viewname* means to use the default view on that display.

Optional appended modifiers include:

**from= *val*** Assume the image is in the named color space. If no **from=** is supplied, it will try to deduce it from the image's metadata or previous **--iscolorspace** directives.

**key= *name*, value= *str*** Adds a key/value pair to the “context” that OpenColorIO will use when applying the look. Multiple key/value pairs may be specified by making each one a comma-separated list.

**unpremult= *val*** : If the numeric *val* is nonzero, the pixel values will be “un-premultiplied” (divided by alpha) prior to the actual color conversion, and then re-multiplied by alpha afterwards. The default is 0, meaning the color transformation will not be automatically bracketed by divide-by-alpha / mult-by-alpha operations.

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 **oiiootool help** (**oiiootool --help**), at the very bottom you will see the list of all looks that **oiiootool** knows about.

Examples:

```
oiiootool in.exr --ociodisplay:from=lnf:key=SHOT:value=pe0012 sRGB Film -o cc.jpg
```

#### **--ociofiletransform <name>**

Replace the current image with a new image whose pixels are transformed using the named OpenColorIO file transform. Optional appended arguments include:

- **inverse= *val*** :

If *val* is nonzero, inverts the color transformation.

- **unpremult= *val*** :

If the numeric *val* is nonzero, the pixel values will be “un-premultiplied” (divided by alpha) prior to the actual color conversion, and then re-multiplied by alpha afterwards. The default is 0, meaning the color transformation will not be automatically bracketed by divide-by-alpha / mult-by-alpha operations.

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 **oiiootool help** (**oiiootool --help**), at the very bottom you will see the list of all looks that **oiiootool** knows about.

Examples:

```
oiiootool in.jpg --ociofiletransform footransform.csp -o out.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 off 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 off as “unassociated alpha,” a.k.a. “non-premultiplied.”

## 12.11 oiiotool commands for deep images

A number of **oiiotool** operations are designed to work with “deep” images. These are detailed below. In general, operations not listed in this section should not be expected to work with deep images.

### 12.11.1 Commands specific to deep images

#### **--deepen**

If the top image is not deep, then turn it into the equivalent “deep” image. Pixels with non-infinite  $z$  or with any non-zero color channels will get a single depth sample in the resulting deep image. Pixels in the source that have 0 in all channels, and either no “Z” channel or a  $z$  value indicating an infinite distance, will result in a pixel with no depth samples.

Optional appended modifiers include:

**z= *val*** The depth to use for deep samples if the source image did not have a “Z” channel. (The default is 1.0.)

#### **--flatten**

If the top image is “deep,” then “flatten” it by compositing the depth samples in each pixel.

#### **--deepmerge**

Replace the *two* top images with a new deep image that is the “deep merge” of the inputs. Both input images must be deep images, have the same number and order of channels and must contain an alpha channel and depth channel.

#### **--deepholdout**

Replace the *two* top images with a new deep image that is the “deep holdout” of the first image by the second — that is, the samples from the first image that are closer than the opaque frontier of the second image. Both input inputs must be deep images.

### 12.11.2 General commands that also work for deep images

#### **--addc, --subc, --mulc, --divc**

Adding, subtracting, multiplying, or dividing a per-channel constant will work for deep images, performing the operation for every sample in the image.

#### **--autotrim**

For subsequent outputs, automatically **--trim** before writing the file.

#### **--ch <channellist>**

Reorder, rename, remove, or add channels to a deep image. See Section *oiiotool commands that shuffle channels or subimages*

#### **--crop <size>**

Crop (adjust the pixel data window), removing pixels or adding empty pixels as needed.

#### **--paste <position>**

Replace one image’s pixels with another’s (at an arbitrary offset).

(This functionality was extended to deep images in OIIO 2.1.)



**--resample** <size>  
Resampling (resize without filtering or interpolation, just choosing the closest deep pixel to copy for each output pixel).

**--diff**  
Report on the difference of the top two images.

**--dumpdata**  
Print to the console detailed information about the values in every pixel.  
Optional appended modifiers include:

- `empty= val` : If 0, will cause deep images to skip printing of information about pixels with no samples, and cause non-deep images to skip printing information about pixels that are entirely 0.0 value in all channels.

**--info**  
Prints information about each input image as it is read.

**--trim**  
Replace the current image with a new image that is cropped to contain the minimal rectangular ROI that contains all of the non-empty pixels of the original image.

**--scanline**

**--tile** <x> <y>  
Convert to scanline or to tiled representations upon output.

**--stats**  
Prints detailed statistical information about each input image as it is read.

**--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. The *strategy* may be either `black` or `error`.



## 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 of OpenImageIO, it will print information about images of any formats readable by ImageInput plugins on hand.

### 13.1 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 `emph{verbose mode}`, which exhaustively prints all metadata about each image:

```
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
```

(continues on next page)

(continued from previous page)

```

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 suppression)
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

```

(continues on next page)

(continued from previous page)

```
$ 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
  ...
...
```

## 13.2 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.



## CONVERTING IMAGE FORMATS WITH `ICONVERT`

### 14.1 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.

### 14.2 `iconvert` Recipes

This section will give quick examples of common uses of `iconvert`.

#### 14.2.1 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
```

### 14.2.2 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
```

### 14.2.3 Changing the compression

The following command converts writes a TIFF file, specifically using zip compression:

```
iconvert --compression zip in.tif out.tif
```

The following command writes its results as a JPEG file at a compression quality of 50 (pretty severe compression), illustrating how some compression methods allow a quality metric to be optionally appended to the name:

```
iconvert --compression jpeg:50 50 big.jpg small.jpg
```

### 14.2.4 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
```

### 14.2.5 Converting between scanline and tiled images

Convert a scanline file to a tiled file with \$16 x 16\$ tiles:

```
iconvert --tile 16 16 s.tif t.tif
```

Convert a tiled file to scanline:

```
iconvert --scanline t.tif s.tif
```

### 14.2.6 Converting images in place

You can use the `--inplace` flag to cause the output to `emph{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
```



### 14.2.7 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
```

### 14.2.8 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
```

## 14.3 `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**

**--compression method:quality**

Sets the compression method, and optionally a quality setting, 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 to  $q$ , on a 1–100 floating-point scale. This only has an effect if the particular compression method supports a quality metric (as JPEG does).

DEPRECATED(2.1): This is considered deprecated, and in general we now recommend just appending the quality metric to the `compression name:qual`.

**--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 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~ref{metadata:orientation} 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.



## SEARCHING IMAGE METADATA WITH IGREP

The `igrep` program search one or more image files for metadata that match a string or regular expression.

### 15.1 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
```

### 15.2 `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, surpressing 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.

## COMPARING IMAGES WITH `IDIFF`

### 16.1 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 of `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.

### 16.2 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.

### 16.2.1 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.

### 16.2.2 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).

## 16.3 `idiff` Command Line Argument Reference

The various command-line options are discussed below:



### 16.3.1 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.

**-q**

Quiet mode – output nothing for successful match), output only minimal error messages to stderr for failure / no match. The shell return code also indicates success or failure (successful match returns 0, failure returns nonzero).

**-a**

Compare all subimages. Without this flag, only the first subimage of each file will be compared.

### 16.3.2 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.

### 16.3.3 Difference image output

**-o outputfile**

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.

### 16.3.4 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.

## MAKING TILED MIP-MAP TEXTURE FILES WITH **MAKETX** OR **OIIOTOOL**

### 17.1 Overview

The TextureSystem (Chapter chap-texturesystem\_) will exhibit much higher performance if the image files it uses as textures are tiled (versus scanline) orientation, have multiple subimages at different resolutions (MIP-mapped), and include a variety of header or metadata fields appropriately set for texture maps. Any image that you intend to use as input to TextureSystem, therefore, should first be converted to have these properties. An ordinary image will work as a texture, but without this additional step, it will be drastically less efficient in terms of memory, disk or network I/O, and time.

This can be accomplished programmatically using the ImageBufAlgo `make_texture()` function (see Section *Import / export* for C++ and Section *Import / export* for Python).

OpenImageIO includes two command-line utilities capable of converting ordinary images into texture files: **maketx** and **oiio tool**.<sup>1</sup>

### 17.2 **maketx**

The **maketx** program will convert ordinary images to efficient textures. 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., **:filename: foo.tif** will write a TIFF file).

Command-line arguments are:

**--help**

Prints usage information to the terminal.

**-v**

Verbose status messages, including runtime statistics and timing.

**--runstats**

Print runtime statistics and timing.

---

<sup>1</sup> Why are there two programs? Historical artifact – **maketx** existed first, and much later **oiio tool** was extended to be capable of directly writing texture files. If you are simply converting an image into a texture, **maketx** is more straightforward and foolproof, in that you can't accidentally forget to turn it into a texture, as you might do with and much later **oiio tool** was extended to be capable of directly writing texture files. If you are simply converting an image into a texture, **maketx** is more straightforward and foolproof, in that you can't accidentally forget to turn it into a texture, as you might do with

- 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 x 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>*  
**--compression** *<method:quality>*  
Sets the compression method, and optionally a quality setting, for the output image (the default is to try to use “zip” compression, if it is available).
- u**  
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 **-u** option enables *update mode*: if the output file already exists, and has the same time stamp as the input file, and the command-line arguments that created it are identical to the current ones, then the texture will be left alone and not be recreated. If the output file does not exist, or has a different time stamp than the input file, or was created using different command line arguments, then the texture will be created and 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`, `mitschell`, `bspline`, `cubic`, `keys`, `simon`, `rifman`, `radial-lanczos3`, `disk`, `sinc`.

If you select a filter with negative lobes (including `lanczos3`, `sinc`, `lanczos3`, `keys`, `simon`, `rifman`, 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.

**--sharpen** <contrast>

EXPERIMENTAL: USE AT YOUR OWN RISK!

This option will run an additional sharpening filter when creating the successive MIP-map levels. It uses an unsharp mask (much like in Section `sec-iba-unsharpmask`) to emphasize high-frequency details to make features “pop” visually even at high MIP-map levels. The *contrast* controls the degree to which it does this. Probably a good value to enhance detail but not go overboard is 0.5 or even 0.25. A value of 1.0 may make strange artifacts at high MIP-map levels. Also, if you simultaneously use `--filter unsharp-median`, a slightly different variant of unsharp masking will be used that employs a median filter to separate out the low-frequencies, this may tend to help emphasize small features while not over-emphasizing large edges.

**--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 3x3 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.

**--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:

```
oiiotool --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.

**--sansattrib**

When set, this edits the command line inserted in the “Software” and “ImageHistory” metadata to omit any verbose `--attrib` and `--sattrib` commands.

**--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.

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.

**--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.

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.

**--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), and in the case of TIFF files will omit the Exif directory block which will not be recognized by the older version of libtiff used by PRMan.

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
```

### **--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, OIIO will at least be able to convert among linear, sRGB, and Rec709.

### **--colorconfig** <name>

Explicitly specify a custom OpenColorIO configuration file. Without this, the default is to use the \$OCIO environment variable as a guide for the location of the OpenColorIO configuration file.

### **--unpremult**

When undergoing some color 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 --mipimage 128.tif --mipimage 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. (See <http://www.pauldebevec.com/Probes/> for examples and an explanation of the geometric layout.)

### **--bumpslopes**

For a single channel input image representing height (that you would ordinarily use for a bump or displacement),

this produces a 6-channel texture that contains the first and second moments of bump slope, which can be used to implement certain bump-to-roughness techniques. The channel layout is as follows:

index	channel name	data at MIP level 0
0	b0_h	$h$ (height)
1	b1_dhds	$\partial h / \partial s$
2	b2_dhdt	$\partial h / \partial t$
3	b3_dhds2	$(\partial h / \partial s)^2$
4	b4_dhdt2	$(\partial h / \partial t)^2$
5	b5_dhdsdt	$(\partial h / \partial s) \cdot (\partial h / \partial t)$

(The strange channel names are to guarantee they are in alphabetical order, to prevent reordering by OpenEXR. And also note that the simple relationships between channels 1 & 2, and 3-6, is only for the highest- resolution level of the MIP-map, and will differ for the lower-res filtered versions, and those differences is what gives us the slope momets that we need.)

A reference for explaining how this can be used is here:

Christophe Hery, Michael Kass, and Junhi Ling. “Geometry into Shading.” Technical Memo 14-04, Pixar Animation Studios, 2014. <https://graphics.pixar.com/library/BumpRoughness>

**--bumpformat** <bformat>

In conjunction with `--bumpslopes`, this option can specify the strategy for determining whether a 3-channel source image specifies a height map or a normal map. The value “height” indicates it is a height map (only the first channel will be used). The value “normal” indicates it is a normal map (all three channels will be used for x, y, z). The default value, “auto”, indicates that it should be interpreted as a height map if and only if the R, G, B channel values are identical in all pixels, otherwise it will be interpreted as a 3-channel normal map.

