Technical Introduction to OpenEXR

Florian Kainz, Rod Bogart, Piotr Stanczyk
Industrial Light & Magic

Peter Hillman
Weta Digital

Updated November 5, 2013

Overview

OpenEXR is an open-source high-dynamic-range floating-point image file format for high-quality image processing and storage. This document presents a brief overview of OpenEXR and explains concepts that are specific to this format.

Contents

Features of OpenEXR

Starting in 1999, Industrial Light & Magic developed OpenEXR, a high-dynamic-range image file format for use in digital visual effects production. In early 2003, after using and refining the file format for a few years, ILM released OpenEXR as an open-source C++ library. In 2013 Weta Digital and ILM jointly released OpenEXR version 2.0, which added support for deep images and multi-part files.

A unique combination of features makes OpenEXR a good fit for high-quality image processing and storage applications:

- High dynamic range
  Pixel data are stored as 16-bit or 32-bit floating-point numbers. With 16 bits, the dynamic range that can be represented is significantly higher than the range of most image capture devices: 10^9 or 30 f-stops without loss of precision, and an additional 10 f-stops at the low end with some loss of precision. Most 8-bit file formats have around 7 to 10 f-stops.

- Good color resolution
  With 16-bit floating-point numbers, color resolution is 1024 steps per f-stop, as opposed to somewhere around 20 to 70 steps per f-stop for most 8-bit file formats. Even after significant processing, for example, extensive color correction, images tend to show no noticeable color banding.

- Lossless and lossy data compression
  OpenEXR employs data compression to reduce the size of image files. Most of the available compression methods are lossless; repeatedly compressing and uncompressing an image does not change the pixel data. With the lossless compression methods, photographic images with significant amounts of film grain tend to shrink to somewhere between 35 and 55 percent of their uncompressed size. OpenEXR also supports lossy compression, which tends to shrink image files more than lossless compression, but doesn't preserve the image data exactly.

- Arbitrary image channels
  OpenEXR images can contain an arbitrary number and combination of image channels, for example red, green, blue, and alpha; luminance and sub-sampled chroma channels; depth, surface normal directions, or motion vectors.

- Scan-line and tiled images, multi-resolution images
  Pixels in an OpenEXR image can be stored either as scan lines or as tiles. Tiled image files allow random access to rectangular sub-regions of an image. Multiple versions of a tiled image, each with a different resolution, can be stored in a single multi-resolution OpenEXR file.
  Multi-resolution images, often called “mipmaps” or “ripmaps,” are commonly used as texture maps in 3D rendering programs to accelerate filtering during texture lookup, or for operations like stereo image matching. Tiled multi-resolution images are also useful for implementing fast zooming and panning in programs that interactively display very large images.

- Multiple views
  OpenEXR files can store multi-view images that show the same scene from multiple different points of view. A common application is 3D stereo imagery, where a left-eye and a right-eye view of a scene are stored in a single file.

- Deep images
  OpenEXR can store deep images, where each pixel contains an arbitrarily long list of values or samples in each channel. In computer graphics, applications for deep images include, for example, easy and accurate compositing of objects that occlude one another in ways that make compositing by stacking flat images difficult, or nearly artifact-free motion blur and depth-of-field blur image processing operations.

- Multi-part files
  An OpenEXR file may contain multiple independent images or "parts" with different sets of image channels, resolutions and data compression methods.
  Multi-part files are useful when an image contains a large number of channels, but file readers are expected read only a subset of those channels at a time. Placing each such subset in a separate part speeds up file reading, since channels that are not needed by the reader are never accessed.

- Ability to store additional data
  Often it is necessary to annotate images with additional data; for example, color timing information, process tracking data, or camera position and view direction. OpenEXR allows storing of an arbitrary number of extra attributes, of arbitrary type, in an image file. Software that reads OpenEXR files ignores attributes it does not understand.

- Easy-to-use C++ and C programming interfaces
  In order to make writing and reading OpenEXR files easy, the file format was designed together with a C++ programming interface. Two levels of access to image files are provided: a fully general interface for writing and reading files with arbitrary sets of image channels, and a specialized interface for the most common case (red, green, blue, and alpha channels, or some subset of those). Additionally, a C-callable version of the programming interface supports reading and writing OpenEXR files from programs written in C.
  Many application programs expect image files to be scan line-based and flat. With the OpenEXR programming interface, applications that cannot handle tiled or deep images can treat all OpenEXR files as if they were scan-line based and flat; the interface automatically converts tiles to scan lines and flattens deep pixels.
  The C++ and C interfaces are implemented in the open-source IlmImf library.

- Fast multi-threaded file reading and writing
  The IlmImf library supports multi-threaded reading or writing of an OpenEXR image file: while one thread performs low-level file input or output, multiple other threads simultaneously encode or decode individual pieces of the file.

- Portability
  The OpenEXR file format is hardware and operating system independent. While implementing the C and C++ programming interfaces, an effort was made to use only language features and library functions that comply with the C and C++ ISO standards.

Definitions and Terminology

Flat and Deep Images

A ***flat image*** has at most one stored value or ***sample*** per pixel per channel.

