14-15 Manette St
London, W1D 4AP
UNITED KINGDOM

t: +44 (0)20 7292 0400
f: +44 (0)20 7292 0401
www.filmlight.ltd.uk

FilmLight
DIGITAL FILM TECHNOLOGY

Technical Note

Truelight Software Library 2.0

Richard Kirk

Document ref.	FL-TL-TN-0057-SoftwareLib
Creation date	14 March 2003
Last modified	28 March 2006
Revision no.	1.112

Summary

With high-speed film scanners, digital cameras, and the increasing use of computer generated images, it is now practical and sensible to master an entire feature film digitally. If people are to work effectively with digital material, either for digital grading or for special effects, then their display must match their final projected image.

A Truelight instance takes in film-based image data, and outputs display RGB. This output should match the projection of a print. The typical everyday user just wants to know how their image will appear in a cinema. There is only one correct way for the image to appear, so they should be able to use the Truelight node without touching the controls

In real life, things are not always so simple. Some colours, such as bright green, are possible on a monitor but not on film. Other colours, such as deep reds and yellows, are possible on film but are impossible on monitors. Clearly, we cannot provide a match for every colour.

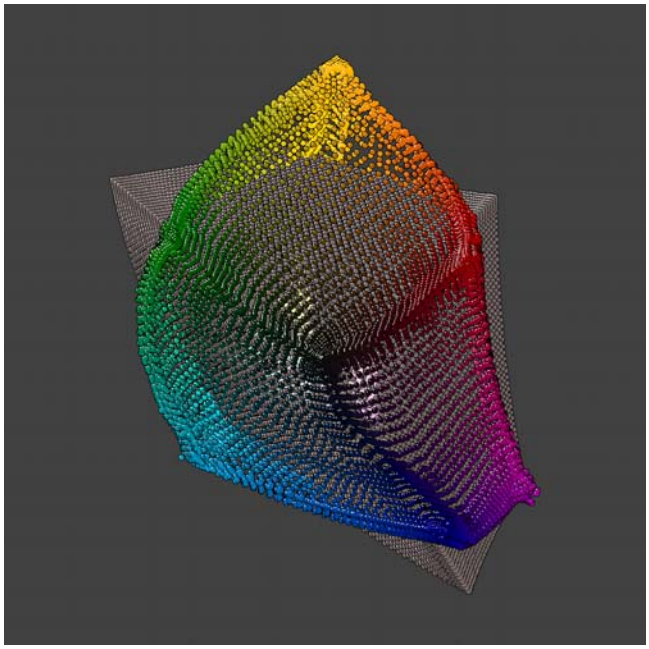
The Truelight user has some simple controls to check their colours are in gamut, to view very dark or very light images, to correct for under- or over-exposure in prints, or to view images under office lighting conditions. However, most of the time the Truelight user should leave these controls turned off, or in their default state.

Truelight also supports inverse transforms from display RGB to film-based data. These inverse transforms and other features are described in this note.

Contents

1 Introduction

A Truelight instance takes in film-based image data, and outputs display RGB. This output should match the projection of a print. The typical everyday user just wants to know how their image will appear in a cinema. There is only one correct way for the image to appear, so they should be able to use the Truelight node without touching the controls.



The colour gamut of a graphics monitor (grey spheres) and of Kodak Vision film (coloured spheres) plotted in CIE L*a*b* space.

The diagram shows one of the many problems we face when matching monitors to film. Some colours, such as bright green, are possible on a monitor but not on film. Other colours, such as deep reds and yellows, are possible on film but are impossible on monitors. Clearly, we cannot provide a match for every colour.

The Truelight user has some simple controls to check their colours are in gamut, to view very dark or very light images, to correct for under- or over-exposure in prints, or to view images under office lighting conditions. However, most of the time the Truelight user should leave these controls turned off, or in their default state.

This document was first written for developers using the Truelight library. If you are not a developer, you might skip sections 1.7 and 5, though even these might give insights on how the applications work.

1.1 Truelight Calibration Files

Most users should be able to use Truelight without needing to know what lies in the calibration files. The nearest they may come to handling calibration files is when they create a display calibration with the Truelight calibration tool.

However, one of the major features of Truelight is that any user can make and edit any of the calibrations using only conventional text processing tools. This can allow advanced users to adapt the Truelight calibration to their particular print process or display.

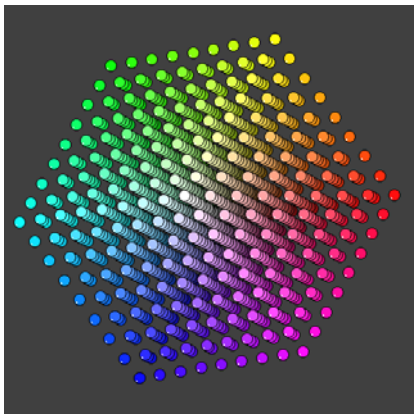
You do not need a full understanding of how Truelight calibrations work to read the rest of this document, but a brief outline may help understanding what the various parameters do, and how monitor calibrations are made. For more details on Truelight calibration files, see section 6.

The Truelight calibration files are normally stored in files in a set of directories under the Truelight root directory. The Truelight library also contains some standard fixed calibrations. For more details, see section 3.

1.2 List transforms

The Truelight library can fit an interpolation function through a random set of points in up to four dimensions. A Truelight list transform is simply a text list of input and output colour values. Each line in a typical list contains a set of three floating-point values for the 3D input space, followed by the corresponding three floating-point values for the 3D output space. The lines are in no particular order, and the points do not have to lie on a regular grid in the colour space.

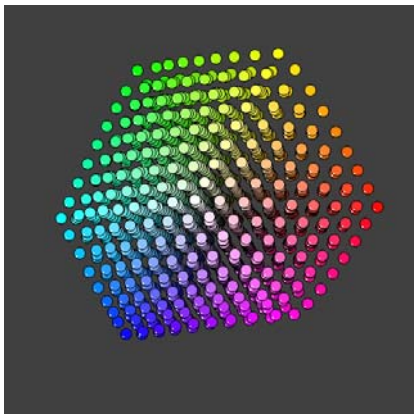
A good list for a perceptual colour space may have about 700 entries. The Cineon colour space contains more than the normal visible range, so we need more points to span it. Here is a typical set of about 1250 points we use for calibration…



A 9*9*9 body-centred lattice of points in Cineon RGB

There is a general scheme for selecting a set of points to use for a particular transform. You start with a large list of points. You find which point which introduces the least error when it is removed, as measured by transforming all the original points, and you remove it. You carry on doing this until some error threshold is exceeded. This is slow, and the resulting grid of points is unlikely to be uniform, but it can give some insights into how the space should be filled.

We recorded a patch on film for each of these colours. We then measured the status M densities of the patches on a densitometer…



The same set of colours plotted in status M density space

A film recorder calibration is just a list of Cineon values and the corresponding status M values.

If we give Truelight a set of RGB Cineon values, it can interpolate the corresponding set of RGB status M density values. If our Cineon colour lies within the scatter of Cineon colours on our list, then the interpolation will probably be accurate. If our Cineon colour lies outside the range of our list data, then the extrapolation may not be accurate.

For more details on list transforms, see section 6.3.

1.3 Formula Transforms

List transforms are good for describing complex colour processes that we do not want to model in detail, but they can be slow, and the interpolation can be inaccurate. You would not want to use a list transform to swap the red and green channels in an RGB image, or range 8-bit data.

The Truelight library can also handle formulas.

These formulas can be very brief. The formula to swap the red and green channels in an RGB image is just **#1,#0,#2**. The formula to range all channels of 8-bit data 0-1 is just **#/255**.

Most display calibrations are formula transforms. They are longer formulas; typically involving mathematical functions, matrices, and tone curves based on experimental data.

For more details on formula transforms, see section 6.4.

1.4 Colour Cube Transforms

The general list and formula transforms in sections 1.2 and 1.3 can convert from one colour space to another, but they are not fast enough for a typical film recorder. Before we can transform millions of pixels of data, we must turn our list or formula transform into a cube transform.

The cube transform interpolates between a series of points lying in neat rows and columns going up the gamut limits. A typical three-dimensional colour cube will have 16 intervals per axis, or about four thousand sets of values. We can interpolate these points using our list or formula transforms.

For more details on colour cube transforms, see section 6.6.

1.5 Truelight Commands

You can control Truelight parameters using commands:

- All commands take the form *name* { *argument string* }.
- All spacing including carriage returns before and after the name is ignored.
- The name cannot contain whitespace or the curly bracket characters '{' and '}'.
- The argument string can contain any spacing including carriage returns. It can also contain the curly bracket characters '{' and '}' provided they are correctly nested.
- Commands without names are comments. Truelight ignores them. You can disable a set of commands by enclosing them in curly brackets, which turns them into a comment.
- Anything else is an error.

Different commands handle the argument string in different ways:

Argument type	Comment	Examples
File name	May contain spaces. Leading and trailing whitespace is ignored.	`Lamp{xenon}` `lamp{./displays/xenon}`
Value Triplets	Commas should separate multiple values. Three identical values may be abbreviated to a single value.	`greyValue{345,456,567}` `greyValue{445}`
On/off switch	Either "on" or "off".	`invert{on}`
Switch	One of a fixed set of strings.	`cubeType{gamutAlarm}`
Formula	See section 1.3 or 6.4.	`rgbFormula{(#*255-16)/219}`
None	Commands without arguments still must have the curly brackets.	`intervals{}`
Default	All commands have a default value. The default often disables the option.	`GreyValue{}`

If you set a parameter more than once, the later setting will override the earlier one.

Some parameters are only available to fully licensed Truelight users (see section 1.7 for details on licensing). Setting an unavailable parameter may give an error. The TruelightHelp( ) library command gives the definitive list of the currently available commands for the build of the library. The ‘tl-utils’ utility prints out this message if you type ‘tl-utils Help’.

1.6 Truelight Profiles

A Truelight 'profile' is a file containing a set of Truelight commands. This is a convenient way to set up Truelight parameters for a typical project. Here is a valid example of a profile:

```
TruelightProfile{
{
  This is a comment.
  Comments and commands can span several lines.
  Comments and commands can contain {curly brackets} provided they balance.
}

profile{default_settings}

greyValue{445} greyStatusA{1.09,1.06,1.03}
aimGamma{ 0.966,
          1.063,
          1.082 }

whiteValue{ 1023 }
linearizeCube{on}

} <- This is closing bracket matches the opening bracket on the first line.
     Anything beyond this closing bracket is ignored.
```

All Truelight profiles start with the **TruelightProfile{** string. When we load a profile, Truelight interprets everything between the opening string and the closing bracket **}** as Truelight commands. Any parameters not set by these commands will have the default values.

The first command in this profile calls the **default_settings** profile. This is equivalent to replacing this command with all the commands in the other profile. If you set a parameter more than once, the later setting will overwrite the earlier one. This way a profile can inherit the settings of another profile. This construction is not common, but it can be useful.

This particular profile shows that profiles are not sensitive to spacing and indentation. Most computer-generated profiles are much tidier.

There are other ways of setting Truelight parameters. For more details on Truelight parameters and how they are set, see section 5.

1.7 Truelight Licensing

Truelight is started using the **TruelightBegin( )** command (see section 5.1). This command must be called before any other Truelight library command. This command asks the FilmLight licence server for a valid licence. If a valid licence is found, all the Truelight functions are enabled. FilmLight issues these licences. They have an expiry date, and are restricted to a single machine, or a single licence server for floating licences.