## 17.3 oiiotool

The **oiiotool** utility (Chapter *oiiotool: the OIIO Swiss Army Knife*) is capable of writing textures using the `-otex` option, and lat-long environment maps using the `-oenv` option. Roughly speaking,

```
maketx [maketx-options] input -o output
```

is equivalent to

```
oiiotool input [oiiotool-options] -otex output
```

and

```
maketx -envlatl [maketx-options] input -o output
```

is equivalent to

```
oiiotool input [oiiotool-options] -oenv output
```

However, the notation for the various options are not identical between the two programs, as will be explained by the remainder of this section.

The most important difference between **oiiotool** and **maketx** is that **oiiotool** can do so much more than convert an existing input image to a texture – literally any image creation or manipulation you can do via **oiiotool** may be output directly as a full texture file using `-otex` (or as a lat-long environment map using `-oenv`).

Note that it is vitally important that you use one of the texture output commands (`-otex` or `-oenv`) when creating textures with **oiiotool** — if you inadvertently forget and use an ordinary `-o`, you will end up with an output image that is much less efficient to use as a texture.



### 17.3.1 Command line arguments useful when creating textures

As with any use of **oiiootool**, you may use the following to control the run generally:

```
--help
-v
--runstats
--threads <n>
    and as with any use of oiiootool, you may use the following command-line options to control aspects of the
    any output files (including those from -otex and -oenv, as well as -o). Only brief descriptions are given
    below, please consult Chapter oiiootool for detailed explanations.

-d <datatype>
    Specify the pixel data type (UINT8, uint16, half, float, etc.) if you wish to override the default of writing
    the same data type as the first input file read.

--tile <x> <y>
    Explicitly override the tile size (though you are strongly urged to use the default, and not use this command).

--compression <method>
    Explicitly override the default compression methods when writing the texture file.

--ch <channellist>
    Shuffle, add, delete, or rename channels (see sec-oiiootool-ch).

--chnames a,b,...
    Renames the channels of the output image.

--fixnan <strategy>
    Repairs any pixels in the input image that contained NaN or Inf values (if the strategy is box3 or black), or
    to simply abort with an error message (if the strategy is error).

--fullpixels
    Resets the “full” (or “display”) pixel range to be the “data” range.

--planarconfig separate
    Forces “separate” (e.g., RRR...GGG...BBB) packing of channels in the output texture. This is almost always
    a bad choice, unless you know that the texture file must be readable by PRMan, in which case it is required.

--attrib <name> <value>
    oiiootool’s --attrib command may be used to set attributes in the metadata of the output texture.

--attrib:type=matrix worldtocam <...16 comma-separated floats...>
--attrib:type=matrix screentocam <...16 comma-separated floats...>
    Set/override the camera and screen matrices.
```

### 17.3.2 Optional arguments to **-otex** and **-oenv**

As with many **oiiootool** commands, the **-otex** and **-oenv** may have various optional arguments appended, in the form **:name=value** (see Section sec-oiiootooloptionalargs).

Optional arguments supported by **-otex** and **-oenv** include all the same options as **-o** (Section *Writing images*) and also the following (explanations are brief, please consult Section sec-maketx for more detailed explanations of each, for the corresponding **maketx** option):

Appended Option	maketx equivalent
wrap= <i>string</i>	--wrap
swrap= <i>string</i>	--swrap
twrap= <i>string</i>	--twrap
resize=1	--resize
nomipmap=1	--nomipmap
updatemode=1	-u
monochrome_detect=1	--monochrome-detect
opaque_detect=1	--opaque-detect
unpremult=1	--unpremult
incolospace= <i>name</i>	--incolospace
outcolospace= <i>name</i>	--outcolospace
highlightcomp=1	--hicomp
sharpen= <i>float</i>	--sharpen
filter= <i>string</i>	--filter
prman_metadata=1	--prman
prman_options=1	--prman-metadata

### 17.3.3 Examples

```
oiiotool in.tif -otex out.tx

oiiotool in.jpg --colorconvert sRGB linear -d uint16 -otex out.tx

oiiotool --pattern:checker 512x512 3 -d uint8 -otex:wrap=periodic checker.tx

oiiotool in.exr -otex:highlightcomp=1:sharpen=0.5 out.exr
```

## METADATA CONVENTIONS

The `ImageSpec` class, described thoroughly in *Image Specification: ImageSpec*, 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_attribs` field. Though literally *anything* can be stored in `extra_attribs` — 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.

### 18.1 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".

Usually this is simply the time that the image data was last modified. It may be wise to also store the "Exif:DateTimeOriginal" and "Exif:DateTimeDigitized" (see Section *Exif metadata*) to further distinguish the original image and its conversion to digital media.

**"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.

## 18.2 Display hints

**"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 different orientation, according to the TIFF/EXIF conventions:

0	normal (top to bottom, left to right)
1	flipped horizontally (top to bottom, right to left)
2	rotated 180° (bottom to top, right to left)
3	flipped vertically (bottom to top, left to right)
4	transposed (left to right, top to bottom)
5	rotated 90° clockwise (right to left, top to bottom)
6	transverse (right to left, bottom to top)
7	rotated 90° counter-clockwise (left to right, bottom to top)

**"PixelAspectRatio"** : float

The aspect ratio ( $x/y$ ) of the size of 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.

**"oio:Movie"** : int

If nonzero, a hint that a multi-image file is meant to be interpreted as an animation (i.e., that the subimages are a time sequence).

**"oio:subimages"** : int

If nonzero, the number of subimages in the file. Not all image file formats can know this without reading the entire file, and in such cases, this attribute will not be set or will be 0. If the value is present and greater than zero, it can be trusted, but if not, nothing should be inferred and you will have to repeatedly seek to subimages to find out how many there are.

**"FramesPerSecond"** : rational

For a multi-image file intended to be played back as an animation, the frame refresh rate. (It's technically a rational, but it may be retrieved as a float also, if you are ok with imprecision.)

## 18.3 Color information

**"oio:ColorSpace"** : string

The name of the color space of the color channels. Values include:

- **"Linear"** : Color pixel values are known to be scene-linear and using facility-default color primaries (presumed sRGB/Rec709 color primaries if otherwise unknown).
- **"sRGB"** : Using standard sRGB response and primaries.
- **"Rec709"** : Using standard Rec709 response and primaries.
- **"ACES"** : ACES color space encoding.

- "AdobeRGB" : Adobe RGB color space.
- "KodakLog" : Kodak logarithmic color space.
- "GammaCorrectedX.Y" : Color values have been gamma corrected (raised to the power  $1/\gamma$ ). The X.Y is the numeric value of the gamma exponent.
- *arbitrary* : The name of any color space known to OpenColorIO (if OCIO support is present).

"oiio:Gamma" : float

If the color space is "GammaCorrectedX.Y", this value is the gamma exponent. (Optional/deprecated; if present, it should match the suffix of the color space.)

"oiio: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).

"ICCProfile" : uint8[]

The ICC color profile takes the form of an array of bytes (unsigned 8 bit chars). The length of the array is the length of the profile blob.

## 18.4 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 file format to file format, and each reader/writer plugin should document the modes it supports. If `ImageOutput::open` is called with an `ImageSpec` that specifies a compression mode not supported by that `ImageOutput`, it will choose a reasonable default. As an example, the OpenEXR writer supports `none`, `rle`, `zip`, `zips`, `piz`, `pxr24`, `b44`, `b44a`, `dwaa`, or `dwab`.

The compression name is permitted to have a quality value to be appended after a colon, for example `dwaa:60`. The exact meaning and range of the quality value can vary between different file formats and compression modes, and some don't support quality values at all (it will be ignored if not supported, or if out of range).

"CompressionQuality" : int

DEPRECATED(2.1)

This is a deprecated method of separately specifying the compression quality. 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.

## 18.5 Substituting an `IOProxy` for custom I/O overrides

Format readers and writers that respond positively to `supports("ioproxy")` have the ability to read or write using an *I/O proxy* object. Among other things, this lets an `ImageOutput` write the file to a memory buffer rather than saving to disk, and for an `ImageInput` to read the file from a memory buffer. (Currently, only PNG and OpenEXR have the ability to do this.) This behavior is controlled by a special attributes

**"oio:ioproxy"** : pointer

Pointer to a `Filesystem::IOProxy` that will handle the I/O.

An explanation of how this feature is used may be found in Sections *Custom I/O proxies (and reading the file from a memory buffer)* and *Custom I/O proxies (and writing the file to a memory buffer)*.

## 18.6 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.

## 18.7 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  $t = j/(yres - 1)$ .

**"oiio:ConstantColor"** : string

If present, is a hint that the texture has the same value in all pixels, and the metadata value is a string containing the channel values as a comma-separated list (no spaces, for example: `0.73,0.9,0.11,1.0`).

**"oiio:AverageColor"** : string

If present, is a hint giving the *average* value of all pixels in the texture, as a string containing a comma-separated list of the channel values (no spaces, for example: `0.73,0.9,0.11,1.0`).

**"oiio:SHA-1"** : string

If present, is a 40-byte SHA-1 hash of the input image (possibly salted with various maketx options) that can serve to quickly compare two separate textures to know if they contain the same pixels. While it's not, technically, 100% guaranteed that no separate textures will match, it's so astronomically unlikely that we discount the possibility (you'd be rendering oovies for centuries before finding a single match).

## 18.8 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

**"Exif:DateTimeDigitized"** : string

Date and time that the original image data was generated or captured, and the time/time that the image was stored as digital data. Both are in YYYY:MM:DD HH:MM:SS format.

To clarify the role of these (and also with respect to the standard `DateTime` metadata), consider an analog photograph taken in 1960 (`Exif:DateTimeOriginal`), which was scanned to a digital image in 2010 (`Exif:DateTimeDigitized`), and had color corrections or other alterations performed in 2015 (`DateTime`).

**"Exif:CompressedBitsPerPixel"** : float

The compression mode used, measured in compressed bits per pixel.

**"Exif:ShutterSpeedValue"** : float

Shutter speed, in APEX units:  $-\log_2(\text{exposure time})$

**"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 suppression
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.

**"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	$\mu m$

**"Exif:ExposureIndex"** : float

The exposure index selected on the camera.

**"Exif:SensingMethod"** : int

The image sensor type on the camra:

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.

## 18.9 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 a 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.

## 18.10 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:

IPTC tag	OpenImageIO metadata convention
Caption	"ImageDescription"
Keyword	IPTC keywords should be concatenated, separated by semicolons (;), and stored as the Keywords attribute.
Exposure-Time	ExposureTime
Copyright-Notice	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.

