OpenColorIO Documentation

Release 1.0.9

Sony Pictures Imageworks

September 20, 2015

CONTENTS

OpenColorIO (OCIO) is a complete color management solution geared towards motion picture production with an emphasis on visual effects and computer animation. OCIO provides a straightforward and consistent user experience across all supporting applications while allowing for sophisticated back-end configuration options suitable for high-end production usage. OCIO is compatible with the Academy Color Encoding Specification (ACES) and is LUT-format agnostic, supporting many popular formats.

OpenColorIO is released as version 1.0 and has been in development since 2003. OCIO represents the culmination of years of production experience earned on such films as SpiderMan 2 (2004), Surf's Up (2007), Cloudy with a Chance of Meatballs (2009), Alice in Wonderland (2010), and many more. OpenColorIO is natively supported in commercial applications like Katana, Mari, Silhouette FX, and others coming soon.

OpenColorIO is free and is one of several open source projects actively sponsored by Sony Imageworks.

http://opencolorio.org

CHAPTER

ONE

MAILING LISTS

There are two mailing lists associated with OpenColorIO:

ocio-users@googlegroups.com For end users (artists, often) interested in OCIO profile design, facility color management, and workflow.

ocio-dev@googlegroups.com For developers interested OCIO APIs, code integration, compilation, etc.

CHAPTER

TWO

USING OPENCOLORIO

Most users will likely want to use the OpenColorIO that comes precompiled with their applications. See the *Compatible Software* for further details on each application.

Note that OCIO configurations aren't required to do any 'real' work, and are available separately on the *Downloads* section of this site. Example images are also available. For assistance customizing .ocio configurations, contact the ocio-users email alias.

- Step 1: set the OCIO environment-variable to /path/to/your/profile.ocio
- Step 2: Launch supported application.

If you are on a platform that is not envvar friendly, most applications also provide a menu option to select a different OCIO configuration after launch.

Please be sure to select a profile that matches your color workflow (VFX work typically requires a different profile than animated features). If you need assistance picking a profile, email ocio-users.

CHAPTER

THREE

DOWNLOADING AND BUILDING THE CODE

Source code is available on Github at http://github.com/imageworks/OpenColorIO

Download a .zip or .tar.gz of the current state of the repository.

Please see the *Developer guide* for more info, and contact ocio-dev with any questions.

3.1 Compatible Software

The following sofware all supports OpenColorIO (to varying degrees).

If you are interested in getting OCIO support for an application not listed here, please contact the ocio-dev mailing list.

If you are a developer and would like assistance integration OCIO into your application, please contant ocio-dev as well.

3.1.1 After Effects

Compositor - Adobe

An OpenColorIO plugin is available for use in After Effects.

See src/aftereffects if you are interested in building your own OCIO plugins.

Pre-built binaries are also available for download, courtesy of fnordware.

3.1.2 Blender

Open Source 3D Application

In version 2.64, Blender has integrated OCIO as part it's redesigned color management system.

3.1.3 Krita

2D Paint - Open Source

Krita now support OpenColorIO for image viewing, allowing for the accurate painting of float32/OpenEXR imagery.

See krita.org for details.

3.1.4 Silhouette

Roto, Paint, Keying - SilhouetteFX

OCIO is natively integrated in 4.5+ Full support is provide for both image import/export, as well as image display.

3.1.5 Nuke

Compositor - The Foundry

Nuke 6.3v7+ ships with native support for OpenColorIO. The OCIO configuration is selectable in the user preferences.

OCIO Nodes: OCIOCDLTransform, OCIOColorSpace, OCIODisplay, OCIOFileTransform, OCIOLookConvert, OCIOLogConvert

The OCIODisplay node is suitable for use in the Viewer as an input process (IP), and a register function is provides to add viewer options for each display upon launch.

The OCIO config "nuke-default" is provided, which matches the built-in Nuke color processing. This profile is useful for those who want to mirror the native nuke color processing in other applications. (The underlying equations are also provided as python code in the config as well).

A video demonstration of the Nuke OCIO workflow.

3.1.6 Mari

3D Paint - The Foundry

Mari 1.4v1+ ships with native support for OpenColorIO in their display toolbar.

A video demonstration of the Mari OCIO workflow.

3.1.7 Katana

CG Pipeline / Lighting Tool - The Foundry

Color management in Katana (all versions) natively relies on OCIO.

2D Nodes: OCIODisplay, OCIOColorSpace, OCIOCDLTransform Monitor Panel: Full OCIO Support

3.1.8 Hiero

Conform & Review - The Foundry

Hiero 1.0 will ship with native support for OCIO in the display and the equivalent of Nuke's OCIOColorSpace in the Read nodes.

It comes with "nuke-default" OCIO config by default, so the Hiero viewer matches when sending files to Nuke for rendering.

3.1.9 Photoshop

OpenColorIO display luts can be exported as ICC profiles for use in photoshop. The core idea is to create an .icc profile, with a valid description, and then to save it to the proper OS icc directory. (On OSX, `~/Library/ColorSync/Profiles/`). Upon a Photoshop relaunch, Edit->Assign Profile, and then select your new OCIO lut.

See the the OCIO user guide for details on baking ICC profiles for Photoshop

3.1.10 OpenImageIO

Open Source Image Library / Renderer Texture Engine

Available in the current code trunk. Integration is with makecolortx (allowing for color space conversion during mipmap generation), and also through the public header src/include/color.h .

Remaining integration tasks include color conversion at runtime .

3.1.11 C++

The core OpenColorIO API is avaiable for use in C++. See the export directory for the C++ API headers. Minimal code examples are also available in the source code distribution. Of particular note are src/apps/ocioconvert/ and src/apps/ociodisplay/

Also see the *Developer guide*

3.1.12 Python

The OpenColorIO API is available for use in python. See the "pyglue" directory in the codebase.

See the devleoper guide for usage examples and API documentation on the PYthon bindings

3.1.13 Vegas Pro

Video editing - Sony

Vegas Pro 12 uses OpenColorIO, supporting workflows such as S-log footage via the ACES colorspace.

3.1.14 Apps w/icc or luts

flame (.3dl), lustre (.3dl), cinespace (.csp), houdini (.lut), iridas_itx (.itx) photoshop (.icc)

Export capabilities through ociobakelut:

```
$ ociobakelut -- create a new LUT or icc profile from an OCIO config or lut file(s)
$
$ usage:  ociobakelut [options] <OUTPUTFILE.LUT>
$
$ example:  ociobakelut --inputspace lg10 --outputspace srgb8 --format flame lg_to_srgb.3dl
$ example:  ociobakelut --lut filmlut.3dl --lut calibration.3dl --format flame display.3dl
$ example:  ociobakelut --lut look.3dl --offset 0.01 -0.02 0.03 --lut display.3dl --format flame disp
$ example:  ociobakelut --inputspace lg10 --outputspace srgb8 --format icc ~/Library/ColorSync/Profil
$ example:  ociobakelut --lut filmlut.3dl --lut calibration.3dl --format icc ~/Library/ColorSync/Prof
$
$
$ Using Existing OCIO Configurations
$     --inputspace %s      Input OCIO ColorSpace (or Role)
$     --outputspace %s     Output OCIO ColorSpace (or Role)
$     --shaperspace %s     the OCIO ColorSpace or Role, for the shaper
$     --iconfig %s         Input .ocio configuration file (default: $OCIO)
$
```

```
$ Config-Free LUT Baking
$     (all options can be specified multiple times, each is applied in order)
$     --lut %s              Specify a LUT (forward direction)
$     --invlut %s           Specify a LUT (inverse direction)
$     --slope %f %f %f      slope
$     --offset %f %f %f     offset (float)
$     --offset10 %f %f %f   offset (10-bit)
$     --power %f %f %f      power
$     --sat %f              saturation (ASC-CDL luma coefficients)
$
$ Baking Options
$     --format %s           the lut format to bake: flame (.3dl), lustre (.3dl),
$                           cinespace (.csp), houdini (.lut), iridas_itx (.itx), icc (.icc)
$     --shapersize %d       size of the shaper (default: format specific)
$     --cubesize %d         size of the cube (default: format specific)
$     --stdout              Write to stdout (rather than file)
$     --v                   Verbose
$     --help                Print help message
$
$ ICC Options
$     --whitepoint %d       whitepoint for the profile (default: 6505)
$     --displayicc %s       an icc profile which matches the OCIO profiles target display
$     --description %s      a meaningful description, this will show up in UI like photoshop
$     --copyright %s        a copyright field
```

See this ocio-dev thread for additional usage discussions.

When exporting an ICC Profile, you will be asked to specify your monitor's profile (it will be selected for you by default). This is because ICC Profile are not LUTs per se. An ICC Profile describes a color space and then needs a destination profile to calculate the transformation. So if you have an operation working and looking good on the monitor you're using (and maybe its profile has been properly measured using a spectrophotometer), then choose your display. If the transform was approved on a different monitor, then maybe you should choose its profile instead.

3.1.15 RV (Beta)

Playback Tool - Tweak Software

OCIO support in RV is currently being developed by Ben Dickson (dbr).

See this email thread for additional details.

This integration is currently considered a work in progress, and should not be relied upon for critical production work.

3.1.16 Java (Beta)

The OpenColorIO API is available for use in Java. See the jniglue directory in the codebase.

This integration is currently considered a work in progress, and should not be relied upon for critical production work.

3.1.17 Ramen (Beta)

Open Source Compositor

Under development, with native OCIO color managment.

3.1.18 CryEngine3 (Beta)

Game Engine - Crytek (Cinema Sandbox)

CryENGINE is a real-time game engine, targeting applications in the motion-picture market. While we don't know many details about the CryEngine OpenColorIO integration, we're looking forward to learning more as information becomes available.

3.2 Configurations

This section gives an overview of what existing (public) OCIO configuration exist, and how to create new ones.

OCIO Configurations can be downloaded here: .zip .tar.gz (OCIO v1.0+)

If you are interested in crafting custom color configurations, and need assistance, please contact: ocio-dev@googlegroups.com

3.2.1 Sony Pictures Imageworks Color Pipeline

This document describes a high-level overview on how to emulate the current color management practice at Sony Imageworks. It applies equally to all profiles used at Imageworks, including both the VFX and Animation profiles. It's by no means a requirement to follow this workflow at your own facility, this is merely a guideline on how we choose to work.

General Pipeline Observations

- All images, on disk, contain colorspace information as a substring withing the filename. This is obeyed by all applications that load image, write images, or view images. File extensions and metadata are ignored with regards to color processing.

  Example:

  ```
  colorimage_lnf.exr  : lnf
  dataimage_ncf.exr : ncf
  plate_lg10.dpx : lg10
  texture_dt8.tif: dt8
  ```

  Note: fileformat extension does NOT imply a color space. Not all dpx files are lg10. Note all tif images are dt8.
- The common file formats we use are exr, tif, dpx.
- render outputs: exr
- render inputs (mipmapped-textures): exr, tif (txtif)
- photographic plates (scans): dpx
- composite outputs: dpx, exr
- on-set reference: (camera raw) NEF, CR2, etc.
- painted textures: psd, tif
- output proxies: jpg
- All pipelines that need to be colorspace aware rely on `Config.parseColorSpaceFromString`.

- Color configurations are show specific. The $OCIO environment variable is set as part of a 'setshot' process, before other applications are launched. Artists are not allowed to work across different shows without using a fresh shell + setshot.
- While the list of colorspaces can be show specific, care is taken to maintain similar naming to the greatest extent feasible. This reduces artist confusion. Even if two color spaces are not identical across shows, if they serve a similar purpose they are named the same. Example: We label 10-bit scanned film negatives as lg10. Even if two different shows use different acquisition film stocks, and rely on different linearization curves, they are both labelled lg10.
- There is no explicit guarantee that image assets copied across shows will be transferable in a color-correct manner. For example, in the above film scan example, one would not expect that the linearized versions of scans processed on different shows to match. In practice, this is not a problematic issue as the colorspaces which are convenient to copy (such as texture assets) happen to be similarly defined across show profiles.

Rendering

- Rendering and shading occurs in a scene-linear floating point space, typically named "ln". Half-float (16-bit) images are labelled lnh, full float images (32-bit) are labelled lnf.
- All image inputs should be converted to ln prior to render-time. Typically, this is done when textures are published. (See below)
- Renderer outputs are always floating-point. Color outputs are typically stored as lnh (16-bit half float). Data outputs (normals, depth data, etc) are stored as ncf ("not color" data, 32-bit full float). Lossy compression is never utilized.
- Render outputs are always viewed with an OCIO compatible image viewer. Thus, for typical color imagery the lnf display transform will be applied. In Nuke, this can be emulated using the OCIODisplay node. A standalone image viewer, ociodisplay, is also included with OpenColorIO src/example.

Texture Painting / Matte Painting

- Textures are painted in non-OCIO color-managed environment. (Photoshop, etc).
- At texture publish time, before mipmaps are generated, all color processing is applied. Internally at SPI we use a modified version of OpenImageIO's maketx that also links to OpenColorIO. We intend to make this code available as soon as possible. In the meantime, the OpenColorIO 'ocioconvert' script included in examples can be relied upon. Color processing (linearization) is applied before mipmap generation in order to assure energy preservation in the render. If the opposite processing order were used, (mipmap in the original space, color convert in the shader), the apparent intensity of texture values would change as the object approached to receeded to the camera.
- The original texture filenames contain the colorspace information as a substring, to signify processing intent.
- Textures that contain data (bump maps, opacity maps, blend maps, etc) are labelled with the nc colorspaces according to their bitdepth. Ex: an 8-bit opacity map -> skin_opacity_nc8.tif
- Painted textures that are intended to modulate diffuse color components are labelled dt (standing for "diffuse texture"). The dt8 colorspace is designed such that, when linearized, values will not extend above 1.0. At texture publishing time these are converted to lnh mipmapped tiffs/exr. Note that as linear textures have greater allocation requirements, a bit depth promotion is required in this case. I.e., even if the original texture as painted was only 8-bits, the mipmapped texture will be stored as a 16-bit float image.
- Painted environment maps, which may be emissive as labelled vd (standing for 'video'). These values, when linearized, have the potential to generate specular information well above 1.0. Note that in the current vd linearization curves, the top code values may be very "sensitive". I.e., very small changes in the initial code value (such as 254->255) may actually result in very large differences in the estimated scene-linear intensity.

All environment maps are store as lnh mipmapped tiffs/exr. The same bit-depth promotion as in the dt8 case is required here.

Compositing

- The majority of compositing operations happen in scene-linear, lnf, colorspace.
- All image inputs are linearized to lnf as they are loaded. Customized input nodes make this processing convenient. Rendered elements, which are stored in linear already, do not require processing. Photographic plates will typically be linearized according to their source type, (lg10 for film scans, gn10 for genesis sources, etc).
- All output images are de-linearized from lnf when they are written. A customized output node makes this convenient.
- On occasion log data is required for certain processing operations. (Plate resizing, pulling keys, degrain, etc). For each show, a colorspace is specified as appropriate for this operation. The artist does not have to keep track of which colorspace is appropriate to use; the OCIOLogConvert node is always intended for this purpose. (Within the OCIO profile, this is specified using the 'compositing_log' role).

3.2.2 spi-vfx

This is a real OCIO color profile in use at Sony Pictures Imageworks, and is suitable for use on visual effects (VFX) work. The concepts utilized in this profile have been successfully validated on a variety of Imageworks visual effects films, including Spider-Man, Alice In Wonderland, G-Force, and Green Lantern.

Conversion from film to/from scene-linear is a simple, trivially invertible 1D transform. The display transforms are complex, 3D film-print emulations.

In production, this profile is typically used before final color details are worked out. Although it sounds temporary, most of a film can be made off this configuration. Final color decisions for a film are often made long after significant work has been done. In some cases shots from a film can be finaled before the color details, such as which Digital Intermediate (DI) house will be used, are decided. Entire projects have been completed using this profile without modification.

This profile embodies two philosophies of color management familiar to those in production: *"Keep It Simple"*, and, *"Don't Be Evil"*.

The following steps outline a simplified visual effects color workflow:

- Load a plate (log film scan, digital motion picture camera, etc)
- Convert device color space to scene-linear
- Render and composite in scene-linear
- Convert from scene-linear to device color space
- Output final plate

It is absolutely critical to guarantee that process - end to end - is colorimetrically a no-op. Under no circumstances are any unintended modifications to the original image allowed.

Thus, this profile uses very simple (1D) conversions for all input and output color space conversions. All of the complexity (the 3D LUT film emulation lifting) is handled at display time, and is never baked (or unbaked) into the imagery. For visualization, this profile includes a generic Kodak Vision print emulation suitable for display on a reference sRGB monitor or a P3 Digital Cinema projector.

Caveot 1: Of course, we realize that there are many other color workflows that may be equally good (or better) than the one presented here. Please, if you have a successful alternative workflows share the details!

Caveot 2: We are not distributing the code that generates the luts from their underlying curve representations. While we hope to publish this eventually, at the current time this process relies on internal tools and we don't have the spare development cycles to port this code to something suitable for distribution.

Invertibility

Elements often need to be transferred back and forth many times between different colorspaces. Since it's impossible to know in advance how many times an image may be transferred between colorspaces it is essential for the majority of transformations to be lossless invertible transformations. By the end of the color pipeline even a 1 value difference in a 10bit transformation can become a significant issue. Invertible transformations can be taken from the source space, to linear and back with no change to the data. A higher value is placed on transformations being predictable and invertible than absolutely correct. All 1-d luts allow for forward and inverse transformations with no loss. Unless specified all channels are equally affected. The luts are 1 bit wider than stated, so lg8 actually defines 9 bits worth of entries. This allows the rounding in the inverse direction to be applied unambiguously (lossless).

Non-invertible transforms contain 3d lookups. 3D transformations can not be inverted due to gamut mapping issues. Non-invertible transformations are only used for final output media (such as QuickTimes) and for display purposes.

Film Emulation Inversion

Inverse film emulation luts aren't supported in a default configuration. Imageworks does not use a film emulation inversion lut for texture or matte paintings. In its place a film emulation preview lut, commonly as an ICC profile, is used. Although most film emulation luts are similar they do differ significantly. The DI facility creating final color is often chosen long after significant vfx work has been begun. The film luts the film will be finished on are not made until weeks, or days, before DI begins. So the 'true' lut that will be used for the finishing is not available until very late in the production, from a VFX perspective. There are many color gamut mapping issues that arise when inverting film to video lut. Using a film inversion lut at this stage would bake in a look that isn't quite right and is very difficult to fully un-bake. It is safer to work with images in a non-constrained way and apply a visualization that can be toggled on and off.

Scene Linear

lnf, lnh, ln16

Middle Gray: 0.18

The linear space is a photometrically linear space, representing high-dynamic range (HDR) pixel values in our virtual world. Middle gray is defined to be at 0.18. While the dynamic range of lnf is technically unbounded, pixel values for natural, well exposed imagery will often fall between +/- 8 stops from gray.

The scene linear colorspace is used for compositing, rendering, and is also the profile connection space. All colorspace conversions are made in relation how they transform into or out of scene linear.

The colors defined in linear are implicitly bounded by film negative sensitivities. The space is based off an OCN film scan where values refer to linear light at the focal plane. 0.18 in linear will correspond to a %18 percent grey card captured on filmplane under the same lighting conditions where diffuse white is 1.0. Values above 1.0 in any channel would indicate a 'specular', or light emitting objects.

lnf is a full precision (32-bit) floating point colorspace. It is used for rendering and compositing.

lnh is a half precision (16-bit) floating point colorspace. It is used for rendering and compositing.

ln16 is a 16 bit integer representation of the [0,1] range of lnf. This is no longer used but is kept if a legacy image needs to be loaded, or if linear images need to be loaded into an application that does not support float. Note that storing a float lnf image using an integer ln16 representation is destructive, and throws away data.

Film Log

lg8, lg10, lg16, lgf

Middle Gray: 445 (of 1023)

The log to linear curve is based on an analysis of several commonly used Kodak acquisition stocks. It was found that Kodak 5218 is right about in the middle in terms of tone response given the input imagery we receive. The curve incorporates some toe compensation. The curve gamma closely matches 5218. The transformation does not represent any single stock. The Imageworks log conversions are not channel specific, all color channels are transformed uniformly. Compositing productivity gains have been found using the toe compensations when compared to using a straight line log to linear curve. Shoulder compensation - while technically correct - detracted from compositing quality, often creating situations where grain film noise would result in larger than desired changes in linear light.

lg8, lg10, and lg16 are similar. They are all the same log to linear transformation but are explicitly defined to be lossless at the specified bit depths. The luts use nearest neighbor interpolation to search for an exact match. Significant performance gains were found when using the proper bitdepth lut. While using the lg16 conversion on an 8 bit image will yield the same result, it is measurably slower than using the 8-bit conversion (assuming 8-bits is all that is needed). This performance gap remains even on current graphics hardware.

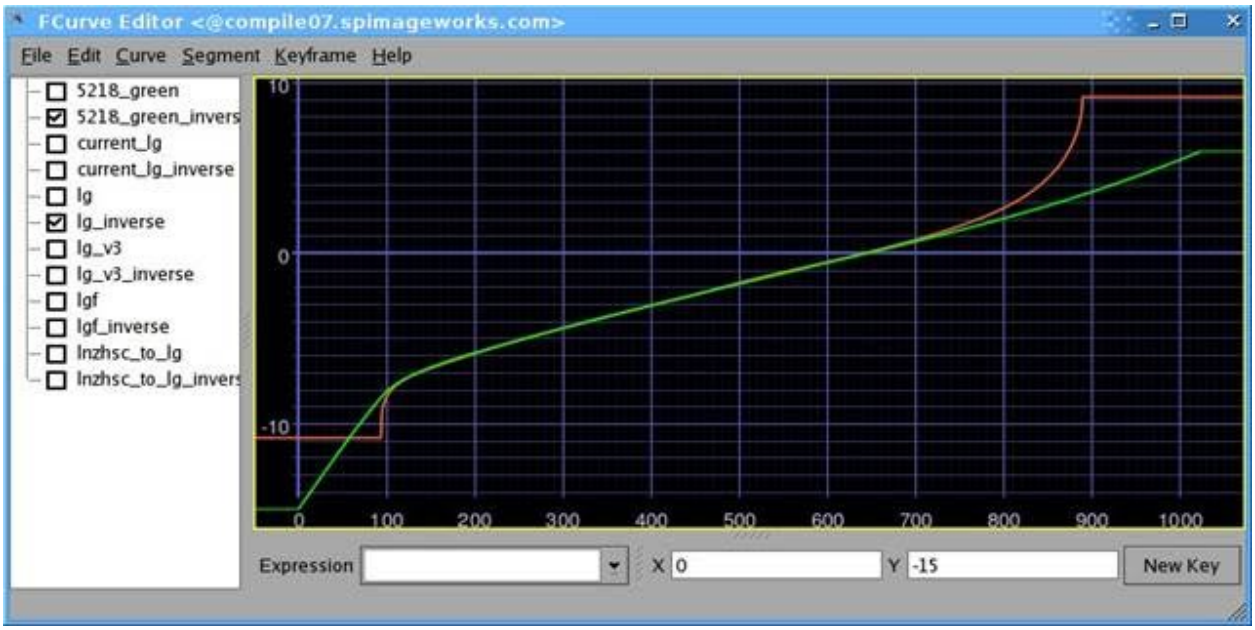


Figure 3.1: lg10 to linear light. The green curve represents the lg to ln conversion. The red curve show the green channel of a Kodak 5218 exposure test. The x-axis is in 10bit input lg the output is in lg base2 linear light units. 0.0 Represents diffuse white.

lg spaces specify 21 of stops of dynamic range. 0 in lg10 space is 15 stops below diffuse white. 445 correspond to 18% grey and is ~2.47 below diffuse white. 1023 in lg10 space is 6 stops above diffuse white.

lgf is identical on the range from 0-1 to the standard lg luts. It specifies an additional range below 0.0 and above 1.0. In 10 bit the spaces is defined from -512 to 2048. Lg color timing number from either on set color correction devices or from a DI house to be applied in a way that can be fully inverted out without loss. Lgf specifies 18 stops above the lg10 max and 36 stops below the log10 min with a total dynamic range of 85 stops. The space is designed to be bigger than needed.