[Example: the most common case is a red-green-blue or RGB image, which contains three channels, and every pixel has exactly one red, one green and one blue sample. Some channels in a flat image may be sub-sampled, as is the case with luminance-chroma images, where the luminance channel has a sample at every pixel, but the chroma channels have samples only at every second pixel of every second scan line.]

A ***deep image*** can store an unlimited number of samples per pixel, and each of those samples is associated with a depth, or distance from the viewer. All channels in a single pixel have the same number of samples, but the number of samples varies from pixel to pixel, and any non-negative number of samples, including zero, is allowed.

Parts, Images, Single-Part and Multi-Part Files

In an OpenEXR file a ***part*** is a separate image, consisting of a header and an array of pixels. A ***single-part file*** contains exactly one part. A ***multi-part file*** contains two or more parts, each with its own header and corresponding pixels.

An ***image*** is the set of all channels in a single part. The images in a multi-part file are largely independent of each other; a file may contain a mixture of flat and deep images.

Note: deep images and multi-part files were added to OpenEXR in version 2.0. OpenEXR 2.0 single-part files that contain flat images are fully compatible with the OpenEXR 1.7 file format. Software that has not been upgraded to version 2.0 will be able read those files, and OpenEXR 2.0-capable software can read any pre-2.0 file. However, files that contain deep images or multiple parts are not backward compatible.

Headers and Attributes

Each part in an OpenEXR file has a ***header***. The header is a list of ***attributes*** that describe the pixels in the part. An attribute is a name-value pair, where the name is a text string and the value is an item of an arbitrary data type. For example, each header includes an attribute whose name is “pixelAspectRatio,” and whose value is a floating-point number that tells application software how to display the corresponding image at the correct aspect ratio.

Pixel space, Display Window and Data Window

Pixel space is a 2D coordinate system with x increasing from left to right and y increasing from top to bottom. Pixels are data samples (for flat images), or lists of data samples (for deep images), that are located at integer coordinate locations in pixel space.

The boundaries of an OpenEXR image are given as an axis-parallel rectangular region in pixel space, the ***display window***. The display window is defined by the positions of the pixels in the upper left and lower right corners.

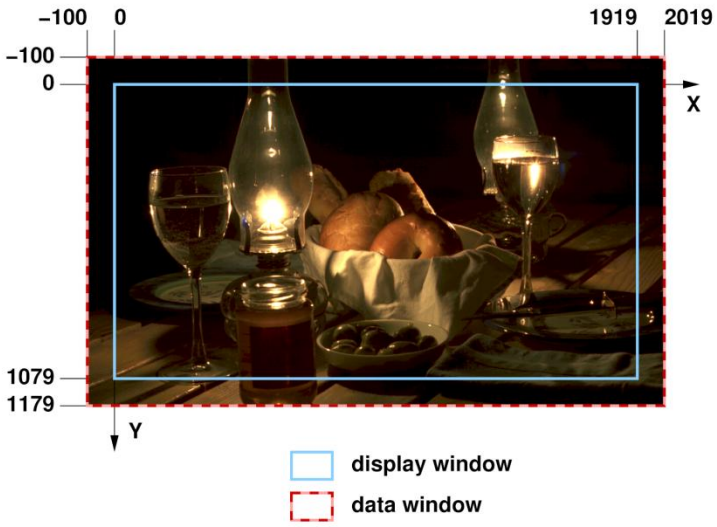
An OpenEXR image may not have pixel data for all the pixels in the display window, or it may have pixel data beyond the boundaries of the display window. The region for which pixel data are available is defined by a second axis-parallel rectangle in pixel space, the ***data window***.

In this document the notation $(x_{min}, y_{min}) - (x_{max}, y_{max})$ is used to describe a window whose upper left and lower right corners are at (x_{min}, y_{min}) and (x_{max}, y_{max}) respectively.

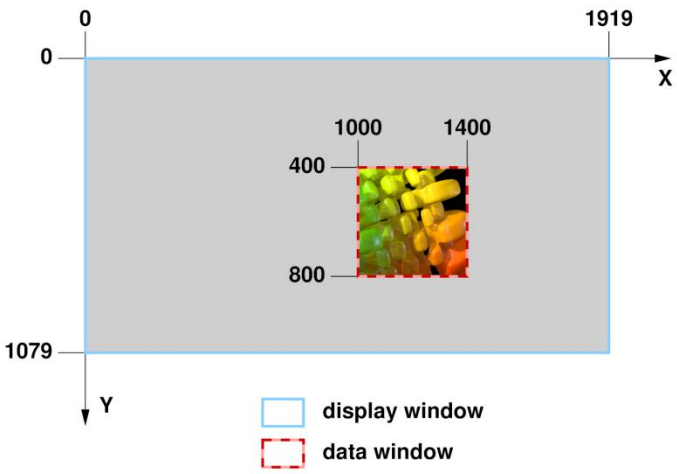
[Example:

Assume that we are producing a movie with a resolution of 1920 by 1080 pixels. The display window for all frames of the movie is (0, 0) - (1919, 1079). For most images, in particular finished frames that will be delivered to the customer, the data window is the same as the display window, but for some images that are used in producing the finished frames, the data window differs from the display window.