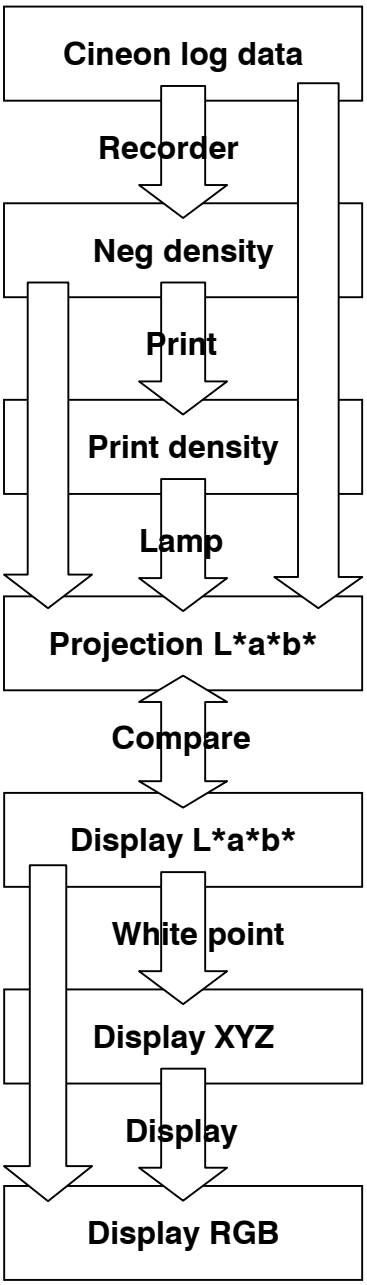
The **TruelightBegin( )** command can take a string argument. The string argument can contain a series of **name{value}** pairs (see section 3.4). The string may contain two entries that look like…

```
name{MyCompany} key{AB35F001249CC00D}
```

If the name and the key do not correspond, these unlock most of the Truelight functions. This is useful for OEM developers who wish to distribute applications with embedded Truelight functions without the FilmLight licence server. This key is not restricted to a particular machine, and does not expire, so the developer can call **TruelightBegin( )** in their code with a fixed string argument.

Truelight can run without a licence or a key, but all the colour-handling operations are disabled. This can be useful if you just want to list the calibrations or the profiles.

2 The Truelight Process



The diagram shows in outline the entire colour transform as supported by Truelight. The square boxes represent the colour space data, and the arrows are the calibrations - the transforms that get us from one colour space to another.

The recorder transform is typically a list transform containing Cineon values and negative densities, similar to the values plotted in section 1.2.

The print transform is a similar list to the recorder transform, but with negative densities as the input, and print densities as the output.

The lamp transform is another list. This has print densities as the input, and visual L*a*b* values of the colours we see when an image is projected.

We usually measure the spectral properties of the projector lamp and the film dyes. This allows us to calculate the density and the projected L*a*b* for any combination of film dyes.

We do not need to measure print densities if we can measure the projected L*a*b* values directly. The arrow on the left bypassing the print density box shows this option. This would only be appropriate for one combination of lamp and print, but it is still useful.

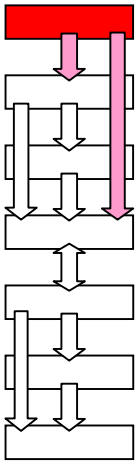
The L*a*b* values of the projection should match the L*a*b* values of the display.

If we have the appropriate instruments, we can measure the display output in absolute tristimulus XYZ units. We can then match the display white to the projection.

Sometimes it is not practical to calibrate the display in absolute units. If the display has a limited dynamic range, it may be better to use the natural white point of the display. In such cases we may chose to calibrate the display in L*a*b* relative to the display white. The arrow on the left that bypasses the display XYZ box shows this option.

We shall now deal with each of these stages in more detail.

2.1 Input Colour Space



The normal Truelight colour space is Cineon log. Kodak also defines fixed formulas for linear and video Cineon colour spaces. For values between 0 and 1…

$$Video > 0.081 \quad Linear = \left(\frac{Video + 0.099}{1.099} \right)^{1/0.45}$$

$$Video < 0.081 \quad Linear = \left(\frac{Video}{4.5} \right)$$

$$Log = \frac{685}{1023} + \frac{300}{1023} . \log_{10} \left(\frac{Linear + 0.0109}{1.0109} \right)$$

NB: Cineon video is not conventional video. It is just Kodak's one-dimensional transform of the Cineon negative density space to give a more 'video-like' tone curve on typical print stocks.

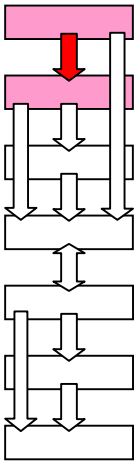
The colour cube transform (see section 6.6) has input look-up tables. We can combine these look-up tables with the one-dimensional video-lin and lin-log transforms to make cubes that can work with video or linear input data without any performance overhead.

The cubeInput{} command supports video and linear Cineon spaces. It was written for plug-ins that select the Cineon space options using radio buttons or pull-down menus. Usually, this transforms the cube look-up table values. If we have an inverted cube, then cubeInput{} does the reverse transform on the cube output values. This was an afterthought to allow plugins to offer an inverse transform.

The valueFormula command is a general transform that always operates on the Truelight input values. If you put the same formula into valueFormula{} - see section 6.4 for details on how to write a formula - you should get similar results. The forward transform will not be quite the same because we will not be stretching the cube look-up tables. To get the same look-up table stretch, copy the formula into cubeLutFormula{} (see section 2.17).

Command	Argument	Default	Notes
cubeInput{}	Log, linear, or video	Log	See truelight.h for recognized spaces.
valueFormula{}	Run formula on input data	None.	See section 6.4 for more about formulae. Not for general use.

2.2 Recorder Calibration



The recorder calibration is typically a list transform containing Cineon values and negative densities, similar to the values plotted in section 1.2. We make a recorder calibration by printing out a calibration strip of known Cineon values, and then measuring the densities.

The negative densities are normally measured on a densitometer using status M filters. The status M filters have narrow pass bands in red, green, and blue, similar to the narrow wavebands used in the lamp house when making a print. The densitometer values should tell us how the image should print.

The SMTPE has defined an alternative standard for measuring negative densities, known as RP-180. It does not matter which standard we use with Truelight, provided we use the same standard for our print calibration (see section 2.4).

You can also load a film recorder with print stock. Direct to print recorder calibrations are described in section 2.5.

The recorder calibration should work if the list values were measured off a calibration film from a correctly set up recorder. Unfortunately, there is more than one way to set up a film recorder. Kodak defines the 10-bit Cineon values…

$$value = (density - Dmin) * gamma * 0.002$$

Dmin and *gamma* are Cineon constants defined by Kodak. *Dmin* is near the film base density. *Gamma* is usually about 1.0. There are different values of *Dmin* and *gamma* for each colour channel.

The recorder{} option is used to select the recorder calibration by name. This option needs a full FilmLight licence (see section 1.7). The library contains a default recorder calibration for an ArriLaser loaded with Kodak 5242.

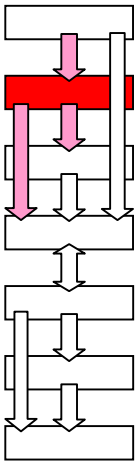
The valueFormula{} option can be used to transform the input Cineon values. This happens before the aim gamma fitting and the grey patch calculations. If you want to transform the input calculations without affecting the aim gamma settings, see the cubeInFormula{} and cubeOutFormula{} commands in section 2.15.

Different companies use different aim gammas. If you use the internal calibrations, you must specify your recorder aim gammas using aimGamma{}.

The listIntervals{} option can be used to interpolate a new list of points based on a BCC lattice. This is probably not useful for everyday operations, but may be useful if the original list had too few entries. A large value may significantly slow up calculations.

Command	Argument	Default	Notes
recorder{}	Recorder calibration file	Internal ArriLaser & Kodak 5242 calibration.	Select recorder calibration by filename. Needs a full Truelight licence.
valueFormula{}	Run formula on input Cineon data	None.	See section 6.4 for more about formulae. Not for general use.
aimGamma{}	Recorder aim gamma values	Do not fit to aim gammas.	Fit to supplied aim gamma values. Comma separated values.
fitGamma{}	on or off	on	Turn on/off aim gamma fitting.
listIntervals{}	Number of BCC intervals	No interpolation	Interpolate a new list based on a BCC lattice.

2.3 Printer Point Exposure Boost



When a laboratory makes a print from the negative, they have controls on the printer lamp house that vary the amount of red, green, and blue light being sent through the negative. The units on these controls are usually referred to as 'points'. There are 12 points to a stop. If you turn up the point setting, more light goes through the negative, and the positive print gets darker.

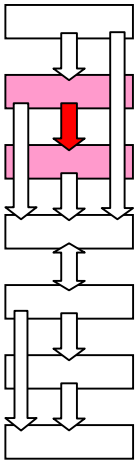
The absolute printer point setting is not much use outside the film laboratory. Different labs use different settings, and these settings are often changed to track the changing chemistry in the baths. The labs do this is by printing a standard 'grey slate', and adjusting the printer light settings until the slate hits a certain density. Outside the printer lab, it is common to ask for "two points more red" or other relative shifts in printer points, rather than ask for absolute settings.

The Truelight printer point boost settings simulate these printer point corrections.

The exposure boosts are used to simulate the effect of changing the print settings. They are also used to compensate for the typical laboratory printing errors when comparing the display to an actual print.

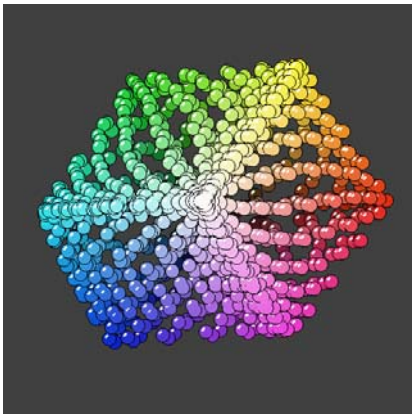
Command	Argument	Default	Notes
printerPoints{}	R,G,B exposure boost in printer points	0,0,0	-1 point equals a shift of 12.5 Cineon values, or 0.025 units of negative density with an aim gamma of 1.0.

2.4 Print Density Calibration



The print calibration is typically a list transform containing negative densities on the input, and print densities on the output.

The print densities are normally measured on a densitometer using status A filters. The status A filters have pass bands in red, green, and blue that are supposed to mimic the human eye response. This is not entirely accurate, but to a first approximation, two pieces of film with similar status A densities will look the same on a light box.



Print densities, plotted in Status A space

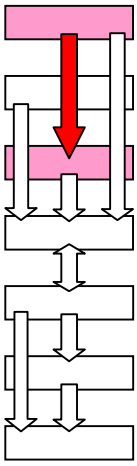
The 'S' shape of the print tone curve compresses the grid of points near the edge of the cube. We must space our experimental points carefully: if our points are too close, the differences between the colour values will be less than the measurement noise.

Because the print calibration is just a list of densities, and contains no explicit physical model of a film's behaviour, we can model unusual print processes such as bleach bypass or solarization just by making another print of our negative, and measuring it.

Usually print processes are controlled by printing out a 'grey slate' - a patch of grey film with known negative densities; then asking the film laboratories to adjust the printer lights (see section 0) until the density hits some standard values. If you use a grey slate, set your grey slate Cineon values and aim densities using the greyValue{} and greyStatusA{} commands.

Command	Argument	Default	Notes
print{}	Print calibration file	Internal Kodak Vision calibration.	Select print calibration by filename. Needs a full Truelight licence.
greyValue{}	Grey slate Cineon values	445, 445, 445	Default = Kodak LAD grey slate
greyStatusA{}	Grey slate Status A densities	No fit	Kodak values 1.09, 1.06, 1.03 are often used.

2.5 Recorder Print Calibration

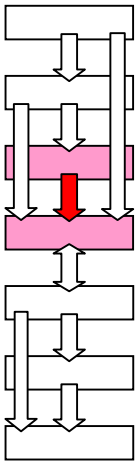


A recorder calibration is usually a list transform containing Cineon values and negative densities, as described in section 2.2. However, it is possible to load a film recorder with print stock.