## 18.11 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).

## 18.12 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.



## 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 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.



## Symbols

"Artist" : string  
     command line option, 367  
 "CompressionQuality" : int  
     command line option, 369  
 "Copyright" : string  
     command line option, 367  
 "DateTime" : string  
     command line option, 367  
 "DocumentName" : string  
     command line option, 367  
 "Exif:ApertureValue" : float  
     command line option, 372  
 "Exif:BrightnessValue" : float  
     command line option, 372  
 "Exif:CompressedBitsPerPixel" : float  
     command line option, 372  
 "Exif:Contrast" : int  
     command line option, 375  
 "Exif:CustomRendered" : int  
     command line option, 374  
 "Exif:DateTimeDigitized" : string  
     command line option, 372  
 "Exif:DateTimeOriginal" : string  
     command line option, 372  
 "Exif:DigitalZoomRatio" : float  
     command line option, 375  
 "Exif:ExposureBiasValue" : float  
     command line option, 372  
 "Exif:ExposureIndex" : float  
     command line option, 374  
 "Exif:ExposureMode" : int  
     command line option, 374  
 "Exif:ExposureProgram" : int  
     command line option, 371  
 "Exif:FileSource" : int  
     command line option, 374  
 "Exif:Flash" int}  
     command line option, 373  
 "Exif:FlashEnergy" : float  
     command line option, 374  
 "Exif:FocalLength" : float  
     command line option, 373  
 "Exif:FocalLengthIn35mmFilm" : int  
     command line option, 375  
 "Exif:FocalPlaneResolutionUnit" : int  
     command line option, 374  
 "Exif:FocalPlaneXResolution" : float  
     command line option, 374  
 "Exif:FocalPlaneYResolution" : float  
     command line option, 374  
 "Exif:GainControl" : float  
     command line option, 375  
 "Exif:ISOSpeedRatings" : int  
     command line option, 372  
 "Exif:ImageHistory" : string  
     command line option, 373  
 "Exif:ImageUniqueID" : string  
     command line option, 376  
 "Exif:LightSource" : int  
     command line option, 373  
 "Exif:MaxApertureValue" : float  
     command line option, 372  
 "Exif:MeteringMode" : int  
     command line option, 372  
 "Exif:PixelXDimension" : int  
     command line option, 374  
 "Exif:PixelYDimension" : int  
     command line option, 374  
 "Exif:Saturation" : int  
     command line option, 375  
 "Exif:SceneCaptureType" : int  
     command line option, 375  
 "Exif:SceneType" : int  
     command line option, 374  
 "Exif:SecurityClassification" :  
     string  
     command line option, 373  
 "Exif:SensingMethod" : int  
     command line option, 374  
 "Exif:Sharpness" : int  
     command line option, 375  
 "Exif:ShutterSpeedValue" : float  
     command line option, 372

```
"Exif:SpectralSensitivity" : string
    command line option,372
"Exif:SubjectDistance" : float
    command line option,372
"Exif:SubjectDistanceRange" : int
    command line option,375
"Exif:SubsecTime" : string
    command line option,373
"Exif:SubsecTimeDigitized" : string
    command line option,373
"Exif:SubsecTimeOriginal" : string
    command line option,373
"Exif:WhiteBalance" : int
    command line option,375
"ExposureTime" : float
    command line option,370
"FNumber" : float
    command line option,370
"FramesPerSecond" : rational
    command line option,368
"GPS:Altitude" : float
    command line option,376
"GPS:AltitudeRef" : string
    command line option,376
"GPS:AreaInformation" : string
    command line option,377
"GPS:DOP" : float
    command line option,376
"GPS:DateStamp" : string
    command line option,377
"GPS:DestBearing" : float
    command line option,377
"GPS:DestBearingRef" : string
    command line option,377
"GPS:DestDistance" : float
    command line option,377
"GPS:DestDistanceRef" : string
    command line option,377
"GPS:DestLatitude" : float[3]
    command line option,377
"GPS:DestLatitudeRef" : string
    command line option,377
"GPS:DestLongitude" : float[3]
    command line option,377
"GPS:DestLongitudeRef" : string
    command line option,377
"GPS:Differential" : int
    command line option,377
"GPS:HPositioningError" : float
    command line option,377
"GPS:ImgDirection" : float
    command line option,377
"GPS:ImgDirectionRef" : string
    command line option,376
"GPS:Latitude" : float[3]
    command line option,376
"GPS:LatitudeRef" : string
    command line option,376
"GPS:Longitude" : float[3]
    command line option,376
"GPS:LongitudeRef" : string
    command line option,376
"GPS:MapDatum" : string
    command line option,377
"GPS:MeasureMode" : string
    command line option,376
"GPS:ProcessingMethod" : string
    command line option,377
"GPS:Satellites" : string
    command line option,376
"GPS:Speed" : float
    command line option,376
"GPS:SpeedRef" : string
    command line option,376
"GPS:Status" : string
    command line option,376
"GPS:TimeStamp" : float[3]
    command line option,376
"GPS:Track" : float
    command line option,376
"GPS:TrackRef" : string
    command line option,376
"HostComputer" : string
    command line option,367
"ICCProfile" : uint8[]
    command line option,369
"IPTC:AuthorsPosition" : string
    command line option,379
"IPTC:CaptionWriter" : string
    command line option,379
"IPTC:Category" : string
    command line option,378
"IPTC:City" : string
    command line option,379
"IPTC:ClassifyState" : string
    command line option,379
"IPTC:Contact" : string
    command line option,379
"IPTC:ContentLocationCode" : string
    command line option,378
"IPTC:ContentLocationName" : string
    command line option,378
"IPTC:Country" : string
    command line option,379
"IPTC:CountryCode" : string
    command line option,379
"IPTC:DateCreated" : string
    command line option,378
```

```

"IPTC:DigitalCreationDate" : string
    command line option, 378
"IPTC:DigitalCreationTime" : string
    command line option, 378
"IPTC:DocumentHistory" : string
    command line option, 379
"IPTC:DocumentNotes" : string
    command line option, 379
"IPTC:EditStatus" : string
    command line option, 378
"IPTC:ExpirationDate" : string
    command line option, 378
"IPTC:ExpirationTime" : string
    command line option, 378
"IPTC:Headline" : string
    command line option, 379
"IPTC:Instructions" : string
    command line option, 378
"IPTC:JobID" : string
    command line option, 379
"IPTC:MasterDocumentID" : string
    command line option, 379
"IPTC:ObjectTypeReference" : string
    command line option, 378
"IPTC:ObjectAttributeReference" :
    string
    command line option, 378
"IPTC:ObjectName" : string
    command line option, 378
"IPTC:OwnerID" : string
    command line option, 379
"IPTC:Prefs" : string
    command line option, 379
"IPTC:ProgramVersion" : string
    command line option, 378
"IPTC:Provider" : string
    command line option, 379
"IPTC:ReferenceDate" : string
    command line option, 378
"IPTC:ReferenceNumber" : string
    command line option, 378
"IPTC:ReferenceService" : string
    command line option, 378
"IPTC:ReleaseDate" : string
    command line option, 378
"IPTC:ReleaseTime" : string
    command line option, 378
"IPTC:ShortDocumentID" : string
    command line option, 379
"IPTC:SimilarityIndex" : string
    command line option, 379
"IPTC:Source" : string
    command line option, 379
"IPTC:State" : string
    command line option, 379
"IPTC:SubjectReference" : string
    command line option, 378
"IPTC:Sublocation" : string
    command line option, 379
"IPTC:TimeCreated" : string
    command line option, 378
"IPTC:UniqueDocumentID" : string
    command line option, 379
"ImageDescription" : string
    command line option, 367
"Keywords" : string
    command line option, 367
"Make" : string
    command line option, 370
"Model" : string
    command line option, 370
"Orientation" : int
    command line option, 368
"PixelAspectRatio" : float
    command line option, 368
"ResolutionUnit" : string
    command line option, 368
"Software" : string
    command line option, 367
"XResolution" : float
    command line option, 368
"YResolution" : float
    command line option, 368
"compression" : string
    command line option, 369
"fovco" : float
    command line option, 370
"oiio:AverageColor" : string
    command line option, 371
"oiio:BitsPerSample" : int
    command line option, 369
"oiio:BorderColor" : float[nchannels]
    command line option, 369
"oiio:ColorSpace" : string
    command line option, 368
"oiio:ConstantColor" : string
    command line option, 371
"oiio:Gamma" : float
    command line option, 369
"oiio:Movie" : int
    command line option, 368
"oiio:SHA-1" : string
    command line option, 371
"oiio:UnassociatedAlpha" : int
    command line option, 369
"oiio:ioproxy" : pointer
    command line option, 370
"oiio:sampleborder" : int

```

```
    command line option, 371
"oiio:subimages" : int
    command line option, 368
"oiio:updirection" : string
    command line option, 371
"planarconfig" : string
    command line option, 369
"smppte:KeyCode" : int[7]
    command line option, 379
"smppte:TimeCode" : int[2]
    command line option, 379
"textureformat" : string
    command line option, 370
"worldtocamera" : matrix44
    command line option, 371
"worldtoscreen" : matrix44
    command line option, 371
"wrapmodes" : string
    command line option, 370
-- divc <value>
    command line option, 317
-- mulc <value>
    command line option, 317
-- subc <value>
    command line option, 317
--Mcamera <...16 floats...>
    command line option, 362
--Mscreen <...16 floats...>
    command line option, 362
--abs
    command line option, 318
--absdiff
    command line option, 318
--absdiffc <value>
    command line option, 318
--add
    command line option, 316
--addc
    command line option, 340
--addc <value>
    command line option, 316
--adjust-time
    command line option, 308
--attrib <name> <value>
    command line option, 309, 362, 365
--attrib:type=matrix worldtocam <...16
    comma-separated floats...>
    command line option, 365
--autocc
    command line option, 305
--autoorient
    command line option, 305
--autopremult
    command line option, 304
--autotile <tilesize>
    command line option, 306
--autotrim
    command line option, 309, 340
--blur <size>
    command line option, 330
--box <x1,y1,x2,y2>
    command line option, 335
--bumpformat <bformat>
    command line option, 364
--bumpslopes
    command line option, 363
--cache <size>
    command line option, 306
--caption <text>
    command line option, 309
--capture
    command line option, 316
--ccmatrix <m00,m01,...>
    command line option, 337
--ch <channellist>
    command line option, 312, 340, 365
--chappend
    command line option, 312
--checknan
    command line option, 361
--chnames <name-list>
    command line option, 311
--chnames a,b,...
    command line option, 361, 365
--chsum
    command line option, 319
--clamp
    command line option, 333
--clear-keywords
    command line option, 310
--colorconfig <filename>
    command line option, 337
--colorconfig <name>
    command line option, 363
--colorconvert <fromspace tospace>
    command line option, 337
--colorconvert <inspace> <outspace>
    command line option, 363
--colorcount r1,g1,b1,...:r2,g2,b2,...:...
    command line option, 302
--colormap <mapname>
    command line option, 321
--compression <method>
    command line option, 308, 360, 365
--constant-color-detect
    command line option, 362
--contrast
    command line option, 320
```

---

```

--convolve
    command line option, 330
--create <size> <channels>
    command line option, 313
--crop <size>
    command line option, 326, 340
--croptofull
    command line option, 326
--cshift <offset>
    command line option, 325
--cut <size>
    command line option, 327
--debug
    command line option, 301
--deepen
    command line option, 340
--deepholdout
    command line option, 340
--deepmerge
    command line option, 340
--diff
    command line option, 302, 341
--dilate <size>
    command line option, 331
--dither
    command line option, 308
--div
    command line option, 317
--divc
    command line option, 340
--dumpdata
    command line option, 302, 341
--dup
    command line option, 312
--echo <message>
    command line option, 302
--envlatl
    command line option, 363
--eraseattrib <pattern>
    command line option, 310
--erode <size>
    command line option, 331
--evaloff
    command line option, 304
--evalon
    command line option, 304
--fail <A> --failpercent <B>
    --hardfail <C>
    command line option, 302
--fft
    command line option, 333
--fill <size>
    command line option, 335
--fillholes
    command line option, 334
--filter <name>
    command line option, 360
--fit <size>
    command line option, 328
--fixnan <strategy>
    command line option, 333, 341
--fixnan <strategy>
    command line option, 365
--fixnan strategy
    command line option, 361
--flatten
    command line option, 340
--flip
    command line option, 324
--flop
    command line option, 324
--format <formatname>
    command line option, 360
--framepadding <n>
    command line option, 303
--frames <seq>
    command line option, 303
--fullpixels
    command line option, 311, 361, 365
--fullsize <size>
    command line option, 311
--hash
    command line option, 302
--help
    command line option, 301, 359, 365
--hicomp
    command line option, 361
--iconfig <name> <value>
    command line option, 306
--iff
    command line option, 333
--ignore-unassoc
    command line option, 362
--info
    command line option, 301, 341
--invert
    command line option, 318
--iscolospace <colospace>
    command line option, 337
--kernel <name> <size>
    command line option, 315
--keyword <text>
    command line option, 310
--label <name>
    command line option, 312
--laplacian
    command line option, 332
--lightprobe