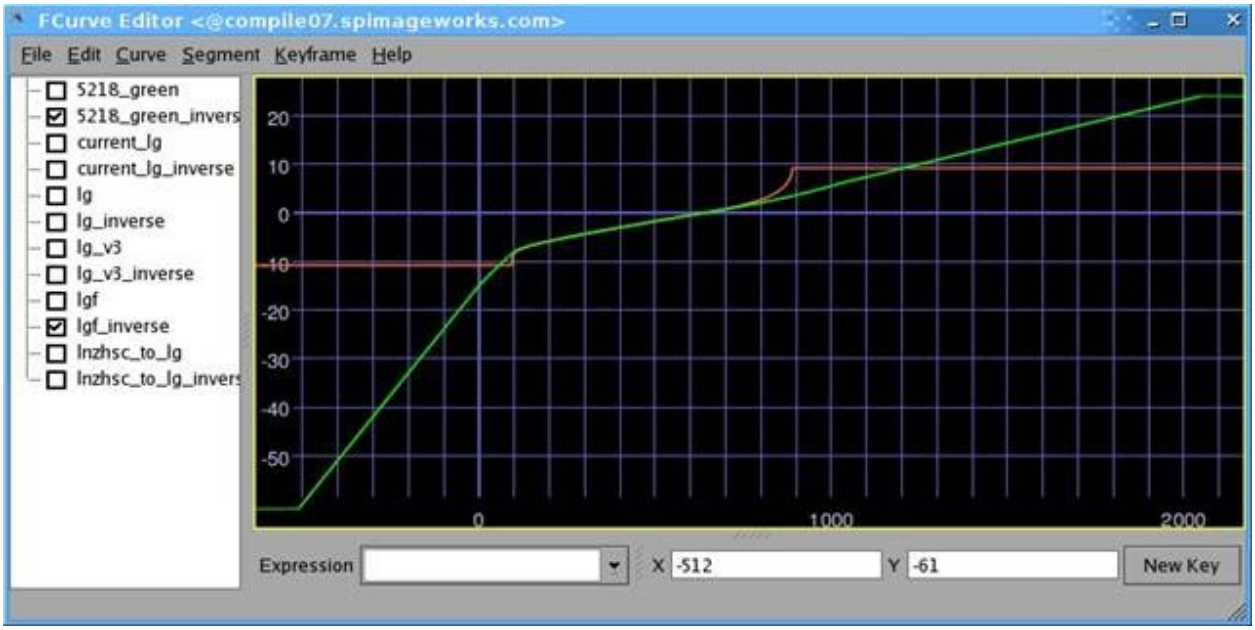


Figure 3.2: lgf to linear light. The green curve represents the lg to ln conversion. The red curve show the green channel of a Kodak 5218 exposure test. The x axis is in 10bit input lg the output is in log(base2) linear light units. 0 Represents diffuse white

Panalog (Genesis)

gn8, gn10, gn16, gnf

GN is the Imageworks Panalog space. It follows the Panalog specification and additionally extrapolates so all of the values from 0 to 1023 are defined. This was necessary due to compression artifacts that could create values below the Panalog specifications. gn8,10,16 are defined with diffuse white at 681, Max white is approximately 2.6 stops above diffuse white and black is approximately 12.6 stops below diffuse white. The dynamic range is less than that of lg.

gnf is similar in purpose and function to lgf. It is identical on the range from 0-1 to the regular gn and specifies an additional range below 0.0 and above 1.0. In 10 bit numbers gnf is defined from -255 to 3125. This allows for color timing number from either on set color correction devices or from a DI house to be applied in a way that can be fully inverted. Additionally it allows for lg10 based image data to be fully represented without clipping.

gnf specifies 14.5 stops above the gn10 max and 18 of stops below the gn10 min. The entire range of gnf is 47 stops.

Reference Art

vd8, vd16, vdf, hd10

The vd spaces are mappings of linear image data into display space. The main part of the transformation is defined as a single curve that is conceptually two parts. The first is a ln to lg conversion. The second is lg to sRGB conversion. This is based off the neutral channel response of the sRGB film emulation lut used in the profile. The dynamic range of the vd colorspace is limited. It is undesirable to map the vd max to the linear max. Such a conversion results in linear values are almost never what an artist intended. The rule of thumb is that, at the high end, single value deltas in an 8 bit image should never create over a half stop of additional linear light. The vd conversion curve is limited to prevent this case.

The dynamic range of the vd is limited to around 2.5 stops above diffuse white. This has two positive attributes. It allows vd to be used to directly on matte paintings. It also allows all of vd to be represented in a gn10 image. The last

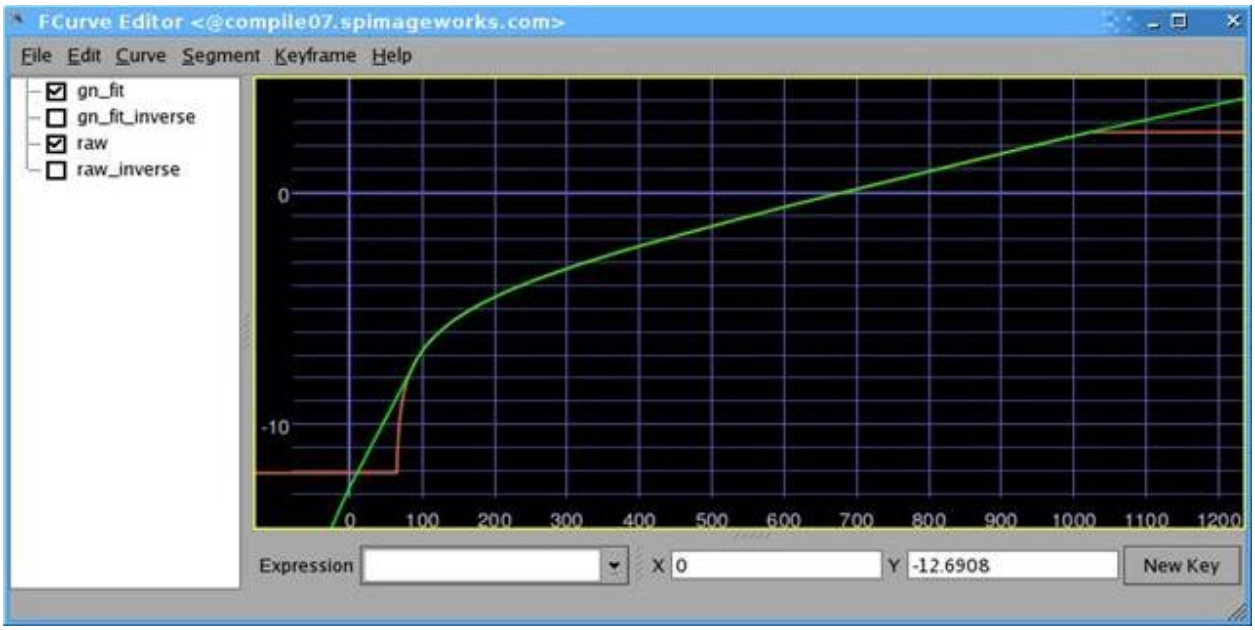


Figure 3.3: gn10 to linear light. the x axis is in 10bit Panalog values. The Y axis is in linear light. The green curve is the gn curve. the red curve is the Panalog data.

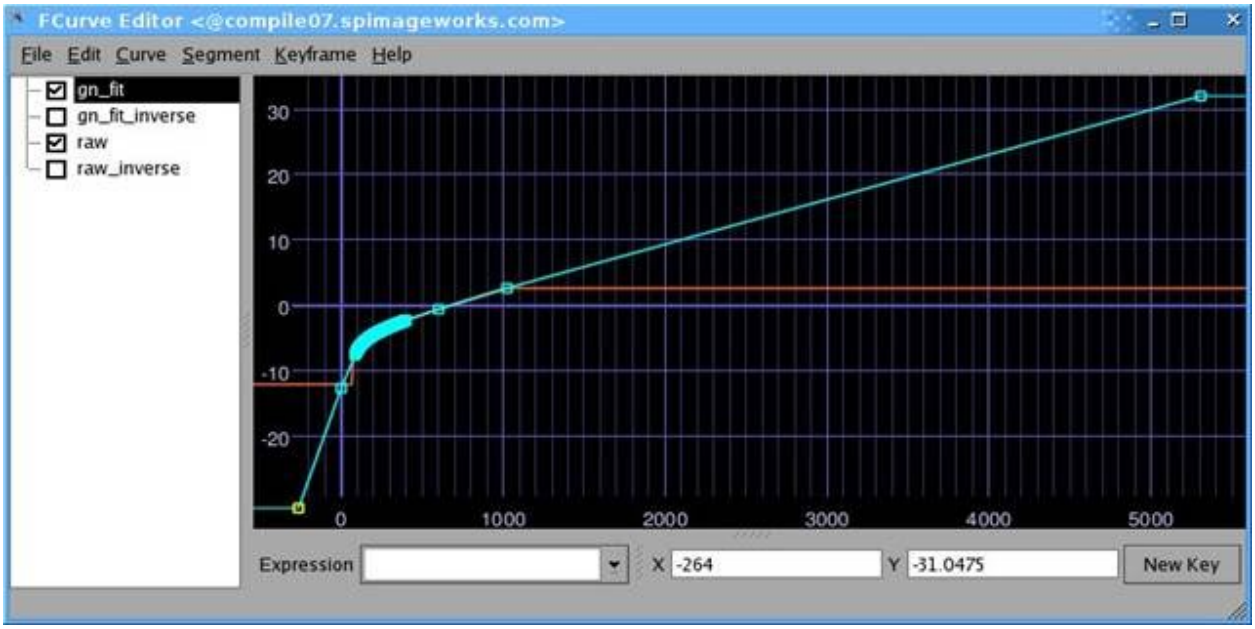


Figure 3.4: gnf to linear light. the x axis is in 10bit Panalog values. The Y axis is in linear light. The green curve is the gn curve. the red curve is the Panalog data. Only a subset of the function is used to define the gnf solorspace

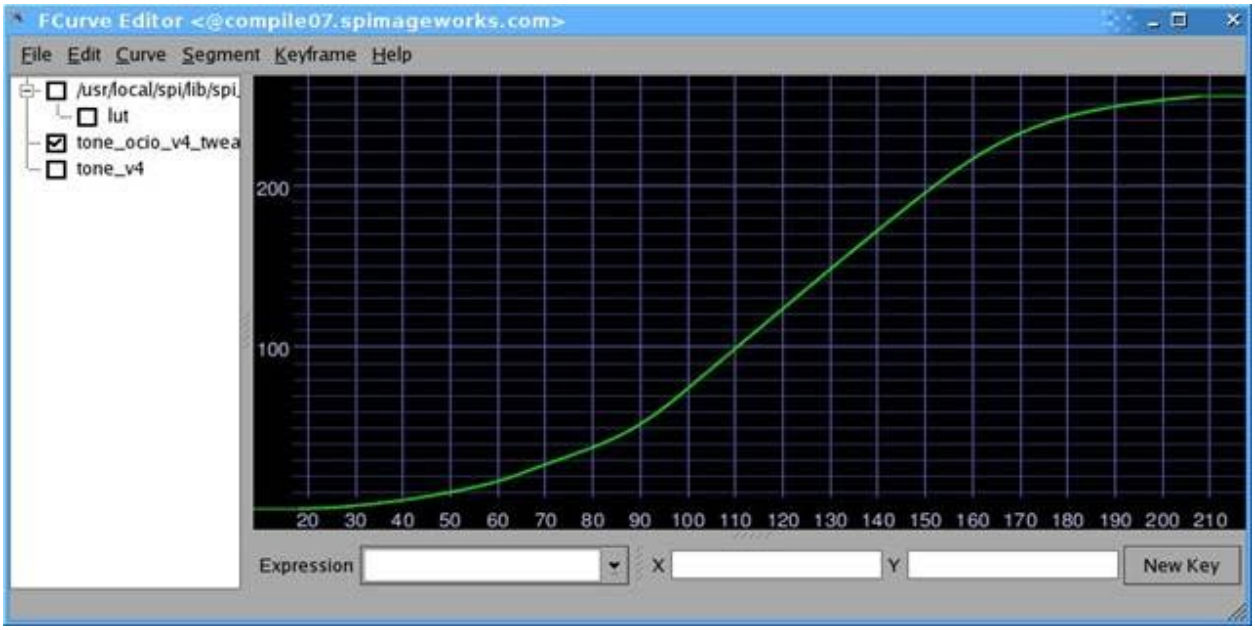


Figure 3.5: The curve used to map from Lg8 to vd 8. The x-axis is in lg8 units, the y-axis is in vd8 units.

part of the transformation is a matrix transformation that moves the whitepoint of film to look correct when displayed with a d65 whitepoint.

The main use of this colorspace is to import RGB images with an unknown colorspace. This colorspace no longer gets much use alone; However it is an integral part of many conversions. It is also part of the matte painting and diffuse texture pipelines.

vdf works differently than the other floating spaces. It still only defines the color transformation from 0-1. This colorspace is used when we receive video space encoded exr's.

HD10 is a vd based space that is used for importing and exporting REC709 range broadcast material. This works very well for broadcast camera native material and poorly for material with a film emulation lut baked in. This transformation works well exporting film based material to tape, even though it lacks a film emulation lut. It does not give an accurate color rendering but created a pleasing image that makes clients happy.

Structurally the conversion is a matrix operation that scales the data then adds an offset to limit the range from 64-940. From there the standard vd transformation is applied.

Texture Painting

dt8, dt16

DT 8,16 - Diffuse texture colorspaces. These colorspaces are used for the conversion of textures painted in video space into a range limited linear space for use in rendering. The Color space is based on the vd transformation but limits the conversion into linear space so that no values above diffuse white can be created. This ensures that textures do not add light to a render. This is achieved by using a matrix transformation to limit the mapping of vd to the linear value of diffuse white.

Data

nc8, nc10, nc16, ncf

Non-Color (NC) spaces are used to hold any data that needs to be processed though the color pipeline unaltered. These are data spaces and can hold anything such as point clouds, normals, untagged reference art, etc. These data spaces do not get transformations applied at any point.

Display Transforms

srgb8, p3dci8, xyz16

srgb8 bakes in the film3d emulation lut. This table can be used for either QuickTime generation or output to the sRGB display. The transformation is a 3d film emulation table with gray balance compensation, so a value of 445,445,445 in lg10 space os modified to become equal RGB values in sRGB. Additionally the lut is scaled so that at least one of the color channels on maximum white uses the display max.

The transformation takes place in three parts. First the linear data is converted to the show log space. Then a film emulation table is applied. Then the grey balance and white scaling compensation are applied. This table is designed to be evaluated in a dimly lit office environment on a sRGB display.

p3dci8 is an implementation of film emulation table that has an output of DCI P3. This is only ever used for driving DLPs for display. The transformation has two parts. First the linear image data is converted to lg10 based image data then the DCI P3 film emulation lut is applied. No additional compensations are made.

xyz16 is designed for the creation of a Digital Cinema Distribution Master (DCDM). The color matches that of the P3 output (dlpqt8), but has an additional output transformation to convert to X'Y'Z'. The transformation takes the linear image data and converts it to lg, then applies the filmlook. The data is then in DCI P3 colorspace. That data is converted to display linear P3, using an inverse gamma curve. A matrix conversion is then used to transform from DCI P3' into XYZ'. The final step is to reapply the gamma 2.6 to result in XYZ16 values.

In this profile each display has three identical looks defined. The names are kept consistent between devices to minimize confusion. OCIO uses a specific tag to associate colorspaces with displays. The tags are nothing more than links to already defined colorspaces.

Film is the image displayed though a film emulation lut. This visualization is display compensated and should visually match between a sRGB display and a P3 projector. The goal is to match how the film will look in a DI. The luts in use for this profile roughly match the Sony ColorWorks environment.

Raw visualization shows the image data on the screen with no display compensation. This is used for image debugging purposes, for instance to see if potential image discontinuities are in the source data or the visualization.

Log visualization displays the image as if it were converted to the show specific log. This is transformation also has no display compensation. The common use for this us to see how well elements fit into the comp without the film emulation lut disguising any flaws. Old school compositors love it for grain matching.

sRGB Film : srgb8 sRGB Raw : nc10 sRGB Log : lg10

DCIP3 Raw : nc10 DCIP3 Log : lg10 DCIP3 Film : dlpqt8

Display Calibration

sRGB is the supported desktop display specification, calibrated to the sRGB standard and viewed in a dim office environment. As Imageworks switched from crt based display devices to LCD based devices a number of possible colorspaces were explored. It was a long decision but sRGB was chosen for a few reasons. An important one was that almost every display manufacturer can implement sRGB, reasonable well. This becomes a boon when we work needs to be done outside of our main facilities. Even a consumer display with calibration can come close to matching the sRGB standard. Since so many monitor manufacturers can hit sRGB calibration we are not tied to purchasing from a specific vendor. It becomes unnecessary to specify a specific display to with productions or external vendors. It also reduces the amount of studio specific color requirements that need to be communicated when working with other facilities. 80 cd/m^2, D65 white point, srgb gamma function (approx 2.4)

P3 was deemed especially unusable on the desktop. The full specification requires a white point of 48 cd/m^2. To adapt P3 for desktop use (in dim ambient environments), the whitepoint luminance needed to be raised. The specified 2.6 gamma is very challenging to the current display technology on very dark colors. This meant that we would have a special Imageworks video P3. A custom colorspace wouldn't make compositing better and would require a conversation, or conversion, every time video was sent out of house.

DCIP3 is a projector calibrated to DCI P3 mastering specification in a theatrical mastering environment. We use a mix of display technologies, SXRD and DLP, depending on application. Gamma 2.6. Traditional DCI calibration.

3.2.3 spi-anim

This is a real OCIO color profile in use at Sony Pictures Imageworks, and is suitable for use on animated features. The concepts utilized in this profile have been successfully validated on a variety of Sony Pictures Animation features including Cloudy With A Chance Of Meatballs, Surf's Up, and Arthur Christmas.

3.2.4 Nuke default

This profile corresponds to the default Nuke color configuration (currently generated from Nuke 6.1).

If you have made modifications to a nuke color configuration, and wish to re-export your own custom OCIO profile, please refer to the nuke_to_ocio utility script distributed with OpenColorIO.

The following color transforms are defined:

- linear
- sRGB
- rec709
- Cineon
- Gamma 1.8
- Gamma 2.2
- Panalog
- REDLog
- ViperLog
- REDSpace

3.2.5 How to Configure ColorSpace Allocation

The allocation / allocation vars are utilized using during GPU 3dlut / shader text generation. (Processor::getGpuShaderText, Processor::getGpuLut3D).

If, in the course of GPU processing, a 3D lut is required, the "allocation / allocation vars" direct how OCIO should sample the colorspace, with the intent being to maintain maximum fidelity and minimize clamping.

Currently support allocations / variables:

ALLOCATION_UNIFORM:: 2 vars: [min, max]

ALLOCATION_LG2:: 2 vars: [lg2min, lg2max] 3 vars: [lg2min, lg2max, linear_offset]

So say you have an srgb image (such as an 8-bit tif), where you know the data ranges between 0.0 - 1.0 (after converting to float). If you wanted to apply a 3d lut to this data, there is no danger in samplingthat space uniformly and clamping data outside (0,1). So for this colorspace we would tag it:

```
allocation: uniform
allocationvars: [0.0, 1.0]
```

These are the defaults, so the tagging could also be skipped.

But what if you were actually first processing the data, where occasionally small undershoot and overshoot values were encountered? If you wanted OCIO to preserve this overshoot / undershoot pixel information, you would do so by modifying the allocation vars.

```
allocation: uniform
allocationvars: [-0.125, 1.125]
```

This would mean that any image data originally within [-0.125, 1.125] will be preserved during GPU processing. (Protip: Data outside this range *may* actually be preserved in some circumstances - such as if a 3d lut is not needed - but it's not required to be preserved).

So why not leave this at huge values (such as [-1000.0, 1000.0]) all the time? Well, there's a cost to supporting this larger dynamic range, and that cost is reduced precision within the 3D luts sample space. So in general you're best served by using sensible allocations (the smallest you can get away with, but no smaller).

Now in the case of high-dynamic range color spaces (such as float linear), a uniform sampling is not sufficient because the max value we need to preserve is so high.

Say you were using a 32x32x32 3d lookup table (a common size). Middle gray is at 0.18, and specular values are very much above that. Say the max value we wanted to preserve in our coding space is 256.0, each 3d lut lattice coordinates would represent 8.0 units of linear light! That means the vast majority of the perceptually significant portions of the space wouldnt be sampled at all!

unform allocation from 0-256: 0 8.0 16.0 ... 240.0 256.0

So another allocation is defined, lg2

```
- !<ColorSpace>
  name: linear
  description: |
      Scene-linear, high dynamic range. Used for rendering and compositing.
  allocation: lg2
  allocationvars: [-8, 8]
```

In this case, we're saying that the appropriate ways to sample the 3d lut are logarithmically, from 2^-8 stops to 2^8 stops.

Sample locations: 2^-8: 0.0039 2^-7: 0.0078 2^-6: 0.0156 ... 2^0: 1.0 ... 2^6: 64.0 2^7: 128.0 2^8: 256.0

Which gives us a much better perceptual sampling of the space.

The one downside of this approach is that it can't represent 0.0, which is why we optionally allow a 3d allocation var, a black point offset. If you need to preserve 0.0 values, and you have a high dynamic range space, you can specify a small offset.

Example:

```
allocation: lg2
allocationvars: [-8, 8, 0.00390625]
```

The [-15.0, 6.0] values in spi-vfx come from the fact that all of the linearizations provided in that profile span the region from 2^-15 stops, to 2^6 stops. One could probably change that black point to a higher number (such as -8), but if you raised it too much you would start seeing black values be clipped. Conversely, on the high end one could

raise it a bit but if you raised it too far the precision would suffer around gray, and if you lowered it further you'd start to see highlight clipping.

3.3 Installation

3.3.1 The easy way

While prebuilt binaries are not yet available for all paltforms, OCIO is available via several platform's package managers.

Fedora and RHEL

In Fedora Core 15 and above, the following command will install OpenColorIO:

```
yum install OpenColorIO
```

Providing you are using the Fedora EPEL repository (see the FAQ for instructions), this same command will work for RedHat Enterprise Linux 6 and higher (including RHEL derivatives such as CentOS 6 and Scientific Linux 6)

OS X using Homebrew

You can use the Homebrew package manager to install OpenColorIO on OS X.

First install Homebrew as per the instructions on the Homebrew homepage (or see the Homebrew wiki for more detailed instructions)

Then simply run the following command to install:

```
brew install opencolorio
```

3.3.2 Building from source

While there is a huge range of possible setups, the following steps should work on OS X and most Linux distros.

The basic requirements are:

- cmake >= 2.8
- (optional) Python 2.x (for the Python bindings)
- (optional) Nuke 6.x (for the Nuke nodes)
- (optional) OpenImageIO (for apps including ocioconvert)
- (optional) Truelight SDK (for TruelightTransform)

To keep things simple, this guide will use the following example paths - these will almost definitely be different for you:

- source code: `/source/ocio`
- the temporary build location: `/tmp/ociobuild`
- the final install directory: `/software/ocio`

First make the build directory and cd to it:

```
$ mkdir /tmp/ociobuild
$ cd /tmp/ociobuild
```

Next step is to run cmake, which looks for things such as the compiler's required arguments, optional requirements like Python, Nuke, OpenImageIO etc

As we want to install OCIO to a custom location (instead of the default `/usr/local`), we will run cmake with `CMAKE_INSTALL_PREFIX`

Still in `/tmp/ociobuild`, run:

```
$ cmake -D CMAKE_INSTALL_PREFIX=/software/ocio /source/ocio
```

The last argument is the location of the OCIO source code (containing the main CMakeLists.txt file). You should see something along the lines of:

```
-- Configuring done
-- Generating done
-- Build files have been written to: /tmp/ociobuild
```

Next, build everything (with the `-j` flag to build using 8 threads):

```
$ make -j8
```

This should complete in a few minutes. Finally, install the files into the specified location:

```
$ make install
```

If nothing went wrong, `/software/ocio` should look something like this:

```
$ cd /software/ocio
$ ls
bin/     include/ lib/
$ ls bin/
ocio2icc    ociobakelut ociocheck
$ ls include/
OpenColorIO/   PyOpenColorIO/ pkgconfig/
$ ls lib/
libOpenColorIO.a      libOpenColorIO.dylib
```

Enabling optional components

The OpenColorIO library is probably not all you want - the Python libraries bindings, the Nuke nodes and several applications are only built if their dependencies are found.

In the case of the Python bindings, the dependencies are the Python headers for the version you wish to use. These may be picked up by default - if so, when you run cmake you would see:

```
-- Python 2.6 okay, will build the Python bindings against .../include/python2.6
```

If not, you can point cmake to correct Python executable using the `-D PYTHON=...` cmake flag:

```
$ cmake -D PYTHON=/my/custom/python2.6 /source/ocio
```

Same process with Nuke (although it less likely to be picked up automatically). Point cmake to your Nuke install directory by adding `-D NUKE_INSTALL_PATH`:

```
$ cmake -D PYTHON=/my/custom/python2.6 -D NUKE_INSTALL_PATH=/Applications/Nuke6.2v1/Nuke6.2v1.app/Co
```

The NUKE_INSTALL_PATH directory should contain the Nuke executable (e.g Nuke6.2v1), and a include/ directory containing DDImage/ and others.

If set correctly, you will see something similar to:

```
-- Found Nuke: /Applications/Nuke6.2v1/Nuke6.2v1.app/Contents/MacOS/include
-- Nuke_API_VERSION: --6.2--
```

The Nuke plugins are installed into lib/nuke$MAJOR.$MINOR/, e.g lib/nuke6.2/OCIODisdplay.so

---

Note: If you are using the Nuke plugins, you should compile the Python bindings for the same version of Python that Nuke uses internally. For Nuke 6.0 and 6.1 this is Python 2.5, and for 6.2 it is Python 2.6

The applications included with OCIO have various dependencies - to determine these, look at the CMake output when first run:

```
-- Not building ocioconvert. Requirement(s) found: OIIO:FALSE
```

3.3.3 Quick environment configuration

The quickest way to set the required *Environment variables* is to source the share/ocio/setup_ocio.sh script installed with OCIO.

For a simple single-user setup, add the following to ~/.bashrc (assuming you are using bash, and the example install directory of /software/ocio):

```
source /software/ocio/share/ocio/setup_ocio.sh
```

The only environment variable you must configure manually is OCIO, which points to the configuration file you wish to use. For prebuilt config files, see the *Downloads* section

To do this, you would add a line to ~/.bashrc (or a per-project configuration script etc), for example:

```
export OCIO="/path/to/my/config.ocio"
```

3.3.4 Nuke Configuration

If you specified the NUKE_INSTALL_PATH option when running cmake, you should have a /software/ocio/lib/nuke6.2 directory containing various files.

If you have followed *Quick environment configuration*, the plugins should be functional. However, one common additional configuration step is to register an OCIODisplay node for each display device/view specified in the config.

To do this, in a menu.py on NUKE_PATH (e.g ~/.nuke/menu.py for a single user setup), add the following:

```
import ocionuke.viewer
ocionuke.viewer.populate_viewer(also_remove = "default")
```

The also_remove argument can be set to either "default" to remove the default sRGB/rec709 options, "all" to remove everything, or "none" to leave existing viewer processes untouched.