We make a recorder calibration by printing out a calibration strip of known Cineon values on print stock, and then measuring the print status A densities. Recorder print calibrations are stored and selected in the same way as ordinary recorder calibrations. They are distinguished from ordinary recorder calibrations by their output colour space. Because they skip negative density and go directly to print density, commands that operate in the negative density colour space, such as printerPoints{}, have no effect.

Command	Argument	Default	Notes
recorder{}	Recorder calibration with status A output.	See section 2.2.	See section 2.2.
valueFormula{}	See section 2.2	None.	See section 6.4 for more about formulae. Not for general use.
greyValue{}	See section 2.4	445, 445, 445	See section 2.4
greyStatusA{}	See section 2.4	No fit	See section 2.4
listIntervals{}	Number of BCC intervals	No interpolation	Interpolate a new list based on a BCC lattice.

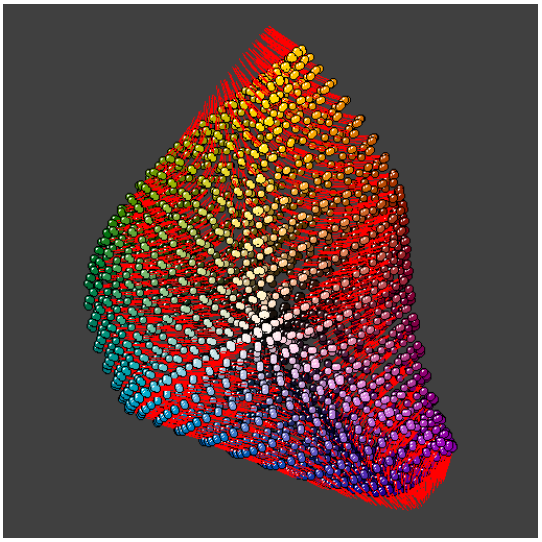
2.6 Projector Lamp Calibration



The lamp transform is typically a list transform containing print densities on the left of each line, and CIE L*a*b* values on the right.

It is possible to measure the L*a*b* values of a projected patch using a spectrometer. In practice, projector lamps are rarely stable enough to measure a typical set of patches. Instead, we usually generate lamp calibration values from a spectral model using a single spectral measurement of the lamp's emission, and the absorption of the film dyes and base.

In principle, our lamp calibration should only apply to the film whose dyes we have measured or modelled. In practise, the dyes of most colour print films are very similar. We should be careful when using black and white prints, or bleach bypass prints, but even these seem to give satisfactory results.

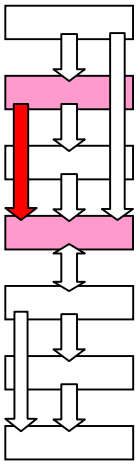


The red vectors represent the colour shift between two different projector lamps. Projected colours plotted in CIE L*a*b*.

The diagram shows the colour shifts we get if we change our projector lamp. The two lamps had similar colour temperatures. We have adjusted the data to make the whites match. We can see the colour shift is complex.

Command	Argument	Default	Notes
lamp{}	Lamp calibration file	Internal Xenon lamp calibration.	Select calibration by filename. Needs a full Truelight licence.

2.7 Print L*a*b* Calibration



The print calibration is typically a list transform containing negative densities on the left of each line, and projected print CIE L*a*b* values on the right.

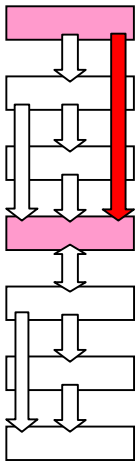
The print L*a*b* calibration can be made by combining a print density calibration (section 2.4) and a projector lamp calibration (section 0). This is a sensible thing to do if our lamp calibration is matched to our particular print stock. It is also possible to make print L*a*b* calibrations directly from measurements.

The L*a*b* print calibrations are kept in the same directory as the density print calibrations. The L*a*b* print calibrations do not need a separate lamp calibration. If a lamp calibration has been selected, it is ignored.

The print grey slate is not graded to specific L*a*b* values. You can explicitly turn off the greyslate fit with greyValue{}.

Command	Argument	Default	Notes
print{}	Print calibration file	Internal Kodak Vision calibration.	Select print calibration by filename. Needs full licence.
lamp{}	Lamp calibration file	Internal Xenon lamp calibration.	Ignored for L*a*b* print calibrations
greyValue{}	Grey slate Cineon values	445, 445, 445	Ignored for L*a*b* print calibrations
greyStatusA{}	Grey slate Status A densities	No grey fit	Ignored for L*a*b* print calibrations

2.8 Input Display Calibration



The input display calibration is used to match one display to another. A typical use is to display images rendered for a video monitor on a calibrated computer display. The input display transform is made by inverting a display calibration taken from the display calibrations directory.

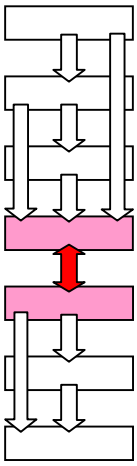
A L*a*b* display calibration (see section 2.12) will transform from CIE L*a*b* to device RGB. Its inverse will convert from device RGB to L*a*b*. This conversion replaces the equivalent conversion made from the recorder, printer, and lamp settings (sections 2.2 to 2.7). If you define inputDisplay{}, then all your other recorder, printer, and lamp settings described in the earlier sections are ignored.

An XYZ display calibration (see section 2.12) is converted to a L*a*b* display calibration using the fixed CIE formulae, with the white point set to the device R=G=B=1.0 white.

The display transform must be inverted to be used as an input display. This inversion is straightforward if the transform has a list (see section 1.2), but display transforms are not often lists. If we have a formula transform with an inverse formula (see section 1.3), then Truelight can easily generate a list from that. If there is no inverse formula, then Truelight can generate an inverted list from the forward transform by iteratively solving for each point. This is generally reliable for smooth functions, but may have difficulty inverting discontinuous functions. If you can make an inverse formula, then it is a good idea to do so.

Command	Argument	Default	Notes
inputDisplay{}	Display calibration file.	None.	Select display calibration by filename.

2.9 Matching Film and Display L*a*b*



The light levels from a projector and a display can match, but the images may still not look the same. There may be external physical reasons for this, such as stray light landing on the projector screen. There may internal physical reasons for this - a bright surround may add light that scatters within the eye, even though no light lands on the screen. There are psychophysical effects in the eye-brain system. There are other effects of experience: we are happy to use tungsten lighting temperatures when looking at reflection copy, but a monitor with a tungsten white point will look very orange.

We can compensate for some of these effects. We know if we add an overall wash of stray light to a display, or we add a bright border, or we turn up the ambient lighting, then we are less able to see the shadow details. We can compensate for this by increasing the slope of the tone curve to boost the contrast of these shadow details.

We have three parameters to the model: the intensity, the adaptivity factor, and the flare factor. We have two sets of these parameters: one for the current display, and one for the reference projection.

The display intensity is set using displayWhite{}. The display white luminance is a similar value, set using white_Y{} (see section 2.10). Under all sensible conditions these two parameters should have the same value. However, it is useful to be able to alter the luminance without the risk of colour side effects from the flare correction. In practice this does not matter much because the intensity only has a significant effect on the flare model when the differences in brightness are much greater than you ever get between one display and another.

The adaptivity factors correct for the grey level surrounding our images. This grey level may belong to the border on the screen, the frame on the monitor, or the colour of the wall behind the monitor, or some combination of all three. More complex colour appearance models handle coloured surrounds with separate parameters for the immediate surround and the distant surround. In practice, three grey levels cover most Truelight user needs: dark surround (0.1), grey surround (0.5) and white surround (1.0).

The flare factor is the estimated flare as a fraction of the display white. This is the most important setting. The effects are severely non-linear - particularly when our display flare goes below the projector flare, and we start adding grey to our blacks. This makes it hard to control with a slider. In practice, three preset values may cover most needs: cinema (0.01), dim office (0.04) and light office (0.10).

These flare correction parameters are not the right way to cure the problem. They may match the appearance of the shadow detail, but they will introduce slight errors in the colour rendition. The right way to deal with stray light, bright borders, or stray ambient lighting is to fix them. However, where people are working on computers in typical office environments, blacking out the office may not be an option. Correcting for flare lets the office worker see the shadow detail. This lets them to get closer to the intended look before the image goes for the final grade in a properly blacked out room.

We cannot base our flare settings on measurements, because the effect is the sum of physical, psychophysical and psychological effects. The best a user can do is to find an image with subtle shadow detail, view the image in a properly blacked out room and memorise the appearance, then adjust their office settings to match.

We have three other controls: we can scale the brightness using brightness{}, scale the contrast using contrast{}, and scale the saturation using saturation{}. These parameters should normally be left in their default settings. However they can be helpful when looking at particularly dark or saturated images, or when viewing images with office lighting…

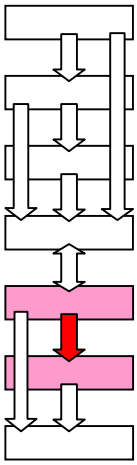
If you have a dim image, and you are having trouble seeing the detail in ordinary office lighting, then you can just turn up the brightness. The image highlights may 'clip' as they go beyond what the display can achieve, but at least the lowlight detail has been rendered with the correct colours.

Truelight will try to fit an exact match for any colour. Some displays such as LCDs have a smaller contrast range than the film they are trying to match. The highlights may be rendered correctly, but the shadow details will be lost. The contrast control can reduce the luminance range, while keeping the colours the same. HD monitors can show the opposite problem: Truelight is matching the light levels in the projection, but differences in flare conditions make the shadow detail more visible. We could add flare, but this makes the shadows look grey, which may look fine on a projected image but can look unnatural on a CRT. Sometimes, it is necessary to artificially extend the range of a video signal to get the right looking range on a scope. In either case, the contrast control can reduce the luminance range, while keeping the colours the same.

The saturation scaling can be used to the same sort of thing with the colour saturation. If our display is brighter than the cinema standard (and most CRTs can manage 50 foot-Lamberts), then we are getting a stronger, less noisy colour signal in our eye. If we reduce the saturation, then our image will look less saturated if it lies next to the cinema image, but in isolation it can seem like it matches (the Hunt effect). Unfortunately, we can see from the image back in section 1 that just reducing the saturation is not an effective way of bringing some film colours into the gamut of a typical display.

Command	Argument	Default	Notes	
displayFlare{}	Display flare as a fraction of white.	0.01	A high value will compensate for ambient lighting. 0.1 (10%) is typical for office lighting conditions.	
displayAdapt{}	Display surround adaption factor	0.1	Dark surround:	0.1
			Grey surround:	0.5
			White surround:	1.0
displayWhite{}	Display luminance in ft-lamberts	16	Matches SMTPE standards	
projectionFlare{}	Film flare as a fraction of white	0.01	A high value will boost the shadow contrast to compensate. 0.01 (1%) is a typical cinema value.	
projectionAdapt{}	Film surround adaption factor	0.1	Dark surround:	0.1
			Grey surround:	0.5
			White surround:	1.0
projectionWhite{}	Film luminance in ft-lamberts	16	Matches SMTPE standards	
brightness{}	Brightness scale factor	1.0	Scales L* value.	
contrast{}	Contrast scale factor	1.0	Scales L* range, leaving white point unchanged.	
saturation{}	Display saturation scale factor	1.0	Scales a*b* values. The L* value of saturated colours is moved towards the gamut centre.	
LabFormula{}	Run formula on L*a*b* data	None.	See section 6.4 for more about formulae. Not for general use.	
whiteValue{}	Cineon white point	None	whiteValue{685} will map the reference white to the L*a*b* white point.	