For a background plate that will be heavily post-processed, extra pixels beyond the edge of the display frame are recorded, and the data window is set to (-100, -100) - (2019, 1179). The extra pixels are not normally displayed, but their existence allows operations such as large-kernel blurs or simulated camera shake to avoid edge artifacts. (In order to compute the color of a pixel near the edge of the display window, a blur kernel must consider pixels outside the display window.)



While tweaking a computer-generated image element, an artist repeatedly renders the same frame. To save time, the artist renders only a small region of interest close to the center of the image. The data window of the image is set to (1000, 400) - (1400, 800). When the image is displayed, the display program fills the area outside of the data window with some default color.



]

Image Channels and Sampling Rates

Every OpenEXR image contains one or more image ***channels***. Each channel has a ***name***, a ***data type***, and x and y ***sampling rates***.

The channel's name is a text string, for example “R”, “Z” or “yVelocity”. The name tells programs that read the image file how to interpret the data in the channel.

For a few channel names, for example, “R”, “G”, “B” and “A”, the interpretation of the data is predefined by code in the IlmImf library, but the interpretation of channels with other names is left to application software.

The data type of a channel determines the data type of the corresponding values in the pixels. OpenEXR supports three channel data types

Type name	Pixel data type
HALF	16-bit binary floating-point numbers according to IEEE standard 754-2008. Used for regular image data.
FLOAT	32-bit binary floating-point numbers according to IEEE standard 754-2008. Used where the range or precision of 16-bit numbers is not sufficient, for example, depth channels.
UINT	32-bit unsigned integers. Used for discrete per-pixel data such as object identifiers.

The x and y sampling rates of a channel, s_x and s_y, determine for which of the pixels within the data window data are stored in the file. Data for a pixel at pixel space coordinates (x, y) are stored only if $x \bmod s_x = 0$ and $y \bmod s_y = 0$. (The notation $x \bmod y$ refers to the remainder of the integer division x/y.)

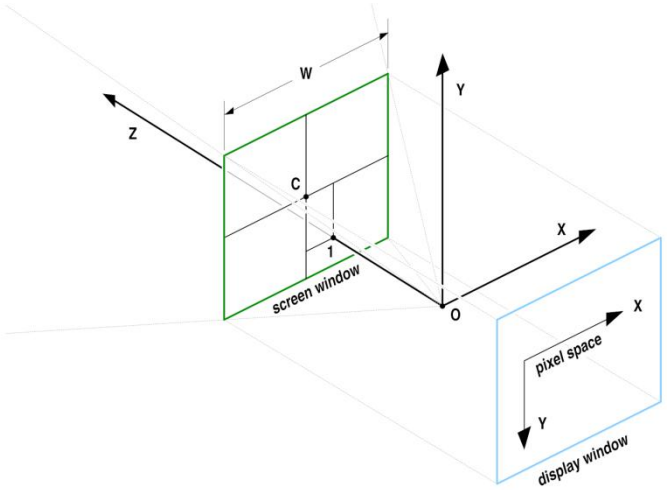
[Examples: For RGBA (red, green, blue, alpha) images, s_x and s_y are 1 for all channels, and each channel contains data for every pixel. For other types of images, some channels may be sub-sampled. In images with one luminance channel and two chroma channels, s_x and s_y would be 1 for the luminance channel, but 2 for the chroma channels, indicating that chroma data are present only at every second pixel of every second scan line.]

Restrictions on Sampling Rates

Channels with an x or y sampling rate other than 1 are allowed only in flat, scan-line based images. If an image is deep or tiled, then the x and y sampling rates for all of its channels must be 1.

Projection, Camera Coordinate System and Screen Window

Many images are generated by a perspective projection. We assume that a camera is located at the origin, O, of a 3D camera coordinate system. The camera looks along the positive z axis. The positive x and y axes correspond to the camera's "left" and "up" directions. The 3D scene is projected onto the $z = 1$ plane. The image recorded by the camera is bounded by a rectangle, the ***screen window***. In pixel space, the screen window corresponds to the image's display window. In the file, the size and position of the screen window are specified by the x and y coordinates of the window's center, C, and by the window's width, W. The height of the screen can be derived from C, W, the display window and the pixel aspect ratio (see below).



Pixel Aspect Ratio

The ***pixel aspect ratio*** of an image is the ratio d_x/d_y, where d_x is the distance between pixel space locations (x, y) and $(x + 1, y)$, and the d_y is the distance between pixel space locations (x, y) and $(x, y + 1)$ on the display when the image is displayed at the correct aspect ratio; that is, when the width divided by the height of the displayed image is as intended.

[Examples: an image whose display window is (0, 0) – (1919, 1079) is meant to be displayed with a 16:9 aspect ratio. The image has a pixel aspect ratio of 1.0. If an image with the same data window is meant to be displayed with an aspect ratio of 2.35:1, then its pixel aspect ratio is 1.322.]