```

command line option, 363  
--line <x1,y1,x2,y2,...>  
command line option, 334  
--mad  
command line option, 318  
--median <size>  
command line option, 331  
--metamatch <regex>  
command line option, 302  
--metamerge  
command line option, 309  
--mipimage <filename>  
command line option, 363  
--monochrome-detect  
command line option, 362  
--mosaic <size>  
command line option, 322  
--mul  
command line option, 317  
--mulc  
command line option, 340  
--native  
command line option, 305  
--nchannels <n>  
command line option, 361  
--no-clobber  
command line option, 303  
--no-metamatch <regex>  
command line option, 302  
--noautocrop  
command line option, 309  
--noise  
command line option, 318  
--nomipmap  
command line option, 361  
--nosoftwareattrib  
command line option, 310  
--ociodisplay <displayname> <viewname>  
command line option, 338  
--ociofiletransform <name>  
command line option, 339  
--ociolook <lookname>  
command line option, 338  
--oio  
command line option, 363  
--opaque-detect  
command line option, 362  
--orient180  
command line option, 310  
--orientation <orient>  
command line option, 310  
--orientccw  
command line option, 310  
--orientcw  
command line option, 310  
--origin <neworigin>  
command line option, 310  
--originoffset <offset>  
command line option, 311  
--over  
command line option, 322  
--paste <location>  
command line option, 321  
--paste <position>  
command line option, 340  
--pattern <patternname> <size>  
<channels>  
command line option, 313  
--pdiff  
command line option, 302  
--pixelaspect <aspect>  
command line option, 329  
--planarconfig <config>  
command line option, 308  
--planarconfig separate  
command line option, 365  
--polar  
command line option, 333  
--pop  
command line option, 312  
--powc <value>  
command line option, 318  
--premult  
command line option, 339  
--prman  
command line option, 363  
--prman-metadata  
command line option, 362  
--quality <q>  
command line option, 308  
--rangecheck Rlow,Glow,Blow,...  
Rhi,Bhi,Ghi,...  
command line option, 303  
--rangecompress  
command line option, 334  
--rangeexpand  
command line option, 334  
--reorient  
command line option, 325  
--resample <size>  
command line option, 327, 340  
--resize  
command line option, 360  
--resize <size>  
command line option, 328  
--rotate <angle>  
command line option, 329  
--rotatel80



command line option, 323  
 --rotate270  
     command line option, 323  
 --rotate90  
     command line option, 322  
 --runstats  
     command line option, 301, 359, 365  
 --sansattrib  
     command line option, 310, 362  
 --sattrib <name> <value>  
     command line option, 309, 362  
 --scanline  
     command line option, 308, 341  
 --selectmip <level>  
     command line option, 311  
 --separate  
     command line option, 360  
 --sharpen <contrast>  
     command line option, 361  
 --siappend  
     command line option, 312  
 --siappendall  
     command line option, 312  
 --sisplit  
     command line option, 311  
 --stats  
     command line option, 302, 341  
 --sub  
     command line option, 317  
 --subc  
     command line option, 340  
 --subimage <n>  
     command line option, 311  
 --swap  
     command line option, 312  
 --swrap <wrapmode>  
     command line option, 360  
 --text <words>  
     command line option, 336  
 --threads <n>  
     command line option, 303, 360, 365  
 --tile <x> <y>  
     command line option, 308, 341, 360, 365  
 --tocolorspace <tospace>  
     command line option, 337  
 --transpose  
     command line option, 325  
 --trim  
     command line option, 327, 341  
 --twrap <wrapmode>  
     command line option, 360  
 --unmip  
     command line option, 311  
 --unpolar  
     command line option, 333  
 --unpremult  
     command line option, 339, 363  
 --unsharp  
     command line option, 332  
 --views <name1,name2,...>  
     command line option, 304  
 --warn <A> --warnpercent <B>  
     --hardwarn <C>  
     command line option, 302  
 --warp <M33>  
     command line option, 330  
 --wildcardoff  
     command line option, 304  
 --wildcardon  
     command line option, 304  
 --wrap <wrapmode>  
     command line option, 360  
 --zover  
     command line option, 322  
 -a  
     command line option, 301  
 -d <datatype>  
     command line option, 308, 360, 365  
 -i <filename>  
     command line option, 304  
 -n  
     command line option, 301  
 -no-autopremult  
     command line option, 304  
 -o <filename>  
     command line option, 306  
 -o outputname  
     command line option, 359  
 -obump <filename>  
     command line option, 307  
 -oenv <filename>  
     command line option, 307  
 -otex <filename>  
     command line option, 307  
 -q  
     command line option, 301  
 -u  
     command line option, 360  
 -v  
     command line option, 301, 359, 365  
 <filename>  
     command line option, 304

## A

A\_channel (*DeepData attribute*), 252  
 AB\_channel (*DeepData attribute*), 252  
 add (*C++ function*), 241  
 AG\_channel (*DeepData attribute*), 252

AGGREGATE (*built-in class*), 243  
aggregate (*TypeDesc attribute*), 244  
All (*ROI attribute*), 245  
allocated() (*DeepData method*), 252  
alpha\_channel (*ImageSpec attribute*), 247  
anisotropic (*C++ member*), 168  
AR\_channel (*DeepData attribute*), 252  
arraylen (*TypeDesc attribute*), 244  
attribute(), 285  
attribute() (*ImageSpec method*), 249

## B

BASETYPE (*built-in class*), 243  
basetype (*TypeDesc attribute*), 244  
built-in function  
    set\_roi(), 246  
    set\_roi\_full(), 246  
    str(), 244

## C

Channel, 381  
channel\_bytes() (*ImageSpec method*), 248  
channel\_name() (*ImageSpec method*), 250  
channelformat() (*ImageSpec method*), 250  
channelformats (*ImageSpec attribute*), 247  
channelindex() (*ImageSpec method*), 248, 250  
channelname() (*DeepData method*), 252  
channelnames (*ImageSpec attribute*), 247  
channels (*C++ function*), 241  
channels (*DeepData attribute*), 252  
channelsize() (*DeepData method*), 252  
channeltype() (*DeepData method*), 252  
chbegin (*ROI attribute*), 245  
chend (*ROI attribute*), 245  
clear() (*ImageBuf method*), 261  
Client, 381  
close() (*ImageInput method*), 253  
close() (*ImageOutput method*), 258  
command line option  
    "Artist" : string, 367  
    "CompressionQuality" : int, 369  
    "Copyright" : string, 367  
    "DateTime" : string, 367  
    "DocumentName" : string, 367  
    "Exif:ApertureValue" : float, 372  
    "Exif:BrightnessValue" : float, 372  
    "Exif:CompressedBitsPerPixel" :  
        float, 372  
    "Exif:Contrast" : int, 375  
    "Exif:CustomRendered" : int, 374  
    "Exif:DateTimeDigitized" : string,  
        372  
    "Exif:DateTimeOriginal" : string,  
        372

    "Exif:DigitalZoomRatio" : float,  
        375  
    "Exif:ExposureBiasValue" : float,  
        372  
    "Exif:ExposureIndex" : float, 374  
    "Exif:ExposureMode" : int, 374  
    "Exif:ExposureProgram" : int, 371  
    "Exif:FileSource" : int, 374  
    "Exif:Flash" int}, 373  
    "Exif:FlashEnergy" : float, 374  
    "Exif:FocalLength" : float, 373  
    "Exif:FocalLengthIn35mmFilm" :  
        int, 375  
    "Exif:FocalPlaneResolutionUnit" :  
        int, 374  
    "Exif:FocalPlaneXResolution" :  
        float, 374  
    "Exif:FocalPlaneYResolution" :  
        float, 374  
    "Exif:GainControl" : float, 375  
    "Exif:ISOSpeedRatings" : int, 372  
    "Exif:ImageHistory" : string, 373  
    "Exif:ImageUniqueID" : string, 376  
    "Exif:LightSource" : int, 373  
    "Exif:MaxApertureValue" : float,  
        372  
    "Exif:MeteringMode" : int, 372  
    "Exif:PixelXDimension" : int, 374  
    "Exif:PixelYDimension" : int, 374  
    "Exif:Saturation" : int, 375  
    "Exif:SceneCaptureType" : int, 375  
    "Exif:SceneType" : int, 374  
    "Exif:SecurityClassification" :  
        string, 373  
    "Exif:SensingMethod" : int, 374  
    "Exif:Sharpness" : int, 375  
    "Exif:ShutterSpeedValue" : float,  
        372  
    "Exif:SpectralSensitivity" :  
        string, 372  
    "Exif:SubjectDistance" : float, 372  
    "Exif:SubjectDistanceRange" : int,  
        375  
    "Exif:SubsecTime" : string, 373  
    "Exif:SubsecTimeDigitized" :  
        string, 373  
    "Exif:SubsecTimeOriginal" :  
        string, 373  
    "Exif:WhiteBalance" : int, 375  
    "ExposureTime" : float, 370  
    "FNumber" : float, 370  
    "FramesPerSecond" : rational, 368  
    "GPS:Altitude" : float, 376  
    "GPS:AltitudeRef" : string, 376

```

"GPS:AreaInformation" : string,377
"GPS:DOP" : float,376
"GPS:DateStamp" : string,377
"GPS:DestBearing" : float,377
"GPS:DestBearingRef" : string,377
"GPS:DestDistance" : float,377
"GPS:DestDistanceRef" : string,377
"GPS:DestLatitude" : float[3],377
"GPS:DestLatitudeRef" : string,377
"GPS:DestLongitude" : float[3],377
"GPS:DestLongitudeRef" : string,
377
"GPS:Differential" : int,377
"GPS:HPositioningError" : float,
377
"GPS:ImgDirection" : float,377
"GPS:ImgDirectionRef" : string,376
"GPS:Latitude" : float[3],376
"GPS:LatitudeRef" : string,376
"GPS:Longitude" : float[3],376
"GPS:LongitudeRef" : string,376
"GPS:MapDatum" : string,377
"GPS:MeasureMode" : string,376
"GPS:ProcessingMethod" : string,
377
"GPS:Satellites" : string,376
"GPS:Speed" : float,376
"GPS:SpeedRef" : string,376
"GPS:Status" : string,376
"GPS:TimeStamp" : float[3],376
"GPS:Track" : float,376
"GPS:TrackRef" : string,376
"HostComputer" : string,367
"ICCPProfile" : uint8[],369
"IPTC:AuthorsPosition" : string,
379
"IPTC:CaptionWriter" : string,379
"IPTC:Category" : string,378
"IPTC:City" : string,379
"IPTC:ClassifyState" : string,379
"IPTC:Contact" : string,379
"IPTC:ContentLocationCode" :
string,378
"IPTC:ContentLocationName" :
string,378
"IPTC:Country" : string,379
"IPTC:CountryCode" : string,379
"IPTC:DateCreated" : string,378
"IPTC:DigitalCreationDate" :
string,378
"IPTC:DigitalCreationTime" :
string,378
"IPTC:DocumentHistory" : string,
379
"IPTC:DocumentNotes" : string,379
"IPTC:EditStatus" : string,378
"IPTC:ExpirationDate" : string,378
"IPTC:ExpirationTime" : string,378
"IPTC:Headline" : string,379
"IPTC:Instructions" : string,378
"IPTC:JobID" : string,379
"IPTC:MasterDocumentID" : string,
379
"IPTC:ObjectTypeReference" :
string,378
"IPTC:ObjectAttributeReference" :
string,378
"IPTC:ObjectName" : string,378
"IPTC:OwnerID" : string,379
"IPTC:Prefs" : string,379
"IPTC:ProgramVersion" : string,378
"IPTC:Provider" : string,379
"IPTC:ReferenceDate" : string,378
"IPTC:ReferenceNumber" : string,
378
"IPTC:ReferenceService" : string,
378
"IPTC:ReleaseDate" : string,378
"IPTC:ReleaseTime" : string,378
"IPTC:ShortDocumentID" : string,
379
"IPTC:SimilarityIndex" : string,
379
"IPTC:Source" : string,379
"IPTC:State" : string,379
"IPTC:SubjectReference" : string,
378
"IPTC:Sublocation" : string,379
"IPTC:TimeCreated" : string,378
"IPTC:UniqueDocumentID" : string,
379
"ImageDescription" : string,367
"Keywords" : string,367
"Make" : string,370
"Model" : string,370
"Orientation" : int,368
"PixelAspectRatio" : float,368
"ResolutionUnit" : string,368
"Software" : string,367
"XResolution" : float,368
"YResolution" : float,368
"compression" : string,369
"fovco" : float,370
"oiio:AverageColor" : string,371
"oiio:BitsPerSample" : int,369
"oiio:BorderColor" :
float[nchannels],369
"oiio:ColorSpace" : string,368