Alternatively, if your workflow has different requirements, you can copy the function and modify it as required, or use it as reference to write your own, better viewer setup function!

```
import nuke


def register_viewers(also_remove = "default"):
```

```
    """Registers the a viewer process for each display device/view, and
    sets the default viewer process.

    ``also_remove`` can be set to either:

    - "default" to remove the default sRGB/rec709 viewer processes
    - "all" to remove all processes
    - "none" to leave existing viewer processes untouched
    """

    if also_remove not in ("default", "none", "all"):
        raise ValueError("also_remove should be set to 'default', 'none' or 'all'")

    if also_remove == "default":
        nuke.ViewerProcess.unregister('rec709')
        nuke.ViewerProcess.unregister('sRGB')
        nuke.ViewerProcess.unregister('None')
    elif also_remove == "all":
        # Unregister all processes, including None, which should be defined in config.ocio
        for curname in nuke.ViewerProcess.registeredNames():
            nuke.ViewerProcess.unregister(curname)

    # Formats the display and transform, e.g "Film1D (sRGB)"
    DISPLAY_UI_FORMAT = "%(view)s (%(display)s)"

    import PyOpenColorIO as OCIO
    config = OCIO.GetCurrentConfig()

    # For every display, loop over every view
    for display in config.getDisplays():
        for view in config.getViews(display):
            # Register the node
            nuke.ViewerProcess.register(
                name = DISPLAY_UI_FORMAT % {'view': view, "display": display},
                call = nuke.nodes.OCIODisplay,
                args = (),
                kwargs = {"display": display, "view": view, "layer": "all"})


    # Get the default display and view, set it as the default used on Nuke startup
    defaultDisplay = config.getDefaultDisplay()
    defaultView = config.getDefaultView(defaultDisplay)

    nuke.knobDefault(
        "Viewer.viewerProcess",
        DISPLAY_UI_FORMAT % {'view': defaultView, "display": defaultDisplay})
```

3.3.5 Environment variables

OCIO
: This variable needs to point to the global OCIO config file, e.g `config.ocio`

DYLD_LIBRARY_PATH
: The `lib/` folder (containing `libOpenColorIO.dylib`) must be on the `DYLD_LIBRARY_PATH` search path, or you will get an error similar to:

```
dlopen(.../OCIOColorSpace.so, 2): Library not loaded: libOpenColorIO.dylib
Referenced from: .../OCIOColorSpace.so
Reason: image not found
```

This applies to anything that links against OCIO, including the Nuke nodes, and the `PyOpenColorIO` Python bindings.

LD_LIBRARY_PATH
Equivalent to the `DYLD_LIBRARY_PATH` on Linux

PYTHONPATH
Python's module search path. If you are using the PyOpenColorIO module, you must add `lib/python2.x` to this search path (e.g `python/2.5`), or importing the module will fail:

```
>>> import PyOpenColorIO
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named PyOpenColorIO
```

Note that `DYLD_LIBRARY_PATH` or `LD_LIBRARY_PATH` must be set correctly for the module to work.

NUKE_PATH
Nuke's customisation search path, where it will look for plugins, gizmos, init.py and menu.py scripts and other customisations.

This should point to both `lib/nuke6.2/` (or whatever version the plugins are built against), and `share/nuke/`

3.4 User Guide

These guides will focus on a specific task (for example, writing a basic config, or setting up per-shot LUT's). For a "broader picture" expiation of how to use OCIO, see the *Configurations* section

3.4.1 Tool overview

OCIO is comprised of many parts. At the lowest level there is the C++ API. This API can be used in applications and plugins

Note that all these plugins use the same config file to define color spaces, roles and so on. For information on setting up configurations, see *Configurations*

The API

Most users will never directly interact with the C++ API. However the API is used by all the supplied applications (e.g *ocio2icc*) and plugins (e.g the *Nuke plugins*)

To get started with the API, see the *Developer guide*

ociocheck

This is a command line tool which shows an overview of an OCIO config file, and check for obvious errors

For example, the following shows the output of a config with a typo - the colorspace used for `compositing_log` is not incorrect:

```
$ ociocheck --iconfig example.ocio

OpenColorIO Library Version: 0.8.3
OpenColorIO Library VersionHex: 525056
Loading example.ocio

** General **
Search Path: luts
Working Dir: /tmp

Default Display: sRGB
Default View: Film

** Roles **
ncf (default)
lnf (scene_linear)
NOT DEFINED (compositing_log)

** ColorSpaces **
lnf
lgf
ncf
srgb8 -- output only

ERROR: Config failed sanitycheck. The role 'compositing_log' refers to a colorspace, 'lgff', which is

Tests complete.
```

It cannot verify the defined color transforms are "correct", only that the config file can be loaded by OCIO without error. Some of the problems it will detect are:

- Duplicate colorspace names
- References to undefined colorspaces
- Required roles being undefined
- At least one display device is defined

As with all the OCIO command line tools, you can use the *–help* argument to read a description and see the other arguments accepted:

```
$ ociocheck --help
ociocheck -- validate an OpenColorIO configuration

usage:  ociocheck [options]

    --help        Print help message
    --iconfig %s  Input .ocio configuration file (default: $OCIO)
    --oconfig %s  Output .ocio file
```

ociobakelut

A command line tool which bakes a color transform into various color lookup file formats ("a LUT")

This is intended for applications that have not directly integrated OCIO, but can load LUT files

If we want to create a `lnf` to `srgb8` viewer LUT for Houdini's MPlay:

```
$ ociobakelut --inputspace scene_linear --shaperspace lg10 --outputspace srgb8 --format houdini houdi
```

The `--inputspace` and `-outputspace` options specify the colorspace of the input image, and the displayed image.

Since a 3D LUT can only practically operate on 0-1 (e.g a Log image), the `--shaperspace` option is specified. This uses the Houdini LUT's 1D "pretransform" LUT to do "lnf" to "lg10", then the 3D LUT part to go from "lg10" to "srgb8" (basically creating a single file containing a 1D linear-to-log LUT, and a 3D log-to-sRGB LUT)

To make a log to sRGB LUT for Flame, the usage is similar, except the shaperspace option is omitted, as the input colorspace does not have values outside 0.0-1.0 (being a Log space):

```
$ ociobakelut --inputspace lg10 --outputspace srgb8 --format flame flame__lg10_to_srgb.3dl
```

See the *What LUT Formats are supported?* section for a list of formats that support baking

ocio2icc

A command line tool to generate an ICC "proofing" profile from a color space transform, which can be used in applications such as Photoshop.

A common workflow is for matte-painters to work on sRGB files in Photoshop. An ICC profile is used to view the work with the same film emulation transform as used in other departments.

ocioconvert

Loads an image, applies a color transform, and saves it to a new file.

OpenImageIO is used to open and save the file, so a wide range of formats are supported.

ociodisplay

A basic image viewer. Uses OpenImageIO to load images, and displays them using OCIO and typical viewer controls (scene-linear exposure control and a post-display gamma control)

May be useful to users to quickly check colorspace configuration, but primarily a demonstration of the OCIO API

Nuke plugins

A set of OCIO nodes for The Foundry's Nuke, including:

- OCIOColorSpace, transforms between two color spaces (similar to the built-in "ColorSpace" node, but the colorspaces are described in the OCIO config file)
- OCIODisplay to be used as viewer processes
- OCIOFileTransform loads a transform from a file (e.g a 1D or 3D LUT), and applies it
- OCIOCDLTransform applies CDL-compliant grades, and includes utilities to create/load ASC CDL files

3.4.2 Baking LUT's

Sometimes it is necessary to write a color transform as a lookup-table file

This is usually because an application does not natively support OCIO (unlike, say, Nuke which various OCIO nodes), but can load a LUT of some kind. This would currently include applications like Autodesk Flame, Adobe Photoshop, SideFX's MPlay (Houdini's "Image Viewer")

Remember that baking a LUT is not a perfect solution. Different LUT formats have various limitations. Certain applications might apply LUT's differently (often incorrectly), and some formats simply cannot accurately represent certain transforms. Others might require carefully selecting shaper spaces and so on.

Be sure to carefully test the generated LUT in the target application. Burning the LUT into a test image (such as Marcie!), and carefully comparing to a reference is often the only way to be sure a LUT is correct.

Config-based baking

This section assumes you have a working OCIO config.

The config can either be specified by setting the OCIO environment variable:

```
bash$ export OCIO=~/path/to/spi-vfx/config.ocio
```

Alternatively the config can be specified as a command-line argument to the ociobakelut command, `--iconfig ~/path/to/spi-vfx/config.ocio`

These examples will use the *spi-vfx* config, specifically the following colorspaces

- `lnf` - scene-referred linear light colorspace (reference colorspace)
- `lg10` - film log colorspace (0-1 colorspace)
- `srgb8` - sRGB display colorspace

Remember these are just for the examples - you can of course use any config and any colorspaces

Log-input display LUT

Say we have a `lg10` image in MPlay (maybe a ".cin" film scan), and wish to view it in our main display colorspace, `srgb8`

The available formats are listed in the `ociobakelut --help` - for MPlay, we use the "houdini" format (see *the FAQ* for a more detailed list)

So, to create a LUT that transforms from `lg10` to `srgb8`:

```
bash$ ociobakelut --format houdini --inputspace lg10 --outputspace srgb8 log_to_display.lut
```

We could then load this LUT into MPlay and view a lg10 image correctly! (note that by MPlay tries to linearise ".cin" files by default, which can be disabled in the preferences, "Convert form 10bit Log")

For most other applications, we could simply change the `--format`

Shaper spaces

Before we create a LUT to view linear-light images, it's important to understand shaper-spaces and preluts.

The two main types of LUT's are 1D and 3D. Such LUT formats typically require input values in the 0.0-1.0 range. Such a LUT would be unsuitable for linear-light values input images (where values are often over 1)

To address this, various LUT formats contain a smaller "prelut" (or "shaper LUT") which is applied before the main LUT. This is used to transform the input values into a 0-1 range (typically a linear-to-log type transform)

In terms of color-transforms, the prelut transforms from "input-space" to "shaper-space", then the main LUT transforms from "shaper-space" to "output-space"

Some formats do not support such a shaper LUT - these are typically used in applications which do not work with floating-point images (e.g Lustre often works with 10-bit integer DPX's, so it's natively supported "–format lustre" (3DL) format has no need for a prelut)

Linear-light input display LUT

With shaper-spaces explained, lets say we have a `lnf` linear light image in MPlay, and wish to view it in the `srgb8` colorspace.

To create this LUT accurately, without clipping, we will use the LUT's prelut to transform from `lnf` to `lg10`, then the 3D LUT will transform from `lg10` to `srgb8`

Sounds complicated, but the command is almost the same as before, just with the `--shaperspace` argument (and `--inputspace` changed, of course):

```
bash$ ociobakelut --format houdini --inputspace lnf --shaperspace lg10 --outputspace srgb8 lin_to_dis
```

Allocation-based prelut

If your *allocation variables* are setup correctly, you can omit the `--shaperspace` argument, and a prelut will be automatically created based on the allocation vars (see the linked page for more information)

Since the colorspaces in the config we are using (*spi-vfx*) have their allocation variables set correctly, we could simplify the `lnf` to `srgb8` bake command:

```
bash$ ociobakelut --format houdini --inputspace lnf --outputspace srgb8 lin_to_display_allocbased.lut
```

This command creates a very different prelut to the explicitly specified `--shaperspace lg10` in the previous example. Explicitly specifying a shaper can produce better results, however the allocation-based prelut usually works nicely, and is convinient

Note that allocation-var based preluts is baker-format dependant, and not all formats currently implement them

Config-free baking

You can perform baking without using a OCIO config. This means you don't have to create a temporary config just to, say, convert from one LUT format to another.

Converting between formats

Say we have a houdini LUT named `log_to_display.lut`. To convert this to a Flame compatible 3dl file, simply run:

```
ociobakelut --format flame --lut log_to_display.lut for_flame.3dl
```

Reversing a 1D LUT

You can apply a LUT in reverse, and write this to a new LUT (this does not work for 3D LUT's, but will for 1D LUT's):

```
bash$ ociobakelut --format flame --invlut logtosrgb.3dl srgbtolog.3dl
```

Creating a grade LUT

You can create a LUT which applies CDL-compliant grades:

```
ociobakelut --format cinespace --slope 1.2 1.0 0.9 mygrade.csp
```

Combining options

These options can be used together, or used multiple times.

For example, to perform a slope offset, then apply "mylut.csp", saving it out for Lustre:

```
bash$ ociobakelut --format lustre --slope 2.0 1.5 0.4 --lut mylut.csp output.3dl
```

ICC profiles (Photoshop)

Photoshop is very focused around print and graphic-design, industries with very different color management concerns to modern feature-film VFX. As such, it can be a pain to integrate.

The main issue is current versions of Photoshop (CS5) are only practical for working with 16-bit integer images (not floating point/linear-light images as is common in compositing software)

The second issue is there is no simple way to load a simple 1D or 3D LUT into Photoshop (and it's API does not make this easy either!)

A working space

First, we need to decide on a colorspace to use for the images in Photoshop. This is the colorspace in which matte-paintings will be performed (likely a different colorspace that used for texture-painting, as these have different requirements)

The working space should be a "0-1 colorspace", reversable, and for matte-paintings ideally allow painting values over "diffuse white" (in other words, to paint values over 1.0 when converted to linear-light in comp)

This is a facility-dependant workflow choice.

For this example we will use the `vd16` colorspace, as described by the *spi-vfx*

Creating display LUT

"Proofing profiles" in Photoshop can be used in a very similar way to a display LUT in applications like Nuke. This proof-profile can be used to apply a 3D color transform from the working-space to a display space (e.g transform from `vd16` to `srgb8` in the spi-vfx config)

These proofing-profiles are ICC profiles - a rather print-specific technology and relatively complex format

Luckily, `ociobakelut` can be used to create these... but, first, there are some important considerations:

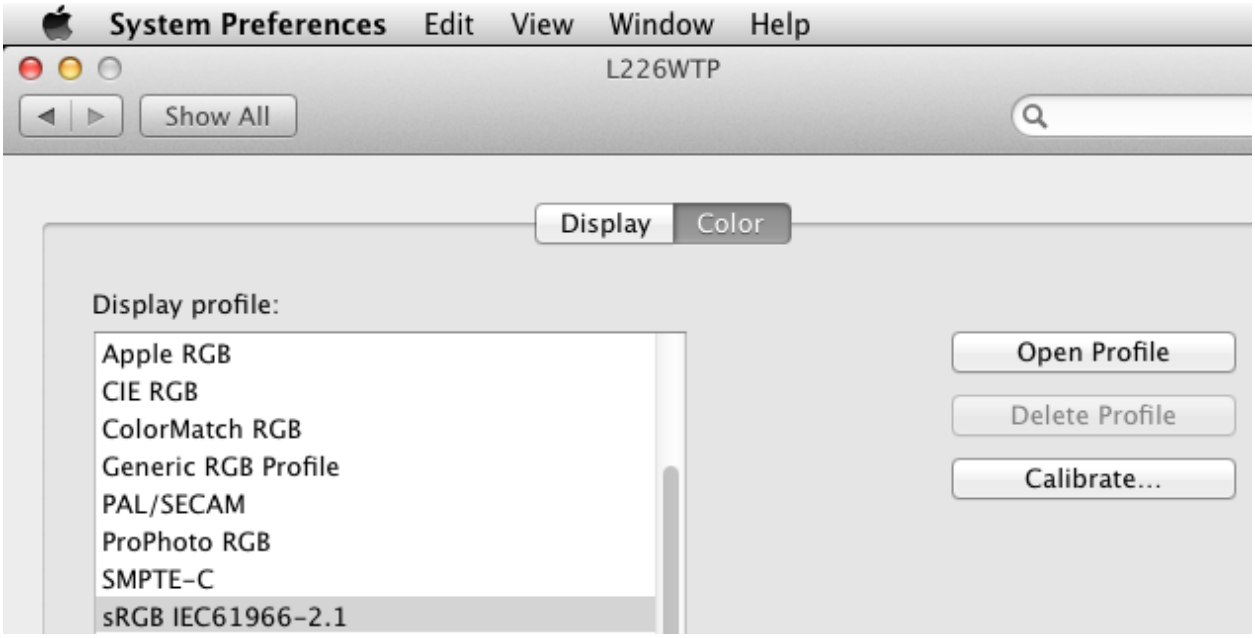
It is important to match the `--displayicc` option to the profile used for the display.

Secondly, Photoshop has a lot of print-focused color-management options, some of which can cause problems.

Determine display ICC

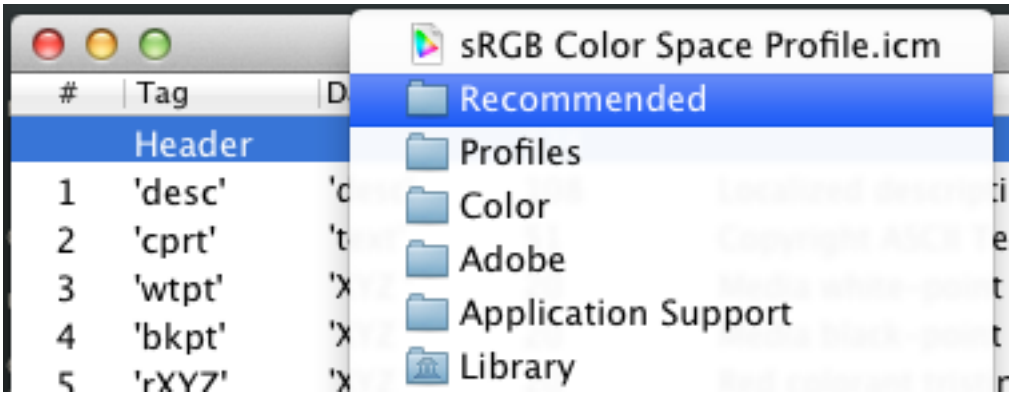
On OS X, launch "System Preferences", open "Displays" and click "Color" tab. The currently active ICC profile is selected.

If you just want something simple that "probably matches" a Linux machine, say, it is easiest to uncheck "Show profiles for this display only" and select the "sRGB IEC61966-2.1" profile. You can skip the rest of this section in this case.



However, if you have a specific display-profile selected (maybe created by monitor-calibration software), you should do the following:

Click "Open Profile", and right-click the icon in the top of the window, and click the folder:



This reveals the current profile in Finder. You can drag the file onto a Terminal.app window to get the full path (or, type it manually)

Create the ICC profile

Almost done now. We can write the ICC profile!

The full command is, using our example colorspaces of `vd16` and `srgb8`:

```
bash$ ociobakelut --format icc --inputspace vd16 --outputspace srgb8 --displayicc /path/to/my/monitor
```

The first three options are the same as any other LUT:

```
bash$ ociobakelut --format icc --inputspace vd16 --outputspace srgb8 [...]
```

Then we specify the display ICC profile:

```
[...] --displayicc /path/to/my/monitorprofile.icc [...]
```

We can set the description (shown in Photoshop), and as the last argument, specify:

```
[...] --description "vd16 to srgb8"  [...]
```

Finally an argument for the output file:

```
[...] vd16_to_srgb.icc
```

If you selected the "sRGB IEC61966-2.1" display profile, you can omit the `--displayicc` argument (it defaults to an standard sRGB profile):

```
bash$ ociobakelut --format icc --inputspace vd16 --outputspace srgb8 --description "vd16 to srgb8"
```

Loading the "display LUT"

Last step is to load the ICC profile into Photoshop, and enable it.

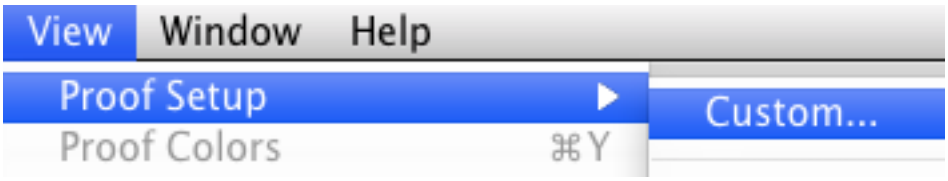
On OS X, these can be put into:

```
/Library/ColorSync/Profiles/
```

(or the equivelant directory in your home-directory)

On Windows, right-click the profile and select "Install profile"

Then on either platform, click "View > Proof Setup > Custom..."

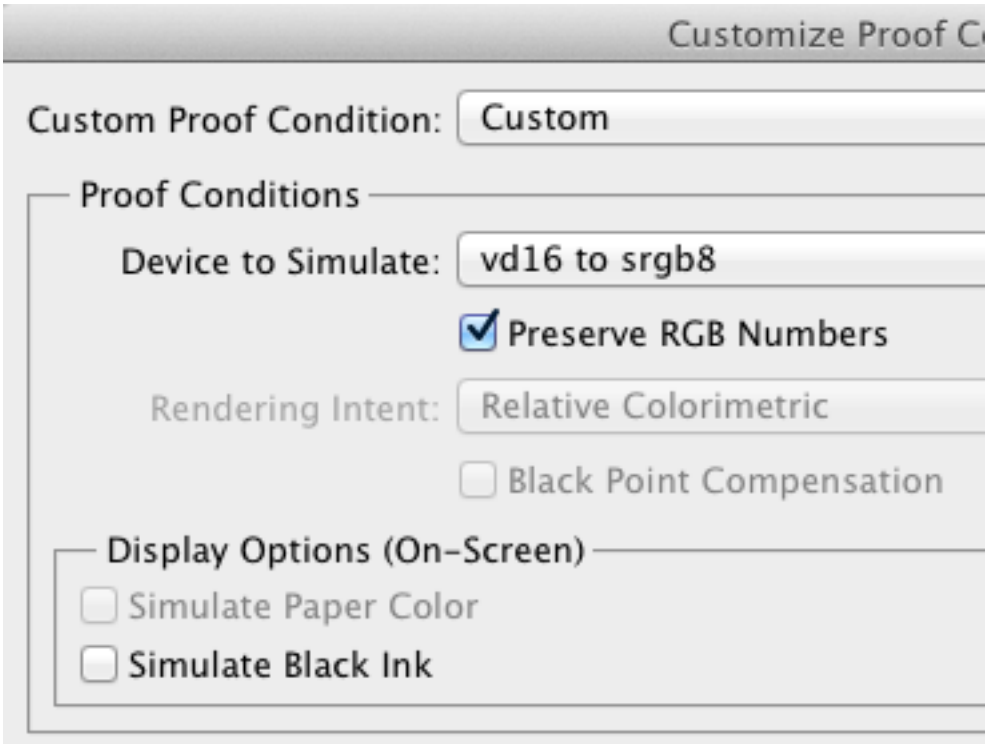


Select your profile from the "Device to simulate" dropdown (the name is what you supplied with `--description`):

As pictured, selecting "Preserve RGB numbers", and deselecting "Simulate Black Ink" is a good starting point (see the next section on "Other color settings")

Finally, you can load an image in your working space, and click "View > Proof Colors", or hit cmd+y (or ctrl+y)

When active, the profile name is shown in the window title (e.g ".... (RGB/16#/vd16 to srgb8", where the part after the "#/" is the profile name, "RGB/16" indicates the current image mode)



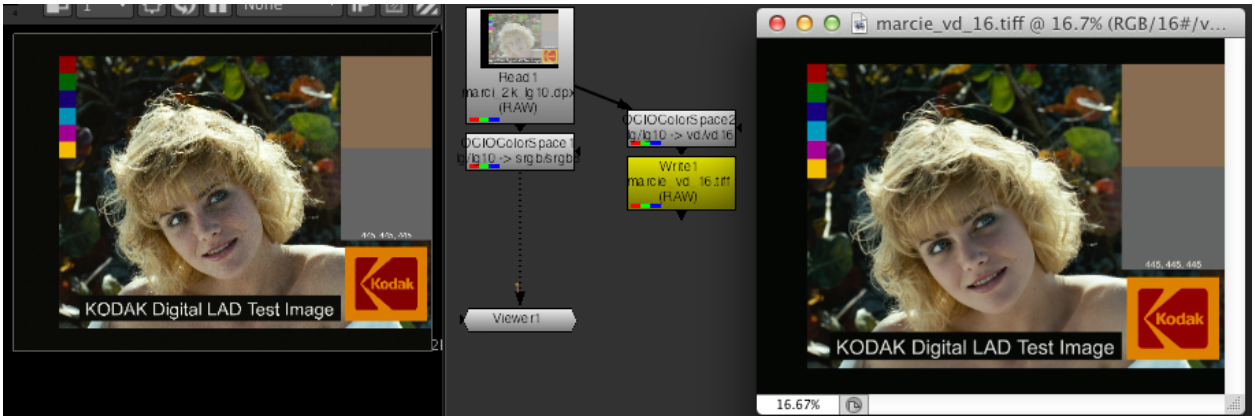
Other color settings

(note this guide is based on Photoshop CS5, and written while checking the OS X version, although most of these apply similarly on Windows 7)

It is usually possible to get a matte-painting to look identical in Photoshop as it does in a more VFX-friendly application such as Nuke.

However, as mentioned Photoshop has a lot of color-management related options, many of which can impair the match between it and other applications. The operating system also has some controls (as seen before with the ColorSync display profile)

The settings that require tweaking have a tendency to change with OS versions, Photoshop versions and the phase of the moon. The only way to be sure is to compare Photoshop side-by-side with a LUT-reference-image (ideally toggling between Photoshop and Nuke):



The most improtant settings are in the "View > Proof Setup > Custom ..." menu.

The recommended "Preserve RGB" setting works sometimes. Other times disabling "Preserve RGB Numbers" and selecting "Rendering Intent: Relative Colorimetric" can be closer.

It is safest to not assign a profile to the images you are working on - this is done by clicking "Edit > Assign Profile", and selecting "Don't Color Manage This Document".

In closing, of course none of this matters if you don't have a calibrated monitor!

3.4.3 Contexts

OCIO's allows different LUT's or grades to be applied based on the current context.

These contexts are usually based on environment variables, but also allows on-the-fly context switching in applications that operate on multiple shots (such as playback tools)

Typically these would be used as part of the display transform, to apply shot-specific looks (such as a CDL color correction, or a 1D grade LUT)

A contrived example

The simplest way to explain this feature is with examples. Say we have two shots, `ab-123` and `sf-432`, and each shot requires a different LUT to view. The current shot name is stored in the environment variable SHOT.