2.10 Display White point



The human eye-brain system is good at compensating for changes in white point. We usually perceive colour as a property of an object, rather than the product of the spectral properties of the object and the illuminant.

CIE L*a*b* is a perceptual colour space that tries to mimic the properties of the eye. It is derived from the absolute XYZ values, but it measures colours with respect to a reference white. The reference white always has the values 100,0,0. Equal vectors in L*a*b* space should correspond to the same visual contrast, even if the different L*a*b* values were calculated with different white points. This is not wholly accurate, but it does work well enough to be useful, especially for white points around the D65 standard.

If we chose a monitor white, we can reverse this transform and go from L*a*b* back to tristimulus XYZ. We enter the white point in CIE Yu'v' coordinates…

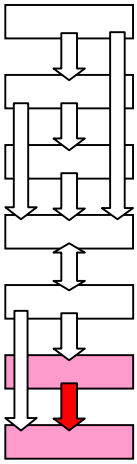
$$u' = \frac{4X}{X + 15Y + 3Z} \qquad v' = \frac{9Y}{X + 15Y + 3Z}$$

The luminance default value matches cinema standards. This is dimmer than monitors normally are. If you are working under office lighting, you will probably want to make it brighter. You can do this by increasing the white_Y{} setting. You can also do this by increasing the brightness{} setting (see section 2.9). The brightness control is usually provided, because it also works with L*a*b* calibrations (see section 2.12).

There is no unique cinema standard chroma, so our default matches D65, the white used in most video standards. This white is a higher colour temperature than that normally quoted for xenon arc lamps, but may be representative of real whites on some modern cinema projectors. These values can be replaced when matching to an actual projector.

Command	Argument	Default	Notes
white_Y{}	Display luminance in ft-lamberts	16	Matches SMTPE standards
white_u{}	CIE u'	0.1978	Matches D65 white point
white_v{}	CIE v'	0.4683	Matches D65 white point

2.11 Display XYZ Calibration



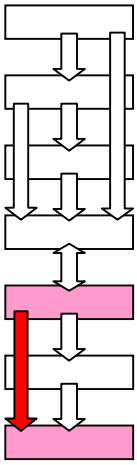
A display XYZ calibration is typically a formula transform that converts absolute XYZ values to display RGB. . The reverse transform may also be present.

The display calibration could be a list of measured XYZ values, and the display RGB values that generated them. However, an XYZ list transform is not a perceptually uniform space, so the interpolation may not work well. Most displays have almost independent red, green, and blue channels, so it is easier to represent them with a formula. This formula may include experimental tone curve figures (see section 6.5).

The XYZ display calibrations are kept in the same directory as the L*a*b* calibrations.

Command	Argument	Default	Notes
display{}	Display calibration file.	None.	Select display calibration by filename. This is the only calibration that does not need a full Truelight licence.

2.12 Display L*a*b* Calibration



A display L*a*b* calibration is typically a formula transform that converts L*a*b* values to display RGB. The reverse transform may also be present.

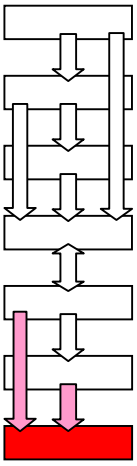
The display calibration could also be a list of values - the measured L*a*b* values, and the display RGB values that generated them. This might be the way to calibrate a digital projector that contained an unknown colour correction algorithm. Most displays have almost independent red, green, and blue channels, so it is easier to represent them with a formula.

Our display L*a*b* calibration bypasses the XYZ colour space, so the white_Y{}, white_u{} and white_v{} commands of section 2.10 will have no effect.

The L*a*b* display calibrations are kept in the same directory as the XYZ calibrations.

Command	Argument	Default	Notes
display{}	Display calibration file.	None.	Select display calibration by filename. This is the only calibration that does not need a full Truelight licence.

2.13 Display Tools



The rgbFormula{} command allows the user to perform an arbitrary function on the display RGB data. This is not usually a sensible thing to do since the display should be properly calibrated. It can be useful for doing simple operations such as video re-ranging. However, if you are doing something like swapping channels or displaying a single channel, you may get side effects. For operations like these, we suggest using cubeOutFormula{} instead. This is applied after the cube is calculated, so it is safer (see section 2.15).

The RGB image we generate should look sensible on the monitor, but the corresponding HD signal will not necessarily look sensible on a scope. The film black is not a true black, so our black signal may have non-zero RGB values. As conventional video is expected to go down to zero in the shadows, we may have to distort the tone curves slightly to get the scope signal to look right on the scope. The videoBlack{r,g,b} command forces the input Cineon values to the video black (0,0,0).

The videoBlack{} correction affects the extreme shadows. It is not supposed to have a significant effect on the image appearance: if this command changes the image appearance significantly, then there is something wrong with your data or your choice of black point. For example if you added a flare correction, the videoBlack{} command may undo it. Get the black point as close as you can to the video black using the other controls, before applying videoBlack{}.

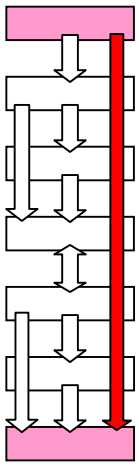
There is also a videoWhite{} command. It maps the given Cineon colour onto the video (1,1,1) white in a similar way.

The cube edit commands editA{}…editH{} operate on the cube output values. If we invert the cube, the cube output will not be display RGB. The cube edit commands editA{}…editH{} are not necessarily display tools, so they are described in a separate section (section 0).

The outputDisplay{} command allows us to simulate one display on another. If we make a cube with output values in display RGB, we convert these values to outputDisplay RGB, using a transform made from the two calibrations, converting any XYZ calibrations to L*a*b* transform using the Yu'v' white point. If we made a gamut alarm cube, this would show the colours that lay outside the display gamut, but the RGB values are made to look right on the outputDisplay device. This does not give sensible results if the cube output is not display RGB, or if the outputDisplay gamut is a much smaller than the display gamut.

Command	Argument	Default	Notes
rgbFormula{}	Run formula on display rgb data	None.	See section 6.4 for more about formulae. Not for general use.
videoBlack{}	Cineon rgb values ranged 0-1.	None	Distorts output values so input rgb gives output of 0,0,0.
videoWhite{}	Cineon rgb values ranged 0-1.	None	Distorts output values so input rgb gives output of 1,1,1.
outputDisplay{}	Display calibration file	None	Maps output values from display{} RGB to outputDisplay{} RGB

2.14 Transform, Formula, & Cube commands



The transform{} command replaces the entire Truelight colour calibration with a single list transform or formula from a file. There is no standard directory for these transforms, so the transform file name will have to include the path.

The formula{} command replaces the entire Truelight colour calibration with a formula supplied as a string.

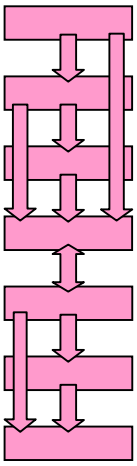
The importCube{} command disables the entire Truelight colour calibration, and instead reads a cube from the cube directory.

These commands are not often used, but they are sometimes handy for debugging or for customized transforms that cannot be achieved using the regular Truelight controls.

The commands that operate on the cube after it is generated, such as monotonicCube{} and the edit commands, will still work.

Command	Argument	Default	Notes
transform{}	Entire transform as a single file.	Conventional Truelight.	File name needs full path.
formula{}	Entire transform as a formula.	Conventional Truelight.	See section 6.4 for more about formulae.
importCube{}	Name of cube file	Cube file made by Truelight	Unencrypted cubes only. Hardware cubes are not read.
formulaIntervals{}	Number of intervals	16	Used when making a list transform from a formula.

2.15 Cube Transform Options



Truelight usually generates a floating-point transform from input log Cineon to output display RGB. It uses this transform to make a cube transform, so it can process image data quickly. The input values are restricted to the input gamut, but the output values are floating point. For more details on cube transforms, see section 6.6.

We introduced cubeInput{} back in section 2.1, because it affected the input colour space at the top of our diagram. However, because this is done when the cube is made, it belongs in this section too. The cubeInFormula{} command is a more general tool that transforms the input values of the list before the cube is generated.

The cubeType{} command lets us make several useful types of cube. The set of supported cube types can be found in xform.h.

The cubeType{straight} command is the default setting. It makes a cube that transforms from the input space to the output space. The cube should be similar to the original floating point transform, except that the input values are restricted to the in-gamut range.

The cubeType{gamutAlarm} command makes a cube with a gamut alarm. The gamut alarm replaces the in-gamut RGB output values with the corresponding R=G=B neutral values. When you transform an image through a gamut alarm cube, the colours that go out of the display gamut show up against a monochrome background. Colours that are interpolated from in gamut and out-of-gamut points are saturated in proportion.

Gamut alarm on standard Kodak LAD test image

The diagram in section 1 shows that there will typically be colours that we can get on film that we cannot get on our display, and vice-versa.

This looks like a serious problem. In fact, these extreme colours are rarely found in real scenes captured on film. Often, reds and yellows go just out of the CRT gamut, but the overall appearance of the image is hardly affected. However, it is important to know which colours on our display should not be trusted.

The cubeType{gamutClip} command generates cubes where all the output values are clipped to the gamut. This will happen anyway when processing integer data, so it is not particularly useful.

The cubeType{videoGamutClip} command generates cubes where all the output values are clipped to the legal 8-bit video gamut range 16..235.

The cubeType{videoGamutAlarm} command generates cubes similar to cubeType{gamutAlarm} but highlighting colours outside the legal 8-bit video gamut range 16..235.

We can generate an inverse cube using the invert{} command. This inverts the list transform before the cube is made. An inverse cube will convert from display RGB to log Cineon. An inverse cube generated with display{SonyHD} can be used to transfer images from video to film.

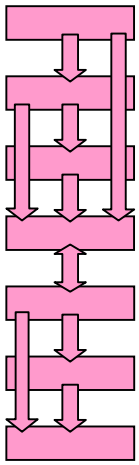
The cubeType{inputGamutAlarm} generates a cube where the output colour space is the same as the input space. Colours that would have transformed to out of gamut values are unchanged: all other colours are replaced with neutrals. This is more useful when we combine it with the invert{} command. Then, when you pass a display RGB image through the cube, the colours that are in the display gamut, but cannot be rendered on film will show up in colour. This is the equivalent of the gamut alarm for people who work in display RGB.

The cubeType{inputGamutClip} also generates a cube where the output colour space is the same as the input space. This cube simulates the effects of transforming to the output space, clipping, then transforming back again. It can be used with the invert{} command to simulate the effects of clipping to the film gamut when transforming from display RGB to Cineon.

The cubeOutFormula{} command operates on the cube output values after everything else. You can convert the 0-1 RGB values to legal video using cubeOutFormula{(#*219+16)/255}. This will operate on the image values and the false colours produced by the gamut alarm.

Command	Argument	Default	Notes
cubeInFormula{}	Run formula on list input values	No formula	See section 6.4 for more about formulae. The formula operates on the list values before the cube is made.
cubeInput{}	log, linear, or video	Log	See truelight.h for recognized spaces.
cubeType{}	gamutAlarm, gamutClip, videoGamutAlarm, videoGamutClip, inputGamutAlarm, inputGamutClip, or straight (default)	straight	See xform.h for recognized cube types.
invert{}	on or off	off	Make an inverse cube
cubeOutFormula{}	Run formula on cube output values	No formula	See section 6.4 for more about formulae. The formula operates on the cube values after everything else.

2.16 Cube Edit Options



Truelight supports colour editing on the cube output values. A straight Truelight cube will have output values in floating-point display RGB, ranged 0..1. These commands can be used to fine-tune the colour transform to remove any remaining colour shifts. These edits should be used with care. If the Truelight calibration has been done properly, then you should have a good match between the colours on the monitor and the film. If there is a significant difference, then you should look for the cause before using any edits.