```

```
"oiio:ConstantColor" : string, 371
"oiio:Gamma" : float, 369
"oiio:Movie" : int, 368
"oiio:SHA-1" : string, 371
"oiio:UnassociatedAlpha" : int, 369
"oiio:ioproxy" : pointer, 370
"oiio:sampleborder" : int, 371
"oiio:subimages" : int, 368
"oiio:updirection" : string, 371
"planarconfig" : string, 369
"smppte:KeyCode" : int[7], 379
"smppte:TimeCode" : int[2], 379
"textureformat" : string, 370
"worldtocamera" : matrix44, 371
"worldtoscreen" : matrix44, 371
"wrapmodes" : string, 370
-- divc <value>, 317
-- mulc <value>, 317
-- subc <value>, 317
--Mcamera <...16 floats...>, 362
--Mscreen <...16 floats...>, 362
--abs, 318
--absdiff, 318
--absdiffc <value>, 318
--add, 316
--addc, 340
--addc <value>, 316
--adjust-time, 308
--attrib <name> <value>, 309, 362, 365
--attrib:type=matrix worldtocam
    <...16 comma-separated
    floats...>, 365
--autocc, 305
--autoorient, 305
--autopremult, 304
--autotile <tilesize>, 306
--autotrim, 309, 340
--blur <size>, 330
--box <x1,y1,x2,y2>, 335
--bumpformat <bformat>, 364
--bumpslopes, 363
--cache <size>, 306
--caption <text>, 309
--capture, 316
--ccmatrix <m00,m01,...>, 337
--ch <channellist>, 312, 340, 365
--chappend, 312
--checknan, 361
--chnames <name-list>, 311
--chnames a,b,..., 361, 365
--chsum, 319
--clamp, 333
--clear-keywords, 310
--colorconfig <filename>, 337
--colorconfig <name>, 363
--colorconvert <fromspace tospace>,
    337
--colorconvert <inspace>
    <outspace>, 363
--colorcount r1,g1,b1,...:r2,g2,b2,...:...,
    302
--colormap <mapname>, 321
--compression <method>, 308, 360, 365
--constant-color-detect, 362
--contrast, 320
--convolve, 330
--create <size> <channels>, 313
--crop <size>, 326, 340
--croptofull, 326
--cshift <offset>, 325
--cut <size>, 327
--debug, 301
--deepen, 340
--deepholdout, 340
--deepmerge, 340
--diff, 302, 341
--dilate <size>, 331
--dither, 308
--div, 317
--divc, 340
--dumpdata, 302, 341
--dup, 312
--echo <message>, 302
--envlatl, 363
--eraseattrib <pattern>, 310
--erode <size>, 331
--evaloff, 304
--evalon, 304
--fail <A> --failpercent <B>
    --hardfail <C>, 302
--fft, 333
--fill <size>, 335
--fillholes, 334
--filter <name>, 360
--fit <size>, 328
--fixnan <strategy>, 333, 341
--fixnan <strategy>, 365
--fixnan strategy, 361
--flatten, 340
--flip, 324
--flop, 324
--format <formatname>, 360
--framepadding <n>, 303
--frames <seq>, 303
--fullpixels, 311, 361, 365
--fullsize <size>, 311
--hash, 302
--help, 301, 359, 365
```

```

--hicomp, 361
--iconfig <name> <value>, 306
--ifft, 333
--ignore-unassoc, 362
--info, 301, 341
--invert, 318
--iscolorspace <colorspace>, 337
--kernel <name> <size>, 315
--keyword <text>, 310
--label <name>, 312
--laplacian, 332
--lightprobe, 363
--line <x1,y1,x2,y2,...>, 334
--mad, 318
--median <size>, 331
--metamatch <regex>, 302
--metamerge, 309
--mipimage <filename>, 363
--monochrome-detect, 362
--mosaic <size>, 322
--mul, 317
--mulc, 340
--native, 305
--nchannels <n>, 361
--no-clobber, 303
--no-metamatch <regex>, 302
--noautocrop, 309
--noise, 318
--nomipmap, 361
--nosoftwareattrib, 310
--ociodisplay <displayname>
    <viewname>, 338
--ociofiletransform <name>, 339
--ociolook <lookname>, 338
--oiio, 363
--opaque-detect, 362
--orient180, 310
--orientation <orient>, 310
--orientccw, 310
--orientcw, 310
--origin <neworigin>, 310
--originoffset <offset>, 311
--over, 322
--paste <location>, 321
--paste <position>, 340
--pattern <patternname> <size>
    <channels>, 313
--pdiff, 302
--pixelaspect <aspect>, 329
--planarconfig <config>, 308
--planarconfig separate, 365
--polar, 333
--pop, 312
--powc <value>, 318
--premult, 339
--prman, 363
--prman-metadata, 362
--quality <q>, 308
--rangecheck Rlow,Glow,Blow,...
    Rhi,Bhi,Ghi,..., 303
--rangecompress, 334
--rangeexpand, 334
--reorient, 325
--resample <size>, 327, 340
--resize, 360
--resize <size>, 328
--rotate <angle>, 329
--rotate180, 323
--rotate270, 323
--rotate90, 322
--runstats, 301, 359, 365
--sansattrib, 310, 362
--sattrib <name> <value>, 309, 362
--scanline, 308, 341
--selectmip <level>, 311
--separate, 360
--sharpen <contrast>, 361
--siappend, 312
--siappendall, 312
--sisplit, 311
--stats, 302, 341
--sub, 317
--subc, 340
--subimage <n>, 311
--swap, 312
--swrap <wrapmode>, 360
--text <words>, 336
--threads <n>, 303, 360, 365
--tile <x> <y>, 308, 341, 360, 365
--tocolorspace <tospace>, 337
--transpose, 325
--trim, 327, 341
--twrap <wrapmode>, 360
--unmip, 311
--unpolar, 333
--unpremult, 339, 363
--unsharp, 332
--views <name1,name2,...>, 304
--warn <A> --warnpercent <B>
    --hardwarn <C>, 302
--warp <M33>, 330
--wildcardoff, 304
--wildcardon, 304
--wrap <wrapmode>, 360
--zover, 322
-a, 301
-d <datatype>, 308, 360, 365
-i <filename>, 304

```

- n, 301
- no-autopremult, 304
- o <filename>, 306
- o outputname, 359
- obump <filename>, 307
- oenv <filename>, 307
- otex <filename>, 307
- q, 301
- u, 360
- v, 301, 359, 365
- <filename>, 304

compare (C++ function), 21, 241

computePixelStats (C++ function), 241

conservative\_filter (C++ member), 168

contains () (ROI method), 246

copy () (ImageBuf method), 265

copy\_deep\_pixel () (DeepData method), 253

copy\_deep\_sample () (DeepData method), 253

copy\_dimensions () (ImageSpec method), 250

copy\_image () (ImageOutput method), 260

copy\_metadata () (ImageBuf method), 265

copy\_pixels () (ImageBuf method), 265

create () (ImageOutput method), 257

crop (C++ function), 241

cspan (C++ type), 23

current\_miplevel () (ImageInput method), 254

current\_subimage () (ImageInput method), 254

## D

Data format, 381

deep (ImageBuf attribute), 267

deep (ImageSpec attribute), 247

deep\_erase\_samples () (ImageBuf method), 268

deep\_insert\_samples () (ImageBuf method), 268

deep\_samples () (ImageBuf method), 267

deep\_value () (DeepData method), 253

deep\_value () (ImageBuf method), 268

deep\_value\_uint () (DeepData method), 253

deep\_value\_uint () (ImageBuf method), 268

DeepData (), 252

default\_channel\_names () (ImageSpec method), 248

defined (ROI attribute), 245

depth (ROI attribute), 245

div (C++ function), 241

## E

environment (C++ function), 169

equivalent (C++ function), 14

equivalent () (TypeDesc method), 245

erase\_attribute () (ImageSpec method), 249

erase\_samples () (DeepData method), 252

extra\_attribs (ImageSpec attribute), 247

## F

file\_format\_name () (ImageBuf method), 263

fill (C++ member), 168

firstchannel (C++ member), 168

fixNonFinite (C++ function), 241

format (ImageSpec attribute), 247

Format Plugin, 381

format\_name () (ImageInput method), 254

format\_name () (ImageOutput method), 257

from\_xml () (ImageSpec method), 250

## G

get\_channelformats () (ImageSpec method), 250

get\_float\_attribute (C++ function), 38

get\_float\_attribute (), 285

get\_float\_attribute () (ImageSpec method), 249

get\_int\_attribute (C++ function), 38

get\_int\_attribute (), 285

get\_int\_attribute () (ImageSpec method), 249

get\_pixels () (ImageBuf method), 267

get\_roi (C++ function), 24

get\_roi\_full (C++ function), 24

get\_string\_attribute (C++ function), 38

get\_string\_attribute (), 285

get\_string\_attribute () (ImageSpec method), 249

getattribute (C++ function), 38

getattribute (), 285

getattribute () (ImageSpec method), 249

getchannel () (ImageBuf method), 266

geterror (), 285

geterror () (ImageBuf method), 267

geterror () (ImageInput method), 256

geterror () (ImageOutput method), 260

## H

has\_error (ImageBuf attribute), 267

height (ROI attribute), 245

## I

Image File Format, 381

Image Format Plugin, 381

image\_bytes () (ImageSpec method), 248

image\_pixels () (ImageSpec method), 248

ImageBuf (), 261

ImageSpec (), 246

init () (DeepData method), 252

init\_spec () (ImageBuf method), 262

initialized () (DeepData method), 252

insert\_samples () (DeepData method), 252

interpode (C++ member), 168

interpixel () (ImageBuf method), 266

interpixel\_bicubic () (ImageBuf method), 266



- interppixel\_bicubic\_NDC() (*ImageBuf method*), 266  
 interppixel\_NDC() (*ImageBuf method*), 266  
 Iterator::deep\_samples (C++ function), 184  
 Iterator::deep\_value (C++ function), 184  
 Iterator::deep\_value\_int (C++ function), 184  
 Iterator::done (C++ function), 184  
 Iterator::exists (C++ function), 184  
 Iterator::operator++ (C++ function), 183  
 Iterator::operator[] (C++ function), 184  
 Iterator::range (C++ function), 184  
 Iterator::set\_deep\_samples (C++ function), 184  
 Iterator::set\_deep\_value (C++ function), 184  
 Iterator::valid (C++ function), 184  
 Iterator::x (C++ function), 184  
 Iterator::y (C++ function), 184  
 Iterator::z (C++ function), 184  
 Iterator<BUFT> (C++ function), 183
- ## L
- localpixels (C++ function), 182
- ## M
- make\_writable() (*ImageBuf method*), 263  
 merge\_deep\_pixels() (*DeepData method*), 253  
 merge\_overlaps() (*DeepData method*), 253  
 Metadata, 381  
 metadata\_val() (*ImageSpec method*), 249  
 miplevel (*ImageBuf attribute*), 264  
 mipmode (C++ member), 168  
 missingcolor (C++ member), 168  
 mul (C++ function), 241
- ## N
- name() (*ImageBuf method*), 263  
 Native data format, 381  
 nativespec() (*ImageBuf method*), 263  
 nchannels (*ImageSpec attribute*), 247  
 nchannels (*ROI attribute*), 245  
 nmiplevels (*ImageBuf attribute*), 264  
 nonzero\_region (C++ function), 241  
 nsubimages (*ImageBuf attribute*), 264
- ## O
- occlusion\_cull() (*DeepData method*), 253  
 OIIO::attribute (C++ function), 35, 37  
 OIIO::declare\_imageio\_format (C++ function), 38  
 OIIO::DeepData (C++ class), 33  
 OIIO::DeepData::allocated (C++ function), 33  
 OIIO::DeepData::capacity (C++ function), 34  
 OIIO::DeepData::channelname (C++ function), 33  
 OIIO::DeepData::channels (C++ function), 33  
 OIIO::DeepData::channelsize (C++ function), 33  
 OIIO::DeepData::channeltype (C++ function), 33  
 OIIO::DeepData::clear (C++ function), 33  
 OIIO::DeepData::copy\_deep\_pixel (C++ function), 34  
 OIIO::DeepData::copy\_deep\_sample (C++ function), 34  
 OIIO::DeepData::data\_ptr (C++ function), 34  
 OIIO::DeepData::deep\_value (C++ function), 34  
 OIIO::DeepData::deep\_value\_uint (C++ function), 34  
 OIIO::DeepData::DeepData (C++ function), 33  
 OIIO::DeepData::erase\_samples (C++ function), 34  
 OIIO::DeepData::free (C++ function), 33  
 OIIO::DeepData::get\_pointers (C++ function), 34  
 OIIO::DeepData::init (C++ function), 33  
 OIIO::DeepData::initialized (C++ function), 33  
 OIIO::DeepData::insert\_samples (C++ function), 34  
 OIIO::DeepData::merge\_deep\_pixels (C++ function), 35  
 OIIO::DeepData::merge\_overlaps (C++ function), 35  
 OIIO::DeepData::occlusion\_cull (C++ function), 35  
 OIIO::DeepData::opaque\_z (C++ function), 35  
 OIIO::DeepData::operator= (C++ function), 33  
 OIIO::DeepData::pixels (C++ function), 33  
 OIIO::DeepData::samples (C++ function), 34  
 OIIO::DeepData::samplesize (C++ function), 33  
 OIIO::DeepData::set\_all\_samples (C++ function), 34  
 OIIO::DeepData::set\_capacity (C++ function), 34  
 OIIO::DeepData::set\_deep\_value (C++ function), 34  
 OIIO::DeepData::set\_samples (C++ function), 34  
 OIIO::DeepData::sort (C++ function), 34  
 OIIO::DeepData::split (C++ function), 34  
 OIIO::getattribute (C++ function), 37  
 OIIO::geterror (C++ function), 38  
 OIIO::ImageBuf::APPBUFFER (C++ enumerator), 171  
 OIIO::ImageBuf::contains\_roi (C++ function), 178