Scan Lines

In scan-line based images, pixels are stored in horizontal rows, or ***scan lines***. An image whose data window is (x_{min}, y_{min}) - (x_{max}, y_{max}) contains $x_{max} - x_{min} + 1$ scan lines. Each scan line contains $y_{max} - y_{min} + 1$ pixels.

Scan-line based images cannot be multi-resolution images.

Tiles

In tiled images, the data window is subdivided into an array of smaller rectangles, called ***tiles***. Each tile contains p_x by p_y pixels. An image whose data window is (x_{min}, y_{min}) - (x_{max}, y_{max}) contains

$$\left\lceil \frac{w}{p_x} \right\rceil$$

by

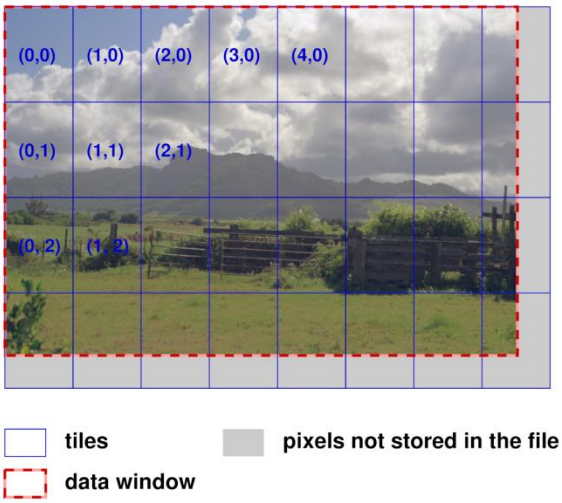
$$\left\lceil \frac{w}{p_x} \right\rceil$$

tiles, where w and h are the width and height of the data window,

$$w = x_{max} - x_{min} + 1$$
$$h = y_{max} - y_{min} + 1'$$

and the notation $\lceil x \rceil$ refers to the ceiling of x, that is, the lowest integer not less than x.

The upper left corner of the upper left tile is aligned with the upper left corner of the data window at (x_{min}, y_{min}). The rightmost column and the bottom row of tiles may extend outside the data window. If a tile contains pixels that are outside the data window, then those extra pixels are discarded when the tile is stored in the file.



Levels and Level Modes

A single tiled part of OpenEXR file may contain multiple versions of the same image, each with a different resolution. Each version is called a ***level***. The number of levels in a part and their resolutions depend on the part's ***level mode***. OpenEXR supports three level modes:

Mode name	Description
ONE_LEVEL	The part contains only a single full-resolution level. A tiled ONE_LEVEL image is equivalent to a scan line based image; the only difference is that pixels are accessed by tile rather than by scan line.
MIPMAP_LEVELS	The part contains multiple versions of the image. Each successive level is half the resolution of the previous level in both dimensions. The lowest-resolution level contains only a single pixel. [Example: if the first level, with full resolution, contains 16×8 pixels, then the file contains four more levels with 8×4, 4×2, 2×1, and 1×1 pixels respectively.]
RIPMAP_LEVELS	Like MIPMAP_LEVELS, but with more levels. The levels include all combinations of reducing the resolution of the first level by powers of two independently in both dimensions. [Example: if the full-resolution level contains 4×4 pixels, then the part contains eight more levels, with the following resolutions: 2×4 1×4 4×2 2×2 1×2 4×1 2×1 1×1]

Level Numbers, Level Sizes and Rounding Mode

Levels are identified by ***level numbers***. A level number is a pair of integers, (l_x, l_y). Level $(0, 0)$ is the highest-resolution level, with w by h pixels. Level (l_x, l_y) contains

$$rf\left(\frac{w}{2^{l_x}}\right)$$

by

$$rf\left(\frac{h}{2^{l_y}}\right)$$

pixels, where $rf(x)$ is a rounding function that depends on the ***level size rounding mode***:

Rounding mode	Rounding function
ROUND_DOWN	$\lfloor x \rfloor$ (floor of x, highest integer not greater than x)
ROUND_UP	$\lceil x \rceil$ (ceiling of x, lowest integer not less than x)

Parts whose level mode is MIPMAP_LEVELS contain only levels where $l_x = l_y$. Parts whose level mode is ONE_LEVEL contain only level $(0, 0)$.

[Examples:

The levels in a RIPMAP_LEVELS part whose highest-resolution level contains 4 by 4 pixels have the following level numbers:

		width		
		4	2	1
	4	$(0,0)$	$(1,0)$	$(2,0)$
height	2	$(0,1)$	$(1,1)$	$(2,1)$
	1	$(0,2)$	$(1,2)$	$(2,2)$

In an equivalent MIPMAP_LEVELS part, only levels $(0,0)$, $(1,1)$ and $(2,2)$ are present.

In a MIPMAP_LEVELS part with a highest-resolution level of 15 by 17 pixels, the resolutions of the remaining levels depend on the level size rounding mode:

Rounding mode	Level resolutions
ROUND_DOWN	15×17, 7×8, 3×4, 1×2, 1×1
ROUND_UP	15×17, 8×9, 4×5, 2×3, 1×2, 1×1

]

Tile Coordinates

In a part with multiple levels, tiles have the same size, regardless of their level. Lower-resolution levels contain fewer, rather than smaller, tiles. Within a level, a tile is identified by a pair of integer ***tile coordinates***, which specify the tile's column and row. The upper left tile has coordinates $(0,0)$, and the coordinates increase from left to right and top to bottom respectively. In order to identify a tile uniquely within a multi-resolution part, both the tile coordinates and the level number are needed.