In the OCIO config, you can use this SHOT environment variable to construct the LUT's path/filename. This path can be absolute (e.g `/example/path/${SHOT}.spi1d`), or relative to any directory on the OCIO search path, which includes the resource path (e.g `${SHOT}.spi1d`)

This is a simplified example, to demonstrate the context feature. Typically this "contextual LUT" would be used in conjuction with other LUT's (e.g before a scene-linear to log transform, followed by a 3D film emulation LUT), this will be covered in *Per-shot grades*

So, we have our empty OCIO config in `~/showcfg`, and our two LUTs in `~/showcfg/luts` which are named `af-123.spi1d` and `sf-432.spi1d`:

```
~/showcfg/
    config.ocio
    luts/
        af-123.spi1d
        sf-432.spi1d
```

In the config, we first specify the config version, and the resource path (usually this is relative to the directory containing `config.ocio`, although can be an absolute path):

```
ocio_profile_version: 1
resource_path: luts
```

Next, we define a colorspace that transforms from the show reference space to the display colorspace:

```
colorspaces:
  - !<ColorSpace>
    name: srgb8
    family: srgb
    bitdepth: 8ui
    from_reference: !<FileTransform> {src: ${SHOT}.spi1d}
```

Then add a display alias for this transform:

```
displays:
  - !<Display> {device: sRGB, name: "Shot LUT", colorspace: srgb8}
```

Finally, we point the OCIO env-variable to the config, set the SHOT env-variable to the shot being worked on, and launch Nuke (or any other OCIO-enabled application):

```
export OCIO=~/showcfg/config.ocio
export SHOT=af-123
nuke
```

In Nuke, we create an OCIODisplay node, select our "sRGB" device with the "Shot LUT" transform, and this will apply the `af-123.spi1d` LUT.

Per-shot grades

Similarly to LUTs, we use a `.cc` file (an XML file containing a single ASC CDL `<ColorCorrection>`), or a `.ccc` file (an XML file containing multiple ASC CDL color corrections, each with a unique ID)

The `.cc` file is applied identically to a regular LUT files, using a `FileTransform`. For example, if we have `af-123.cc` in the `luts/` directory:

```
<ColorCorrection id="mygrade">
        <SOPNode>
             <Slope>2 1 1</Slope>
             <Offset>0 0 0</Offset>
             <Power>1 1 1</Power>
        </SOPNode>
        <SATNode>
             <Saturation>1</Saturation>
        </SATNode>
  </ColorCorrection>
```

We wish to apply this grade on the scene-linear image, then transform into log and apply a 3D print emulation LUT. Since this requires multiple transforms, instead of using a single `FileTransform`, we use a `GroupTransform` (which is is just a collection of other transforms):

```
colorspaces:
  - !<ColorSpace>
    name: lnh
    family: ln
    bitdepth: 16f:
    isdata: false

  - !<ColorSpace>
    name: lg10
    family: lg
    bitdepth: 10ui
    isdata: false
    to_reference: !<FileTransform> {src: lg10.spi1d, interpolation: nearest}

  - !<ColorSpace>
    name: srgb8
    family: srgb
    bitdepth: 8ui
    isdata: false
    from_reference: !<GroupTransform>
      children:
        - !<FileTransform> {src: ${SHOT}.cc}
```

```
        - !<ColorSpaceTransform> {src: lnh, dst: lg10}
        - !<FileTransform> {src: film_emulation.spi3d, interpolation: linear}
```

A .ccc file is a collection of `<ColorCorrection>`'s. The only difference is when defining the `FileTransform`, you must specify the `cccdid` key, which you can also construct using the context's environment variables. This means we could create a `grades.ccc` file containing the grade for all our shots:

```
<ColorCorrectionCollection xmlns="urn:ASC:CDL:v1.2">
      <ColorCorrection id="af-123">
              <SOPNode>
                   <Slope>2 1 1</Slope>
                   <Offset>0 0 0</Offset>
                   <Power>1 1 1</Power>
              </SOPNode>
              <SATNode>
                   <Saturation>1</Saturation>
              </SATNode>
        </ColorCorrection>
        <ColorCorrection id="mygrade">
                <SOPNode>
                     <Slope>0.9 0.7 0.9</Slope>
                     <Offset>0 0 0</Offset>
                     <Power>1 1 1</Power>
                </SOPNode>
                <SATNode>
                     <Saturation>1</Saturation>
                </SATNode>
          </ColorCorrection>
</ColorCorrectionCollection>
```

And the colorspace definition to utilise this:

```
- !<ColorSpace>
  name: srgb8
  family: srgb
  bitdepth: 8ui
  isdata: false
  from_reference: !<GroupTransform>
    children:
      - !<FileTransform> {src: grades.ccc, cccid: ${SHOT}}
      - !<ColorSpaceTransform> {src: lnh, dst: lg10}
      - !<FileTransform> {src: film_emulation.spi3d, interpolation: linear}
```

A complete example

Warning: This is incomplete, the lnh_graded space is likely wrong

The context feature can be used to accommodate complex grading pipelines. In this example, we have a "neutral grade" for each shot, to neutralise color casts and exposure variations, keeping plates consistent throughout a sequence.

To view a shot, we reverse this neutral grade, apply a "beauty grade", then apply the display transform (the usual lin-to-log and a film emulation LUT)

We will use the same two example shots from before, af-123 (which is in the af sequence) and sg-432 (in the sg sequence). Imagine we have many shots in each sequence, so we wish to put the grades for each sequence in a separate file.

Using the same directory structure as above, in `~/showcfg/luts` we first create two grade files, `grades_af.ccc`:

```
<ColorCorrectionCollection xmlns="urn:ASC:CDL:v1.2">
      <ColorCorrection id="af/af-123/neutral">
              <SOPNode>
                   <Slope>2 1 1</Slope>
                   <Offset>0 0 0</Offset>
                   <Power>1 1 1</Power>
              </SOPNode>
              <SATNode>
                   <Saturation>1</Saturation>
              </SATNode>
        </ColorCorrection>

      <ColorCorrection id="af/af-123/beauty">
              <SOPNode>
                   <Slope>1.5 1.2 0.9</Slope>
                   <Offset>0 0 0</Offset>
                   <Power>1 1 1</Power>
              </SOPNode>
              <SATNode>
                   <Saturation>0.8</Saturation>
              </SATNode>
        </ColorCorrection>

        <!-- More ColorCorrection's... -->
</ColorCorrectionCollection>
```

And `grades_sg.ccc`:

```
<ColorCorrectionCollection xmlns="urn:ASC:CDL:v1.2">
        <ColorCorrection id="sg/sg-432/neutral">
                <SOPNode>
                     <Slope>0.9 0.7 0.9</Slope>
                     <Offset>0 0 0</Offset>
                     <Power>1 1 1</Power>
                </SOPNode>
                <SATNode>
                     <Saturation>1</Saturation>
                </SATNode>
          </ColorCorrection>

        <ColorCorrection id="sg/sg-432/beauty">
                <SOPNode>
                     <Slope>1.1 0.9 0.8</Slope>
                     <Offset>0 0 0</Offset>
                     <Power>1.2 0.9 1.5</Power>
                </SOPNode>
                <SATNode>
                     <Saturation>1</Saturation>
                </SATNode>
          </ColorCorrection>

          <!-- More ColorCorrection's.. -->
</ColorCorrectionCollection>
```

Next, we create the `config.ocio` file, containing a colorspace to define several colorspaces:

- `lnh`, the scene-linear, 16-bit half-float space in which compositing will happen

- `lg10`, the 10-bit log space in which material will be received (e.g in .dpx format)
- `srgb8`, the display colorspace, for viewing the neutrally graded footage on an sRGB display
- `srgb8graded`, another display colorspace, for viewing the final "beauty grade"

```
ocio_profile_version: 1

# The directory relative to the location of this config
resource_path: "luts"

roles:
  scene_linear: lnh
  compositing_log: lgf

displays:
  # Reference to display transform, without reversing the working grade
  - !<Display> {device: sRGB, name: Film1D, colorspace: srgb8}

  # Reference to display, reversing the working grade, and applying
  # the beauty grade
  - !<Display> {device: sRGB, name: Film1DGraded, colorspace: srgb8graded}

colorspaces:

  # The source space, containing a log to scene-linear LUT
  - !<ColorSpace>
    name: lg10
    family: lg
    bitdepth: 10ui
    isdata: false
    to_reference: !<FileTransform> {src: lg10.spi1d, interpolation: nearest}

  # Our scene-linear space (reference space)
  - !<ColorSpace>
    name: lnh
    family: ln
    bitdepth: 16f
    isdata: false

  # Neutrally graded scene-linear
  - !<ColorSpace>
    name: lnh_graded
    family: ln
    bitdepth: 16f
    isdata: false
    to_reference: !<FileTransform> {src: "grades_${SEQ}.ccc", cccid: "${SEQ}/${SHOT}/neutral"}


  # The display colorspace - how to get from scene-linear to sRGB
  - !<ColorSpace>
    name: srgb8
    family: srgb
    bitdepth: 8ui
    isdata: false
    from_reference: !<GroupTransform>
      children:
        - !<ColorSpaceTransform> {src: lnh, dst: lg10}
        - !<FileTransform> {src: lg_to_srgb.spi3d, interpolation: linear}
```

```
  # Display color, with neutral grade reversed, and beauty grade applied
  - !<ColorSpace>
    name: srgb8graded
    family: srgb
    bitdepth: 8ui
    isdata: false
    from_reference: !<GroupTransform>
      children:
        - !<FileTransform> {src: "grades_${SEQ}.ccc", cccid: "${SEQ}/${SHOT}/neutral", direction: inv
        - !<FileTransform> {src: "grades_${SEQ}.ccc", cccid: "${SEQ}/${SHOT}/beauty", direction: forw
        - !<ColorSpaceTransform> {src: lnh, dst: srgb8}
```

3.4.4 Looks

A "look" is a named color transform, intended to modify the look of an image in a "creative" manner (as opposed to a colorspace definion which tends to be technically/mathematically defined)

Examples of looks may be a neutral grade, to be applied to film scans prior to VFX work, or a per-shot DI grade decided on by the director, to be applied just before the viewing transform

Looks are defined similarly to colorspaces, you specify a name and a transform (possibly a GroupTransform containing several other transforms), and optionally an inverse transform.

Where looks differ from colorspace definions are in how they are applied. With a look, you also specify the "process space" - the colorspace in which the transform is applied.

Example configuration

Step 1: Setup a Look

A look is a top-level OCIO configuration object. Conceptually, it's a named transform which gets applied in a specific color space. All of the changes below to the .ocio configs can be done manually by editing the text, or using the Python API.

Example look definition in a OCIO config:

```
looks:
- !<Look>
  name: di
  process_space: rclg16
  transform: !<FileTransform> {src: look_di.cc, interpolation: linear}
```

The src file can be any LUT type that OCIO supports (in this case, it's a file that contains the `<ColorCorrection>` element from a CDL file.) You could also specify a .3dl, etc.

Once you define a look in your configuration, you'll see that the `OCIOLookTransform` node in Nuke will provide the named option. In this example, the 'DI' look conceptually represents a look that will be applied in DI. Other look names we often used are 'onset', 'editorial', etc. The `process_space` specifies that the transform should be applied in that space. In this example, if you provide linear input to the `OCIOLookTransform` node, the pixels will be converted to `rclg16` before applying the `look_di.cc` file-transform.

Step 2: Update the Display to use a look.

You can specify an optional 'looks' tag in the View tag, which will apply the specified look(s). This lets application in the viewer provide options which use the looks.

Example:

```
displays:
  DLP:
    - !<View> {name: Raw, colorspace: nc10}
    - !<View> {name: Log, colorspace: rclg10}
    - !<View> {name: Film, colorspace: p3dci16}
    - !<View> {name: Film DI, colorspace: p3dci16, looks: di}
  sRGB:
    - !<View> {name: Raw, colorspace: nc10}
    - !<View> {name: Log, colorspace: rclg10}
    - !<View> {name: Film, colorspace: srgb10}
    - !<View> {name: Film DI, colorspace: srgb10, looks: di}
```

Option for advanced users: The looks tag is actually a comma-delimited list supporting +/- modifiers. So if you you wanted to specify a View that undoes DI, and then adds Onset, you could do "-di,+onset".

Step 3: Get per-shot looks supported.

In the top example, look_di.cc, being a relative path location, will check each location in the config's search_path. The first file that's found will be used.

So if your config contains:

```
search_path: luts
```

... then only the 'luts' subdir relative to the OCIO config will be checked.

However if you specify:

```
search_path: /shots/show/$SHOT/cc/data:luts
```

...the directory '/shots/show/$SHOT/cc/data/' will be evaluated first, and only if not found will the 'luts' directory be checked.

env-vars, absolute, and relative paths can be used both in the config's `search_path`, as well as the View's src specification.

Example:

```
- !<Look>
  name: di
  process_space: rclg16
  transform: !<FileTransform> {src: looks/$SHOT_di/current/look_$SHOT_di.cc, interpolation: linear}
```

Note that if the per-shot lut is not found, you can control whether a fallback LUT succeeds based on if it's in the master location. You can also use this for multiple levels (show, shot, etc).

Advanced option: If some shots use .cc files, and some use 3d-luts currently there's no simple way to handle this. What we'd recommend as a work around is to label all of your files with the same extension (such as .cc), and then rely on OCIO's resiliance to misnamed lut files to just load them anyways. Caveat: this only works in 1.0.1+ (commit sha-1: 6da3411ced)

Advanced option: In the Nuke OCIO nodes, you often want to preview looks 'across shots' (often for reference, same-as, etc). You can override the env-vars in each node, using the 'Context' menu. For example, if you know that $SHOT is being used, in the context key1 you should specify 'SHOT', and the in value1 specify the shot to use (such as dev.lookdev). You can also use expressions, to say parse a shot name out of `[metadata "input/filename"]`

Advanced option: If you are writing your own OCIO integration code, `getProcessor` will fail if the per-shot lut is not found, and you may want to distinguish this error from other OCIO errors. For this reason, we provide OCIO::ExceptionMissingFile, which can be explicitly caught (this can then handled using `OCIO::DisplayTransform::setLooksOverride()`). I'd expect image flipbook applications to use this approach.

3.4.5 Config syntax

OpenColorIO is primarily controlled by a central configuration file, usually named `config.ocio`. This page will only describe how to *syntactically* write this OCIO config - e.g what transforms are available, or what sections are optional.

This page alone will not help you to write a useful config file! See the *Configurations* section for examples of complete, practical configs, and discussion of how they fit within a facitilies workflow.

YAML basics

This config file is a YAML document, so it is important to have some basic knowledge of this format.

The Wikipedia article on YAML has a good overview.

OCIO configs typically use a small subset of YAML, so looking at existing configs is probably the quickest way to familiarise yourself (just remember the indentation is important!)

Checking for errors

Use the `ociocheck` command line tool to validate your config. It will inform you of YAML-syntax errors, but more importantly it performs various OCIO-specific "sanity checks".

For more information, see the overview of *ociocheck*

Config sections

`ocio_profile_version`

Required.

By convention, the profile starts with `ocio_profile_version`.

This is an integer, specifying which version of the OCIO config syntax is used.

Currently there is only one OCIO profile version, so the value is always `1` (one)

```
ocio_profile_version: 1
```

`search_path`

Optional. Default is an empty search path.

`search_path` is a colon-seperated list of directories. Each directory is checked in order to locate a file (e.g a LUT).

This works is very similar to how the UNIX `$PATH` env-var works for executables.

A common directory structure for a config is:

```
config.ocio
luts/
  lg10_to_lnf.spi1d
  lg10_to_p3.3dl
```

For this, we would set `search_path` as follows:

```
search_path: "luts"
```

In a colorspace definition, we might have a FileTransform which refers to the LUT `lg10_to_lnf.spi1d`. It will look in the `luts` directory, relative to the `config.ocio` file's location.

Paths can be relative (to the directory containing `config.ocio`), or absolute (e.g `/mnt/path/to/my/luts`)

Multiple paths can be specified, including a mix of relative and absolute paths. Each path is separated with a colon `:`

```
search_path: "/mnt/path/to/my/luts:luts"
```

Finally, paths can reference OCIO's context variables:

```
search_path: "/shots/show/$SHOT/cc/data:luts"
```

This allows for some clever setups, for example per-shot LUT's with fallbacks to a default. For more information, see the examples in *Looks*

strictparsing

Optional. Valid values are `true` and `false`. Default is `true` (assuming a config is present):

```
strictparsing: true
```

OCIO provides a mechanism for applications to extract the colorspace from a filename (the `parseColorSpaceFromString` API method)

So for a file like `example_render_v001_lnf.0001.exr` it will determine the colorspace `lnf` (it being the right-most substring containing a colorspace name)

However, if the colorspace cannot be determined and `strictparsing: true`, it will produce an error.

If the colorspace cannot be determined and `strictparsing: false`, the default role will be used. This allows unhandled images to operate in "non-color managed" mode.

Application authors should note: when no config is present (e.g via `$OCIO`), the default internal profile specifies `strictparsing=false`, and the default color space role is `raw`. This means that ANY string passed to OCIO will be parsed as the default `raw`. This is nice because in the absence of a config, the behavior from your application perspective is that the library essentially falls back to "non-color managed".

luma

Deprecated. Optional. Default is the Rec.709 primaries specified by the ASC:

```
luma: [0.2126, 0.7152, 0.0722]
```

These are the luminance coeficients, which can be used by OCIO-supporting applications when adjusting saturation (e.g in an image-viewer when displaying a single channel)

Note: While the API method is not yet officially deprecated, `luma` is a legacy option from Imageworks' internal, closed-source predecessor to OCIO.

The `luma` value is not respected anywhere within the OCIO library. Also very few (if any) applications supporting OCIO will respect the value either.

---

roles

Required.

A "role" is an alias to a colorspaces, which can be used by applications to perform task-specific color transforms without requiring the user to select a colorspace by name.

For example, the Nuke node OCIOLogConvert: instead of requiring the user to select the appropriate log colorspace, the node performs a transform between `scene_linear` and `compositing_log`, and the OCIO config specifies the project-appropriate colorspaces. This simplifies life for artists, as they don't have to remember which is the correct log colorspace for the current project - the OCIOLogConvert always does the correct thing.

A typical role definition looks like this, taken from the *spi-vfx* example configuration:

```
roles:
  color_picking: cpf
  color_timing: lg10
  compositing_log: lgf
  data: ncf
  default: ncf
  matte_paint: vd8
  reference: lnf
  scene_linear: lnf
  texture_paint: dt16
```

All values in this example (such as `cpf`, `lg10` and `ncf`) refer to colorspaces defined later the config, in the `colorspaces` section.

A description of all roles. Note that applications may interpret or use these differently.

- `color_picking` - Colors in a color-selection UI can be displayed in this space, while selecting colors in a different working space (e.g `scene_linear` or `texture_paint`)
- `color_timing` - colorspace used for applying color corrections, e.g user-specified grade within an image viewer (if the application uses the `DisplayTransform::setDisplayCC` API method)
- `compositing_log` - a log colorspace used for certain processing operations (plate resizing, pulling keys, degrain, etc). Used by the OCIOLogConvert Nuke node.
- `data` - used when writing data outputs such as normals, depth data, and other "non color" data. The colorspace in this role should typically have `data: true` specified, so no color transforms are applied.
- `default` - when `strictparsing: false`, this colorspace is used as a fallback. If not defined, the `scene_linear` role is used
- `matte_paint` - Colorspace which matte-paintings are created in (for more information, *see the guide on baking ICC profiles for Photoshop*, and *spi-vfx*)
- `reference` - Colorspace used for reference imagery (e.g sRGB images from the internet)
- `scene_linear` - The scene-referred linear-to-light colorspace, typically used as reference space (see *Terminology*)
- `texture_paint` - Similar to `matte_paint` but for painting textures for 3D objects (see the description of texture painting in *SPI's pipeline*)

displays

Required.

This section defines all the display devices which will be used. For example you might have a sRGB display device for artist workstations, a DCIP3 display device for the screening room projector.

Each display device has a number of "views". These views provide different ways to display the image on the selected display device. Examples of common views are:

- "Film" to emulate the final projected result on the current display
- "Log" to send log-space pixel values directly to the display, resulting in a "lifted" image useful for checking black-levels.
- "Raw" when assigned a colorspace with `raw: yes` set will show the unaltered image, useful for tech-checking images

An example of the `displays` section from the *spi-vfx* config:

```
displays:
  DCIP3:
    - !<View> {name: Film, colorspace: p3dci8}
    - !<View> {name: Log, colorspace: lg10}
    - !<View> {name: Raw, colorspace: nc10}
  sRGB:
    - !<View> {name: Film, colorspace: srgb8}
    - !<View> {name: Log, colorspace: lg10}
    - !<View> {name: Raw, colorspace: nc10}
    - !<View> {name: Film, colorspace: srgb8}
```

All the colorspaces (`p3dci8`, `srgb8` etc) refer to colorspaces defined later in the config.

Unless the `active_displays` and `active_views` sections are defined, the first display and first view will be the default.

active_displays

Optional. Default is for all displays to be visible, and to respect order of items in `displays` section.

You can choose what display devices to make visible in UI's, and change the order in which they appear.

Given the example `displays` block in the previous section - to make the sRGB device appear first:

```
active_displays: [sRGB, DCIP3]
```

To display only the `DCIP3` device, simply remove `sRGB`:

```
active_displays: [DCIP3]
```

The value can be overridden by the *OCIO_ACTIVE_DISPLAYS* env-var. This allows you to make the `sRGB` the only active display, like so:

```
active_displays: [sRGB]
```

Then on a review machine with a DCI P3 projector, set the following environment variable, making `DCIP3` the only visible display device:

```
export OCIO_ACTIVE_DISPLAYS="DCIP3"
```

Or specify multiple active displays, by separating each with a colon:

```
export OCIO_ACTIVE_DISPLAYS="DCIP3:sRGB"
```

active_views

Optional. Default is for all views to be visible, and to respect order of the views under the display.

Works identically to `active_displays`, but controls which *views* are visible.

Overridden by the `OCIO_ACTIVE_VIEWS` env-var:

```
export OCIO_ACTIVE_DISPLAYS="Film:Log:Raw"
```

looks

Optional.

This section defines a list of "looks". A look is a color transform defined similarly to a colorspace, with a few important differences.

For example, a look could be defined for a "first pass DI beauty grade", which is used to view shots with a rough approximation of the final grade.

When the look is defined in the config, you must specify a name, the color transform, and the colorspace in which the grade is performed (the "process space"). You can optionally specify an inverse transform for when the look transform is not trivially invertable (e.g it applies a 3D LUT)

When an application applies a look, OCIO ensures the grade is applied in the correct colorspace (by converting from the input colorspace to the process space, applies the look's transform, and converts the image to the output colorspace)

Here is a simple `looks:` section, which defines two looks:

```
looks:
  - !<Look>
    name: beauty
    process_space: lnf
    transform: !<CDLTransform> {slope: [1, 2, 1]}

  - !<Look>
    name: neutral
    process_space: lg10
    transform: !<FileTransform> {src: 'neutral-${SHOT}-${SEQ}.csp', interpolation: linear }
    inverse_transform: !<FileTransform> {src: 'neutral-${SHOT}-${SEQ}-reverse.csp', interpolation: li
```

Here, the "beauty" look applies a simple, static ASC CDL grade, making the image very green (for some artistic reason!). The beauty look is appied in the scene-linear `lnf` colorspace (this colorspace is defined elsewhere in the config.

Next is a definition for a "neutral" look, which applies a shot-specific CSP LUT, dynamically finding the correct LUT based on the SEQ and SHOT *context variables*.

For example if `SEQ=ab` and `SHOT=1234`, this look will search for a LUT named `neutral-ab-1234.csp` in locations specified in `search_path`.

The `process_space` here is `lg10`. This means when the look is applied, OCIO will perform the following steps:

- Transform the image from it's current colorspace to the `lg10` process space
- Apply apply the FileTransform (which applies the grade LUT)
- Transform the graded image from the process space to the output colorspace

The "beauty" look specifies the optional `inverse_transform`, because in this example the neutral CSP files contain a 3D LUT. For many transforms, OCIO will automatically calculate the inverse transform (as with the "beauty" look), however with a 3D LUT the inverse transform needs to be defined.

If the look was applied in reverse, and `inverse_transform` as not specified, then OCIO would give a helpful error message. This is commonly done for non-invertable looks

As in colorspace definitions, the transform can be specified as a series of transforms using the `GroupTransform`, for example:

```
looks:
  - !<Look>
    name: beauty
    process_space: lnf
    transform: !<GroupTransform>
      children:
        - !<CDLTransform> {slope: [1, 2, 1]}
        - !<FileTransform> {src: beauty.spi1d, interpolation: nearest}
```

colorspaces

Required.

This section is a list of all the colorspaces known to OCIO. A colorspace can be referred to elsewhere within the config (including other colorspace definitions), and are used within OCIO-supporting applications.

to_reference and from_reference Here is a example of a very simple `colorspaces` section, modified from the *spi-vfx* example config:

```
colorspaces:
  - !<ColorSpace>
    name: lnf
    bitdepth: 32f
    description: |
      lnf : linear show space

  - !<ColorSpace>
    name: lg16
    bitdepth: 16ui
    description: |
      lg16 : conversion from film log
    to_reference: !<FileTransform> {src: lg16_to_lnf.spi1d, interpolation: nearest}
```

First the `lnf` colorspace (short for linear float) is used as our reference colorspace. The name can be anything, but the idea of a reference colorspace is an important convention within OCIO: **all other colorspaces are defined as transforms either to or from this colorspace.**

The `lg16` colorspace is a 16-bit log colorspace (see *spi-vfx* for an explaination of this colorspace). It has a name, a bitdepth and a description.

The `lg16` colorspace is defined as a transform from `lg16` to the reference colorspace (`lnf`). That transform is to apply the LUT `lg16_to_lnf.spi1d`. This LUT has an input of `lg16` integers and outputs linear 32-bit float values

Since the 1D LUT is automatically invertable by OCIO, we can use this colorspace both to convert `lg16` images to `lnf`, and `lnf` images to `lg16`

Importantly, because of the reference colorspace concept, we can convert images from `lg16` to the reference colorspace, and then on to any other colorspace.