OIIO::ImageBuf::copy (C++ function), 179  
 OIIO::ImageBuf::copy\_metadata (C++ function), 179  
 OIIO::ImageBuf::copy\_pixels (C++ function), 179  
 OIIO::ImageBuf::deep (C++ function), 181  
 OIIO::ImageBuf::deep\_erase\_samples (C++ function), 181  
 OIIO::ImageBuf::deep\_insert\_samples (C++ function), 181  
 OIIO::ImageBuf::deep\_pixel\_ptr (C++ function), 182  
 OIIO::ImageBuf::deep\_samples (C++ function), 181  
 OIIO::ImageBuf::deep\_value (C++ function), 182  
 OIIO::ImageBuf::deep\_value\_uint (C++ function), 182  
 OIIO::ImageBuf::deepdata (C++ function), 182  
 OIIO::ImageBuf::errorf (C++ function), 182  
 OIIO::ImageBuf::file\_format\_name (C++ function), 177  
 OIIO::ImageBuf::get\_pixels (C++ function), 181  
 OIIO::ImageBuf::getchannel (C++ function), 180  
 OIIO::ImageBuf::geterror (C++ function), 182  
 OIIO::ImageBuf::getpixel (C++ function), 180  
 OIIO::ImageBuf::has\_error (C++ function), 182  
 OIIO::ImageBuf::IBStorage (C++ enum), 171  
 OIIO::ImageBuf::ImageBuf (C++ function), 172, 173  
 OIIO::ImageBuf::IMAGECACHE (C++ enumerator), 171  
 OIIO::ImageBuf::init\_spec (C++ function), 175  
 OIIO::ImageBuf::initialized (C++ function), 177  
 OIIO::ImageBuf::interppixel (C++ function), 180  
 OIIO::ImageBuf::interppixel\_bicubic (C++ function), 180  
 OIIO::ImageBuf::interppixel\_bicubic\_NDC (C++ function), 181  
 OIIO::ImageBuf::interppixel\_NDC (C++ function), 180  
 OIIO::ImageBuf::LOCALBUFFER (C++ enumerator), 171  
 OIIO::ImageBuf::make\_writeable (C++ function), 173  
 OIIO::ImageBuf::miplevel (C++ function), 177  
 OIIO::ImageBuf::name (C++ function), 177  
 OIIO::ImageBuf::nativespec (C++ function), 177  
 OIIO::ImageBuf::nchannels (C++ function), 177  
 OIIO::ImageBuf::nmiplevels (C++ function), 177  
 OIIO::ImageBuf::nsubimages (C++ function), 177  
 OIIO::ImageBuf::operator= (C++ function), 179  
 OIIO::ImageBuf::orientation (C++ function), 178  
 OIIO::ImageBuf::pixelindex (C++ function), 182  
 OIIO::ImageBuf::pixeltype (C++ function), 178  
 OIIO::ImageBuf::read (C++ function), 174  
 OIIO::ImageBuf::reset (C++ function), 172–174  
 OIIO::ImageBuf::roi (C++ function), 178  
 OIIO::ImageBuf::roi\_full (C++ function), 178  
 OIIO::ImageBuf::set\_deep\_samples (C++ function), 181  
 OIIO::ImageBuf::set\_deep\_value (C++ function), 182  
 OIIO::ImageBuf::set\_full (C++ function), 178  
 OIIO::ImageBuf::set\_orientation (C++ function), 178  
 OIIO::ImageBuf::set\_origin (C++ function), 178  
 OIIO::ImageBuf::set\_pixels (C++ function), 181  
 OIIO::ImageBuf::set\_roi\_full (C++ function), 178  
 OIIO::ImageBuf::set\_write\_format (C++ function), 176  
 OIIO::ImageBuf::set\_write\_tiles (C++ function), 176  
 OIIO::ImageBuf::setpixel (C++ function), 181  
 OIIO::ImageBuf::spec (C++ function), 177  
 OIIO::ImageBuf::specmod (C++ function), 177  
 OIIO::ImageBuf::storage (C++ function), 177  
 OIIO::ImageBuf::subimage (C++ function), 177  
 OIIO::ImageBuf::swap (C++ function), 179  
 OIIO::ImageBuf::threads (C++ function), 178  
 OIIO::ImageBuf::UNINITIALIZED (C++ enumerator), 171  
 OIIO::ImageBuf::WrapMode\_from\_string (C++ function), 183  
 OIIO::ImageBuf::write (C++ function), 175, 176  
 OIIO::ImageBufAlgo::abs (C++ function), 208  
 OIIO::ImageBufAlgo::absdiff (C++ function), 208  
 OIIO::ImageBufAlgo::add (C++ function), 207  
 OIIO::ImageBufAlgo::capture\_image (C++ function), 238



OIIO::ImageBufAlgo::channel\_append (C++ function), 198, 199  
 OIIO::ImageBufAlgo::channel\_sum (C++ function), 213  
 OIIO::ImageBufAlgo::channels (C++ function), 197, 198  
 OIIO::ImageBufAlgo::checker (C++ function), 192, 193  
 OIIO::ImageBufAlgo::circular\_shift (C++ function), 202, 203  
 OIIO::ImageBufAlgo::clamp (C++ function), 213, 214  
 OIIO::ImageBufAlgo::color\_count (C++ function), 220  
 OIIO::ImageBufAlgo::color\_map (C++ function), 216  
 OIIO::ImageBufAlgo::color\_range\_check (C++ function), 221  
 OIIO::ImageBufAlgo::colorconvert (C++ function), 230, 231  
 OIIO::ImageBufAlgo::colormatrixtransform (C++ function), 231, 232  
 OIIO::ImageBufAlgo::compare (C++ function), 218  
 OIIO::ImageBufAlgo::compare\_Yee (C++ function), 219  
 OIIO::ImageBufAlgo::complex\_to\_polar (C++ function), 226  
 OIIO::ImageBufAlgo::computePixelHashSHA1 (C++ function), 222  
 OIIO::ImageBufAlgo::computePixelStats (C++ function), 217  
 OIIO::ImageBufAlgo::contrast\_remap (C++ function), 214, 215  
 OIIO::ImageBufAlgo::convolve (C++ function), 223  
 OIIO::ImageBufAlgo::copy (C++ function), 199  
 OIIO::ImageBufAlgo::crop (C++ function), 199, 200  
 OIIO::ImageBufAlgo::cut (C++ function), 200  
 OIIO::ImageBufAlgo::deep\_holdout (C++ function), 240  
 OIIO::ImageBufAlgo::deep\_merge (C++ function), 240  
 OIIO::ImageBufAlgo::deepen (C++ function), 239  
 OIIO::ImageBufAlgo::dilate (C++ function), 228  
 OIIO::ImageBufAlgo::div (C++ function), 209  
 OIIO::ImageBufAlgo::erode (C++ function), 228, 229  
 OIIO::ImageBufAlgo::fft (C++ function), 225  
 OIIO::ImageBufAlgo::fill (C++ function), 192  
 OIIO::ImageBufAlgo::fillholes\_pushpull (C++ function), 227  
 OIIO::ImageBufAlgo::fit (C++ function), 206  
 OIIO::ImageBufAlgo::fixNonFinite (C++ function), 226  
 OIIO::ImageBufAlgo::flatten (C++ function), 239  
 OIIO::ImageBufAlgo::flip (C++ function), 202  
 OIIO::ImageBufAlgo::flop (C++ function), 202  
 OIIO::ImageBufAlgo::from\_OpenCV (C++ function), 238  
 OIIO::ImageBufAlgo::histogram (C++ function), 222  
 OIIO::ImageBufAlgo::ifft (C++ function), 225  
 OIIO::ImageBufAlgo::invert (C++ function), 211, 212  
 OIIO::ImageBufAlgo::isConstantChannel (C++ function), 219  
 OIIO::ImageBufAlgo::isConstantColor (C++ function), 219  
 OIIO::ImageBufAlgo::isMonochrome (C++ function), 220  
 OIIO::ImageBufAlgo::laplacian (C++ function), 224  
 OIIO::ImageBufAlgo::mad (C++ function), 209, 210  
 OIIO::ImageBufAlgo::make\_kernel (C++ function), 223  
 OIIO::ImageBufAlgo::make\_texture (C++ function), 237  
 OIIO::ImageBufAlgo::median\_filter (C++ function), 227  
 OIIO::ImageBufAlgo::mul (C++ function), 209  
 OIIO::ImageBufAlgo::noise (C++ function), 193, 194  
 OIIO::ImageBufAlgo::nonzero\_region (C++ function), 221  
 OIIO::ImageBufAlgo::ociodisplay (C++ function), 233  
 OIIO::ImageBufAlgo::ociofiletransform (C++ function), 234  
 OIIO::ImageBufAlgo::ociolook (C++ function), 232  
 OIIO::ImageBufAlgo::over (C++ function), 210, 211  
 OIIO::ImageBufAlgo::paste (C++ function), 200  
 OIIO::ImageBufAlgo::polar\_to\_complex (C++ function), 226  
 OIIO::ImageBufAlgo::pow (C++ function), 212  
 OIIO::ImageBufAlgo::premult (C++ function), 235  
 OIIO::ImageBufAlgo::rangecompress (C++ function), 217  
 OIIO::ImageBufAlgo::rangeexpand (C++

*function*), 217  
 OIIO::ImageBufAlgo::render\_box (C++ *function*), 195  
 OIIO::ImageBufAlgo::render\_line (C++ *function*), 194  
 OIIO::ImageBufAlgo::render\_point (C++ *function*), 194  
 OIIO::ImageBufAlgo::render\_text (C++ *function*), 196  
 OIIO::ImageBufAlgo::reorient (C++ *function*), 202  
 OIIO::ImageBufAlgo::resample (C++ *function*), 205  
 OIIO::ImageBufAlgo::resize (C++ *function*), 205  
 OIIO::ImageBufAlgo::rotate (C++ *function*), 203, 204  
 OIIO::ImageBufAlgo::rotate180 (C++ *function*), 201  
 OIIO::ImageBufAlgo::rotate270 (C++ *function*), 201  
 OIIO::ImageBufAlgo::rotate90 (C++ *function*), 201  
 OIIO::ImageBufAlgo::sub (C++ *function*), 207, 208  
 OIIO::ImageBufAlgo::text\_size (C++ *function*), 197  
 OIIO::ImageBufAlgo::to\_OpenCV (C++ *function*), 238  
 OIIO::ImageBufAlgo::transpose (C++ *function*), 202  
 OIIO::ImageBufAlgo::unpremult (C++ *function*), 235  
 OIIO::ImageBufAlgo::unsharp\_mask (C++ *function*), 227, 228  
 OIIO::ImageBufAlgo::warp (C++ *function*), 206, 207  
 OIIO::ImageBufAlgo::zero (C++ *function*), 191  
 OIIO::ImageBufAlgo::zover (C++ *function*), 211  
 OIIO::ImageCache (C++ *class*), 138  
 OIIO::ImageCache::add\_file (C++ *function*), 149  
 OIIO::ImageCache::add\_tile (C++ *function*), 149  
 OIIO::ImageCache::attribute (C++ *function*), 141, 142  
 OIIO::ImageCache::close (C++ *function*), 148  
 OIIO::ImageCache::close\_all (C++ *function*), 148  
 OIIO::ImageCache::create (C++ *function*), 138  
 OIIO::ImageCache::create\_thread\_info (C++ *function*), 143  
 OIIO::ImageCache::destroy (C++ *function*), 138  
 OIIO::ImageCache::destroy\_thread\_info (C++ *function*), 143  
 OIIO::ImageCache::get\_image\_handle (C++ *function*), 143  
 OIIO::ImageCache::get\_image\_info (C++ *function*), 144, 146  
 OIIO::ImageCache::get\_imagespec (C++ *function*), 146  
 OIIO::ImageCache::get\_perthread\_info (C++ *function*), 143  
 OIIO::ImageCache::get\_pixels (C++ *function*), 147  
 OIIO::ImageCache::get\_tile (C++ *function*), 148  
 OIIO::ImageCache::getattribute (C++ *function*), 142  
 OIIO::ImageCache::geterror (C++ *function*), 150  
 OIIO::ImageCache::getstats (C++ *function*), 150  
 OIIO::ImageCache::good (C++ *function*), 143  
 OIIO::ImageCache::ImageHandle (C++ *type*), 143  
 OIIO::ImageCache::imagespec (C++ *function*), 146, 147  
 OIIO::ImageCache::invalidate (C++ *function*), 148  
 OIIO::ImageCache::invalidate\_all (C++ *function*), 148  
 OIIO::ImageCache::Perthread (C++ *type*), 143  
 OIIO::ImageCache::release\_tile (C++ *function*), 148  
 OIIO::ImageCache::reset\_stats (C++ *function*), 150  
 OIIO::ImageCache::resolve\_filename (C++ *function*), 144  
 OIIO::ImageCache::tile\_format (C++ *function*), 148  
 OIIO::ImageCache::tile\_pixels (C++ *function*), 149  
 OIIO::ImageCache::tile\_roi (C++ *function*), 149  
 OIIO::ImageInput (C++ *class*), 78  
 OIIO::ImageInput::close (C++ *function*), 86  
 OIIO::ImageInput::create (C++ *function*), 78  
 OIIO::ImageInput::Creator (C++ *type*), 84  
 OIIO::ImageInput::current\_miplevel (C++ *function*), 86  
 OIIO::ImageInput::current\_subimage (C++ *function*), 86  
 OIIO::ImageInput::error (C++ *function*), 86  
 OIIO::ImageInput::errorf (C++ *function*), 86  
 OIIO::ImageInput::fmterror (C++ *function*),