Views

A ***view*** is a set of image channels, identified by a naming convention and a header attribute. This is commonly used to store stereo files, with one view for each eye. For flat images, views can be stored in separate files, separate parts of a single file, or together in a single part. For deep images, views can be stored in separate files or separate parts of a single file, but they cannot be stored together in a single part.

For details, see the separate “Storing Multi-View Images in OpenEXR Files” document.

OpenEXR File Structure

An OpenEXR file consists of one or more parts. Each part has a header and a corresponding array of pixels.

Header

The header is a list of attributes that describe the image in a part. To ensure that OpenEXR files written by one program can be read by other programs, certain required attributes must be present in every header:

Attribute Name	Description
displayWindow, dataWindow	The display and data window of the image.
pixelAspectRatio	The pixel aspect ratio of the image.
channels	Description of the image channels stored in the image.
compression	Specifies the compression method applied to the pixel data of all channels in the image.
lineOrder	Specifies in what order the scan lines in the file are stored in the file (increasing Y, decreasing Y, or, for tiled images, also random Y).
screenWindowWidth, screenWindowCenter	Describe the perspective projection that produced the image (see page 8). Programs that deal with images as purely two-dimensional objects may not be able to generate a description of a perspective projection. Those programs should set screenWindowWidth to 1, and screenWindowCenter to (0, 0).
tiles	This attribute is required only for tiled flat or tiled deep images. It specifies the size of the tiles, the level mode and the level size rounding mode of the image.

In multi-part and deep files, additional attributes must be present the header of each part:

Attribute Name	Description
name	The name of the part. Part names must be unique, that is, no two parts in a file may share the same name.

Attribute Name	Description
type	The type of image stored in the part. Four types are possible: scanlineimage — Flat, scan-line based. tiledimage — Flat, tiled. deepscanline — Deep, scan-line based. deeptile — Deep, tiled.
maxSamplesPerPixel	This attribute is automatically added to deep files, and its value is set by the IlmImf library when the file is written. The attribute indicates the maximum number of samples in any single pixel within the image. Application code may use this number to make memory allocation more efficient for image processing: if the maximum number of samples is small, then the application may want to allocate that many samples for every pixel, thus making the in-memory layout of an image more regular by eliminating many pointers.

In addition to the required attributes, application software may place any number of additional attributes in the headers. Often it is necessary to annotate an image with additional data, for example, color timing information, process tracking data, or camera position and view direction. Those data can be packaged as extra attributes in the header of the image.

Constraints on Attribute Values

The parts in a multi-part OpenEXR file are not entirely independent. The values of the displayWindow and pixelAspectRatio attributes must be the same for all parts of a file. For example, it is not possible to have one part with a pixel aspect ratio of 1.0 and another part with a pixel aspect ratio of 1.5.

In addition, if the headers include timeCode and chromaticities attributes, then the values of those attributes must also be the same for all parts of a file. (The timeCode and chromaticities attributes are defined in the IlmImf library. The purpose of the chromaticities attribute is described under “RGB Color,” on page 18.)

Scan Lines

When a scan-line based image is written, application code must set the lineOrder attribute in the header to INCREASING_Y order (top scan line first) or DECREASING_Y order (bottom scan line first). Application code must then write the scan lines in the specified order. When a scan-line based file is read, random access to the scan lines is possible; the scan lines can be read in any order. However, reading the scan lines in the same order as they were written causes the file to be read sequentially, without “seek” operations, and as fast as possible.

Note that the lineOrder attribute does not affect the orientation of the pixel space coordinate system. Pixel space y coordinates always increase from top to bottom. The lineOrder attribute determines only how the scan lines are arranged in the file.

Tiles

When application code writes or reads a tiled image, the tiles can be accessed in any order. When a tiled file is written, application code must set the lineOrder attribute. The IlmImf library may buffer and sort the tiles, depending on the setting of the lineOrder attribute and the order in which the tiles arrive. On reading a tiled image, application code can avoid "seek" operations and read the file as fast as possible by reading the tiles sequentially, in the order as they appear in the file.

For tiled files, line order is interpreted as follows:

Line order	Description
INCREASING_Y	The tiles for each level are stored in a contiguous block. The levels are ordered like this: $$(0,0) \quad (1,0) \quad \cdots \quad (n_x-1,0)$$ $$(0,1) \quad (1,1) \quad \cdots \quad (n_x-1,1)$$ $$\cdots$$ $$(0,n_y-1) \quad (1,n_y-1) \quad \cdots \quad (n_x-1,n_y-1),$$ where $$n_x = rf\left(\log_2(w)\right) + 1,$$ $$n_y = rf\left(\log_2(h)\right) + 1$$ if the file's level mode is RIPMAP_LEVELS, or $$n_x = n_y = rf\left(\log_2(\max(w,h))\right) + 1$$ if the level mode is MIPMAP_LEVELS, or $$n_x = n_y = 1$$ if the level mode is ONE_LEVEL. In each level, the tiles are stored in the following order: $$(0,0) \quad (1,0) \quad \cdots \quad (t_x-1,0)$$ $$(0,1) \quad (1,1) \quad \cdots \quad (t_x-1,1)$$ $$\cdots$$ $$(0,t_y-1) \quad (1,t_y-1) \quad \cdots \quad (t_x-1,t_y-1),$$ where $t_x$and $t_y$are the number of tiles in the x and y direction respectively, for that particular level.