A typical edit command might look like this…

```
editA{ in{123,150,115} out{120,150,110} r{80} e{4} /{255} }
```

This edit will map the input colour (123,150,115) onto the output colour (120,150,110). This edit will affect colours in the input colour space within a radius of 80 of the input colour. The optional scale argument /**{255}** ranges the input and output colours and the range values from 0..255 to 0..1. The optical eccentricity argument **e{4}** increases the range of the edit in the luminance direction: this is often useful to change all instances of a particular colour over a wide luminance range. These values are best set using some interactive user interface, such as the Truelight Viewer Edit RGB panel. The default scale is 1.0. The default radius is 0.5 after scaling. The default eccentricity is 1.0.

Do not try and make the radius too small. The tool is intended to produce long-ranged smooth edits over several cube intervals. If your radius is less than a cube interval you may get unexpected results. You can counter this by increasing the number of cube intervals (see section 2.17), but this will increase the cube calculation time..

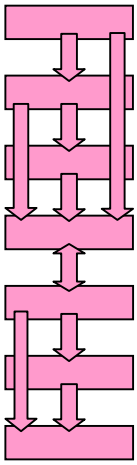
You can view the extent the edit as a greyscale by adding gamut{on} to an edit argument. Colours affected by the edit will appear light against a dark background.

There are 8 possible edits from **editA** to **editH**. The edits are applied in alphabetical order, stopping at the first edit with gamut{on}.

All these edits can be disabled using the command **editCube{off}**.

Command	Argument	Default	Notes
editA{}..editH{}	in{r,g,b} out{r,g,b} r{} scale{} gamut{on|off}	No edit	See above for argument defaults.
editCube{}	on or off	on	Can be used to turn all edits off without losing all the settings.

2.17 Cube Generation Options



Truelight generates a floating-point transform from input log Cineon to output display RGB. It uses this transform to make a cube transform, so it can process image data quickly. The input values are restricted to the input gamut, but the output values are floating point. For more details on cube transforms, see section 6.6.

Section 2.14 described the parameters that changed the cube transform. There is also a set of cube generation parameters that do not affect the transform itself, but modify how the cube does the transform. The average user will always use the default settings, but there are occasions when these parameters are useful.

We can vary the number of intervals per cube axis. You can make cubes with between 2 and 40 points per axis. The default value of 16 seems to work for most conditions. You can try increasing the number of levels for the badly behaved colour spaces such as the gamut alarm. More intervals can give better results and the interpolation is just as fast, but the cubes take longer to calculate. Larger cubes may not be compatible with Truelight hardware.

Intervals{0} is a special case. This gives a transform with input look-up tables but no cube. This is faster than the regular cube transform, but it is incapable of doing any 3-D correction. This can be used to generate look-up tables for conventional display hardware. It has mostly been used to show off the advantages of the full 3-D transform.

The cube generation includes a monotonic filter. In general, we would expect each output value to increase if we increased the corresponding input value. However, small experimental errors and/or interpolation overshoot can give values that go against the trend. The monotonic filter corrects these values. Usually, this is a good thing to do. However, you can turn off this filter for output spaces that are intentionally non-monotonic.

The cube generation has an option to linearize the cube. The input LUTs are warped to give cube intervals with uniform intervals along the diagonal in the output space. In general this gives smoother transforms, so it is enabled by default. However, the effects of following this with an arbitrary cubeLutFormula{} setting are not fully explored. The linearization can be disabled with linearizeCube{off}.

The fitGamut{} switch determines how the out of gamut cube points are generated. The traditional fitGamut{off} approach was to generate the output RGB values, and then clip them to the 0..1 gamut. The newer fitGamut{on} option projects the out of gamut points onto the gamut boundaries in L*a*b* space, preserving hue, and preserving brightness too where this is practical. This has been found to cure some subtle problems with colours close to the gamut limit, where the cube is interpolating between in gamut and out of gamut colours.

Truelight makes cubes with smooth input LUTs that give good results for perceptual colour spaces. Linear exposure is not a perceptually uniform colour space: a linear cube with evenly spaced intervals would have very few shadow tones. The cubeInput{} command (see section 2.1) makes a linear cube from a log cube using the fixed formulas of section 2.1. The cubeLutFormula{} command offers a more general solution without changing the transform..

Suppose you wanted to transform rendered images from linear display RGB to Cineon. You could make a cube with Truelight using a linear display calibration and with invert{on}. The cubeLutFormula{#^0.4} command makes a cube with gamma 0.4 input LUTs. This formula stretches out the shadows and compress the highlights in your linear display RGB, so the cube interpolation effectively happens in gamma 2.5 space. This gives the same transform, but the more perceptually uniform distribution of cube intervals gives smoother-looking results.

If you have video input, cubeLutFormula{(#*255-16)/219} generates a cube with the first interval at 16 and the last at 235 when working with 8-bit data. This is useful for legalized video data: it puts the first and last cube intervals at the gamut limits, and avoids wasting LUT range on the illegal values. See section 6.4 for more on formulas.

Because cubeLutFormula{} operates on the cube look-up tables, it can only use transforms with independent channels. If the cubeLutFormula{} argument contains some cross coupling between the channels, then Truelight will report an error.

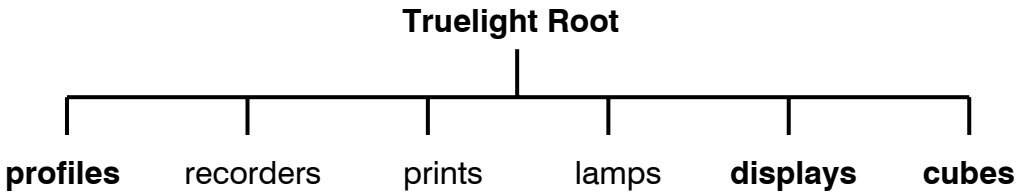
You can combine cubeLutFormula{}, linearizeCube{}, and cubeInput{}. If you combine them, this is the order in which they operate on the LUTs. Combining them is not harmful, but the results may be difficult to predict.

The last two commands are used for generating and exporting cube files to the Truelight hardware. They do not usually belong in a profile.

Command	Argument	Default	Notes
intervals{}	Number of cube intervals.	16	Special value 0 gives us a set of 1-D look-up tables.
lutLength{}	Cube input LUT length	101 (0-100%)	Special value 0 gives us a cube without input LUTs.
monotonicCube{}	on or off	on	Enables or disables monotonic filter
linearizeCube{}	on or off	on	Enables or disables LUT linearization
fitGamut{}	on or off	off	Project out of gamut points in L*a*b*
cubeLutFormula{}	Run formula on cube input LUTs	No formula	See section 6.4 for more about formulae. This moves the cube intervals, but does not change the underlying transform.
cubeHardware{}	8-bit, 10-bit, or none	None	Generate cube for Truelight hardware. See truelight.h for latest list of supported hardware options
cubeFile{}	Cube file name	None	May be restricted to hardware cubes. See section 3.3 for the default directory.

3 The Truelight Directories

The standard Truelight directory looks like this:



A Truelight installation should create all these directories. A fully licensed Truelight application may use all these directories. A Truelight application without a licence, but with an OEM developer key (see section 1.7) uses fixed built-in calibrations for everything but the display, so it will only use the directories in bold type.

Truelight files do not have to live in these directories. You can to refer to files in Truelight outside these directories by including the directory path with the file name. You can change the Truelight root directory. Some tasks can search non-standard directories using the TruelightBegin optional arguments (see section 3.4).Some operating systems allow symbolic links, where one file can point to another. In general, we recommend keeping all calibration in the usual default directories. This way, if you make a full copy of the Truelight root directory, you will save all the data you will need to get Truelight back to a given state.

Truelight instances have internal cubes, so there is usually no need to write cubes to files: the 'cubes' directory is needed for old and experimental software, or for Truelight hardware cubes

On a Unix workstation, 'Truelight' is normally **`/usr/fl/truelight`**. We can override this by defining the environment variable **`TRUELIGHT_ROOT`**. On an MS-Windows workstation, 'Truelight' is normally given by the registry variable **`TRUELIGHT_ROOT`** in **`HKEY_LOCAL_MACHINE`** in **`SOFTWARE\Filmlight\Truelight\`** in the appropriate key for the library version (see **`TL_VERSION`** in truelight.h). For backward compatibility, there is a default directory if the registry entry is not found (**`C:Program Files\FilmLight\truelight`**).

The TruelightHelp( ) library command prints out a lot of information about the current Truelight set-up. The current Truelight root directory can be found towards the bottom of this report. The ‘tl-utils’ utility prints out this message if you type ‘tl-utils Help’.

3.1 Internal Calibrations

Calibrations in external files are referred to by their filename. The Truelight library also contains some fixed calibrations, including the default calibrations that are used when no calibration of a particular type is specified. Sometimes we need to specify these internal calibrations by name. These transforms do not have a separate file, so they are called by their internal name with the prefix “internal-“. For example, you could use the internal Kodak2383 print calibration with the command `print{internal-Kodak2383}`.

The TruelightHelp( ) library command lists the internal calibrations in the current library. The ‘tl-utils’ utility prints out this message if you type ‘tl-utils Help’. This listing comes right at the bottom of the ‘Help’ message.

3.2 Updating Calibrations

When you create a new calibration or profile, the tool that creates the new file should check for a name clash, and take appropriate action. When creating a new **monitor** profile, for example, the monitor calibration tool should move the previous **displays**/**monitor** file to **displays**/**monitor.OLD**. Older copies are not necessarily backed up. The profiles that refer to **monitor** will probably want to refer to the latest calibration, so this is the right thing to do in this case.

When you install Truelight, you will also install a set of standard profiles and calibrations. This time, the emphasis is on preserving the old files. If you have used a profile with the line `recorder{arri}` on a job, you will not want the appearance of that job under Truelight to change because the **arri** calibration has been replaced. In this case, the installation too should keep the old version of the file with its original name, and rename the new version. If you already have a **recorders**/**arri** calibration, and the new one is different, you should see **recorders**/**arri.NEW** after an installation. When you do a Truelight installation or upgrade, check for these new files, and see if the differences are significant.

3.3 The Default Directory Entry

The Truelight library contains a set of hard-coded set of recorder, print, and lamp calibrations. If a particular recorder, print, or lamp calibration is not specified, one of the hard-coded calibrations is used instead. The default calibrations simulate output on an ArriLaser recorder loaded with Kodak 5242 stock, printing on Kodak Vision 2383, and projected using a generic Xenon lamp. These defaults are not suitable for accurate grading, but they should be enough to give a reasonable 'film-like' look. You will need to enter your aim gammas and densities.

There is no hard-coded display calibration. If a particular display is not specified, Truelight will look for a calibration called 'monitor' in the displays directory. A freshly installed Truelight system does not come with a default display calibration. The display calibration has to be created on site by calibrating the display.

3.4 Truelight Directory Listing

When Truelight starts up, it makes lists of the calibration and profile directories. These lists contain the full addresses of the files. These lists are intended to support simple pull-down menus.

Truelight normally lists the standard directories (see section 3), followed by the internal calibrations (see section 3.1). You can change the search paths by passing a string argument to the TruelightBegin( ) function (see section 5.1). This string should contain a set of name{value} pairs. The recognised options are…

Argument	Action
name{..}	OEM developer name (see section 1.7)
key{..}	OEM developer key (see section 1.7)
profiles{..}	List this directory instead of the usual 'profiles' directory
recorders{..}	List this directory instead of the usual 'recorders' directory
prints{..}	List this directory instead of the usual 'prints' directory
lamps{..}	List this directory instead of the usual 'lamps' directory
displays{..}	List this directory instead of the usual 'displays' directory
cubes{..}	Write to this directory instead of the usual "cubes" directory

The directory name must end with the filename separation character appropriate for the system.