- 86
- OIIO::ImageInput::format\_name (C++ function), 84
- OIIO::ImageInput::geterror (C++ function), 86
- OIIO::ImageInput::lock (C++ function), 87
- OIIO::ImageInput::open (C++ function), 78, 85
- OIIO::ImageInput::read\_image (C++ function), 82
- OIIO::ImageInput::read\_native\_deep\_image (C++ function), 83
- OIIO::ImageInput::read\_native\_deep\_scanline (C++ function), 82
- OIIO::ImageInput::read\_native\_deep\_tiles (C++ function), 83
- OIIO::ImageInput::read\_native\_scanline (C++ function), 83
- OIIO::ImageInput::read\_native\_scanlines (C++ function), 83
- OIIO::ImageInput::read\_native\_tile (C++ function), 83
- OIIO::ImageInput::read\_native\_tiles (C++ function), 84
- OIIO::ImageInput::read\_scanline (C++ function), 79, 80
- OIIO::ImageInput::read\_scanlines (C++ function), 80
- OIIO::ImageInput::read\_tile (C++ function), 80, 81
- OIIO::ImageInput::read\_tiles (C++ function), 81
- OIIO::ImageInput::seek\_subimage (C++ function), 86
- OIIO::ImageInput::spec (C++ function), 85
- OIIO::ImageInput::spec\_dimensions (C++ function), 85
- OIIO::ImageInput::supports (C++ function), 84
- OIIO::ImageInput::threads (C++ function), 86
- OIIO::ImageInput::try\_lock (C++ function), 87
- OIIO::ImageInput::unique\_ptr (C++ type), 84
- OIIO::ImageInput::unlock (C++ function), 87
- OIIO::ImageInput::valid\_file (C++ function), 84
- OIIO::ImageOutput (C++ class), 59
- OIIO::ImageOutput::AppendMIPLevel (C++ enumerator), 59
- OIIO::ImageOutput::AppendSubimage (C++ enumerator), 59
- OIIO::ImageOutput::close (C++ function), 61
- OIIO::ImageOutput::copy\_image (C++ function), 66
- OIIO::ImageOutput::Create (C++ enumerator), 59
- OIIO::ImageOutput::create (C++ function), 59
- OIIO::ImageOutput::Creator (C++ type), 65
- OIIO::ImageOutput::error (C++ function), 66
- OIIO::ImageOutput::errorf (C++ function), 66
- OIIO::ImageOutput::fmterror (C++ function), 66
- OIIO::ImageOutput::format\_name (C++ function), 66
- OIIO::ImageOutput::geterror (C++ function), 66
- OIIO::ImageOutput::open (C++ function), 60, 61
- OIIO::ImageOutput::OpenMode (C++ enum), 59
- OIIO::ImageOutput::spec (C++ function), 61
- OIIO::ImageOutput::supports (C++ function), 59
- OIIO::ImageOutput::threads (C++ function), 66
- OIIO::ImageOutput::unique\_ptr (C++ type), 65
- OIIO::ImageOutput::write\_deep\_image (C++ function), 65
- OIIO::ImageOutput::write\_deep\_scanlines (C++ function), 65
- OIIO::ImageOutput::write\_deep\_tiles (C++ function), 65
- OIIO::ImageOutput::write\_image (C++ function), 64
- OIIO::ImageOutput::write\_rectangle (C++ function), 63
- OIIO::ImageOutput::write\_scanline (C++ function), 62
- OIIO::ImageOutput::write\_scanlines (C++ function), 62
- OIIO::ImageOutput::write\_tile (C++ function), 62
- OIIO::ImageOutput::write\_tiles (C++ function), 63
- OIIO::ImageSpec (C++ class), 25
- OIIO::ImageSpec::alpha\_channel (C++ member), 27
- OIIO::ImageSpec::attribute (C++ function), 28, 29
- OIIO::ImageSpec::auto\_stride (C++ function), 32
- OIIO::ImageSpec::channel\_bytes (C++ function), 27
- OIIO::ImageSpec::channel\_name (C++ function), 31
- OIIO::ImageSpec::channelformat (C++ function), 31
- OIIO::ImageSpec::channelformats (C++ member), 26

OIIO::ImageSpec::channelindex (C++ function), 31  
 OIIO::ImageSpec::channelnames (C++ member), 27  
 OIIO::ImageSpec::copy\_dimensions (C++ function), 31  
 OIIO::ImageSpec::decode\_compression\_metadata (C++ function), 31  
 OIIO::ImageSpec::deep (C++ member), 27  
 OIIO::ImageSpec::default\_channel\_names (C++ function), 27  
 OIIO::ImageSpec::depth (C++ member), 26  
 OIIO::ImageSpec::erase\_attribute (C++ function), 29  
 OIIO::ImageSpec::extra\_attribs (C++ member), 27  
 OIIO::ImageSpec::find\_attribute (C++ function), 29  
 OIIO::ImageSpec::format (C++ member), 26  
 OIIO::ImageSpec::from\_xml (C++ function), 31  
 OIIO::ImageSpec::full\_depth (C++ member), 26  
 OIIO::ImageSpec::full\_height (C++ member), 26  
 OIIO::ImageSpec::full\_width (C++ member), 26  
 OIIO::ImageSpec::full\_x (C++ member), 26  
 OIIO::ImageSpec::full\_y (C++ member), 26  
 OIIO::ImageSpec::full\_z (C++ member), 26  
 OIIO::ImageSpec::get\_channelformats (C++ function), 31  
 OIIO::ImageSpec::get\_float\_attribute (C++ function), 30  
 OIIO::ImageSpec::get\_int\_attribute (C++ function), 30  
 OIIO::ImageSpec::get\_string\_attribute (C++ function), 31  
 OIIO::ImageSpec::getattribute (C++ function), 30  
 OIIO::ImageSpec::getattributetype (C++ function), 30  
 OIIO::ImageSpec::height (C++ member), 26  
 OIIO::ImageSpec::image\_bytes (C++ function), 28  
 OIIO::ImageSpec::image\_pixels (C++ function), 28  
 OIIO::ImageSpec::ImageSpec (C++ function), 27  
 OIIO::ImageSpec::metadata\_val (C++ function), 32  
 OIIO::ImageSpec::nchannels (C++ member), 26  
 OIIO::ImageSpec::operator[] (C++ function), 32  
 OIIO::ImageSpec::pixel\_bytes (C++ function), 28  
 OIIO::ImageSpec::roi (C++ function), 31  
 OIIO::ImageSpec::roi\_full (C++ function), 31  
 OIIO::ImageSpec::scanline\_bytes (C++ function), 28  
 OIIO::ImageSpec::serialize (C++ function), 31  
 OIIO::ImageSpec::set\_format (C++ function), 27  
 OIIO::ImageSpec::set\_roi (C++ function), 31  
 OIIO::ImageSpec::set\_roi\_full (C++ function), 31  
 OIIO::ImageSpec::size\_t\_safe (C++ function), 28  
 OIIO::ImageSpec::tile\_bytes (C++ function), 28  
 OIIO::ImageSpec::tile\_depth (C++ member), 26  
 OIIO::ImageSpec::tile\_height (C++ member), 26  
 OIIO::ImageSpec::tile\_pixels (C++ function), 28  
 OIIO::ImageSpec::tile\_width (C++ member), 26  
 OIIO::ImageSpec::to\_xml (C++ function), 31  
 OIIO::ImageSpec::undefined (C++ function), 32  
 OIIO::ImageSpec::valid\_tile\_range (C++ function), 31  
 OIIO::ImageSpec::width (C++ member), 26  
 OIIO::ImageSpec::x (C++ member), 26  
 OIIO::ImageSpec::y (C++ member), 26  
 OIIO::ImageSpec::z (C++ member), 26  
 OIIO::ImageSpec::z\_channel (C++ member), 27  
 OIIO::openimageio\_version (C++ function), 38  
 OIIO::ROI (C++ struct), 23  
 OIIO::ROI::All (C++ function), 24  
 OIIO::ROI::contains (C++ function), 24  
 OIIO::ROI::defined (C++ function), 24  
 OIIO::ROI::depth (C++ function), 23  
 OIIO::ROI::height (C++ function), 23  
 OIIO::ROI::nchannels (C++ function), 24  
 OIIO::ROI::npixels (C++ function), 24  
 OIIO::ROI::ROI (C++ function), 24  
 OIIO::ROI::width (C++ function), 23  
 OIIO::roi\_intersection (C++ function), 24  
 OIIO::roi\_union (C++ function), 24  
 OIIO::span (C++ class), 21  
 OIIO::span::first (C++ function), 22  
 OIIO::span::last (C++ function), 22  
 OIIO::span::operator= (C++ function), 22  
 OIIO::span::span (C++ function), 22

OIIO::string\_view (C++ class), 14  
 OIIO::string\_view::at (C++ function), 15  
 OIIO::string\_view::c\_str (C++ function), 15  
 OIIO::string\_view::empty (C++ function), 15  
 OIIO::string\_view::find (C++ function), 15, 16  
 OIIO::string\_view::operator  
     std::string (C++ function), 15  
 OIIO::string\_view::operator[] (C++ function), 15  
 OIIO::string\_view::rfind (C++ function), 16  
 OIIO::string\_view::str (C++ function), 15  
 OIIO::string\_view::string\_view (C++ function), 15  
 OIIO::Tex::BatchWidth (C++ member), 167  
 OIIO::Tex::FloatWide (C++ type), 167  
 OIIO::Tex::IntWide (C++ type), 167  
 OIIO::Tex::RunMask (C++ type), 167  
 OIIO::TextureSystem (C++ class), 152  
 OIIO::TextureSystem::attribute (C++ function), 154, 155  
 OIIO::TextureSystem::close (C++ function), 166  
 OIIO::TextureSystem::close\_all (C++ function), 167  
 OIIO::TextureSystem::create (C++ function), 153  
 OIIO::TextureSystem::create\_thread\_info (C++ function), 157  
 OIIO::TextureSystem::destroy (C++ function), 153  
 OIIO::TextureSystem::destroy\_thread\_info (C++ function), 157  
 OIIO::TextureSystem::environment (C++ function), 159, 160, 162, 163  
 OIIO::TextureSystem::get\_imagespec (C++ function), 165  
 OIIO::TextureSystem::get\_perthread\_info (C++ function), 156  
 OIIO::TextureSystem::get\_texels (C++ function), 166  
 OIIO::TextureSystem::get\_texture\_handle (C++ function), 157  
 OIIO::TextureSystem::get\_texture\_info (C++ function), 163, 165  
 OIIO::TextureSystem::getattribute (C++ function), 155, 156  
 OIIO::TextureSystem::geterror (C++ function), 167  
 OIIO::TextureSystem::getstats (C++ function), 167  
 OIIO::TextureSystem::good (C++ function), 157  
 OIIO::TextureSystem::imagecache (C++ function), 167  
 OIIO::TextureSystem::imagespec (C++ function), 165, 166  
 OIIO::TextureSystem::invalidate (C++ function), 166  
 OIIO::TextureSystem::invalidate\_all (C++ function), 166  
 OIIO::TextureSystem::reset\_stats (C++ function), 167  
 OIIO::TextureSystem::resolve\_filename (C++ function), 163  
 OIIO::TextureSystem::texture (C++ function), 157, 158, 160, 161  
 OIIO::TextureSystem::texture3d (C++ function), 158, 159, 161, 162  
 OIIO::TypeDesc (C++ struct), 9  
 OIIO::TypeDesc::AGGREGATE (C++ enum), 10  
 OIIO::TypeDesc::aggregate (C++ member), 13  
 OIIO::TypeDesc::arraylen (C++ member), 13  
 OIIO::TypeDesc::basesize (C++ function), 12  
 OIIO::TypeDesc::BASETYPE (C++ enum), 9  
 OIIO::TypeDesc::basetype (C++ member), 13  
 OIIO::TypeDesc::basevalues (C++ function), 12  
 OIIO::TypeDesc::c\_str (C++ function), 12  
 OIIO::TypeDesc::CHAR (C++ enumerator), 10  
 OIIO::TypeDesc::COLOR (C++ enumerator), 11  
 OIIO::TypeDesc::DOUBLE (C++ enumerator), 10  
 OIIO::TypeDesc::elementsize (C++ function), 12  
 OIIO::TypeDesc::elementtype (C++ function), 12  
 OIIO::TypeDesc::equivalent (C++ function), 13  
 OIIO::TypeDesc::FLOAT (C++ enumerator), 10  
 OIIO::TypeDesc::fromstring (C++ function), 13  
 OIIO::TypeDesc::HALF (C++ enumerator), 10  
 OIIO::TypeDesc::INT (C++ enumerator), 10  
 OIIO::TypeDesc::INT16 (C++ enumerator), 10  
 OIIO::TypeDesc::INT32 (C++ enumerator), 10  
 OIIO::TypeDesc::INT64 (C++ enumerator), 10  
 OIIO::TypeDesc::INT8 (C++ enumerator), 9  
 OIIO::TypeDesc::is\_array (C++ function), 12  
 OIIO::TypeDesc::is\_floating\_point (C++ function), 12  
 OIIO::TypeDesc::is\_signed (C++ function), 12  
 OIIO::TypeDesc::is\_sized\_array (C++ function), 12  
 OIIO::TypeDesc::is\_unknown (C++ function), 12  
 OIIO::TypeDesc::is\_unsized\_array (C++ function), 12  
 OIIO::TypeDesc::is\_vec2 (C++ function), 13