Line order	Description
DECREASING_Y	Levels are ordered as for INCREASING_Y, but within each level, the tiles are stored in this order: $(0, t_y - 1)\ (1, t_y - 1)\ \cdots\ (t_x - 1, t_y - 1),$ $\cdots$ $(0, 1)\quad (1, 1)\quad \cdots\quad (t_x - 1, 1)$ $(0, 0)\quad (1, 0)\quad \cdots\quad (t_x - 1, 0)$
RANDOM_Y	When a file is written, tiles are not sorted; they are stored in the file in the order they are produced by the application program.

If an application program produces tiles in an essentially random order, then selecting INCREASING_Y or DECREASING_Y line order may force the IlmImf library to allocate significant amounts of memory to buffer tiles until they can be stored in the file in the proper order. If memory is scarce, allocating the extra memory can be avoided by setting the file's line order to RANDOM_Y. In this case the library doesn't buffer and sort tiles; each tile is immediately stored in the file.

Data Compression

OpenEXR offers several different data compression methods, with various speed versus compression ratio tradeoffs. Optionally, the pixels can be stored in uncompressed form. With fast file systems, uncompressed files can be written and read significantly faster than compressed files.

Compressing an image with a lossless method preserves the image exactly; the pixel data are not altered. Compressing an image with a lossy method preserves the image only approximately; the compressed image looks like the original, but the data in the pixels may have changed slightly.

In a multi-part file the compression method can be selected separately for each part.

The following compression schemes are supported:

Name	Description
PIZ (lossless)	A wavelet transform is applied to the pixel data, and the result is Huffman-encoded. This scheme tends to provide the best compression ratio for the types of images that are typically processed at Industrial Light & Magic. Files are compressed and decompressed at roughly the same speed. For photographic images with film grain, the files are reduced to between 35 and 55 percent of their uncompressed size. PIZ compression works well for scan-line based files, and also for tiled files with large tiles, but small tiles do not shrink much. (PIZ-compressed data start with a relatively long header; if the input to the compressor is short, adding the header

Name	Description
	tends to offset any size reduction of the input.) PIZ compression is only supported for flat images.
ZIP (lossless)	Differences between horizontally adjacent pixels are compressed using the open-source zlib library. ZIP decompression is faster than PIZ decompression, but ZIP compression is significantly slower. Photographic images tend to shrink to between 45 and 55 percent of their uncompressed size. Multi-resolution files are often used as texture maps for 3D renderers. For this application, fast read accesses are usually more important than fast writes, or maximum compression. For texture maps, ZIP is probably the best compression method. In scan-line based files, 16 rows of pixels are accumulated and compressed together as a single block.
ZIPS (lossless)	Uses the open-source zlib library for compression. Like ZIP compression, but operates on one scan line at a time.
RLE (lossless)	Differences between horizontally adjacent pixels are run-length encoded. This method is fast, and works well for images with large flat areas, but for photographic images, the compressed file size is usually between 60 and 75 percent of the uncompressed size.
PXR24 (lossy)	After reducing 32-bit floating-point data to 24 bits by rounding (while leaving 16-bit floating-point data unchanged), differences between horizontally adjacent pixels are compressed with zlib, similar to ZIP. PXR24 compression preserves image channels of type HALF and UINT exactly, but the relative error of FLOAT data increases to about 3×10^{-5}. This compression method works well for depth buffers and similar images, where the possible range of values is very large, but where full 32-bit floating-point accuracy is not necessary. Rounding improves compression significantly by eliminating the pixels' 8 least significant bits, which tend to be very noisy, and therefore difficult to compress. PXR24 compression is only supported for flat images.
B44 (lossy)	Channels of type HALF are split into blocks of four by four pixels or 32 bytes. Each block is then packed into 14 bytes, reducing the data to 44 percent of their uncompressed size. When B44 compression is applied to RGB images in combination with luminance/chroma encoding (see below), the size of the compressed pixels is about 22 percent of the size of the original RGB data. Channels of type UINT or FLOAT are not compressed. Decoding is fast enough to allow real-time playback of B44-compressed OpenEXR image sequences on commodity hardware. The size of a B44-compressed file depends on the number of pixels in the image, but not on the data in the pixels. All images with the same resolution and the same set of channels have the same size. This can be advantageous for systems that support real-time playback of image sequences; the predictable file size makes it easier to allocate space on storage media efficiently. B44 compression is only supported for flat images.