Here is another example colorspace, which is defined using `from_reference`.

```
- !<ColorSpace>
  name: srgb8
  bitdepth: 8ui
  description: |
    srgb8 :rgb display space for the srgb standard.
  from_reference: !<FileTransform> {src: srgb8.spi3d, interpolation: linear}
```

We use `from_reference` here because we have a LUT which transforms from the reference colorspace (`lnf` in this example) to sRGB.

In this case `srgb8.spi3d` is a complex 3D LUT which cannot be inverted, so it is considered a "display only" colorspace. If we did have a second 3D LUT to apply the inverse transform, we can specify both `to_reference` and `from_reference`

```
- !<ColorSpace>
  name: srgb8
  bitdepth: 8ui
  description: |
    srgb8 :rgb display space for the srgb standard.
  from_reference: !<FileTransform> {src: lnf_to_srgb8.spi3d, interpolation: linear}
  to_reference: !<FileTransform> {src: srgb8_to_lnf.spi3d, interpolation: linear}
```

Using multiple transforms The previous example colorspaces all used a single transform each, however it is often useful to use multiple transforms to define a colorspace.

```
- !<ColorSpace>
  name: srgb8
  bitdepth: 8ui
  description: |
    srgb8 :rgb display space for the srgb standard.
  from_reference: !<GroupTransform>
    children:
      - !<ColorSpaceTransform> {src: lnf, dst: lg16}
      - !<FileTransform> {src: lg16_to_srgb8.spi3d, interpolation: linear}
```

Here to get from the reference colorspace, we first use the `ColorSpaceTransform` to convert from `lnf` to `lg16`, then apply our 3D LUT on the log-encoded images.

This primarily demonstrates the meta-transform `GroupTransform`: a transform which simply composes two or more transforms together into one. Anything that acceptsa transform like `FileTransform` or `CDLTransform` will also accept a `GroupTransform`

It is also worth noting the `ColorSpaceTransform`, which transforms between `lnf` and `lg16` colorspaces (which are defined within the current config).

Example transform steps This section explains how OCIO internally applies all the transforms. It can be skipped over if you understand how the reference colorspace works.

```
colorspaces:
  - !<ColorSpace>
    name: lnf
    bitdepth: 32f
    description: |
```

```
    lnf : linear show space

- !<ColorSpace>
  name: lg16
  bitdepth: 16ui
  description: |
    lg16 : conversion from film log
  to_reference: !<FileTransform> {src: lg16.spi1d, interpolation: nearest}

- !<ColorSpace>
  name: srgb8
  bitdepth: 8ui
  description: |
    srgb8 :rgb display space for the srgb standard.
  from_reference: !<GroupTransform>
    children:
      - !<ColorSpaceTransform> {src: lnf, dst: lg16}
      - !<FileTransform> {src: lg16_to_srgb8.spi3d, interpolation: linear}
```

To explain how this all ties together to display an image, say we have an image in the `lnf` colorspace (e.g a linear EXR) and wish to convert it to `srgb8` - the transform steps are:

- `ColorSpaceTransform` is applied, converting from lnf to lg16
- The `FileTransform` is applied, converting from lg16 to srgb8.

A more complex example: we have an image in the `lg16` colorspace, and convert to `srgb8` (using the lg16 definition from earlier, or the *spi-vfx* config):

First OCIO converts from lg16 to the reference space, using the transform defined in lg16's to_reference:

- `FileTransform` applies the lg16.spi1d

With the image now in the reference space, srgb8's transform is applied:

- ColorSpaceTransform to transform from lnf to lg16
- FileTransform applies the `lg16_to_srgb8.spi3d` LUT.

Note: OCIO has an transform optimiser which removes redunant steps, and combines similar transforms into one operation.

In the previous example, the complete transform chain would be "lg16 -> lnf, lnf -> lg16, lg16 -> srgb8". However the optimiser will reduce this to "lg16 -> srgb".

`bitdepth` Optional. Default: `32f`

Specify an appropriate bitdepth for the colorspace, and applications can use this to automatically output images in the correct bit-depth.

Valid options are:

- `8ui`
- `10ui`
- `12ui`
- `14ui`
- `16ui`

- 32ui
- 16f
- 32f

The number is in bits. ui stands for unsigned integer. f stands for floating point.

Example:

```
- !<ColorSpace>
  name: srgb8
  bitdepth: 8ui

  from_reference: [...]
```

isdata: Optional. Default: false. Boolean.

The isdata key on a colorspace informs OCIO that this colorspace is used for non-color data channels, such as the "normals" output of a a multipass 3D render.

Here is example "non-color" colorspace from the *spi-vfx* config:

```
- !<ColorSpace>
  name: ncf
  family: nc
  equalitygroup:
  bitdepth: 32f
  description: |
    ncf :nc,Non-color used to store non-color data such as depth or surface normals
  isdata: true
  allocation: uniform
```

equalitygroup: Optional.

If two colorspaces are in the "equality group", transforms between them are considered non-operations.

You might have multiple colorspaces which are identical, but operate at different bit-depths.

For example, see the lg10 and lg16 colorspaces in the *spi-vfx* config. If loading a lg10 image and converting to lg16, no transform is required. This is of course faster, but may cause an unexpected increase in precision (e.g it skip potential clamping caused by a LUT)

```
- !<ColorSpace>
  name: lg16
  equalitygroup: lg
  bitdepth: 16ui
  to_reference: !<FileTransform> {src: lg16.spi1d, interpolation: nearest}

- !<ColorSpace>
  name: lg10
  equalitygroup: lg
  bitdepth: 10ui
  to_reference: !<FileTransform> {src: lg10.spi1d, interpolation: nearest}
```

Do not put different colorspaces in the same equality group. For logical grouping of "similar" colorspaces, use the family option.

family: Optional.

Allows for logical grouping of colorspaces within a UI.

For example, a series of "log" colorspaces could be put in one "family". Within a UI like the Nuke `OCIOColorSpace` node, these will be grouped together.

```
- !<ColorSpace>
  name: kodaklog
  family: log
  equalitygroup: kodaklog
  [...]

- !<ColorSpace>
  name: si2klog
  family: log
  equalitygroup: si2klog
  [...]

- !<ColorSpace>
  name: rec709
  family: display
  equalitygroup: rec709
  [...]
```

Unlike `equalitygroup`, the `family` has no impact on image processing.

allocation and allocationvars Optional.

These two options are used when OCIO transforms are applied on the GPU.

It is also used to automatically generate a "shaper LUT" when *baking LUT's* unless one is explicitly specified (not all output formats utilise this)

For a detailed description, see *How to Configure ColorSpace Allocation*

Example of a "0-1" colorspace

```
allocation: uniform
allocationvars: [0.0, 1.0]

allocation: lg2
allocationvars: [-15, 6]
```

description Optional.

A human-readable description of the colorspace.

The YAML syntax allows for either single-line descriptions:

```
- !<ColorSpace>
  name: kodaklog
  [...]
  description: A concise description of the kodaklog colorspace.
```

Or multiple-lines:

```
- !<ColorSpace>
  name: kodaklog
  [...]
```

```
  description:
    This is a multi-line description of the kodaklog colorspace,
    to demonstrate the YAML syntax for doing so.

    Here is the second line. The first one will be unwrapped into
    a single line, as will this one.
```

It's common to use literal | block syntax to preserve all newlines:

```
- !<ColorSpace>
  name: kodaklog
  [...]
  description: |
    This is one line.
    This is the second.
```

Available transforms

AllocationTransform

Transforms from reference space to the range specified by the vars:

Keys:

- allocation
- vars
- direction

CDLTransform

Applies an ASC CDL compliant grade

Keys:

- slope
- offset
- power
- sat
- direction

ColorSpaceTransform

Transforms from src colorspace to dst colorspace.

Keys:

- src
- dst
- direction

ExponentTransform

Raises pixel values to a given power (often referred to as "gamma")

```
!<ExponentTransform> {value: [1.8, 1.8, 1.8, 1]}
```

Keys:

- value
- direction

FileTransform

Applies a lookup table (LUT)

Keys:

- src
- cccid
- interpolation
- direction

GroupTransform

Combines multiple transforms into one.

```
colorspaces:

  - !<ColorSpace>
    name: adx10

    [...]

    to_reference: !<GroupTransform>
      children:
        - !<FileTransform> {src: adx_adx10_to_cdd.spimtx}
        - !<FileTransform> {src: adx_cdd_to_cid.spimtx}
```

A group transform is accepted anywhere a "regular" transform is.

LogTransform

Applies a mathematical logarithm with a given base to the pixel values.

Keys:

- base

LookTransform

Applies a named look

MatrixTransform

Applies a matrix transform to the pixel values

Keys:

- matrix
- offset
- direction

TruelightTransform

Applies a transform from a Truelight profile.

Keys:

- config_root
- profile
- camera
- input_display
- recorder
- print
- lamp
- output_camera
- display
- cube_input
- direction

Note: This transform requires OCIO to be compiled with the Truelight SDK present.

3.5 Developer guide

Some information on contributing to OCIO:

3.5.1 Getting started

Checking Out The Codebase

The master code repository is available on Github: http://github.com/imageworks/OpenColorIO

For those unfamiliar with git, the wonderful part about it is that even though only a limited number people have write access to the master repository, anyone is free to create, and even check in, changes to their own local git repository. Your local changes will not automatically be pushed back to the master repository, so everyone feel free to informally play around with the codebase. Also - unlike svn - when you download the git repository you have a full copy of the project's history (including revision history, logs, etc), so the majority of code changes you will make, including commits, do not require network server access.

The first step is to install git on your system. For those new to the world of git, github has an excellent tutorial stepping you through the process, available at: http://help.github.com/

To check out a read-only version of the repository (no github signup required):

```
git clone git://github.com/imageworks/OpenColorIO.git ocio
```

For write-access, you must first register for a Github account (free). Then, you must create a local fork of the OpenColorIO repository by visiting http://github.com/imageworks/OpenColorIO and clicking the "Fork" icon. If you get hung up on this, further instructions on this process are available at http://help.github.com/forking/

To check out a read-write version of the repository (github acct required):

```
git clone git@github.com:yourgitusername/OpenColorIO.git ocio

Initialized empty Git repository in /mcp/ocio/.git/
remote: Counting objects: 2220, done.
remote: Compressing objects: 100% (952/952), done.
remote: Total 2220 (delta 1434), reused 1851 (delta 1168)
Receiving objects: 100% (2220/2220), 2.89 MiB | 2.29 MiB/s, done.
Resolving deltas: 100% (1434/1434), done.
```

Both read + read/write users should then add the Imageworks (SPI) master branch as a remote. This will allow you to more easily fetch updates as they become available:

```
cd ocio
git remote add spi git://github.com/imageworks/OpenColorIO.git
```

Optionally, you may then add any additional users who have individual working forks (just as you've done). This will allow you to track, view, and potentially merge intermediate changes before they're been pushed into the main trunk. (For really bleeding edge folks). For example, to add Jeremy Selan's working fork:

```
git remote add js git://github.com/jeremyselan/OpenColorIO.git
```

You should then do a git fetch, and git merge (detailed below) to download the remote branches you've just added.

Merging changes

More detailed guide coming soon, for now, see http://help.github.com/remotes/

3.5.2 Coding guidelines

There are only two important rules:

1. When making changes, conform to the style and conventions of the surrounding code.
2. Strive for clarity, even if that means occasionally breaking the guidelines. Use your head and ask for advice if your common sense seems to disagree with the conventions.

File Conventions

C++ implementation should be named `*.cpp`. Headers should be named `*.h`. All source files should begin with the copyright and license, which can be cut and pasted from other source files). For NEW source files, please do change the copyright year to the present. However DO NOT edit existing files just to update the copyright year, it just creates pointless deltas and offers no increased protection

Line Length

Each line of text in your code should be at most 80 characters long.

Generally the only exceptions are for comments with example commands or URLs - to make cut and paste easier.

The other exception is for those rare cases where letting a line be longer (and wrapping on an 80-character window) is actually a better and clearer alternative than trying to split it into two lines. Sometimes this happens, but it's rare.

DO NOT alter somebody else's code to re-wrap lines (or change whitespace) just because you found something that violates the rules. Let the group/author/leader know, and resist the temptation to change it yourself.

Formatting

- Indent 4 spaces at a time, and use actual spaces, not tabs. This is particularly critical on python code. The only exception currently allowed in within Makefiles, where tab characters are sometimes required.
- Opening brace on the line following the condition or loop.
- The contents of namespaces are not indented.
- Function names should be on the same line as their return values.
- Function calls should NOT have a space between the function name and the opening parenthesis. A single space should be added after each required comma.

Here is a short code fragement that shows these concepts in action:

```
namespace
{


int MungeMyNumbers(int a, int b)
{
    int x = a + b;

    if (a == 0 || b==0)
    {
        x += 1;
        x += 2;
    }
    else
    {
        for (int i=0; i<16; ++i)
        {
            x += a * i;
        }
    }

    return x;
}


} // namespace
```

Misc. Rules

- Avoid macros when possible.

- Anonymous namespaces are preferred when sensible.
- Variable names should be camelCase, as well as longAndExplicit.
- Avoid long function implementations. Break up work into small, manageable chunks.
- Use "TODO:" comments for code that is temporary, a short-term solution, or good-enough but not perfect. This is vastly preferred to leaving subtle corner cases undocumented.
- Always initialize variables on construction.
- Do not leave dead code paths around. (This is what revision history is for)
- Includes should always be ordered as follows: C library, C++ library, other libraries' .h, OCIO public headers, OCIO private headers. Includes within a category should be alphabetized.
- The C++ "using" directive is not allowed.
- Static / global variables should be avoided at all costs.
- Use const whenever it makes sense to do so.
- The use of Boost is not allowed, except for unit_test_framework and shared_ptr.
- Default function arguments are not allowed.

Bottom Line

When in doubt, look elsewhere in the code base for examples of similar structures and try to format your code in the same manner.

Portions of this document have been blatantly lifted from OpenImageIO, and Google.

3.5.3 Documentation guidelines

OpenColorIO is documentated using reStructuredText, processed by Sphinx.

The documentation primarily lives in the `docs/` folder, within the main OpenColorIO repoistory.

The rST source for the C++ API documentation is extracted from comments in the public header files in `export/`

The Python API documentation is extracted from dummy .py files within the `src/pyglue/DocStrings/` folder

Building the docs

Just like a *regular build from source*, but specify the `-D OCIO_BUILD_DOCS=yes` argument to CMake.

Then run the `make doc` target. The default HTML output will be created in `build_dir/docs/build-html/`

Note that CMake must be run before each invokation of `make` to copy the edited rST files.

Initial run:

```
$ mkdir build && cd build
```

Then after each change you wish to preview:

```
$ cmake -D OCIO_BUILD_DOCS=yes .. && make doc
```

Basics

- Try to keep the writing style consistent with surrounding docs.
- Fix all warnings output by the Sphinx build process. An example of such an warning is:

```
checking consistency... [...]/build/docs/userguide/writing_configs.rst:: WARNING: document isn't
```

- Use the following hierarchy of header decorations:

```
Level 1 heading
===============

Level 2 heading
***************

Level 3 heading
+++++++++++++++

Level 4 heading
---------------
```

- To add a new page, create a new `.rst` file in the appropriate location. In that directory's `index.rst`, add the new file to the `toctree` directive.

  The new file should contain a top-level heading (decorated with `=====` underline), and an approriate label for referencing from other pages. For example, a new file `docs/userguide/baking_luts.rst` might start like this:

```
.. _userguide-bakingluts:

Baking LUT's
============

In order to bake a LUT, ...
```

Emacs rST mode

Emacs' includes a mode for editing rST files. It is documented on the docutils site

One of the features it includes is readjusting the hierarchy of heading decorations (the underlines for different heading levels). To configure this to use OCIO's convention, put the following in your `.emacs.d/init.el`:

```
(setq rst-preferred-decorations
      '((?= simple 0)
        (?* simple 0)
        (?+ simple 0)
        (?- simple 0)))
```

3.5.4 Submitting Changes

Code Review

Ask early, and ask often!

All new contributors are highly encouraged to post development ideas, questions, or any other thoughts to the *Mailing lists* before starting to code. This greatly improves the process and quality of the resulting library. Code reviews (particularly for non-trivial changes) are typically far simpler if the reviewers are aware of a development task beforehand. (And, who knows? Maybe they will have implementation suggestions as well!)

3.5.5 Issues

Please visit http://github.com/imageworks/OpenColorIO/issues for an up to date listing of bugs, feature requests etc

Instructions on using OCIO:

3.5.6 Usage Examples

Some examples of using the OCIO API, via both C++ and the Python bindings.

For further examples, see the `src/apps/` directory in the git repository

Applying a basic ColorSpace transform, using the CPU

This describes what code is used to convert from a specified source `ColorSpace` to a specified destination `ColorSpace`. If you are using the OCIO Nuke plugins, the OCIOColorSpace node performs these steps internally.

1. **Get the Config.** This represents the entirety of the current color "universe". It can either be initialized by your app at startup or created explicitly. In common usage, you can just query `GetCurrentConfig()`, which will auto initialize on first use using the `OCIO` environment variable.
2. **Get Processor from the Config.** A processor corresponds to a 'baked' color transformation. You specify two arguments when querying a processor: the *ColorSpace* you are coming from, and the *ColorSpace* you are going to. *ColorSpaces* ColorSpaces can be either explicitly named strings (defined by the current configuration) or can be *Roles* (essentially *ColorSpace* aliases) which are consistent across configurations. Constructing a `Processor` object is likely a blocking operation (thread-wise) so care should be taken to do this as infrequently as is sensible. Once per render 'setup' would be appropriate, once per scanline would be inappropriate.
3. **Convert your image, using the Processor.** Once you have the processor, you can apply the color transformation using the "apply" function. In *C++*, you apply the processing in-place, by first wrapping your image in an `ImageDesc` class. This approach is intended to be used in high performance applications, and can be used on multiple threads (per scanline, per tile, etc). In *Python* you call "applyRGB" / "applyRGBA" on your sequence of pixels. Note that in both languages, it is far more efficient to call "apply" on batches of pixels at a time.

C++

```
#include <OpenColorIO/OpenColorIO.h>
namespace OCIO = OCIO_NAMESPACE;

try
{
    OCIO::ConstConfigRcPtr config = OCIO::GetCurrentConfig();
    ConstProcessorRcPtr processor = config->getProcessor(OCIO::ROLE_COMPOSITING_LOG,
                                                         OCIO::ROLE_SCENE_LINEAR);

    OCIO::PackedImageDesc img(imageData, w, h, 4);
    processor->apply(img);
}
catch(OCIO::Exception & exception)
```

```
{
    std::cerr << "OpenColorIO Error: " << exception.what() << std::endl;
}
```

Python

```
import PyOpenColorIO as OCIO

try:
    config = OCIO.GetCurrentConfig()
    processor = config.getProcessor(OCIO.Constants.ROLE_COMPOSITING_LOG,
                                    OCIO.Constants.ROLE_SCENE_LINEAR)

    # Apply the color transform to the existing RGBA pixel data
    img = processor.applyRGBA(img)
except Exception, e:
    print "OpenColorIO Error",e
```

Displaying an image, using the CPU (simple ColorSpace conversion)

Converting an image for display is similar to a normal color space conversion. The only difference is that one has to first determine the name of the display (destination) ColorSpace by querying the config with the device name and transform name.

1. **Get the Config.** See *Applying a basic ColorSpace transform, using the CPU* for details.
2. **Lookup the display ColorSpace.** The display `ColorSpace` is queried from the configuration using `Config::getDisplayColorSpaceName()`. If the user has specified value for the `device` or the `displayTransformName`, use them. If these values are unknown default values can be queried (as shown below).
3. **Get the processor from the Config.** See *Applying a basic ColorSpace transform, using the CPU* for details.
4. **Convert your image, using the processor.** See *Applying a basic ColorSpace transform, using the CPU* for details.

C++

```
#include <OpenColorIO/OpenColorIO.h>
namespace OCIO = OCIO_NAMESPACE;

OCIO::ConstConfigRcPtr config = OCIO::GetCurrentConfig();

// If the user hasn't picked a display, use the defaults...
const char * device = config->getDefaultDisplayDeviceName();
const char * transformName = config->getDefaultDisplayTransformName(device);
const char * displayColorSpace = config->getDisplayColorSpaceName(device, transformName);

ConstProcessorRcPtr processor = config->getProcessor(OCIO::ROLE_SCENE_LINEAR,
                                                     displayColorSpace);

OCIO::PackedImageDesc img(imageData, w, h, 4);
processor->apply(img);
```

Python

```
import PyOpenColorIO as OCIO

config = OCIO.GetCurrentConfig()

device = config.getDefaultDisplayDeviceName()
transformName = config.getDefaultDisplayTransformName(device)
displayColorSpace = config.getDisplayColorSpaceName(device, transformName)

processor = config.getProcessor(OCIO.Constants.ROLE_SCENE_LINEAR, displayColorSpace)

processor.applyRGB(imageData)
```

Displaying an image, using the CPU (Full Display Pipeline)

This alternative version allows for a more complex displayTransform, allowing for all of the controls typically added to real-world viewer interfaces. For example, options are allowed to control which channels (red, green, blue, alpha, luma) are visible, as well as allowing for optional color corrections (such as an exposure offset in scene linear). If you are using the OCIO Nuke plugins, the OCIODisplay node performs these steps internally.

1. **Get the Config.** See *Applying a basic ColorSpace transform, using the CPU* for details.
2. **Lookup the display ColorSpace.** See *Displaying an image, using the CPU (simple ColorSpace conversion)* for details
3. **Create a new DisplayTransform.** This transform will embody the full 'display' pipeline you wish to control. The user is required to call `DisplayTransform::setInputColorSpaceName()` to set the input ColorSpace, as well as `DisplayTransform::setDisplayColorSpaceName()` (with the results of `Config::getDisplayColorSpaceName()`).
4. **Set any additional DisplayTransform options.** If the user wants to specify a channel swizzle, a scene-linear exposure offset, an artistic look, this is the place to add it. See below for an example. Note that although we provide recommendations for display, any transforms are allowed to be added into any of the slots. So if for your app you want to add 3 transforms into a particular slot (chained together), you are free to wrap them in a `GroupTransform` and set it accordingly!
5. **Get the processor from the Config.** The processor is then queried from the config passing the new `DisplayTransform` as the argument. Once the processor has been returned, the original `DisplayTransform` is no longer necessary to hold onto. (Though if you'd like to for re-use, there is no problem doing so).
6. **Convert your image, using the processor.** See *Applying a basic ColorSpace transform, using the CPU* for details.

C++

```
// Step 1: Get the config
OCIO::ConstConfigRcPtr config = OCIO::GetCurrentConfig();

// Step 2: Lookup the display ColorSpace
const char * device = config->getDefaultDisplayDeviceName();
const char * transformName = config->getDefaultDisplayTransformName(device);
const char * displayColorSpace = config->getDisplayColorSpaceName(device, transformName);

// Step 3: Create a DisplayTransform, and set the input and display ColorSpaces
```

```
// (This example assumes the input is scene linear. Adapt as needed.)

OCIO::DisplayTransformRcPtr transform = OCIO::DisplayTransform::Create();
transform->setInputColorSpaceName( OCIO::ROLE_SCENE_LINEAR );
transform->setDisplayColorSpaceName( displayColorSpace );

// Step 4: Add custom transforms for a 'canonical' Display Pipeline

// Add an fstop exposure control (in SCENE_LINEAR)
float gain = powf(2.0f, exposure_in_stops);
const float slope3f[] = { gain, gain, gain };
OCIO::CDLTransformRcPtr cc =  OCIO::CDLTransform::Create();
cc->setSlope(slope3f);
transform->setLinearCC(cc);

// Add a Channel view 'swizzle'

// 'channelHot' controls which channels are viewed.
int channelHot[4] = { 1, 1, 1, 1 };  // show rgb
//int channelHot[4] = { 1, 0, 0, 0 };  // show red
//int channelHot[4] = { 0, 0, 0, 1 };  // show alpha
//int channelHot[4] = { 1, 1, 1, 0 };  // show luma

float lumacoef[3];
config.getDefaultLumaCoefs(lumacoef);

float m44[16];
float offset[4];
OCIO::MatrixTransform::View(m44, offset, channelHot, lumacoef);
OCIO::MatrixTransformRcPtr swizzle = OCIO::MatrixTransform::Create();
swizzle->setValue(m44, offset);
transform->setChannelView(swizzle);

// And then process the image normally.
OCIO::ConstProcessorRcPtr processor = config->getProcessor(transform);

OCIO::PackedImageDesc img(imageData, w, h, 4);
processor->apply(img);
```

Python

```
import PyOpenColorIO as OCIO

# Step 1: Get the config
config = OCIO.GetCurrentConfig()

# Step 2: Lookup the display ColorSpace
device = config.getDefaultDisplayDeviceName()
transformName = config.getDefaultDisplayTransformName(device)
displayColorSpace = config.getDisplayColorSpaceName(device, transformName)

# Step 3: Create a DisplayTransform, and set the input and display ColorSpaces
# (This example assumes the input is scene linear. Adapt as needed.)

transform = OCIO.DisplayTransform()
transform.setInputColorSpaceName(OCIO.Constants.ROLE_SCENE_LINEAR)
```

```
transform.setDisplayColorSpaceName(displayColorSpace)

# Step 4: Add custom transforms for a 'canonical' Display Pipeline

# Add an fstop exposure control (in SCENE_LINEAR)
gain = 2**exposure
slope3f = (gain, gain, gain)

cc = OCIO.CDLTransform()
cc.setSlope(slope3f)

transform.setLinearCC(cc)

# Add a Channel view 'swizzle'

channelHot = (1, 1, 1, 1) # show rgb
# channelHot = (1, 0, 0, 0) # show red
# channelHot = (0, 0, 0, 1) # show alpha
# channelHot = (1, 1, 1, 0) # show luma

lumacoef = config.getDefaultLumaCoefs()

m44, offset = OCIO.MatrixTransform.View(channelHot, lumacoef)

swizzle = OCIO.MatrixTransform()
swizzle.setValue(m44, offset)
transform.setChannelView(swizzle)

# And then process the image normally.
processor = config.getProcessor(transform)

print processor.applyRGB(imageData)
```

Displaying an image, using the GPU

Applying OpenColorIO's color processing using GPU processing is straightforward, provided you have the capability to upload custom shader code and a custom 3D Lookup Table (3DLUT).