The list entries will contain just the filename if the file is in the default directory. Entries from other directories will contain the full directory address followed by the filename. Internal library entries will have the calibration prefixed with “internal-“. We can have local, networked, and internal calibrations with the same name.

This allows us to support a user interface for each calibration type using a pull-down menu. The entries on the pull-down menu may need editing if the full directory addresses are too long. The menu might look a bit neater if the internal calibration “internal-“ prefix was replaced by a change in type colour or something similar.

If this user interface is felt to be too restrictive, then we can use a full browser interface instead. The Truelight library has support for filtering lists of files, so you can make the browser list only recorder calibrations, only print calibrations, and so on.

4 The User Interface

Truelight has a great many parameters. All of them have their uses. However, many of these parameters - choice of film recorder, choice of film stock, etc - remain fixed for an entire project. Even the 'printer light' parameters we used in the example will only be needed for a few grading seats in well blacked-out rooms with a projector.

Sometimes, inexperienced users may mistake a Truelight node in a grader for a retouch tool. They may have an image with an overall yellow colour cast. They change the yellow exposure boost setting (see section 0), and the yellow cast goes away. They get the image printed, and they see the yellow cast has 'come back'.

The art is to give the users only the controls they need. The house colour expert can set up the other settings -the ones that do not change very often, and do not need interactive controls - in a text file. Optional controls can be hidden, unless some 'advanced controls' option is selected. Some of these options are only available to fully licensed Truelight users (see section 1.7). Here are suggestions for the visibility of the Truelight parameters…

Command	Notes	Value set by…
profile{}	This loads a fixed set of settings that have been previously prepared by an expert.	All users
printerPoints{} brightness{} shift_a{} shift_b{} displayFlare{} cubeType{gamutAlarm} cubeInput{log|linear}	These controls match the simulation to the local viewing and print conditions. These settings are typically short-lived, and do not belong in a profile.	All users
invert{} + cubeType{inputGamutAlarm} cubeType{inputGamutClip}	These are used to simulate film output when working in display RGB (see section 2.13). The user interface should offer the choice of "gamut alarm" and "gamut clip", and set these parameters for the user	Display RGB users
cubeHardware{} cubeFile	The user interface should offer the hardware option, and set these parameters for the user.	Hardware users
white_Y white_u white_v	Small shifts in these controls can look like printer light settings. Any potential user must be trusted to understand the difference. The default D65 white point settings are good enough for most office environment work.	Higher-end users. Other users as option
recorder{} print{} lamp{} display{}	These should be set in the profile. A high-end user grading to a projection reference may also want to swap their display or lamp while keeping the other controls.	High-end users with full Truelight licence (see section 1.7 for details on licensing)
greyValue{} aimGamma{} whiteValue{}	The house expert should define these parameters in profiles. They will rarely change.	Expert
All other commands	Any command is available to any user that can edit a profile file. A user interface could support any command with a command line interface.	All users

4.1 Different ways to set Truelight Parameters

There are several ways to set a parameter in Truelight:

1. Pass the value in using a specialised Truelight library function. This is the sensible route for UI controls. Some parameters do not have specialised functions. See section 5.2 for details.
2. Pass the setting as a command string using a general Truelight library function. The same function can set all the Truelight parameters. See section 1.5 for more on commands.
3. Put the setting in a profile. Profiles are user editable files with one or more command strings. This is a good way for the house colour expert to set all the values that rarely change. See section 1.6 for more on profiles. The Truelight viewer is a good tool for editing profiles because it checks the command syntax and reports any errors, and it gives a visual check if you load an image.
4. Use the default value. Most parameters have a valid default value.

If you set a parameter more than once, the later setting will override the earlier one.

Some parameters are only available to fully licensed Truelight users (see section 1.7 for details on licensing). Setting an unavailable parameter may give an error. You can get the definitive list of the available parameter setting commands for the current library using the TruelightHelp( ) library command. The ‘tl-utils’ utility prints out this message if you type ‘tl-utils Help’.

5 Using the Truelight Library

This document was first written for developers using the Truelight library. If you are not a developer, you can probably skip this section.

Many Truelight library routines return an integer value. Unless stated otherwise, the value 1 signifies success and the value 0 signifies an error. You can get more details about errors in a string by calling…

```
char *TruelightGetErrorString(void);
```

There are some combinations of Truelight commands that do not cause an error, and yet may not yield sensible data. If you have a Truelight instance (see section 5.1 for how to create an instance) then you can get a simple one-line warning message by calling…

```
const char *TruelightInstanceGetWarning(void *instance);
```

If there are several warning messages, then the most severe one is returned. If no warning is needed, then it returns a NULL. This is useful for short error prompts in user interfaces. You can get a longer description of the current settings with…

```
int TruelightInstanceWriteNotes(void *instance, const char *filename);
```

This message covers several lines, and is written out to a named file, or to the Truelight standard output if you supply a NULL filename pointer. This does not repeat any of the warning or error messages covered by the other commands.

The Truelight library has its own standard output streams for output and error messages. Most The Truelight library also has its own standard output streams for output and error messages. Most library routines do not use these, but they can be handy for logging, debugging or error messages. On some systems, they default to **stdout** and **stderr**; on others they default to **null**. You can set the streams by calling…

```
void TruelightSetStandardOut(FILE *);
void TruelightSetStandardErr(FILE *);
```

The Truelight library can send a lengthy string describing itself, and the commands it supports, with…

```
void TruelightHelp(FILE *);
```

This string will depend on the licences and the default settings. If you want to get the current listing, call it after calling TruelightBegin( ) (see section 5.1).

5.1 Starting up Truelight

The first library call should be…

```
int TruelightBegin(const char *args);
```

This initialises any global data used by the libraries. It checks for licences and keys (see section 1.7). It also sets up lists of all the calibrations and profiles in the standard directories (see section 3.4).

You can stop and restart Truelight to refresh these lists without affecting the current Truelight instances. You can also force a re-listing of the standard directories with…

```
int TruelightRestart(const char *args);
```

However, if your files have changed, it might be safer to restart the application.

These lists are useful when making pull-down menus listing the choices. You can get the number of entries in each list, and the full filename of each entry using the following commands…

```
int TruelightGetNProfiles(void);
char *TruelightGetProfileName(int n);
int TruelightGetNRecorders(void);
char *TruelightGetRecorderName(int n);
int TruelightGetNPrints(void);
char *TruelightGetPrintName(int n);
int TruelightGetNLamps(void);
char *TruelightGetLampName(int n);
int TruelightGetNDisplays(void);
char *TruelightGetDisplayName(int n);
```

You can get the Truelight root directory (see section 3) with…

```
char *TruelightGetRoot(void);
```

Before you can set any of the parameters, you have to create a Truelight instance…

```
void *TruelightCreateInstance(void);
```

You can have more than one Truelight instance per application.

The default entry (if there is one) is no longer entry zero in the lists. If you need to get the default entry, see section you will have to create an instance, and then set the default entry first. You can get the default print name, for example, using some of the commands from section 5.2…

```
void * instance;
char * defaultPrint;
instance = TruelightCreateInstance();
TruelightSetPrint(instance, NULL);              /* ß This sets the default print calibration */
defaultPrint = TruelightGetPrint(instance);     /* ß This returns the curent print cal. name */
```

5.2 Setting the parameters

When you have a Truelight instance, you can set the parameters. There are two ways of doing this. You can set any of the parameters with a general command that parses a string:

```
int TruelightInstanceSetParameter (void *instance, char *str);
```

When Truelight instance reads a profile, it calls TruelightInstanceSetParameter( ) once for each command. See section 1.5 for details on Truelight commands.

Some parameters have special commands to set the parameter directly…

```
int TruelightInstanceSetProfile(void *instance, char *profile);
int TruelightInstanceSetRecorder(void *instance, char *deviceName);
int TruelightInstanceSetPrint(void *instance, char *deviceName);
int TruelightInstanceSetLamp(void *instance, char *deviceName);
int TruelightInstanceSetDisplay(void *instance, char *deviceName);
int TruelightInstanceSetPrinterPoints(void *instance, float lights[3]);
int TruelightInstanceSetFlareCorrection(void *instance, float flare);
int TruelightInstanceSetWhiteY(void *instance, float Y);
int TruelightInstanceSetWhiteU(void *instance, float U);
int TruelightInstanceSetWhiteV(void *instance, float V);
int TruelightInstanceSetBrightness(void *instance, float brightness);
int TruelightInstanceSetContrast(void *instance, float contrast);
int TruelightInstanceSetSaturation(void *instance, float saturation);
int TruelightInstanceSetCubeType(void *instance, int cubeType);
int TruelightInstanceSetInMax(void *instance, int max);
int TruelightInstanceSetOutMax(void *instance, int max);
int TruelightInstanceSetMax(void *instance, int max);
int TruelightInstanceSetMax(void *instance, int max);
int TruelightInstanceSetCubeInput(void *instance, int cubeInput);
int TruelightInstanceSetInvertFlag(void *instance, int invertFlag);
```

Each of these commands has a similar command that gets the current parameter value rather than setting it. The names are similar, but with **Get** instead of **Set**. The 'truelight.h' for details. These are used to find the current values of parameters to make user controls appear with the correct settings.

TruelightSetProfile( ) keeps the local viewing condition parameters (see table in section 4), and resets all the others to the default values. It then reads in and executes the commands in the profile. This allows users to swap their profiles but keep their printerPoints{} and other local settings.

Many Truelight library commands have equivalent string commands:

Library Command	Equivalent Truelight Command
TruelightInstanceSetProfile( )	profile{*file*}
TruelightInstanceSetRecorder( )	recorder{*file*}
TruelightInstanceSetPrint( )	print{*file*}
TruelightInstanceSetLamp( )	lamp{*file*}
TruelightInstanceSetDisplay( )	display{*file*}
TruelightInstanceSetPrinterPoints( )	printerPoints{*r.g.b*}
TruelightInstanceSetDisplayFlare( )	displayFlare{*value*}
TruelightInstanceSetWhiteY( )	white_Y{*value*}
TruelightInstanceSetWhiteU( )	white_u{*value*}
TruelightInstanceSetWhiteV( )	white_v{*value*}
TruelightInstanceSetBrightness( )	brightness{*value*}
TruelightInstanceSetContrast( )	contrast{*value*}
TruelightInstanceSetSaturation( )	saturation{*value*}
TruelightInstanceSetCubeType( )	cubeType{straight | gamutAlarm | inputGamutAlarm |…}
TruelightInstanceSetMax( ) etc	none (see section 5.4)
TruelightInstanceSetCubeInput( )	cubeInput{log | linear | video}
TruelightInstanceSetInvertFlag()	invert{on | off}

5.3 Checking the Parameters

You can turn the current settings back into a list of commands with…

```
char *TruelightInstanceGetCommands(void *instance, const char *delimeter);
```

The delimiter string is put after each command. Profiles are often made using a whitespace delimiter such as “\n”. Commands that have been overwritten, commands that can be defaulted, and most commands that do not affect the transform are not output. The command string is owned by the instance, and is destructed with it. This command is used in the Truelight applications when we save the current commands as a profile.

If two instances give the same command string, they should give the same transform. The Truelight library keeps a copy of this string when a cube is made, and uses it to check whether it is necessary to remake the cube later.

The command string does not tell us everything about the Truelight instance. If you want to check the instance has got to a particular state, try…

```
unsigned long TruelightInstanceChecksum (void *instance);
```

This returns a deep checksum over the instance. If two instances return the same checksum, it is very likely (though not certain) that they are in the same state.

You can check whether a command is used with…

```
int TruelightInstanceUnusedCommand(void *instance, const char *command);
```

This returns 0 if command{...} would affect the instance, and 1 if it doesn't. The command should contain just the command name, no spaces, and no brackets. The Truelight viewer uses this so it can grey out unused controls and command lines. The transform does not need to be set up (see section 5.5) to use this command.