Name	Description
B44A (lossy)	Like B44, except for blocks of four by four pixels where all pixels have the same value, which are packed into 3 instead of 14 bytes. For images with large uniform areas, B44A produces smaller files than B44 compression. B44A compression is only supported for flat images.

Luminance/Chroma Images

Encoding flat images with one luminance and two chroma channels, rather than as RGB data, allows a simple but effective form of lossy data compression that is independent of the compression methods listed above. The chroma channels can be stored at lower resolution than the luminance channel. This leads to significantly smaller files, with only a small reduction in image quality. The specialized RGBA interface in the IlmImf library directly supports reading and writing luminance/chroma images. When an application program writes an image, it can choose either RGB or luminance/chroma format. When an image with luminance/chroma data is read, the library automatically converts the pixels back to RGB.

Given linear RGB data, luminance, Y, is computed as a weighted sum of R, G, and B:

$$Y = R \cdot w_R + G \cdot w_G + B \cdot w_B,$$

where the values of the weighting factors, w_R, w_G, and w_B, are derived from the chromaticities of the image's primaries and white point. (See “RGB Color,” on page 18.)

Chroma information is stored in two channels, RY and BY, which are computed like this:

$$RY = \frac{R - Y}{Y}$$

$$BY = \frac{B - Y}{Y}$$

The RY and BY channels can be low-pass filtered and sub-sampled without degrading the original image very much. The RGBA interface in IlmImf uses vertical and horizontal sampling rates of 2. Even though the resulting luminance/chroma images contain only half as much data, they usually do not look noticeably different from the original RGB images.

Converting RGB data to luminance/chroma format also allows space-efficient storage of gray-scale images. Only the Y channel needs to be stored in the file. The RY and BY channels can be discarded. If the original is already a gray-scale image, that is, every pixel's red, green, and blue are equal, then storing only Y preserves the image exactly; the Y channel is not sub-sampled, and the RY and BY channels contain only zeroes.

The HALF Data Type

Image channels of type HALF are stored as 16-bit binary floating-point numbers with 1 sign bit, 5 exponent bits and 10 mantissa bits. The 16-bit floating-point data type is implemented as a C++ class, *half*, which was designed to behave as much as possible like the standard floating-point data types built into the C++ language. In arithmetic expressions, numbers of type *half* can be mixed freely with float and double numbers; in most cases, conversions to and from *half* happen automatically.

The OpenEXR half data type predates the IEEE 754-2008 standard by several years, but it is fully compatible with the standard, except for rounding: OpenEXR implements only the "round to nearest, ties to even" mode.

Recommendations

Scene-Referred Images

By convention, OpenEXR stores scene-referred linear values in the RGB floating-point numbers. By this we mean that red, green and blue values in the pixels are linear relative to the amount of light in the depicted scene. This implies that displaying an image requires some processing to account for the non-linear response of a typical display device. In its simplest form, this is a power function to perform gamma correction, but processing may be more complex. For example, one may want to apply a "film look" or reduce the dynamic range of the original image before displaying it. By storing linear data in the file (double the number, double the light in the scene), we have the best starting point for these downstream algorithms.

With this linear relationship established, the question remains, what number is white? The convention employed by OpenEXR is to determine a middle gray object, and assign it the photographic 18% gray value, or 0.18 in the floating point scheme. Other pixel values can be easily determined from there (a stop brighter is 0.36, another stop is 0.72). The value 1.0 has no special significance (it is not a clamping limit, as in other formats); it roughly represents light coming from a 100% reflector (slightly brighter than white paper). However, there are many brighter pixel values available to represent objects such as fire and highlights.

The range of normalized 16-bit floating-point numbers can represent thirty stops of information with 1024 steps per stop. We have eighteen and a half stops above middle gray, and eleven and a half below. Denormalized numbers provide an additional ten stops at the low end, with gradually decreasing precision.

RGB Color

Simply calling the R channel red is not sufficient information to determine accurately the color that is represented by a given pixel value. The IlmImf library defines a "chromaticities" attribute, which specifies the CIE x,y coordinates for red, green and blue primaries, and white; that is, for the RGB triples (1, 0, 0), (0, 1, 0), (0, 0, 1), and (1, 1, 1). The x,y coordinates of all possible RGB triples can be derived from the chromaticities attribute. If the primaries and white point for a given display are known, a file-to-display color transform can correctly be done. The IlmImf library does not perform this transformation; it is left to the display software. The chromaticities attribute is optional, and many

programs that write OpenEXR omit it. If a file doesn't have a chromaticities attribute, display software should assume that the file's primaries and the white point match Rec. ITU-R BT.709-3:

	CIE x, y
red	0.6400, 0.3300
green	0.3000, 0.6000
blue	0.1500, 0.0600
white	0.3127, 0.3290

CIE XYZ Color

In an OpenEXR file whose pixels represent CIE XYZ tristimulus values, the pixels' X, Y and Z components should be stored in the file's R, G and B channels. The file header should contain a chromaticities attribute with the following values:

	CIE x, y
red	1, 0
green	0, 1
blue	0, 0
white	1/3, 1/3

Channel Names

An OpenEXR image can have any number of channels with arbitrary names. The specialized RGBA image interface in the IlmImf library assumes that channels with the names “R”, “G”, “B” and “A” mean red, green, blue and alpha. No predefined meaning has been assigned to any other channels. However, for a few channel names we recommend the interpretations given in the two tables below. We expect this table to grow over time as users employ OpenEXR for data such as shadow maps, motion-vector fields or images with more than three color channels.