1. **Get the Processor.** This portion of the pipeline is identical to the CPU approach. Just get the processor as you normally would have, see above for details.
2. **Create a GpuShaderDesc.**
3. **Query the GPU Shader Text + 3D LUT.**
4. **Configure the GPU State.**
5. **Draw your image.**

C++

This example is available as a working app in the OCIO source: src/apps/ociodisplay.

```
// Step 0: Get the processor using any of the pipelines mentioned above.
OCIO::ConstConfigRcPtr config = OCIO::GetCurrentConfig();
const char * device = config->getDefaultDisplayDeviceName();
const char * transformName = config->getDefaultDisplayTransformName(device);
```

```
const char * displayColorSpace = config->getDisplayColorSpaceName(device, transformName);
ConstProcessorRcPtr processor = config->getProcessor(OCIO::ROLE_SCENE_LINEAR,
                                                     displayColorSpace);

// Step 1: Create a GPU Shader Description
GpuShaderDesc shaderDesc;
shaderDesc.setLanguage(OCIO::GPU_LANGUAGE_GLSL_1_0);
shaderDesc.setFunctionName("OCIODisplay");
const int LUT3D_EDGE_SIZE = 32;
shaderDesc.setLut3DEdgeLen(LUT3D_EDGE_SIZE);

// Step 2: Compute and the 3D LUT
// Optional Optimization:
//     Only do this the 3D LUT's contents
//     are different from the last drawn frame.
//     Use getGpuLut3DCacheID to compute the cacheID.
//     cheaply.
//
// std::string lut3dCacheID = processor->getGpuLut3DCacheID(shaderDesc);
int num3Dentries = 3*LUT3D_EDGE_SIZE*LUT3D_EDGE_SIZE*LUT3D_EDGE_SIZE;
std::vector<float> g_lut3d;
g_lut3d.resize(num3Dentries);
processor->getGpuLut3D(&g_lut3d[0], shaderDesc);

// Load the data into an OpenGL 3D Texture
glGenTextures(1, &g_lut3d_textureID);
glBindTexture(GL_TEXTURE_3D, g_lut3d_textureID);
glTexImage3D(GL_TEXTURE_3D, 0, GL_RGB,
             LUT3D_EDGE_SIZE, LUT3D_EDGE_SIZE, LUT3D_EDGE_SIZE,
             0, GL_RGB,GL_FLOAT, &g_lut3d[0]);

// Step 3: Query
```

C++ API documentation:

3.5.7 C++ API

Usage Example: *Compositing plugin that converts from "log" to "lin"*

```
#include <OpenColorIO/OpenColorIO.h>
namespace OCIO = OCIO_NAMESPACE;

try
{
    // Get the global OpenColorIO config
    // This will auto-initialize (using $OCIO) on first use
    OCIO::ConstConfigRcPtr config = OCIO::GetCurrentConfig();

    // Get the processor corresponding to this transform.
    OCIO::ConstProcessorRcPtr processor = config->getProcessor(OCIO::ROLE_COMPOSITING_LOG,
                                                               OCIO::ROLE_SCENE_LINEAR);

    // Wrap the image in a light-weight ImageDescription
    OCIO::PackedImageDesc img(imageData, w, h, 4);

    // Apply the color transformation (in place)
    processor->apply(img);
```

```
}
catch(OCIO::Exception & exception)
{
    std::cerr << "OpenColorIO Error: " << exception.what() << std::endl;
}
```

Exceptions

class `Exception`
An exception class to throw for errors detected at runtime.

> **Warning:** All functions in the Config class can potentially throw this exception.

`Exception::`**`Exception`**`(`*const char**`)`
Constructor that takes a string as the exception message.

`Exception::`**`Exception`**`(`*const Exception&*`)`
Constructor that takes an exception pointer.

Exception& `Exception::`**`operator=`**`(`*const Exception&*`)`
Constructor that takes an exception pointer and returns an exception pointer (???).

`Exception::`**`~Exception`**`()`

const char* `Exception::`**`what`**`() const`

class `ExceptionMissingFile`
An exception class for errors detected at runtime, thrown when OCIO cannot find a file that is expected to exist. This is provided as a custom type to distinguish cases where one wants to continue looking for missing files, but wants to properly fail for other error conditions.

`ExceptionMissingFile::`**`ExceptionMissingFile`**`(`*const char**`)`

`ExceptionMissingFile::`**`ExceptionMissingFile`**`(`*const ExceptionMissingFile&*`)`

Global

void **`ClearAllCaches`**`()`
OpenColorIO, during normal usage, tends to cache certain information (such as the contents of LUTs on disk, intermediate results, etc.). Calling this function will flush all such information. Under normal usage, this is not necessary, but it can be helpful in particular instances, such as designing OCIO profiles, and wanting to re-read luts without restarting.

const char* **`GetVersion`**`()`
Get the version number for the library, as a dot-delimited string (e.g., "1.0.0"). This is also available at compile time as OCIO_VERSION.

int **`GetVersionHex`**`()`
Get the version number for the library, as a single 4-byte hex number (e.g., 0x01050200 for "1.5.2"), to be used for numeric comparisons. This is also available at compile time as OCIO_VERSION_HEX.

LoggingLevel **`GetLoggingLevel`**`()`
Get the global logging level. You can override this at runtime using the `OCIO_LOGGING_LEVEL` environment variable. The client application that sets this should use `SetLoggingLevel()`, and not the environment variable. The default value is INFO.

void **SetLoggingLevel**(LoggingLevel *level*)
Set the global logging level.

Config

A config defines all the color spaces to be available at runtime.

The color configuration (`Config`) is the main object for interacting with this library. It encapsulates all of the information necessary to use customized `ColorSpaceTransform` and `DisplayTransform` operations.

See the *User Guide* for more information on selecting, creating, and working with custom color configurations.

For applications interested in using only one color config at a time (this is the vast majority of apps), their API would traditionally get the global configuration and use that, as opposed to creating a new one. This simplifies the use case for plugins and bindings, as it alleviates the need to pass around configuration handles.

An example of an application where this would not be sufficient would be a multi-threaded image proxy server (daemon), which wished to handle multiple show configurations in a single process concurrently. This app would need to keep multiple configurations alive, and to manage them appropriately.

Roughly speaking, a novice user should select a default configuration that most closely approximates the use case (animation, visual effects, etc.), and set the `OCIO` environment variable to point at the root of that configuration.

Note: Initialization using environment variables is typically preferable in a multi-app ecosystem, as it allows all applications to be consistently configured.

See *Usage Examples*

ConstConfigRcPtr **GetCurrentConfig**()
Get the current configuration.

void **SetCurrentConfig**(const ConstConfigRcPtr& *config*)
Set the current configuration. This will then store a copy of the specified config.

class Config

Initialization

static ConfigRcPtr `Config::`**Create**()
Constructor...ELABORATE

static ConstConfigRcPtr `Config::`**CreateFromEnv**()

static ConstConfigRcPtr `Config::`**CreateFromFile**(const char* *filename*)

static ConstConfigRcPtr `Config::`**CreateFromStream**(std::istream& *istream*)

ConfigRcPtr `Config::`**createEditableCopy**() `const`

void `Config::`**sanityCheck**() `const`
This will throw an exception if the config is malformed. The most common error occurs when references are made to colorspaces that do not exist.

const char* `Config::`**getDescription**() `const`

void `Config::`**setDescription**(const char* *description*)

void `Config::`**serialize**(std::ostream& *os*) `const`
Returns the string representation of the Config in YAML text form. This is typically stored on disk in a file with the extension .ocio.

const char* Config::**getCacheID**() const
This will produce a hash of the all colorspace definitions, etc. All external references, such as files used in FileTransforms, etc., will be incorporated into the cacheID. While the contents of the files are not read, the file system is queried for relavent information (mtime, inode) so that the config's cacheID will change when the underlying luts are updated. If a context is not provided, the current Context will be used. If a null context is provided, file references will not be taken into account (this is essentially a hash of Config::serialize).

const char* Config::**getCacheID**(const ConstContextRcPtr& *context*) const

Resources

Given a lut src name, where should we find it?

ConstContextRcPtr Config::**getCurrentContext**() const

void Config::**addEnvironmentVar**(const char* *name*, const char* *defaultValue*)

int Config::**getNumEnvironmentVars**() const

const char* Config::**getEnvironmentVarNameByIndex**(int *index*) const

const char* Config::**getEnvironmentVarDefault**(const char* *name*) const

void Config::**clearEnvironmentVars**()

const char* Config::**getSearchPath**() const

void Config::**setSearchPath**(const char* *path*)

const char* Config::**getWorkingDir**() const

void Config::**setWorkingDir**(const char* *dirname*)

ColorSpaces

int Config::**getNumColorSpaces**() const

const char* Config::**getColorSpaceNameByIndex**(int *index*) const
This will null if an invalid index is specified

Note: These fcns all accept either a color space OR role name. (Colorspace names take precedence over roles.)

ConstColorSpaceRcPtr Config::**getColorSpace**(const char* *name*) const
This will return null if the specified name is not found.

int Config::**getIndexForColorSpace**(const char* *name*) const

void Config::**addColorSpace**(const ConstColorSpaceRcPtr& *cs*)

Note: If another color space is already registered with the same name, this will overwrite it. This stores a copy of the specified color space.

---

void Config::**clearColorSpaces**()

const char* Config::**parseColorSpaceFromString**(const char* *str*) const
Given the specified string, get the longest, right-most, colorspace substring that appears.

- If strict parsing is enabled, and no color space is found, return an empty string.

- If strict parsing is disabled, return ROLE_DEFAULT (if defined).
- If the default role is not defined, return an empty string.

bool `Config::isStrictParsingEnabled() const`

void `Config::setStrictParsingEnabled`(bool *enabled*)

Roles

A role is like an alias for a colorspace. You can query the colorspace corresponding to a role using the normal getColorSpace fcn.

void `Config::setRole`(const char* *role*, const char* *colorSpaceName*)

> **Note:** Setting the `colorSpaceName` name to a null string unsets it.

int `Config::getNumRoles() const`

bool `Config::hasRole`(const char* *role*) `const`
Return true if the role has been defined.

const char* `Config::getRoleName`(int *index*) `const`
Get the role name at index, this will return values like 'scene_linear', 'compositing_log'. Return empty string if index is out of range.

Display/View Registration

Looks is a potentially comma (or colon) delimited list of lookNames, Where +/- prefixes are optionally allowed to denote forward/inverse look specification. (And forward is assumed in the absense of either)

const char* `Config::getDefaultDisplay() const`

int `Config::getNumDisplays() const`

const char* `Config::getDisplay`(int *index*) `const`

const char* `Config::getDefaultView`(const char* *display*) `const`

int `Config::getNumViews`(const char* *display*) `const`

const char* `Config::getView`(const char* *display*, int *index*) `const`

const char* `Config::getDisplayColorSpaceName`(const char* *display*, const char* *view*) `const`

const char* `Config::getDisplayLooks`(const char* *display*, const char* *view*) `const`

void `Config::addDisplay`(const char* *display*, const char* *view*, const char* *colorSpaceName*, const char* *looks*)
For the (display,view) combination, specify which colorSpace and look to use. If a look is not desired, then just pass an empty string

void `Config::clearDisplays()`

void `Config::setActiveDisplays`(const char* *displays*)
Comma-delimited list of display names.

const char* `Config::getActiveDisplays() const`

void `Config::setActiveViews`(const char* *views*)
Comma-delimited list of view names.

const char* Config::**getActiveViews**() const

Luma

Get the default coefficients for computing luma.

Note: There is no "1 size fits all" set of luma coefficients. (The values are typically different for each colorspace, and the application of them may be nonsensical depending on the intensity coding anyways). Thus, the 'right' answer is to make these functions on the Config class. However, it's often useful to have a config-wide default so here it is. We will add the colorspace specific luma call if/when another client is interesting in using it.

---

void Config::**getDefaultLumaCoefs**(float* *rgb*) const

void Config::**setDefaultLumaCoefs**(const float* *rgb*)
These should be normalized (sum to 1.0 exactly).

Look

Manager per-shot look settings.

ConstLookRcPtr Config::**getLook**(const char* *name*) const

int Config::**getNumLooks**() const

const char* Config::**getLookNameByIndex**(int *index*) const

void Config::**addLook**(const ConstLookRcPtr& *look*)

void Config::**clearLooks**()

Processors

Convert from inputColorSpace to outputColorSpace

Note: This may provide higher fidelity than anticipated due to internal optimizations. For example, if the inputcolorspace and the outputColorSpace are members of the same family, no conversion will be applied, even though strictly speaking quantization should be added.

If you wish to test these calls for quantization characteristics, apply in two steps; the image must contain RGB triples (though arbitrary numbers of additional channels can be supported (ignored) using the pixelStrideBytes arg).

ConstProcessorRcPtr Config::**getProcessor**(const ConstContextRcPtr& *context*, const ConstColorSpaceRcPtr& *srcColorSpace*, const ConstColorSpaceRcPtr& *dstColorSpace*) const

ConstProcessorRcPtr Config::**getProcessor**(const ConstColorSpaceRcPtr& *srcColorSpace*, const ConstColorSpaceRcPtr& *dstColorSpace*) const

ConstProcessorRcPtr Config::**getProcessor**(const char* *srcName*, const char* *dstName*) const

> **Note:** Names can be colorspace name, role name, or a combination of both.

ConstProcessorRcPtr Config::**getProcessor**(const ConstContextRcPtr& *context*, const char* *srcName*, const char* *dstName*) const

Get the processor for the specified transform.

Not often needed, but will allow for the re-use of atomic OCIO functionality (such as to apply an individual LUT file).

ConstProcessorRcPtr `Config::`**`getProcessor`**(const ConstTransformRcPtr& *transform*) `const`

ConstProcessorRcPtr `Config::`**`getProcessor`**(const ConstTransformRcPtr& *transform*, TransformDirection *direction*) `const`

ConstProcessorRcPtr `Config::`**`getProcessor`**(const ConstContextRcPtr& *context*, const ConstTransformRcPtr& *transform*, TransformDirection *direction*) `const`

ColorSpace

The *ColorSpace* is the state of an image with respect to colorimetry and color encoding. Transforming images between different *ColorSpaces* is the primary motivation for this library.

While a complete discussion of colorspaces is beyond the scope of header documentation, traditional uses would be to have *ColorSpaces* corresponding to: physical capture devices (known cameras, scanners), and internal 'convenience' spaces (such as scene linear, logarithmic).

ColorSpaces are specific to a particular image precision (float32, uint8, etc.), and the set of ColorSpaces that provide equivalent mappings (at different precisions) are referred to as a 'family'.

class `ColorSpace`

static ColorSpaceRcPtr `ColorSpace::`**`Create`**()

ColorSpaceRcPtr `ColorSpace::`**`createEditableCopy`**() `const`

const char* `ColorSpace::`**`getName`**() `const`

void `ColorSpace::`**`setName`**(const char* *name*)

const char* `ColorSpace::`**`getFamily`**() `const`
Get the family, for use in user interfaces (optional)

void `ColorSpace::`**`setFamily`**(const char* *family*)
Set the family, for use in user interfaces (optional)

const char* `ColorSpace::`**`getEqualityGroup`**() `const`
Get the ColorSpace group name (used for equality comparisons) This allows no-op transforms between different colorspaces. If an equalityGroup is not defined (an empty string), it will be considered unique (i.e., it will not compare as equal to other ColorSpaces with an empty equality group). This is often, though not always, set to the same value as 'family'.

void `ColorSpace::`**`setEqualityGroup`**(const char* *equalityGroup*)

const char* `ColorSpace::`**`getDescription`**() `const`

void `ColorSpace::`**`setDescription`**(const char* *description*)

BitDepth `ColorSpace::`**`getBitDepth`**() `const`

void `ColorSpace::`**`setBitDepth`**(BitDepth *bitDepth*)

Data

ColorSpaces that are data are treated a bit special. Basically, any colorspace transforms you try to apply to them are ignored. (Think of applying a gamut mapping transform to an ID pass). Also, the `DisplayTransform` process obeys special 'data min' and 'data max' args.

This is traditionally used for pixel data that represents non-color pixel data, such as normals, point positions, ID information, etc.

bool `ColorSpace::isData() const`

void `ColorSpace::setIsData`(bool *isData*)

Allocation

If this colorspace needs to be transferred to a limited dynamic range coding space (such as during display with a GPU path), use this allocation to maximize bit efficiency.

Allocation `ColorSpace::getAllocation() const`

void `ColorSpace::setAllocation`(Allocation *allocation*)

Specify the optional variable values to configure the allocation. If no variables are specified, the defaults are used.

ALLOCATION_UNIFORM:

```
2 vars: [min, max]
```

ALLOCATION_LG2:

```
2 vars: [lg2min, lg2max]
3 vars: [lg2min, lg2max, linear_offset]
```

int `ColorSpace::getAllocationNumVars() const`

void `ColorSpace::getAllocationVars`(float* *vars*) `const`

void `ColorSpace::setAllocationVars`(int *numvars*, const float* *vars*)

Transform

ConstTransformRcPtr `ColorSpace::getTransform`(ColorSpaceDirection *dir*) `const`
: If a transform in the specified direction has been specified, return it. Otherwise return a null ConstTransformRcPtr

void `ColorSpace::setTransform`(const ConstTransformRcPtr& *transform*, ColorSpaceDirection *dir*)
: Specify the transform for the appropriate direction. Setting the transform to null will clear it.

Look

The *Look* is an 'artistic' image modification, in a specified image state. The processSpace defines the ColorSpace the image is required to be in, for the math to apply correctly.

class `Look`

static LookRcPtr `Look::Create()`

LookRcPtr `Look::createEditableCopy() const`

const char* Look::**getName**() const

void Look::**setName**(const char* *name*)

const char* Look::**getProcessSpace**() const

void Look::**setProcessSpace**(const char* *processSpace*)

ConstTransformRcPtr Look::**getTransform**() const

void Look::**setTransform**(const ConstTransformRcPtr& *transform*)
Setting a transform to a non-null call makes it allowed.

ConstTransformRcPtr Look::**getInverseTransform**() const

void Look::**setInverseTransform**(const ConstTransformRcPtr& *transform*)
Setting a transform to a non-null call makes it allowed.

Processor

class Processor

static ProcessorRcPtr Processor::**Create**()

bool Processor::**isNoOp**() const

bool Processor::**hasChannelCrosstalk**() const

does the processor represent an image transformation that introduces crosstalk between the image channels

ConstProcessorMetadataRcPtr Processor::**getMetadata**() const

CPU Path

void Processor::**apply**(ImageDesc& *img*) const
Apply to an image.

Apply to a single pixel.

Note: This is not as efficient as applying to an entire image at once. If you are processing multiple pixels, and have the flexibility, use the above function instead.

---

void Processor::**applyRGB**(float* *pixel*) const

void Processor::**applyRGBA**(float* *pixel*) const

const char* Processor::**getCpuCacheID**() const

GPU Path

Get the 3d lut + cg shader for the specified DisplayTransform.

cg signature will be:

```
shaderFcnName(in half4 inPixel, const uniform sampler3D lut3d)
```

lut3d should be size: 3 * edgeLen * edgeLen * edgeLen return 0 if unknown

const char* Processor::**getGpuShaderText**(const GpuShaderDesc& *shaderDesc*) const

const char* `Processor::`**`getGpuShaderTextCacheID`**(const GpuShaderDesc& *shaderDesc*) `const`

void `Processor::`**`getGpuLut3D`**(float* *lut3d*, const GpuShaderDesc& *shaderDesc*) `const`

const char* `Processor::`**`getGpuLut3DCacheID`**(const GpuShaderDesc& *shaderDesc*) `const`

class `ProcessorMetadata`
This class contains meta information about the process that generated this processor. The results of these functions do not impact the pixel processing.

static ProcessorMetadataRcPtr `ProcessorMetadata::`**`Create`**()

int `ProcessorMetadata::`**`getNumFiles`**() `const`

const char* `ProcessorMetadata::`**`getFile`**(int *index*) `const`

int `ProcessorMetadata::`**`getNumLooks`**() `const`

const char* `ProcessorMetadata::`**`getLook`**(int *index*) `const`

void `ProcessorMetadata::`**`addFile`**(const char* *fname*)

void `ProcessorMetadata::`**`addLook`**(const char* *look*)

Baker

In certain situations it is nessary to serilize transforms into a variety of application specific lut formats. The Baker can be used to create lut formats that ocio supports for writing.

Usage Example: *Bake a houdini sRGB viewer lut*

```
OCIO::ConstConfigRcPtr config = OCIO::Config::CreateFromEnv();
OCIO::BakerRcPtr baker = OCIO::Baker::Create();
baker->setConfig(config);
baker->setFormat("houdini"); // set the houdini type
baker->setType("3D"); // we want a 3D lut
baker->setInputSpace("lnf");
baker->setShaperSpace("log");
baker->setTargetSpace("sRGB");
std::ostringstream out;
baker->bake(out); // fresh bread anyone!
std::cout << out.str();
```

static BakerRcPtr **`Create`**()
create a new Baker

BakerRcPtr **`createEditableCopy`**() `const`
create a copy of this Baker

void **`setConfig`**(const ConstConfigRcPtr& *config*)
set the config to use

ConstConfigRcPtr **`getConfig`**() `const`
get the config to use

void **`setFormat`**(const char* *formatName*)
set the lut output format

const char* **`getFormat`**() `const`
get the lut output format

void **setType**(const char* *type*)
set the lut output type (1D or 3D)

const char* **getType**() const
get the lut output type

void **setMetadata**(const char* *metadata*)
set *optional* meta data for luts that support it

const char* **getMetadata**() const
get the meta data that has been set

void **setInputSpace**(const char* *inputSpace*)
set the input ColorSpace that the lut will be applied to

const char* **getInputSpace**() const
get the input ColorSpace that has been set

void **setShaperSpace**(const char* *shaperSpace*)
set an *optional* ColorSpace to be used to shape / transfer the input colorspace. This is mostly used to allocate an HDR luminance range into an LDR one. If a shaper space is not explicitly specified, and the file format supports one, the ColorSpace Allocation will be used

const char* **getShaperSpace**() const
get the shaper colorspace that has been set

void **setLooks**(const char* *looks*)
set the looks to be applied during baking Looks is a potentially comma (or colon) delimited list of lookNames, Where +/- prefixes are optionally allowed to denote forward/inverse look specification. (And forward is assumed in the absense of either)

const char* **getLooks**() const
get the looks to be applied during baking

void **setTargetSpace**(const char* *targetSpace*)
set the target device colorspace for the lut

const char* **getTargetSpace**() const
get the target colorspace that has been set

void **setShaperSize**(int *shapersize*)
override the default the shaper sample size, default: <format specific>

int **getShaperSize**() const
get the shaper sample size

void **setCubeSize**(int *cubesize*)
override the default cube sample size default: <format specific>

int **getCubeSize**() const
get the cube sample size

void **bake**(std::ostream& *os*) const
bake the lut into the output stream

static int **getNumFormats**()
get the number of lut writers

static const char* **getFormatNameByIndex**(int *index*)
get the lut writer at index, return empty string if an invalid index is specified

ImageDesc

const ptrdiff_t **AutoStride**
AutoStride

class ImageDesc
This is a light-weight wrapper around an image, that provides a context for pixel access. This does NOT claim ownership of the pixels or copy image data

ImageDesc::**~ImageDesc**()

PackedImageDesc

class PackedImageDesc

PackedImageDesc::**PackedImageDesc**(float* *data*, long *width*, long *height*, long *numChannels*, ptrdiff_t *chanStrideBytes=AutoStride*, ptrdiff_t *xStrideBytes=AutoStride*, ptrdiff_t *yStrideBytes=AutoStride*)
Pass the pointer to packed image data: rgbrgbrgb, etc. The number of channels must be greater than or equal to 3 If a 4th channel is specified, it is assumed to be alpha information. Channels > 4 will be ignored.

PackedImageDesc::**~PackedImageDesc**()

float* PackedImageDesc::**getData**() const

long PackedImageDesc::**getWidth**() const

long PackedImageDesc::**getHeight**() const

long PackedImageDesc::**getNumChannels**() const

ptrdiff_t PackedImageDesc::**getChanStrideBytes**() const

ptrdiff_t PackedImageDesc::**getXStrideBytes**() const

ptrdiff_t PackedImageDesc::**getYStrideBytes**() const

PlanarImageDesc

class PlanarImageDesc

PlanarImageDesc::**PlanarImageDesc**(float* *rData*, float* *gData*, float* *bData*, float* *aData*, long *width*, long *height*, ptrdiff_t *yStrideBytes=AutoStride*)
Pass the pointer to the specified image planes: rrrr gggg bbbb, etc. aData is optional, pass NULL if no alpha exists. {r,g,b} Data must be specified

PlanarImageDesc::**~PlanarImageDesc**()

float* PlanarImageDesc::**getRData**() const

float* PlanarImageDesc::**getGData**() const

float* PlanarImageDesc::**getBData**() const

float* PlanarImageDesc::**getAData**() const

long PlanarImageDesc::**getWidth**() const

long PlanarImageDesc::**getHeight**() const

ptrdiff_t PlanarImageDesc::**getYStrideBytes**() const

GpuShaderDesc

class GpuShaderDesc

GpuShaderDesc::**GpuShaderDesc**()

GpuShaderDesc::**~GpuShaderDesc**()

void GpuShaderDesc::**setLanguage**(GpuLanguage *lang*)

GpuLanguage GpuShaderDesc::**getLanguage**() const

void GpuShaderDesc::**setFunctionName**(const char* *name*)

const char* GpuShaderDesc::**getFunctionName**() const

void GpuShaderDesc::**setLut3DEdgeLen**(int *len*)

int GpuShaderDesc::**getLut3DEdgeLen**() const

const char* GpuShaderDesc::**getCacheID**() const

Context

class Context

static ContextRcPtr Context::**Create**()

ContextRcPtr Context::**createEditableCopy**() const

const char* Context::**getCacheID**() const

void Context::**setSearchPath**(const char* *path*)

const char* Context::**getSearchPath**() const

void Context::**setWorkingDir**(const char* *dirname*)

const char* Context::**getWorkingDir**() const

void Context::**setStringVar**(const char* *name*, const char* *value*)

const char* Context::**getStringVar**(const char* *name*) const

int Context::**getNumStringVars**() const

const char* Context::**getStringVarNameByIndex**(int *index*) const

void Context::**clearStringVars**()

void Context::**setEnvironmentMode**(EnvironmentMode *mode*)

EnvironmentMode Context::**getEnvironmentMode**() const

void Context::**loadEnvironment**()
Seed all string vars with the current environment.

const char* Context::**resolveStringVar**(const char* *val*) const
Do a file lookup.