5.4 Fixing the Cube Data Precision

The cube structure has a set of floating-point look-up and cube tables. The ranges of the input and output values are mapped onto the range of the input LUTs and the floating point range 0.0 to 1.0.

This is a precision-independent version of the cube. The floating-point values can use these values directly. Before we can apply the cube efficiently to fixed-point data, we must generate a second set of cube data fitted to our particular input and output data precisions. If we have 10-bit Cineon data, we need a set of input tables that accept values between 0 and 1023. If we have 8-bit unsigned output data, we need a set of cube table values that are scaled and clipped to range from 0 to 255. At the same time, it is convenient to offset and bit-shift the look-up tables to make the interpolation faster.

The cube library supports reading and writing of precision-independent cube data, and run-time fitting of cubes to particular input and output data precisions. The Truelight library supports one set of input and output data precisions per instance. The input and the output have the same data precision when set by TruelightInstanceSetMax( ). Here are some typical values…

TruelightInstanceSetMax(instance, 255)	Unsigned byte data
TruelightInstanceSetMax(instance, 1023)	Unsigned 10-bit data
TruelightInstanceSetMax(instance, 65535)	Unsigned 16-bit data
TruelightInstanceSetMax(instance, 1)	Float data (default)

TruelightInstanceSetInMax( ) and TruelightInstanceSetOutMax( ) set the input and output data precisions independently. TruelightInstanceGetInMax( ) and TruelightInstanceGetOutMax( ) return the current values.

5.5 Setting up the Transform

When the parameters have all been set, we set up the instance

```
int TruelightInstanceSetUp(void *instance);
```

Every time TruelightInstanceSetUp( ) is called, it reads the calibration files and generates a fresh cube transform. This may take several seconds for large cubes. It may fail to generate a cube if there are errors in the files or the parameters

5.6 Using the Transform

The instance error flag will be zero if the current transform data is ready to use, and non-zero if the data is uninitialized or corrupted, or otherwise unusable. See truelight.h for a listing of the other possible states. The error flag can be read using…

```
int TruelightInstanceGetErrorFlag(void *instance);
```

Truelight also has an error string which is used to report errors in the Truelight library or in any current instance. You can get the error string using…

```
char *TruelightGetErrorString(void);
```

This is used to generate most of the Truelight error messages in the applications. Once you have used the error message, you will probably want to clear the error string using…

```
void TruelightResetErrorString(void);
```

Single-pixel integer and float data can be transformed using…

```
void TruelightInstanceTransformI(void *instance, int rgb[3]);
void TruelightInstanceTransformF(void *instance, float rgb[3]);
```

For speed, these routines have no checks on the integrity of the transform data: it is wise to check the instance error flag before using these transform routines. The integer transform routine contains no check that the input data lies in the range 0 to cubeMax (see section 5.4). If there is a risk that your data may go outside this range, you will have to clip it or risk an address error.

If you have access to the cube libraries, you can get at the cube data using…

```
void *TruelightInstanceGetCubeHeader(void *instance);
char *TruelightInstanceGetSerialCube(void *instance, int nscale);
```

5.7 Closing Down

When you have finished transforming, free up the cube and other data using…

```
void TruelightInstanceSetDown(void *instance);
```

The parameters are still set. You can recreate the cube by calling TruelightInstanceSetUp() again. To destroy the instance and all its parameters, you call…

```
void TruelightDestroyInstance(void *instance);
```

If you have not already called TruelightInstanceSetDown(), it will be set down for you. Finally, you can shut down Truelight, and free up all the lists using…

```
void TruelightEnd(void);
```

This does not affect the data in the Truelight instances. These will have to be freed separately.

6 Truelight Calibration Files

A major feature of Truelight is that any user can make and edit any of the calibrations using only conventional text processing tools. This can allow advanced users to adapt the Truelight calibration to their particular print process or display.

6.1 The Calibration Header

Here is a header from a recorder calibration…

```
xfm{
  in{
    type{ 3D }
    space{ RGB:Cineon }
    range{ #/1023 }
  }
  out{
    type{ 3D }
    space{ statusM }
  }
}
```

The syntax is a nested series of **name{value}** pairs. The names can have leading and trailing whitespace, but they cannot contain whitespace or curly brackets. The values can contain whitespace and curly brackets, provided the curly brackets nest properly. The indentation is purely to make the file easy to read.

The whole header is enclosed by **xfm{..}**. The first four bytes of a calibration file should be **xfm{**. Although any leading whitespace will be ignored, it is good style to start all calibration files with the same four bytes.

The header must contain an **in{..}** and **out{..}** entry for the input and output spaces. You can add other entries if you like.

6.2 Colour Space Entries

The **in{..}** and **out{..}** entries must both contain a **type{..}** entry which specifies the type of colour data. The type can be **string** for colour names, or **1D**, **2D**, **3D**, etc for a set of floating-point values.

All other entries are optional.

The **space{..}** entry is a string that describes the colour space. It is useful to test the data types agree before operations such as combining two colour transforms. When comparing two **space{..}** strings, leading and trailing whitespace is ignored. If the two strings are the same, or one is a subset of the other, then the colour spaces are assumed to match. Thus an **RGB:Cineon** space will match **RGB**, or **RGB:Cineon(0-1023)** but it will not match **RGB:video**. If the **space{..}** strings do not match, then a warning or error may be generated.

The **range{..}** entry contains a brief formula that will range in-gamut input values 0-1. The input **range{ #/1023 }** entry is appropriate for 10-bit unsigned Cineon data. The output data is status M density, which has no fixed range, so there is no output **range{..}** entry. See section 1.3 for more about formulas.

You can add other entries if you like. This can be useful for adding comments, or extra parameters that are not used within Truelight.

6.3 List transforms

In section 1.2 we plotted out Cineon and Status M density information for a typical recorder calibration. The header for this calibration appears in section 6.1. After the closing bracket of the **xfm{..}** block the calibration continues like this…

```
list{
  52.86  52.50  49.14  0.0050 0.0050 -0.002
 169.43  52.50  49.14  0.1960 0.0150  0.002
 287.14  52.50  49.14  0.3980 0.0340  0.011
 407.71  52.50  49.14  0.6040 0.0500  0.019
 526.00  52.50  49.14  0.8050 0.0600  0.025
 643.14  52.50  49.14  1.0160 0.0670  0.030
 760.29  52.50  49.14  1.2360 0.0760  0.038
 882.57  52.50  49.14  1.4660 0.0750  0.042
 998.57  52.50  49.14  1.7100 0.0860  0.051
  52.86 174.00  49.14  0.0180 0.2300 -0.007
 169.43 174.00  49.14  0.2100 0.2410 -0.001

 ( many lines omitted )

}
```

Each line contains a set of three floating-point values for the 3D input space, followed by a further three floating-point values for the 3D output space. The lines are in no particular order, and the points do not have to lie on a regular grid in the colour space.

The Truelight library can fit an interpolation function through a random set of points in up to four dimensions. If we had a set of RGB Cineon values, we could interpolate the corresponding RGB status M density values. If the points in our list surround our point, then the interpolation will probably be accurate. If the point lies outside the range of our list data, then the extrapolation may not be accurate.

The interpolation also assumes that our values are in a 'well-behaved' colour space. It is hard to define a 'well-behaved' space. Generally, it means that colours that look like each other are numerically close to each other. It is perhaps easier to provide a counter-example: HSV (Hue, Saturation, and Value) is a 'badly-behaved' space. The hue is described by an angle between 0 and 360 degrees. The colour (0,100,50) is a bright red. The colour (359,100,500) is an almost identical bright red. The colour (0,0,50) is the same grey as (180,0,50). The saturation and hue of black is undefined. These uncertainties make interpolation in HSV very difficult.

You can invert a list transform. If you swap the contents of the **in{..}** and **out{..}** entries, and you swap around the entries in the **list{..}** data, then you will have a list transform that will convert from status M to Cineon.

Not all inverse transforms are meaningful. Suppose you recorded your image on black and white film. The status M densities would all be neutrals. The forward transform would convert colour Cineon image values into neutral densities. The reverse transform would probably not be able to put colour back into the data in any meaningful way.

The input data in our example has a 10-bit range. The transform **in{..}** might look like this…

```
in{
  type{ 3D }
  space { RGB:Cineon }
  range{ #/1023 }
}
```

Before the library uses the input values, the 0-1023 range is mapped onto the floating-point range 0-1 using the range formula. This is a convenient tool that allows us to use the familiar 10-bit Cineon values in our list. Our list could contain 8-bit values and a different range formula, or floating point values with no range setting, and the result would be the same.

The values in the list can go outside the range implied by the range formula. Tools such as the gamut alarm (see section 0) are made to detect these out-of-range values.

The **out{..}** can also have a **range{..}** argument. This works in the same was for the output values.

6.4 Formula Transforms

Formula transforms complement list transforms. They can be precise, even with poorly behaved spaces like HSV; but they cannot be inverted automatically, and they are harder to use with experimental data.

Here is the formula transform for converting from CIE L*a*b* to ITU rec709 video RGB with a D65 white point. The interesting bit is contained in the **formula{..}** entry…

```
xfm{
in{
type{ 3D }
space{ L*a*b* }
formula{
  /* Input is L*a*b* relative to D65 white point */
  L = #0; a = #1; b = #2;

  fY = (L+16.0)/116.0;
  fX = fY + a/500.0;
  fZ = fY - b/200.0;

  X = (fX < 24.0/116.0) ? (fX - 16.0/116.0)/7.787 : fX*fX*fX * 0.9505;
  Y = (fY < 24.0/116.0) ? (fY - 16.0/116.0)/7.787 : fY*fY*fY;
  Z = (fZ < 24.0/116.0) ? (fZ - 16.0/116.0)/7.787 : fZ*fZ*fZ * 1.089;

  /* Convert to R-709 primaries */
  R =  3.240479*X - 1.537150*Y - 0.498535*Z;
  G = -0.969256*X + 1.875992*Y + 0.041556*Z;
  B =  0.055648*X - 0.204043*Y + 1.057311*Z;

  /* R-709 gamma */
  r = R<0.018 ? 4.5*R : 1.099*R^0.45 - 0.099;
  g = G<0.018 ? 4.5*G : 1.099*G^0.45 - 0.099;
  b = B<0.018 ? 4.5*B : 1.099*B^0.45 - 0.099;

  /* Output... */
  r*255,g*255,b*255
}
}
out{
type{ 3D }
space{ RGB }
range{ #/255 }
}
}
```

In our experience, the ITU rec-709 formula does not look much like real video monitors. If you want to convert to video, we recommend using the Truelight SonyHD transform. Here we are just using the ITU rec-709 standard to illustrate how a formula works.

A full and rigorous description of the formula language is beyond the scope of this note. If you are familiar with the ‘C” language, then you should be able to read most of the example. We shall just describe the places where the syntax deviates from ‘C’.

There is no need to declare the variables. They are all assumed to be floats. Integer operations such as modular division are not supported. There should be no need for them as we are trying to describe continuous colour transforms. Many other 'C' operators are not supported. The only comparisons are '<' and '>'. The Boolean operators are 'and' 'or;' and 'not'. '^' is the raise-to-the power operator. There are no bitwise operations.

The three L*a*b* input values are #0, #1, and #2. When the expressions ending in ';' have been evaluated and removed, we end up with three comma-separated values. These are our RGB output values. For clarity, we usually put all the commas at the end of a long formula, after all the semicolons.

If we are doing the same operation on all the input channels, we can just use the single value #. The range formula "#/1023" in section 6.2 divides all three channels by 1023.

The maximum numbers of dimensions in or out is limited to TR_NDIMS defined in xformdef.h. This can be greater than four. We have used up to 101 dimensions to handle spectroscopic data.