For flat images:

Name	Interpretation
Y	Luminance, used either alone, for gray-scale images, or in combination with RY and BY for color images.
RY, BY	Chroma for luminance/chroma images, see above.
AR, AG, AB	Red, green and blue alpha/opacity, for colored mattes (required to composite images of objects like colored glass correctly).

For deep images:

Name	Interpretation
Z	Distance of the front of a sample from the viewer.
ZBack	Distance of the back of a sample from the viewer.

Name	Interpretation
A	Alpha/opacity of a sample.
R, G, B	Red, green and blue values of a sample.
AR, AG, AB	Red, green and blue alpha/opacity of a sample. Required for representing images of objects like colored glass, as well as for computing colored shadows.
id	A numerical identifier for the object represented by a sample. Samples that belong to the same object have the same numerical identifier.

For more information on the channels in deep images, see the separate "Interpreting OpenEXR Deep Pixels" document.

Layers

In an image file with many channels it is sometimes useful to group the channels into layers, that is, into sets of channels that logically belong together. Grouping is done using a naming convention: channel *C* in layer *L* is called *L.C*.

For example, an image may contain separate R, G and B channels for light that originated at each of several different virtual light sources. The channels in such an image might be called "light1.R", "light1.G", "light1.B", "light2.R", "light2.G", "light2.B", etc.

Layers can be nested. A name of the form $L_1.L_2.L_3 \ldots L_n.C$ means that layer L_1 contains a nested layer L_2, which in turn contains another nested layer L_3, and so on to layer L_n, which contains channel C.

For example, "light1.specular.R" identifies the "R" channel in the "specular" sub-layer of layer "light1".

Note that this naming convention does not describe a back-to-front stacking order or any compositing operations for combining the layers into a final image.

Color, Alpha and Compositing of Flat Images

The "over" operation for flat images creates a new composite image by stacking a foreground image and a background image so that the foreground image appears to be in front of the background image.

The A, AR, AG, and AB channels in a flat image represent a pixel's alpha or opacity: 0.0 means the pixel is transparent; 1.0 means the pixel is opaque.

By convention, the color channels of a pixel directly represent the amount of light that reaches the viewer from that pixel (as opposed to an amount of light that must first be multiplied by alpha). With this representation, computing

$$composite = foreground + (1 - alpha) \cdot background$$

performs a correct "over" operation.

This color representation is usually referred to as “premultiplied color.” Some other image file formats employ a representation called “straight” or “un-premultiplied color,” where an “over” operation requires computing

$$composite = alpha \cdot foreground + (1 - alpha) \cdot background.$$

Calling the color channels “premultiplied” does not mean that the color values in an image have actually been multiplied by alpha at some point during the creation of the image, or that pixels with zero alpha and non-zero color channels are illegal. Non-zero color with zero alpha is legal; such a pixel represents an object that emits light even though it is completely transparent, for example, a candle flame or a lens flare.

In the visual effects industry premultiplied color channels are the norm, and application software packages typically use internal image representations that are also premultiplied.

However, some application programs have an internal image representation with straight color. Since pixels with zero alpha and non-zero color can and do occur in OpenEXR images, application programs with straight color channels should take care to avoid discarding the color information in pixels with zero alpha. After reading an OpenEXR image such an application must convert the pixels to straight color by dividing the color channels by alpha. This division fails when alpha is zero. The application software could set all color channels to zero wherever the alpha channel is zero, but this might alter the image in an irreversible way. For example, a fully transparent flame on top of a candle would simply disappear and could not be recovered.

If the application’s internal straight image representation uses 32-bit floating-point numbers then one way around this problem might be to set alpha to $\max(h, alpha)$ before dividing, where h is a very small but positive value (h should be a power of two and less than half of the smallest positive 16-bit floating-point value). This way the result of the division becomes well-defined, and the division can be undone later, when the image is saved in a new OpenEXR file. Depending on the application software there may be other ways to preserve color information in pixels with zero alpha.

Credits

OpenEXR was developed at Industrial Light & Magic, a division of Lucas Digital Ltd. LLC, Marin County, California.

The OpenEXR file format was designed and implemented by Florian Kainz, Wojciech Jarosz, and Rod Bogart. The PIZ compression scheme is based on an algorithm by Christian Rouet. Josh Pines helped extend the PIZ algorithm for 16-bit and found optimizations for the float-to-half conversions. Drew Hess packaged and adapted ILM's internal source code for public release. The PXR24 compression method is based on an algorithm written by Loren Carpenter at Pixar Animation Studios.

The deep image representation in OpenEXR 2.0 is based on the ODZ file format that was developed by Peter Hillman. The OpenEXR 2.0 file format was designed and implemented by Peter Hillman at Weta Digital, and Piotr Stanczyk and Yunfeng Bai at Industrial Light & Magic.