Evaluate the specified variable (as needed). Will not throw exceptions.

const char* Context::**resolveFileLocation**(const char* *filename*) const
Do a file lookup.

Evaluate all variables (as needed). Also, walk the full search path until the file is found. If the filename cannot be found, an exception will be thrown.

3.5.8 C++ Transforms

Typically only needed when creating and/or manipulating configurations

---

class `Transform`

std::ostream& **`operator<<`**(*std::ostream&*, *const Transform&*)

---

class `AllocationTransform`

Forward direction wraps the 'expanded' range into the specified, often compressed, range.

static AllocationTransformRcPtr `AllocationTransform::`**`Create`**`()`

TransformRcPtr `AllocationTransform::`**`createEditableCopy`**`() const`

TransformDirection `AllocationTransform::`**`getDirection`**`() const`

void `AllocationTransform::`**`setDirection`**(TransformDirection *dir*)

Allocation `AllocationTransform::`**`getAllocation`**`() const`

void `AllocationTransform::`**`setAllocation`**(Allocation *allocation*)

int `AllocationTransform::`**`getNumVars`**`() const`

void `AllocationTransform::`**`getVars`**(float* *vars*) `const`

void `AllocationTransform::`**`setVars`**(int *numvars*, const float* *vars*)

std::ostream& **`operator<<`**(*std::ostream&*, *const AllocationTransform&*)

---

class `CDLTransform`

An implementation of the ASC CDL Transfer Functions and Interchange - Syntax (Based on the version 1.2 document)

Note: the clamping portion of the CDL is only applied if a non-identity power is specified.

---

static CDLTransformRcPtr `CDLTransform::`**`Create`**`()`

static CDLTransformRcPtr `CDLTransform::`**`CreateFromFile`**(const char* *src*, const char* *cccid*)

Load the CDL from the src .cc or .ccc file. If a .ccc is used, the cccid must also be specified src must be an absolute path reference, no relative directory or envvar resolution is performed.

TransformRcPtr `CDLTransform::`**`createEditableCopy`**`() const`

TransformDirection `CDLTransform::`**`getDirection`**`() const`

void `CDLTransform::`**`setDirection`**(TransformDirection *dir*)

bool `CDLTransform::`**`equals`**(const ConstCDLTransformRcPtr& *other*) `const`

const char* `CDLTransform::`**`getXML`**`() const`

void `CDLTransform::`**`setXML`**(const char* *xml*)

ASC_SOP

Slope, offset, power:

```
out = clamp( (in * slope) + offset ) ^ power
```

void CDLTransform::**setSlope**(const float* *rgb*)

void CDLTransform::**getSlope**(float* *rgb*) const

void CDLTransform::**setOffset**(const float* *rgb*)

void CDLTransform::**getOffset**(float* *rgb*) const

void CDLTransform::**setPower**(const float* *rgb*)

void CDLTransform::**getPower**(float* *rgb*) const

void CDLTransform::**setSOP**(const float* *vec9*)

void CDLTransform::**getSOP**(float* *vec9*) const

ASC_SAT

void CDLTransform::**setSat**(float *sat*)

float CDLTransform::**getSat**() const

void CDLTransform::**getSatLumaCoefs**(float* *rgb*) const
These are hard-coded, by spec, to r709

Metadata

These do not affect the image processing, but are often useful for pipeline purposes and are included in the serialization.

void CDLTransform::**setID**(const char* *id*)
Unique Identifier for this correction

const char* CDLTransform::**getID**() const

void CDLTransform::**setDescription**(const char* *desc*)
Textual description of color correction (stored on the SOP)

const char* CDLTransform::**getDescription**() const

std::ostream& **operator<<**(*std::ostream&*, *const CDLTransform&*)

---

class ColorSpaceTransform

static ColorSpaceTransformRcPtr ColorSpaceTransform::**Create**()

TransformRcPtr ColorSpaceTransform::**createEditableCopy**() const

TransformDirection ColorSpaceTransform::**getDirection**() const

void ColorSpaceTransform::**setDirection**(TransformDirection *dir*)

const char* ColorSpaceTransform::**getSrc**() const

void ColorSpaceTransform::**setSrc**(const char* *src*)

const char* ColorSpaceTransform::**getDst**() const

void ColorSpaceTransform::**setDst**(const char* *dst*)

std::ostream& **operator<<**(*std::ostream&*, *const ColorSpaceTransform&*)

---

class DisplayTransform

static DisplayTransformRcPtr `DisplayTransform::`**`Create`**`()`

TransformRcPtr `DisplayTransform::`**`createEditableCopy`**`() const`

TransformDirection `DisplayTransform::`**`getDirection`**`() const`

void `DisplayTransform::`**`setDirection`**`(`TransformDirection *dir*`)`

void `DisplayTransform::`**`setInputColorSpaceName`**`(`const char* *name*`)`
Step 0. Specify the incoming color space

const char* `DisplayTransform::`**`getInputColorSpaceName`**`() const`

void `DisplayTransform::`**`setLinearCC`**`(`const ConstTransformRcPtr& *cc*`)`
Step 1: Apply a Color Correction, in ROLE_SCENE_LINEAR

ConstTransformRcPtr `DisplayTransform::`**`getLinearCC`**`() const`

void `DisplayTransform::`**`setColorTimingCC`**`(`const ConstTransformRcPtr& *cc*`)`
Step 2: Apply a color correction, in ROLE_COLOR_TIMING

ConstTransformRcPtr `DisplayTransform::`**`getColorTimingCC`**`() const`

void `DisplayTransform::`**`setChannelView`**`(`const ConstTransformRcPtr& *transform*`)`
Step 3: Apply the Channel Viewing Swizzle (mtx)

ConstTransformRcPtr `DisplayTransform::`**`getChannelView`**`() const`

void `DisplayTransform::`**`setDisplay`**`(`const char* *display*`)`
Step 4: Apply the output display transform This is controlled by the specification of (display, view)

const char* `DisplayTransform::`**`getDisplay`**`() const`

void `DisplayTransform::`**`setView`**`(`const char* *view*`)`
Specify which view transform to use

const char* `DisplayTransform::`**`getView`**`() const`

void `DisplayTransform::`**`setDisplayCC`**`(`const ConstTransformRcPtr& *cc*`)`
Step 5: Apply a post display transform color correction

ConstTransformRcPtr `DisplayTransform::`**`getDisplayCC`**`() const`

void `DisplayTransform::`**`setLooksOverride`**`(`const char* *looks*`)`
A user can optionally override the looks that are, by default, used with the expected display / view combination. A common use case for this functionality is in an image viewing app, where per-shot looks are supported. If for some reason a per-shot look is not defined for the current Context, the Config::getProcessor fcn will not succeed by default. Thus, with this mechanism the viewing app could override to looks = "", and this will allow image display to continue (though hopefully) the interface would reflect this fallback option.)

Looks is a potentially comma (or colon) delimited list of lookNames, Where +/- prefixes are optionally allowed to denote forward/inverse look specification. (And forward is assumed in the absense of either)

const char* `DisplayTransform::`**`getLooksOverride`**`() const`

void `DisplayTransform::`**`setLooksOverrideEnabled`**`(`bool *enabled*`)`
Specifiy whether the lookOverride should be used, or not. This is a speparate flag, as it's often useful to override "looks" to an empty string

bool `DisplayTransform::`**`getLooksOverrideEnabled`**`() const`

std::ostream& **`operator<<`**`(`*std::ostream&*, *const DisplayTransform&*`)`

---

class ExponentTransform

Represents exponent transform: pow(clamp(color), value)

If the exponent is 1.0, this will not clamp. Otherwise, the input color will be clamped between [0.0, inf]

static ExponentTransformRcPtr `ExponentTransform::`**Create**`()`

TransformRcPtr `ExponentTransform::`**createEditableCopy**`() const`

TransformDirection `ExponentTransform::`**getDirection**`() const`

void `ExponentTransform::`**setDirection**(TransformDirection *dir*)

void `ExponentTransform::`**setValue**(const float* *vec4*)

void `ExponentTransform::`**getValue**(float* *vec4*) `const`

std::ostream& **operator<<**(*std::ostream&*, *const ExponentTransform&*)

---

class FileTransform

static FileTransformRcPtr `FileTransform::`**Create**`()`

TransformRcPtr `FileTransform::`**createEditableCopy**`() const`

TransformDirection `FileTransform::`**getDirection**`() const`

void `FileTransform::`**setDirection**(TransformDirection *dir*)

const char* `FileTransform::`**getSrc**`() const`

void `FileTransform::`**setSrc**(const char* *src*)

const char* `FileTransform::`**getCCCId**`() const`

void `FileTransform::`**setCCCId**(const char* *id*)

Interpolation `FileTransform::`**getInterpolation**`() const`

void `FileTransform::`**setInterpolation**(Interpolation *interp*)

static int `FileTransform::`**getNumFormats**`()`
get the number of lut readers

static const char* `FileTransform::`**getFormatNameByIndex**(int *index*)
get the lut readers at index, return empty string if an invalid index is specified

static const char* `FileTransform::`**getFormatExtensionByIndex**(int *index*)
get the lut reader extension at index, return empty string if an invalid index is specified

std::ostream& **operator<<**(*std::ostream&*, *const FileTransform&*)

---

class GroupTransform

static GroupTransformRcPtr `GroupTransform::`**Create**`()`

TransformRcPtr `GroupTransform::`**createEditableCopy**`() const`

TransformDirection `GroupTransform::`**getDirection**`() const`

void `GroupTransform::`**setDirection**(TransformDirection *dir*)

ConstTransformRcPtr `GroupTransform::`**getTransform**(int *index*) `const`

int `GroupTransform::`**size**`() const`

void GroupTransform::**push_back**(const ConstTransformRcPtr& *transform*)

void GroupTransform::**clear**()

bool GroupTransform::**empty**() const

std::ostream& **operator<<**(*std::ostream&*, *const GroupTransform&*)

---

class LogTransform

Represents log transform: log(color, base)

- The input will be clamped for negative numbers.
- Default base is 2.0
- Only the rgb channels are affected

static LogTransformRcPtr LogTransform::**Create**()

TransformRcPtr LogTransform::**createEditableCopy**() const

TransformDirection LogTransform::**getDirection**() const

void LogTransform::**setDirection**(TransformDirection *dir*)

void LogTransform::**setBase**(float *val*)

float LogTransform::**getBase**() const

std::ostream& **operator<<**(*std::ostream&*, *const LogTransform&*)

---

class LookTransform

static LookTransformRcPtr LookTransform::**Create**()

TransformRcPtr LookTransform::**createEditableCopy**() const

TransformDirection LookTransform::**getDirection**() const

void LookTransform::**setDirection**(TransformDirection *dir*)

const char* LookTransform::**getSrc**() const

void LookTransform::**setSrc**(const char* *src*)

const char* LookTransform::**getDst**() const

void LookTransform::**setDst**(const char* *dst*)

void LookTransform::**setLooks**(const char* *looks*)

Specify looks to apply. Looks is a potentially comma (or colon) delimited list of look names, Where +/- prefixes are optionally allowed to denote forward/inverse look specification. (And forward is assumed in the absense of either)

const char* LookTransform::**getLooks**() const

std::ostream& **operator<<**(*std::ostream&*, *const LookTransform&*)

---

class MatrixTransform

Represents an MX+B Matrix transform

static MatrixTransformRcPtr MatrixTransform::**Create**()

TransformRcPtr `MatrixTransform::`**`createEditableCopy`**`() const`

TransformDirection `MatrixTransform::`**`getDirection`**`() const`

void `MatrixTransform::`**`setDirection`**(TransformDirection *dir*)

bool `MatrixTransform::`**`equals`**(const MatrixTransform& *other*) `const`

void `MatrixTransform::`**`setValue`**(const float* *m44*, const float* *offset4*)

void `MatrixTransform::`**`getValue`**(float* *m44*, float* *offset4*) `const`

void `MatrixTransform::`**`setMatrix`**(const float* *m44*)

void `MatrixTransform::`**`getMatrix`**(float* *m44*) `const`

void `MatrixTransform::`**`setOffset`**(const float* *offset4*)

void `MatrixTransform::`**`getOffset`**(float* *offset4*) `const`

Convenience functions

to get the mtx and offset corresponding to higher-level concepts

---

Note: These can throw an exception if for any component `oldmin == oldmax.  (divide by 0)`

---

static void `MatrixTransform::`**`Fit`**(float* *m44*, float* *offset4*, const float* *oldmin4*, const float* *oldmax4*, const float* *newmin4*, const float* *newmax4*)

static void `MatrixTransform::`**`Identity`**(float* *m44*, float* *offset4*)

static void `MatrixTransform::`**`Sat`**(float* *m44*, float* *offset4*, float *sat*, const float* *lumaCoef3*)

static void `MatrixTransform::`**`Scale`**(float* *m44*, float* *offset4*, const float* *scale4*)

static void `MatrixTransform::`**`View`**(float* *m44*, float* *offset4*, int* *channelHot4*, const float* *lumaCoef3*)

std::ostream& **`operator<<`**(*std::ostream&*, *const MatrixTransform&*)

---

class `TruelightTransform`

Truelight transform using its API

static TruelightTransformRcPtr `TruelightTransform::`**`Create`**`()`

TransformRcPtr `TruelightTransform::`**`createEditableCopy`**`() const`

TransformDirection `TruelightTransform::`**`getDirection`**`() const`

void `TruelightTransform::`**`setDirection`**(TransformDirection *dir*)

void `TruelightTransform::`**`setConfigRoot`**(const char* *configroot*)

const char* `TruelightTransform::`**`getConfigRoot`**`() const`

void `TruelightTransform::`**`setProfile`**(const char* *profile*)

const char* `TruelightTransform::`**`getProfile`**`() const`

void `TruelightTransform::`**`setCamera`**(const char* *camera*)

const char* `TruelightTransform::`**`getCamera`**`() const`

void `TruelightTransform::`**`setInputDisplay`**(const char* *display*)

const char* `TruelightTransform::`**`getInputDisplay`**`() const`

void `TruelightTransform::`**`setRecorder`**(const char* *recorder*)

const char* TruelightTransform::**getRecorder**() const

void TruelightTransform::**setPrint**(const char* *print*)

const char* TruelightTransform::**getPrint**() const

void TruelightTransform::**setLamp**(const char* *lamp*)

const char* TruelightTransform::**getLamp**() const

void TruelightTransform::**setOutputCamera**(const char* *camera*)

const char* TruelightTransform::**getOutputCamera**() const

void TruelightTransform::**setDisplay**(const char* *display*)

const char* TruelightTransform::**getDisplay**() const

void TruelightTransform::**setCubeInput**(const char* *type*)

const char* TruelightTransform::**getCubeInput**() const

std::ostream& **operator<<**(*std::ostream&*, *const TruelightTransform&*)

3.5.9 C++ Types

Core

type ConstConfigRcPtr

type ConfigRcPtr

type ConstColorSpaceRcPtr

type ColorSpaceRcPtr

type ConstLookRcPtr

type LookRcPtr

type ConstContextRcPtr

type ContextRcPtr

type ConstProcessorRcPtr

type ProcessorRcPtr

type ConstProcessorMetadataRcPtr

type ProcessorMetadataRcPtr

type ConstBakerRcPtr

type BakerRcPtr

Transforms

type ConstTransformRcPtr

type TransformRcPtr

type ConstAllocationTransformRcPtr

type AllocationTransformRcPtr

type ConstCDLTransformRcPtr

type CDLTransformRcPtr

type ConstColorSpaceTransformRcPtr

type ColorSpaceTransformRcPtr

type ConstDisplayTransformRcPtr

type DisplayTransformRcPtr

type ConstExponentTransformRcPtr

type ExponentTransformRcPtr

type ConstFileTransformRcPtr

type FileTransformRcPtr

type ConstGroupTransformRcPtr

type GroupTransformRcPtr

type ConstLogTransformRcPtr

type LogTransformRcPtr

type ConstLookTransformRcPtr

type LookTransformRcPtr

type ConstMatrixTransformRcPtr

type MatrixTransformRcPtr

type ConstTruelightTransformRcPtr

type TruelightTransformRcPtr

Enums

type ColorSpaceDirection

type TransformDirection

type Interpolation
Specify the interpolation type to use If the specified interpolation type is not supported in the requested context (for example, using tetrahedral interpolationon 1D luts) an exception will be throw.

INTERP_BEST will choose the best interpolation type for the requested context:

Lut1D INTERP_BEST: LINEAR Lut3D INTERP_BEST: LINEAR

Note: INTERP_BEST is subject to change in minor releases, so if you care about locking off on a specific interpolation type, we'd recommend directly specifying it.

type BitDepth

type Allocation

type GpuLanguage
Used when there is a choice of hardware shader language.

type EnvironmentMode

Conversion

const char* **BoolToString** (bool *val*)

bool **BoolFromString** (const char* *s*)

const char* **LoggingLevelToString** (LoggingLevel *level*)

LoggingLevel **LoggingLevelFromString** (const char* *s*)

const char* **TransformDirectionToString** (TransformDirection *dir*)

TransformDirection **TransformDirectionFromString** (const char* *s*)

TransformDirection **GetInverseTransformDirection** (TransformDirection *dir*)

TransformDirection **CombineTransformDirections** (TransformDirection *d1*, TransformDirection *d2*)

const char* **ColorSpaceDirectionToString** (ColorSpaceDirection *dir*)

ColorSpaceDirection **ColorSpaceDirectionFromString** (const char* *s*)

const char* **BitDepthToString** (BitDepth *bitDepth*)

BitDepth **BitDepthFromString** (const char* *s*)

bool **BitDepthIsFloat** (BitDepth *bitDepth*)

int **BitDepthToInt** (BitDepth *bitDepth*)

const char* **AllocationToString** (Allocation *allocation*)

Allocation **AllocationFromString** (const char* *s*)

const char* **InterpolationToString** (Interpolation *interp*)

Interpolation **InterpolationFromString** (const char* *s*)

const char* **GpuLanguageToString** (GpuLanguage *language*)

GpuLanguage **GpuLanguageFromString** (const char* *s*)

const char* **EnvironmentModeToString** (EnvironmentMode *mode*)

EnvironmentMode **EnvironmentModeFromString** (const char* *s*)

Roles

ColorSpace Roles are used so that plugins, in addition to this API can have abstract ways of asking for common colorspaces, without referring to them by hardcoded names.

Internal:

```
GetGPUDisplayTransform - (ROLE_SCENE_LINEAR (fstop exposure))
                         (ROLE_COLOR_TIMING (ASCColorCorrection))
```

External Plugins (currently known):

```
Colorpicker UIs       - (ROLE_COLOR_PICKING)
Compositor LogConvert - (ROLE_SCENE_LINEAR, ROLE_COMPOSITING_LOG)
```

const char* **ROLE_DEFAULT**
: "default"

const char* **ROLE_REFERENCE**
: "reference"

const char* **ROLE_DATA**
"data"

const char* **ROLE_COLOR_PICKING**
"color_picking"

const char* **ROLE_SCENE_LINEAR**
"scene_linear"

const char* **ROLE_COMPOSITING_LOG**
"compositing_log"

const char* **ROLE_COLOR_TIMING**
"color_timing"

const char* **ROLE_TEXTURE_PAINT**
This role defines the transform for painting textures. In some workflows this is just a inverse display gamma with some limits

const char* **ROLE_MATTE_PAINT**
This role defines the transform for matte painting. In some workflows this is a 1D HDR to LDR allocation. It is normally combined with another display transform in the host app for preview.

Python API documentation:

3.5.10 Python API

Description

A color configuration (`PyOpenColorIO.Config`) defines all the color spaces to be available at runtime.

(`PyOpenColorIO.Config`) is the main object for interacting with this library. It encapsulates all the information necessary to use customized `PyOpenColorIO.ColorSpaceTransform` and `PyOpenColorIO.DisplayTransform` operations.

See the *User Guide* for more information on selecting, creating, and working with custom color configurations.

For applications interested in using only one color configuration at a time (this is the vast majority of apps), their API would traditionally get the global configuration and use that, as opposed to creating a new one. This simplifies the use case for plugins and bindings, as it alleviates the need to pass around configuration handles.

An example of an application where this would not be sufficient would be a multi-threaded image proxy server (daemon) that wants to handle multiple show configurations concurrently in a single process. This app would need to keep multiple configurations alive, and manage them appropriately.

Roughly speaking, a novice user should select a default configuration that most closely approximates the use case (animation, visual effects, etc.), and set the `OCIO` environment variable to point at the root of that configuration.

Note: Initialization using environment variables is typically preferable in a multi-app ecosystem, as it allows all applications to be consistently configured.

Note: Paths to LUTs can be relative. The search paths are defined in `PyOpenColorIO.Config`.

See *Usage Examples*

Examples of Use

```
import PyOpenColorIO as OCIO

# Load an existing configuration from the environment.
# The resulting configuration is read-only. If $OCIO is set, it will use that.
# Otherwise it will use an internal default.
config = OCIO.GetCurrentConfig()

# What color spaces exist?
colorSpaceNames = [ cs.getName() for cs in config.getColorSpaces() ]

# Given a string, can we parse a color space name from it?
inputString = 'myname_linear.exr'
colorSpaceName = config.parseColorSpaceFromString(inputString)
if colorSpaceName:
    print 'Found color space', colorSpaceName
else:
    print 'Could not get color space from string', inputString

# What is the name of scene-linear in the configuration?
colorSpace = config.getColorSpace(OCIO.Constants.ROLE_SCENE_LINEAR)
if colorSpace:
    print colorSpace.getName()
else:
    print 'The role of scene-linear is not defined in the configuration'

# For examples of how to actually perform the color transform math,
# see 'Python: Processor' docs.

# Create a new, empty, editable configuration
config = OCIO.Config()

# Create a new color space, and add it
cs = OCIO.ColorSpace(...)
# (See ColorSpace for details)
config.addColorSpace(cs)

# For additional examples of config manipulation, see
# https://github.com/imageworks/OpenColorIO-Configs/blob/master/nuke-default/make.py
```

Exceptions

Global

Config

ColorSpace

Look

Processor

Context

3.5.11 Python Transforms

Transform

AllocationTransform

```
import PyOpenColorIO as OCIO
transform = OCIO.AllocationTransform()
transform.setAllocation(OCIO.Constants.ALLOCATION)
```

CDLTransform

```
import PyOpenColorIO as OCIO

cdl = OCIO.CDLTransform()
# Set the slope, offset, power, and saturation for each channel.
cdl.setSOP([, , , , , , , , ])
cdl.setSat([, , ])
cdl.getSatLumaCoefs()
```

ColorSpaceTransform

This class is meant so that ColorSpace conversions can be reused, referencing ColorSpaces that already exist.

Note: Careless use of this may create infinite loops, so avoid referencing the colorspace you're in.

```
import PyOpenColorIO as OCIO
transform = OCIO.ColorSpaceTransform()
```

DisplayTransform

```
import PyOpenColorIO as OCIO
transform = OCIO.DisplayTransform()
```

ExponentTransform

```
import PyOpenColorIO as OCIO
transform = OCIO.ExponentTransform()
```

FileTransform

GroupTransform

LogTransform

```
import PyOpenColorIO as OCIO
```

`PyOpenColorIO.LogTransform` is used to define a log transform. The direction of the transform and its numerical base can be specified.

LookTransform

MatrixTransform

3.5.12 Python Types

Constants

Internal Architecture:

3.5.13 Internal Architecture Overview

External API

Configs

At the highest level, we have OCIO::Configs. This represents the entirety of the current color "universe". Configs are serialized as .ocio files, read at runtime, and are often used in a 'read-only' context.

Config are loaded at runtime to allow for customized color handling in a show- dependent manner.

Example Configs:

- ACES (Acacdemy's standard color workflow)
- spi-vfx (Used on some Imageworks VFX shows such as spiderman, etc).
- and others

ColorSpaces

The meat of an OCIO::Config is a list of named ColorSpaces. ColorSpace often correspond to input image states, output image states, or image states used for internal processing.

Example ColorSpaces (from ACES configuration):

- aces (HDR, scene-linear)

- adx10 (log-like density encoding space)
- slogf35 (sony F35 slog camera encoding)
- rrt_srgb (baked in display transform, suitable for srgb display)
- rrt_p3dci (baked in display transform, suitable for dcip3 display)

Transforms

ColorSpaces contain an ordered list of transforms, which define the conversion to and from the Config's "reference" space.

Transforms are the atomic units available to the designer in order to specify a color conversion.