OIIO::TypeDesc::is\_vec3 (C++ function), 13  
 OIIO::TypeDesc::is\_vec4 (C++ function), 13  
 OIIO::TypeDesc::KEYCODE (C++ enumerator), 11  
 OIIO::TypeDesc::LASTBASE (C++ enumerator), 10  
 OIIO::TypeDesc::LONGLONG (C++ enumerator), 10  
 OIIO::TypeDesc::MATRIX33 (C++ enumerator), 11  
 OIIO::TypeDesc::MATRIX44 (C++ enumerator), 11  
 OIIO::TypeDesc::NONE (C++ enumerator), 9  
 OIIO::TypeDesc::NORMAL (C++ enumerator), 11  
 OIIO::TypeDesc::NOSEMANTICS (C++ enumerator), 11  
 OIIO::TypeDesc::NOXFORM (C++ enumerator), 11  
 OIIO::TypeDesc::numelements (C++ function), 12  
 OIIO::TypeDesc::operator bool (C++ function), 12  
 OIIO::TypeDesc::operator!= (C++ function), 13  
 OIIO::TypeDesc::operator== (C++ function), 13  
 OIIO::TypeDesc::operator< (C++ function), 13  
 OIIO::TypeDesc::POINT (C++ enumerator), 11  
 OIIO::TypeDesc::PTR (C++ enumerator), 10  
 OIIO::TypeDesc::RATIONAL (C++ enumerator), 11  
 OIIO::TypeDesc::reserved (C++ member), 13  
 OIIO::TypeDesc::SCALAR (C++ enumerator), 10  
 OIIO::TypeDesc::scalartype (C++ function), 12  
 OIIO::TypeDesc::SHORT (C++ enumerator), 10  
 OIIO::TypeDesc::size (C++ function), 12  
 OIIO::TypeDesc::STRING (C++ enumerator), 10  
 OIIO::TypeDesc::TIMECODE (C++ enumerator), 11  
 OIIO::TypeDesc::TypeDesc (C++ function), 11, 12  
 OIIO::TypeDesc::UCHAR (C++ enumerator), 9  
 OIIO::TypeDesc::UINT (C++ enumerator), 10  
 OIIO::TypeDesc::UINT16 (C++ enumerator), 10  
 OIIO::TypeDesc::UINT32 (C++ enumerator), 10  
 OIIO::TypeDesc::UINT64 (C++ enumerator), 10  
 OIIO::TypeDesc::UINT8 (C++ enumerator), 9  
 OIIO::TypeDesc::ULONGLONG (C++ enumerator), 10  
 OIIO::TypeDesc::unarray (C++ function), 13  
 OIIO::TypeDesc::UNKNOWN (C++ enumerator), 9  
 OIIO::TypeDesc::USHORT (C++ enumerator), 10  
 OIIO::TypeDesc::VEC2 (C++ enumerator), 10  
 OIIO::TypeDesc::VEC3 (C++ enumerator), 11  
 OIIO::TypeDesc::VEC4 (C++ enumerator), 11  
 OIIO::TypeDesc::VECSEMANTICS (C++ enum), 11  
 OIIO::TypeDesc::vecsemantics (C++ member), 13  
 OIIO::TypeDesc::VECTOR (C++ enumerator), 11  
 OIIO::ustring (C++ class), 16  
 OIIO::ustring::~~ustring (C++ function), 17  
 OIIO::ustring::assign (C++ function), 18  
 OIIO::ustring::begin (C++ function), 19  
 OIIO::ustring::c\_str (C++ function), 18  
 OIIO::ustring::clear (C++ function), 18  
 OIIO::ustring::compare (C++ function), 19  
 OIIO::ustring::concat (C++ function), 20  
 OIIO::ustring::copy (C++ function), 19  
 OIIO::ustring::data (C++ function), 18  
 OIIO::ustring::empty (C++ function), 19  
 OIIO::ustring::end (C++ function), 19  
 OIIO::ustring::fmtformat (C++ function), 20  
 OIIO::ustring::format (C++ function), 20  
 OIIO::ustring::from\_unique (C++ function), 21  
 OIIO::ustring::getstats (C++ function), 20  
 OIIO::ustring::hash (C++ function), 19  
 OIIO::ustring::is\_unique (C++ function), 20  
 OIIO::ustring::length (C++ function), 19  
 OIIO::ustring::make\_unique (C++ function), 20  
 OIIO::ustring::memory (C++ function), 20  
 OIIO::ustring::operator std::string (C++ function), 18  
 OIIO::ustring::operator string\_view (C++ function), 18  
 OIIO::ustring::operator!= (C++ function), 19, 20  
 OIIO::ustring::operator= (C++ function), 18  
 OIIO::ustring::operator== (C++ function), 19  
 OIIO::ustring::operator< (C++ function), 20  
 OIIO::ustring::operator[] (C++ function), 19  
 OIIO::ustring::rbegin (C++ function), 19  
 OIIO::ustring::rend (C++ function), 19  
 OIIO::ustring::size (C++ function), 19  
 OIIO::ustring::sprintf (C++ function), 20  
 OIIO::ustring::string (C++ function), 18  
 OIIO::ustring::substr (C++ function), 19  
 OIIO::ustring::ustring (C++ function), 17  
 opaque\_z () (DeepData method), 253  
 open () (ImageInput method), 253  
 open () (ImageOutput method), 258  
 openimageio\_version, 285  
 operator!= (C++ function), 14, 21, 24  
 operator== (C++ function), 14, 21, 24  
 operator<< (C++ function), 21, 24

orientation (*ImageBuf attribute*), 264  
 oriented\_full\_height (*C++ function*), 178  
 oriented\_full\_height (*ImageBuf attribute*), 264  
 oriented\_full\_width (*C++ function*), 178  
 oriented\_full\_width (*ImageBuf attribute*), 264  
 oriented\_full\_x (*C++ function*), 178  
 oriented\_full\_x (*ImageBuf attribute*), 264  
 oriented\_full\_y (*C++ function*), 178  
 oriented\_full\_y (*ImageBuf attribute*), 264  
 oriented\_height (*C++ function*), 178  
 oriented\_height (*ImageBuf attribute*), 264  
 oriented\_width (*C++ function*), 178  
 oriented\_width (*ImageBuf attribute*), 264  
 oriented\_x (*C++ function*), 178  
 oriented\_x (*ImageBuf attribute*), 264  
 oriented\_y (*C++ function*), 178  
 oriented\_y (*ImageBuf attribute*), 264

## P

Pixel, **381**  
 pixel\_bytes() (*ImageSpec method*), 248  
 pixeladdr (*C++ function*), 182  
 pixelindex() (*ImageBuf method*), 267  
 pixels (*DeepData attribute*), 252  
 pixels\_valid (*ImageBuf attribute*), 265  
 pixeltype() (*ImageBuf method*), 265  
 Plugin, **381**

## R

rblur (*C++ member*), 168  
 read() (*ImageBuf method*), 262  
 read\_image() (*ImageInput method*), 254  
 read\_native\_deep\_image() (*ImageInput method*), 256  
 read\_native\_deep\_scanlines() (*ImageInput method*), 256  
 read\_native\_deep\_tiles() (*ImageInput method*), 256  
 read\_scanline() (*ImageInput method*), 255  
 read\_scanlines() (*ImageInput method*), 255  
 read\_tile() (*ImageInput method*), 255  
 read\_tiles() (*ImageInput method*), 255  
 render\_point() (*ImageBufAlgo method*), 269  
 resample (*C++ function*), 241  
 reset() (*ImageBuf method*), 262  
 roi (*ImageBuf attribute*), 264  
 roi (*Imagespec attribute*), 248  
 ROI(), 245  
 roi\_full (*ImageBuf attribute*), 264  
 roi\_full (*ImageSpec attribute*), 248  
 RunMaskOn (*C++ enumerator*), 168  
 rwidth (*C++ member*), 168

## S

samples() (*DeepData method*), 252  
 samplesize() (*DeepData method*), 252  
 sblur (*C++ member*), 168  
 Scanline, **381**  
 Scanline Image, **381**  
 scanline\_bytes() (*ImageSpec method*), 248  
 seek\_subimage() (*ImageInput method*), 254  
 serialize() (*ImageSpec method*), 250  
 set\_deep\_samples() (*ImageBuf method*), 268  
 set\_deep\_value() (*DeepData method*), 252  
 set\_deep\_value() (*ImageBuf method*), 268  
 set\_deep\_value\_uint() (*DeepData method*), 253  
 set\_deep\_value\_uint() (*ImageBuf method*), 268  
 set\_format() (*ImageSpec method*), 248  
 set\_full() (*ImageBuf method*), 264  
 set\_origin() (*ImageBuf method*), 264  
 set\_pixels() (*ImageBuf method*), 267  
 set\_roi (*C++ function*), 24  
 set\_roi()  
     built-in function, 246  
 set\_roi\_full (*C++ function*), 24  
 set\_roi\_full()  
     built-in function, 246  
 set\_samples() (*DeepData method*), 252  
 set\_write\_format() (*ImageBuf method*), 263  
 set\_write\_tiles() (*ImageBuf method*), 263  
 setpixel() (*ImageBuf method*), 266  
 sort() (*DeepData method*), 253  
 spec() (*ImageBuf method*), 263  
 spec() (*ImageInput method*), 254  
 spec() (*ImageOutput method*), 258  
 specmod() (*ImageBuf method*), 263  
 split() (*DeepData method*), 253  
 str()  
     built-in function, 244  
 sub (*C++ function*), 241  
 subimage (*C++ member*), 168  
 subimage (*ImageBuf attribute*), 264  
 subimagename (*C++ member*), 168  
 supports() (*ImageOutput method*), 258  
 swap() (*ImageBuf method*), 265  
 swidth (*C++ member*), 168

## T

tblur (*C++ member*), 168  
 texture3d (*C++ function*), 169  
 TextureSystem::texture (*C++ function*), 168  
 Tile, **381**  
 tile\_bytes() (*ImageSpec method*), 248  
 tile\_pixels() (*ImageSpec method*), 248  
 Tiled Image, **381**  
 to\_native\_rectangle (*C++ function*), 103  
 to\_native\_scanline (*C++ function*), 103

to\_native\_tile (C++ function), 103  
to\_xml () (ImageSpec method), 250  
twidth (C++ member), 168

## U

undefined () (ImageSpec method), 250

## V

valid\_tile\_range () (ImageSpec method), 250  
VECSEMANTICS (built-in class), 243  
vecsemantics (TypeDesc attribute), 244  
Volume Image, **381**

## W

width (ROI attribute), 245  
write () (ImageBuf method), 262  
write\_deep\_image () (ImageOutput method), 260  
write\_deep\_scanlines () (ImageOutput method), 260  
write\_deep\_tiles () (ImageOutput method), 260  
write\_image () (ImageOutput method), 258  
write\_scanline () (ImageOutput method), 259  
write\_scanlines () (ImageOutput method), 259  
write\_tile () (ImageOutput method), 259  
write\_tiles () (ImageOutput method), 259

## X

xbegin (C++ function), 177  
xbegin (ImageBuf attribute), 264  
xbegin (ROI attribute), 245  
xend (C++ function), 177  
xend (ImageBuf attribute), 264  
xend (ROI attribute), 245  
xmax (ImageBuf attribute), 264  
xmin (ImageBuf attribute), 264

## Y

ybegin (C++ function), 177  
ybegin (ImageBuf attribute), 264  
ybegin (ROI attribute), 245  
yend (C++ function), 177  
yend (ImageBuf attribute), 264  
yend (ROI attribute), 245  
ymax (ImageBuf attribute), 264  
ymin (ImageBuf attribute), 264

## Z

Z\_channel (DeepData attribute), 252  
z\_channel (ImageSpec attribute), 247  
Zback\_channel (DeepData attribute), 252  
zbegin (C++ function), 177  
zbegin (ImageBuf attribute), 264  
zbegin (ROI attribute), 245

zend (C++ function), 177  
zend (ImageBuf attribute), 264  
zend (ROI attribute), 245  
zero () (ImageBufAlgo method), 269  
zmax (ImageBuf attribute), 264  
zmin (ImageBuf attribute), 264