Examples of OCIO::Transforms are:

- File-based transforms (1d lut, 3d lut, mtx... anything, really.)
- Math functions (gamma, log, mtx)
- The 'meta' GroupTransform, which contains itself an ordered lists of transforms
- The 'meta' LookTransform, which contains an ordered lists of transforms

For example, the adx10 ColorSpace (in one particular ACES configuration) -Transform FROM adx, to our reference ColorSpace:

1. Apply FileTransform adx_adx10_to_cdd.spimtx
2. Apply FileTransform adx_cdd_to_cid.spimtx
3. Apply FileTransform adx_cid_to_rle.spi1d
4. Apply LogTransform base 10 (inverse)
5. Apply FileTransform adx_exp_to_aces.spimtx

If we have an image in the reference ColorSpace (unnamed), we can convert TO adx by applying each in the inverse direction:

1. Apply FileTransform adx_exp_to_aces.spimtx (inverse)
2. Apply LogTransform base 10 (forward)
3. Apply FileTransform adx_cid_to_rle.spi1d (inverse)
4. Apply FileTransform adx_cdd_to_cid.spimtx (inverse)
5. Apply FileTransform adx_adx10_to_cdd.spimtx (inverse)

Note that this isn't possible in all cases (what if a lut or matrix is not invertible?), but conceptually it's a simple way to think about the design.

Summary

Configs and ColorSpaces are just a bookkeeping device used to get and ordered lists of Transforms corresponding to image color transformation.

Transforms are visible to the person AUTHORING the OCIO config, but are NOT visible to the client applications. Client apps need only concern themselves with Configs and Processors.

OCIO::Processors

A processor corresponds to a 'baked' color transformation. You specify two arguments when querying a processor: the *ColorSpace* you are coming from, and the *ColorSpace* you are going to.

Once you have the processor, you can apply the color transformation using the "apply" function. For the CPU veseion, first wrap your image in an ImageDesc class, and then call apply to process in place.

Example:

```
#include <OpenColorIO/OpenColorIO.h>
namespace OCIO = OCIO_NAMESPACE;

try
{
   // Get the global OpenColorIO config
   // This will auto-initialize (using $OCIO) on first use
   OCIO::ConstConfigRcPtr config = OCIO::GetCurrentConfig();

   // Get the processor corresponding to this transform.
   // These strings, in this example, are specific to the above
   // example. ColorSpace names should NEVER be hard-coded into client
   // software, but should be dynamically queried at runtime from the library
   OCIO::ConstProcessorRcPtr processor = config->getProcessor("adx10", "aces");

   // Wrap the image in a light-weight ImageDescription
   OCIO::PackedImageDesc img(imageData, w, h, 4);

   // Apply the color transformation (in place)
   processor->apply(img);
}
catch(OCIO::Exception & exception)
{
   std::cerr << "OpenColorIO Error: " << exception.what() << std::endl;
}
```

The GPU code path is similar. You get the processor from the config, and then query the shaderText and the lut3d. The client loads these to the GPU themselves, and then makes the appropriate calls to the newly defined function.

See *src/apps/ociodisplay* for an example.

Internal API

The Op Abstraction

It is a useful abstraction, both for code-reuse and optimization, to not relying on the transforms to do pixel processing themselves.

Consider that the FileTransform represents a wide-range of image processing operations (basically all of em), many of which are really complex. For example, the houdini lut format in a single file may contain a log convert, a 1d lut, and then a 3d lut; all of which need to be applied in a row! If we dont want the FileTransform to know how to process all possible pixel operations, it's much simpler to make light-weight processing operations, which the transforms can create to do the dirty work as needed.

All image processing operations (ops) are a class that present the same interface, and it's rather simple:

```
virtual void apply(float* rgbaBuffer, long numPixels)
```

Basically, given a packed float array with the specified number of pixels, process em.

Examples of ops include Lut1DOp, Lut3DOp, MtxOffsetOp, LogOp, etc.

Thus, the job of a transform becomes much simpler and they're only responsible for converting themselves to a list of ops. A simple FileTransform that only has a single 1D lut internally may just generate a single Lut1DOp, but a FileTransform that references a more complex format (such as the houdini lut case referenced above) may generate a few ops:

```
void FileFormatHDL::BuildFileOps(OpRcPtrVec & ops,
                         const Config& /*config*/,
                         const ConstContextRcPtr & /*context*/,
                         CachedFileRcPtr untypedCachedFile,
                         const FileTransform& fileTransform,
                         TransformDirection dir) const {

// Code omitted which loads the lut file into the file cache...

CreateLut1DOp(ops, cachedFile->lut1D,
                   fileTransform.getInterpolation(), dir);
CreateLut3DOp(ops, cachedFile->lut3D,
                   fileTransform.getInterpolation(), dir);
```

See (`src/core/*Ops.h`) for the available ops.

Note that while compositors often have complex, branching trees of image processing operations, we just have a linear list of ops, lending itself very well to optimization.

Before the ops are run, they are optimized. (Collapsed with appropriate neighbors, etc).

An Example

Let us consider the internal steps when getProcessor() is called to convert from ColorSpace 'adx10' to ColorSpace 'aces':

- The first step is to turn this ColorSpace conversion into an ordered list of transforms.

We do this by creating a single of the conversions from 'adx10' to reference, and then adding the transforms required to go from reference to 'aces'. * The Transform list is then converted into a list of ops. It is during this stage luts, are loaded, etc.

CPU CODE PATH

The master list of ops is then optimized, and stored internally in the processor.

```
FinalizeOpVec(m_cpuOps);
```

During Processor::apply(...), a subunit of pixels in the image are formatted into a sequential rgba block. (Block size is optimized for computational (SSE) simplicity and performance, and is typically similar in size to an image scanline)

```
float * rgbaBuffer = 0;
long numPixels = 0;
while(true) {
   scanlineHelper.prepRGBAScanline(&rgbaBuffer, &numPixels);
   ...
```

Then for each op, op->apply is called in-place.

```
for(OpRcPtrVec::size_type i=0, size = m_cpuOps.size(); i<size; ++i)
{
   m_cpuOps[i]->apply(rgbaBuffer, numPixels);
}
```

After all ops have been applied, the results are copied back to the source

```
scanlineHelper.finishRGBAScanline();
```

GPU CODE PATH

1. The master list of ops is partitioned into 3 ordered lists:

- As many ops as possible from the BEGINNING of the op-list that can be done analytically in shader text. (called gpu-preops)
- As many ops as possible from the END of the op-list that can be done analytically in shader text. (called gpu-postops)
- The left-over ops in the middle that cannot support shader text, and thus will be baked into a 3dlut. (called gpu-lattice)

#. Between the first an the second lists (gpu-preops, and gpu-latticeops), we anaylze the op-stream metadata and determine the appropriate allocation to use. (to minimize clamping, quantization, etc). This is accounted for here by interserting a forward allocation to the end of the pre-ops, and the inverse allocation to the start of the lattice ops.

See https://github.com/imageworks/OpenColorIO/blob/master/src/core/NoOps.cpp#L183

#. The 3 lists of ops are then optimized individually, and stored on the processor. The Lut3d is computed by applying the gpu-lattice ops, on the CPU, to a lut3d image.

The shader text is computed by calculating the shader for the gpu-preops, adding a sampling function of the 3d lut, and then calculating the shader for the gpu post ops.

3.6 FAQ

3.6.1 License?

New BSD.

You are welcome to include the OpenColorIO in commercial, or open source applications. See the *License* for further details.

3.6.2 Terminology

- Transform - a function that alters RGB(A) data (e.g transform an image from scene linear to sRGB)
- Reference space - a space that connects colorspaces
- Colorspace - a meaningful space that can be transferred to and from the reference space
- Display - a virtual or physical display device (e.g an sRGB display device)
- View - a meaningful view of the reference space on a Display (e.g a film emulation view on an sRGB display device)

- Role - abstract colorspace naming (e.g specify the "lnh" colorspace as the scene_linear role, or the color-picker UI uses color_picking role)
- Look - a color transform which applies a creative look (for example a per-shot netural grade to remove color-casts from a sequence of film scans, or a DI look)

3.6.3 What LUT Formats are supported?

Ext	Details	Notes
3dl	Autodesk Apps: Lustre, Flame, etc. Supports shaper LUT + 3D	Read + Write Support.
ccc	ASC CDL ColorCorrectionCollection	Full read support.
cc	ASC CDL ColorCorrection	Full read support.
csp	Cinespace (Rising Sun Research) LUT. Spline-based shaper LUT, with either 1D or 3D LUT.	Read + Write Support. Shaper is resampled into simple 1D LUT with 2^16 samples.
cub	Truelight format. Shaper Lut + 3D	Full read support.
cube	Iridas format. Either 1D or 3D Lut.	Full read support
hdl	Houdini. 1D Lut, 3D lut, 1D shaper Lut	Only 'C' type is supported. Need to add R G B A RGB RGBA ALL. No support for Sampling tag. Header fields must be in strict order.
look	IRIDAS .look	Read baked 3D LUT embedded in file. No mask support.
mga/m3d	Pandora 3D lut	Full read support.
spi1d	1D format. Imageworks native 1D lut format. HDR friendly, supports arbitrary input and output domains	Full read support.
spi3d	3D format. Imageworks native 3D lut format.	Full read support.
spimtx	3x3 matrix + color offset. Imageworks native color matrix format	Full read support.
vf	Inventor 3d lut.	Read support for 3d lut data and global_transform element

Note: Shaper LUT application in OCIO currently only supports linear interpolation. For very small shaper LUT sizes this may not be sufficient. (CSP shaper luts excluded; they do use spline interpolation at load-time).

3.6.4 Can you query a color space by name (like "Rec709") and get back XYZ coordinates of its primaries and whitepoint?

Not currently.

OCIO is a color configuration 'playback' tool that tries to be as flexible as possible; color information such as this is often only needed / relevant at configuration authoring time. Making primaries / whitepoint required would limit the pipeline OCIO could service. In the strictest sense, we would consider OCIO to be a 'baked' representation of color processes, similar to how Alembic files do not store animation rig data, but rather only the baked geometry.

Also, remember that not all colorspaces using in visual effects even have strongly defined color space definitions. For example, scanned film negatives, when linearized with 1d transfer curves (the historical norm in vfx), do not have defined primaries/white point. Each rgb value could of course individually be tied to a specific color, but if you were to do a sweep of the pure 'red channel', for example, you'd find that it creates a curves in chromaticity space, not a single point. (This is due to the 1d linearization not attempting to undo the subtractive processes that created the pixels in the first place.

But many color spaces in OCIO *do* have very strongly defined white points/chromaticities. On the display side, for example, we have very precise information on this.

Perhaps OCIO should include optional metadata to tag outputs? We are looking at this as as a OCIO 1.2 feature.

3.6.5 Can you convert XYZ <-> named color space RGB values?

OCIO includes a MatrixTransform, so the processing capability is there. But there is no convenience function to compute this matrix for you. (We do include other Matrix convenience functions though, so it already has a place to be added. See MatrixTransform in export/OpenColorTransforms.h)

There's talk of extended OCIO 1.2 to have a plugin api where colorspaces could be dynamically added at runtime (such as after reading exr chromaticity header metadata). This would necessitate adding such a feature.

3.6.6 What are the differences between Nuke's Vectorfield and OCIOFileTransform?

(All tests done with Nuke 6.3)

Ext	Details	Notes
3dl	Matched Results	
ccc	n/a	
cc	n/a	
csp	*Difference*	Gain error. Believe OCIO is correct, but need to verify.
cub	Matched Results	Note: Nuke's .cub exporter is broken (gain error)
cube	Matched Results	
hdl	n/a	
mga/m3d	n/a	
spi1d	n/a	
spi3d	n/a	
spimtx	n/a	
vf	*Difference*	Gain error. Believe OCIO is correct, but need to verify.

All gain differences are due to a common 'gotcha' when interpolating 3d luts, related to internal index computation. If you have a 32x32x32 3dlut, when sampling values from (0,1) do you internally scale by 31.0 or 32.0? This is typically well-defined for each format, (in this case the answer is usually 31.0) but when incorrectly handled in an application, you occationally see gain errors that differ by this amount. (In the case of a 32-sized 3dlut, 32/31 = ~3% error)

3.6.7 What do ColorSpace::setAllocation() and ColorSpace::setAllocationVars() do?

These hints only come into play during GPU processing, and are used to determine proper colorspace allocation handling for 3D LUTs. See this page *How to Configure ColorSpace Allocation* for further information.

3.7 Downloads

- Sample OCIO Configurations – .zip .tar.gz
- Reference Images v1.0v4 – .tgz
- Core Library v1.0.9 – .zip .tar.gz
- Core Library latest – .zip .tar.gz

Per-version updates: *ChangeLog*.

Build instructions: *Building from source*.

3.7.1 Contributor License Agreements

Please see the Imageworks Open Source website

3.7.2 Deprecated Downloads

- Reference Images v1.0v2 tgz
- Reference Images v1.0v1 tgz
- Core Library v1.0.8 – .zip .tar.gz
- Core Library v1.0.7 – .zip .tar.gz
- Core Library v1.0.6 – .zip .tar.gz
- Core Library v1.0.5 – .zip .tar.gz
- Core Library v1.0.4 – .zip .tar.gz
- Core Library v1.0.3 – .zip .tar.gz
- Core Library v1.0.2 – .zip .tar.gz
- Core Library v1.0.1 – .zip .tar.gz
- Core Library v1.0.0 – .zip .tar.gz
- Color Config v0.7v4 tgz (OCIO v0.7.6+)
- Core Library v0.8.7 – .zip .tar.gz
- Core Library v0.7.9 – .zip .tar.gz
- Core Library v0.6.1 – .zip .tar.gz
- Core Library v0.5.16 – .zip .tar.gz
- Core Library v0.5.8 – .zip .tar.gz

3.8 ChangeLog

Version 1.0.9 (Sep 2 2013):

- CDL cccid supports both named id and index lookups
- ociobakelut / ocioconvert updates
- FreeBSD compile dixes
- FileTransform disk cache allows concurrent disk lookups
- CSP windows fix
- Python 3 support
- Fix envvar abs/relative path testing
- Can explicitly declare config envvars
- gcc44 compile warning fixes

Version 1.0.8 (Dec 11 2012):

- After Effects plugin

- Core increased precision for matrix inversion
- Core md5 symbols no longer leaked
- CMake compatibility with OIIO 1.0 namespacing
- Cmake option to control python soname
- Nuke register_viewers defaults OCIODisplay to "all"
- Nuke ColorLookup <-> spi1d lut examples
- Windows uses boost shared_ptr by default
- Windows fixed csp writing
- Windows build fixes
- ociobakelut supports looks

Version 1.0.7 (April 17 2012):

- IRIDAS .look support
- ociolutimage utility added (handles image <-> 3dlut)
- CMake build allows optional reliance on system libraries
- CMake layout changes for python and nuke installs
- Bumped internals to yaml 0.3.0, pystring 1.1.2
- Optimized internal handling of Matrix / Exponent Ops
- Added INTERP_BEST interpolation option
- Python config.clearLooks() added
- Python docs revamp
- Nuke config-dependent knob values now baked into .nk scripts
- Nuke OCIOLookTransform gets reload button
- Nuke nodes get updated help text

Version 1.0.6 (March 12 2012):

- JNI (Java) updates
- ocioconvert arbitrary attr support

Version 1.0.5 (Feb 22 2012):

- Internal optimization framework added
- SetLoggingLevel(..) bugfix
- Python API Documentation / website updates
- Clang compilation fix

Version 1.0.4 (Jan 25 2012):

- ocio2icc deprecated (functionality merged into ociobakelut)
- rv integration (beta)
- nuke: updated channel handling
- Documentation / website updates

Version 1.0.3 (Dec 21 2011):

- Tetrahedral 3dlut interpolation (CPU only)
- ociocheck and Config.sanityCheck() have improved validation
- Mari: updated examples
- Makefile: misc updates to match distro library conventions

Version 1.0.2 (Nov 30 2011):

- 3D lut processing (cpu) is resiliant to nans
- Nuke OCIOFileTransform gains Reload buttons
- Installation on multi-lib *nix systems improved
- Installation handling of soversion for C++/python improved
- ociobakelut improvements
- Initial version of Java bindings (alpha)

Version 1.0.1 (Oct 31 2011):

- Luts with incorrect extension are properly loaded
- ocio2icc / ociobakelut get –lut option (no longer requires ocio config)
- DisplayTransform looks do not apply to 'data' passes.

Version 1.0.0 (Oct 3 2011):

- ABI Lockoff for 1.0 branch
- General API Cleanup (removed deprecated / unnecessary functions)
- New features can be added, but the ABI will only be extended in a binary compatible manner. Profiles written from 1.0 will always be readable in future versions.
- Fixed Truelight Reading Bug
- ocio2icc no longer requires ocio config (can provide raw lut(s)

Version 0.8.7 (Oct 3 2011):

- Fixed Truelight Reading Bug

Version 0.8.6 (Sept 7 2011):

- Updated .ocio config reading / writing to be forwards compatibile with 1.0 (Profiles written in 0.8.6+ will be 1.0 compatible. Compatibility from prior versions is likely, though not guaranteed.)
- Better logging
- Added ColorSpace.equalitygroup (makes ColorSpace equality explicit)
- Substantial Nuke node updates
- Added support for Iridas .itx read/write
- Windows Build Support

Version 0.8.5 (Aug 2 2011):

- Nuke OCIODisplay fixed (bug from 0.8.4)
- Updated Houdini HDL Reader / Writer

Version 0.8.4 (July 25 2011):

- Native Look Support
- Core / Nuke OCIODisplay supports alpha viewing
- Added Houdini (.lut) writing
- Added Pandora (.mga,.m3d) reading
- Additional internal bug fixes

Version 0.8.3 (June 27 2011):

- OCIO::Config symlink resolution fixed (bugfix)
- Nuke OCIODisplay knobs use string storage (bugfix)
- Makefile cleanup
- Documentation cleanup

Version 0.8.2 (June 7 2011):

- Numerous Windows compatibility fixes
- Python binding improvements
- OCIO::Baker / ociobakelut improvements
- Lut1D/3D do not crash on nans (bugfix)
- Nuke UI doesnt crash in known corner case (bugfix)

Version 0.8.1 (May 9 2011):

- New roles: TEXTURE_PAINT + MATTE_PAINT
- Mari API Example (src/mari)
- FileFormat registry updated to allow Windows + Debug support
- boost_ptr build compatibility option

Version 0.8.0 (Apr 19 2011):

- ABI Lockoff for stable 0.8 branch New features can be added, but the ABI will only be extended in a binary compatible manner
- Otherwise identical to 0.7.9

Version 0.7.9 (Apr 18 2011):

- Added support for .vf luts
- Misc. Nuke enhancements
- Adds optional boost ptr support (backwards compatibility)
- Makefile enhancements (Nuke / CMAKE_INSTALL_EXEC_PREFIX)
- cdlTransform.setXML crash fix

Version 0.7.8 (March 31 2011):

- Iridas lut (.cube) bugfix, DOMAIN_MIN / DOMAIN_MAX now obeyed
- Exposed GPU functions in python (needed for Mari)

- Nuke OCIODisplay cleanup: Fixed knob names and added envvar support
- ociobaker cleanup
- tinyxml ABI visibility cleaned up
- Removed Boost dependency, tr1::shared_ptr now used instead

Version 0.7.7 (March 1 2011):

- Lut baking API + standalone app
- Truelight runtime support (optional)
- Cinespace (3d) lut writing
- CSP prelut support
- Boost argparse dependency removed
- SanityCheck is more exhaustive
- FileTransform supports relative path dirs
- Python enhancements (transform kwarg support)
- Makefile enhancements (OIIO Path)
- Processor API update (code compatible, binary incompatible)

Version 0.7.6 (Feb 1 2011):

- Updated Config Display API (.ocio config format updated)
- Added ocio2icc app (ICC Profile Generation)
- Revamp of ASC CDL, added .cc and .ccc support
- Documentation Improvements
- Makefile enhancements (Boost_INCLUDE_DIR, etc)

Version 0.7.5 (Jan 13 2011):

- ociodisplay enhancements
- gpu display bugfix (glsl profile 1.0 only)
- Makefile enhancements
- Nuke installation cleanup
- Doc generation using sphinx (html + pdf)

Version 0.7.4 (Jan 4 2011):

- Added 'Context', allowing for 'per-shot' Transforms
- Misc API Cleanup: removed old functions + fixed const-ness
- Added config.sanityCheck() for validation
- Additional Makefile configuration options, SONAME, etc.

Version 0.7.3 (Dec 16 2010):

- Added example applications: ociodisplay, ocioconvert
- Makefile: Add boost header dependency
- Makefile: Nuke plugins are now version specific

- Fixed bug in GLSL MatrixOp

Version 0.7.2 (Dec 9 2010):

- GPUAllocation refactor (API tweak)
- Added AllocationTransform
- Added LogTransform
- Removed CineonLogToLinTransform
- A few bug fixes

Version 0.7.1 (Nov 15 2010):

- Additional 3d lut formats: Truelight .cub + Iridas .cube
- FileTransform supports envvars and search paths
- Added Nuke plugins: LogConvert + FileTransform
- Improved OCIO profile formatting
- GCC visibility used (when available) to hide private symbols
- A few bug fixes

Version 0.7.0 (Oct 21 2010):

- Switched file format from XML to Yaml

Version 0.6.1 (Oct 5 2010):

- Exposed ExponentTransform
- Added CineonLogToLinTransform - a simple 'straight-line' negative linearization. Not strictly needed (could be done previously with LUTs) but often convenient to have.
- Added DisplayTransform.displayCC for post display lut CC.
- Many python improvements
- A few bug fixes
- A few Makefile enhancements

Version 0.6.0 (Sept 21 2010):

- Start of 0.6, "stable" branch

  All 0.6.x builds will be ABI compatible (forward only). New features (even experimental ones) will be added to the 0.6 branch, as long as binary and source compatibility is maintained. Once this no longer is possible, a 0.7 "dev" branch will be forked.

- Split public header into 3 parts for improved readability (you still only import <OpenColorIO/OpenColorIO.h> though)
- Added MatrixTransform
- Refactored internal unit testing
- Fixed many compile warnings

Version 0.5.16 (Sept 16 2010):

- PyTransforms now use native python class inheritance
- OCIO namespace can now be configured at build time (for distribution in commercial apps)
- Updated make install behavior
- DisplayTransform accepts generic cc operators (instead of CDL only)
- A few bug fixes / compile warning fixes

Version 0.5.15 (Sept 8 2010):

- OCIO is well behaved when $OCIO is unset, allowing un-color-managed use.
- Color Transforms can be applied in python config->getProcessor
- Simplification of API (getColorSpace allows cs name, role names, and cs objects)
- Makefile enhancements (courtesy Malcolm Humphreys)
- A bunch of bug fixes

Version 0.5.14 (Sept 1 2010):

- Python binding enhancements
- Simplified class implmentations (reduced internal header count)

Version 0.5.13 (Aug 24 2010):

- GPU Processing now supports High Dynamic Range color spaces
- Added log processing operator
- Numerous bug fixes
- Numerous enhancements to python glue
- Exposed PyOpenColorIO header, for use in apps that require custom python glue
- Matrix op is optimized for diagonal-only subcases
- Numerous changes to Nuke Plugin (now with an addition node, OCIODisplay)

Version 0.5.12 (Aug 18 2010):

- Additional DisplayTransform improvements
- Additional GPU Improvements
- Added op hashing (processor->getGPULut3DCacheID)

Version 0.5.11 (Aug 11 2010):

- Initial DisplayTransform implementation
- ASC CDL Support
- Config Luma coefficients

Version 0.5.10 (July 22 2010):

- Updated Nuke Plugin, now works in OSX
- Fixed misc. build warnings.
- Continued GPU progress (still under development)

Version 0.5.9 (June 28 2010):

- Renamed project, classes, namespaces to OpenColorIO (OCIO)

- Added single-pixel processor path
- Improved python path makefile detection
- Continued GPU progress (still under development)

Version 0.5.8 (June 22 2010):

- Support for .3dl
- Support for matrix ops
- Code refactor (added Processors) to support gpu/cpu model
- Much better error checking
- Compilation support for python 2.5
- Compilation support for OSX

Version 0.5.7 (June 14 2010):

- Python API is much more fleshed out
- Improved public C++ header

Version 0.5.6 (June 8 2010):

- PyConfig stub implementation
- Dropped ImageDesc.init; added PlanarImageDesc / PackedImageDesc
- Dropped tr1::shared_ptr; added boost::shared_ptr

Version 0.5.5 (June 4 2010):

- .ocio supports path references
- Switch config envvar to $OCIO
- Added 3D lut processing ops

Version 0.5.4 (June 1 2010):

- Initial Release
- CMake linux support
- XML OCIO format parsing / saving
- Example colorspace configuration with a few 'trivial' colorspaces
- Mutable colorspace configuration API
- Support for 1D lut processing
- Support for SPI 1D fileformats.
- Nuke plugin

3.9 License

yaml-cpp http://code.google.com/p/yaml-cpp/

PTex (Mutex), courtesy of Brent Burley and Disney http://ptex.us/

Little CMS http://www.littlecms.com/

MD5, courtesy L. Peter Deutsch, Aladdin Enterprises. http://sourceforge.net/projects/libmd5-rfc/files/

argparse, courtesy OpenImageIO and Larry Gritz http://openimageio.org

---

Pystring

TinyXML is released under the zlib license:

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

---

yaml-cpp

Copyright (c) 2008 Jesse Beder.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

PTEX SOFTWARE Copyright 2009 Disney Enterprises, Inc. All rights reserved

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

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The names "Disney", "Walt Disney Pictures", "Walt Disney Animation Studios" or the names of its contributors may NOT be used to endorse or promote products derived from this software without specific prior written permission from Walt Disney Pictures.

Disclaimer: THIS SOFTWARE IS PROVIDED BY WALT DISNEY PICTURES AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT AND TITLE ARE DISCLAIMED. IN NO EVENT SHALL WALT DISNEY PICTURES, 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 BASED 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 DAMAGES.

Little CMS Copyright (c) 1998-2010 Marti Maria Saguer

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

MD5

Copyright (C) 1999, 2002 Aladdin Enterprises. All rights reserved.

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch ghost@aladdin.com

---

argparse

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

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

OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

(This is the Modified BSD License)

search

genindex

test...123