The scientific functions include sqrt, exp, min, max, log10, ln, sin, cos, tan, asin, acos, atan, and pi. The trig functions work with angles in degrees.

The pipe operator ## is used to pipe the output of one function into the input of the next. The Truelight software uses it to combine two formula transforms. It is not intended for programmers.

The 'index' and 'value' functions linearly interpolate values via a list. This is used in the monitor calibrations (see section 6.5).

The formula transform works in two passes. The first pass evaluates everything it can with the input # parameters undefined. This reduces the overhead for calculating any constant values.

A transform can have a **formula{..}** entry in its **out{..}** field. If you put the inverse transform formula in here, then the transform can be inverted, for when a transform is inverted, the contents of **out{..}** and **in{..}** are swapped. However, there is no check that the reverse formula actually undoes the forward formula.

The example has an output **range{..}**. The formula generates values ranged 0-255, and the range formula will re-range these values to 0-1.

This example has not got an input **range{..}**. However, if it had, the formula would start off with the values after the input **range{..}**. If we were taking in RGB values, then they would be ranged 0-1. This may seem strange. However consider merging two transforms, called A and B. The combined formula for the forward transform, where **##** is the pipe operator, would be…

```
input formula A ## output range formula A ## input formula B
```

However, if our formula input values had not been ranged, it would have to be…

```
input formula A ## output range formula A ## inverse of input range formula B ## input formula B
```

We cannot do this unless we supply inverses for all our range formulae. The range formulae do not work as neatly for formula transforms as they do for lists. However, we can still make formulae that generate values with familiar looking ranges.

6.5 Display Transforms

Most calibrations last for a long time. Monitor calibrations are exceptions: most users may need to calibrate their monitor once a month or more. The Truelight library provides support for generating or updating monitor formula transforms.

There are a few more function not mentioned in section 5…

```
int TruelightMoncalSubstitution(FILE *in, FILE *out, const char *table);
```

in is a monitor calibration or a calibration template, opened for reading. **out** is a new file opened for writing, 'table' is a string containing **name{value}** pairs. Here are some lines for the red channel in monitor calibration template…

```
/* moncal:ri = index(R,${ri}); */
replace this line
/* moncal:rv = value(ri,${rv}); */
replace this line
```

This would not work as a calibration - the "replace this line" bits would force an error in the formula interpreter. However, this file was passed through TruelightMoncalSubstitution( ), and the string in 'table' contained the following **name{value}** pairs…

```
ri{-17.80,-16.89,-15.98,-15.07,-14.16,-13.27 … -4.95,-4.61,-4.36)
rv{0,5,10,15,20,25,30,35,40,45,50,60,70,80 … 220,240,255)
```

TruelightMoncalSubstitution( ) reads the input file, line by line. Normally it just echoes the lines to the output. However, when it finds a line of the form…

```
/* moncal: <string 1> ${name} <string 2> */
```

…it then replaces the following line with...

```
<string 1> value <string 2>
```

It returns the number of substitutions. The output would then look like this…

```
/* moncal:ri = index(R,${ri}); */
ri = index(Rh,-17.80,-16.89,-15.98,-15.07,-14.16,-13.27 … -4.95,-4.61,-4.36);
/* moncal:rv = value(ri,${rv}); */
rv = value(ri,0,5,10,15,20,25,30,35,40,45,50,60,70,80 … 220,240,255);
```

The formula interpreter can read this. The **index** function looks up the **R** value in its list, returning 0.0 if **R** was -17.80, and 1.0 if **R** was -4.36. The **value** function looks up the **ri** value in its list the other way, returning 0 if **ri** was 0.0, and 255 if **ri** was 1.0. We have interpolated the red value through a measured tone curve.

If the application can set parameters, it may also want to read the old parameters from the file. This is possible with…

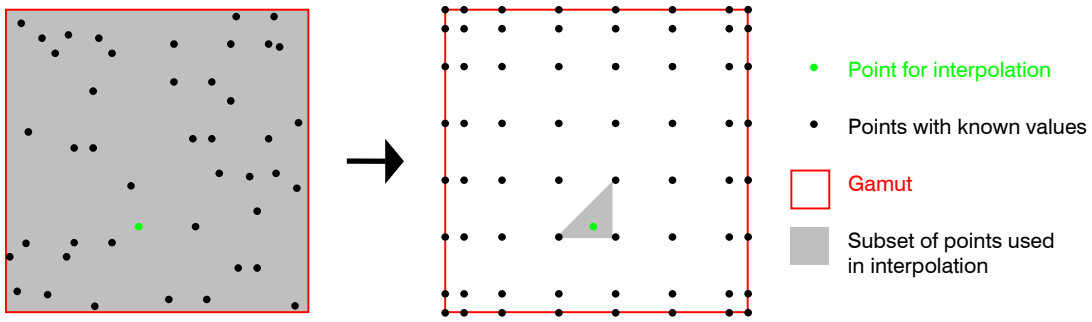
```
int TruelightMoncalRetrieveString(char *out, int l, FILE *in, const char *name);
int TruelightMoncalRetrieveFloat(float *out, FILE *in, const char *name);
```

The routines return 1 if a value is found, and zero if no match is found.

6.6 Colour Cube Transforms

The general list and formula transforms in section 6.3 and 6.4 can convert from one colour space to another, but they are not fast enough for a typical film recorder. Before we can transform millions of pixels of data, we must turn our list or formula transform into a cube transform.

The cube transform interpolates between a series of points lying in neat rows and columns going up the gamut limits. A typical three-dimensional colour cube will have 16 intervals per axis, or about four thousand sets of values. This cube may have more points than the list transform that made it, but the cube transform is faster because we only need a few values for our interpolation…



List transform Colour cube transform

The Truelight instance contains a cube transform structure. When we set up a Truelight instance (see section 5.5), Truelight generates the whole transform from input log values to output display RGB, then interpolates a colour cube from this transform. When we use the Truelight transform routines (see section 5.6), we are using this cube transform.

A typical colour cube has three input dimensions. Let us call them **X**, **Y**, and **Z**.

We can also have any number of output dimensions. For the moment we shall only consider one dimension **S**. If we can interpolate one output dimension, then we can interpolate others the same way.

We have a regular array of known values of **S** for a fixed set of intervals in **X**, **Y**, and **Z**. In 'C' we can write this array like this…

```
float S[Nx][Ny][Nz];
```

…where **Nx**, **Ny**, and **Nz** are the number of intervals in each axis (typically 16).

Let us consider the X-axis. We have **Nx** levels on this axis where we know the output values. We also have an input value **x**. We could calculate which two of our **Nx** levels are on either side of our **x** value. However, if **x** is an integer, as it probably is, it is a lot easier to look up the values in a table…

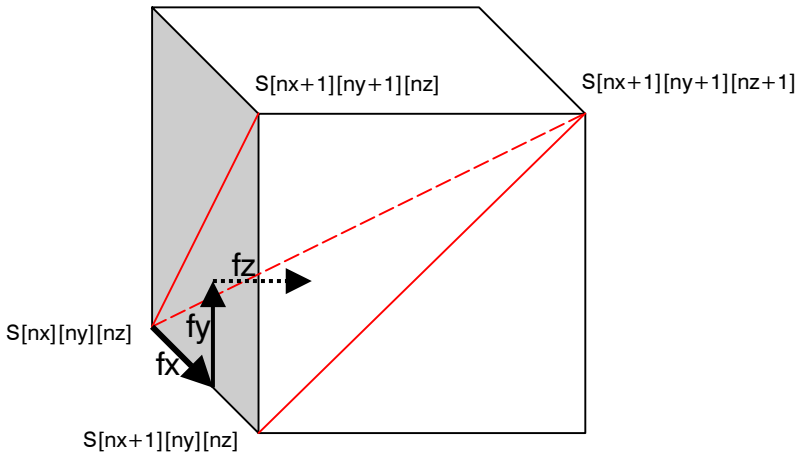
```
int   nx = xLevel[x];          /* 0   <= nx <  Nx  */
float fx = xFraction[x];       /* 0.0 <= fx <= 1.0 */
```

Our **x** value lies on or between level **nx** and level (**nx**+1). **fx** is the fraction of an interval. **fx** has extreme values of 0.0 if **x** lies on level **nx**, and 1.0 if it lies on level (**nx**+1).

Our point will lie in or on a smaller cube bounded by level **nx** and level (**nx**+1), level **ny** and level (**ny**+1), and level **nz** and level (**nz**+1). We then sort our fractional values **fx**..**fz**. There are 3*2*1 = 6 possible rankings of these three values.

Let us consider the ranking **fx** >= **fy** >= **fz**.

The region within our small cube with **fx** >= **fy** >= **fz** is a distorted tetrahedron shape like this…



This tetrahedral shape has three sides parallel to the cube edges: one parallel to the X-axis, one parallel to the Y-axis, and one parallel to the Z-axis. Each of these edges has a known value of **S** at either end, so we can calculate the gradient of **S** in the three perpendicular directions. If we have the value in the first corner, and the gradient in three perpendicular directions, it is easy to calculate the value at our point (**fx**, **fy**, **fz**). If we do the interpolation, we get…

```
Sxyz =   ( 1-fx)* S[ nx ][ ny ][ nz ]
       + (fx-fy)* S[nx+1][ ny ][ nz ]
       + (fy-fz)* S[nx+1][ny+1][ nz ]
       + (fz-0 )* S[nx+1][ny+1][nz+1];
```

Remember: this formula only applies for the ranking **fx** >= **fy** >= **fz**.. We can get the corresponding formulae for the other rankings by swapping around our **x**, **y**, and **z**.

A simple non-optimised version of the tetrahedral interpolation algorithm in 'C' might look something like this…

```
float tetrahedronInterpolation(int x, int y, int z)
{
   int nx=xLevel[x], ny=yLevel[y], nz=zLevel[z];
   float fx=xFraction[x], fy=yFraction[y], fz=zFraction[z];
   float Sxyz
   if (fx > fy) {
     if (fy > fz) {
        Sxyz =  (1-fx)* S[ nx ][ ny ][ nz ]
             + (fx-fy)* S[nx+1][ ny ][ nz ]
             + (fy-fz)* S[nx+1][ny+1][ nz ]
             +    (fz)* S[nx+1][ny+1][nz+1];
     } else if (fx > fz) {
        Sxyz =  (1-fx)* S[ nx ][ ny ][ nz ]
             + (fx-fz)* S[nx+1][ ny ][ nz ]
             + (fz-fy)* S[nx+1][ ny ][nz+1]
             +    (fy)* S[nx+1][ny+1][nz+1];
     } else {
        Sxyz =  (1-fz)* S[ nx ][ ny ][ nz ]
             + (fz-fx)* S[ nx ][ ny ][nz+1]
             + (fx-fy)* S[nx+1][ ny ][nz+1]
             +    (fy)* S[nx+1][ny+1][nz+1];
     }
   } else {
     if (fz > fy) {
        Sxyz =  (1-fz)* S[ nx ][ ny ][ nz ]
             + (fz-fy)* S[ nx ][ ny ][nz+1]
             + (fy-fx)* S[ nx ][ny+1][nz+1]
             +    (fx)* S[nx+1][ny+1][nz+1];
     } else if (fz > fx) {
        Sxyz =  (1-fy)* S[ nx ][ ny ][ nz ]
             + (fy-fz)* S[ nx ][ny+1][ nz ]
             + (fz-fx)* S[ nx ][ny+1][nz+1]
             +    (fx)* S[nx+1][ny+1][nz+1];
     } else {
        Sxyz =  (1-fy)* S[ nx ][ ny ][ nz ]
             + (fy-fx)* S[ nx ][ny+1][ nz ]
             + (fx-fz)* S[nx+1][ny+1][ nz ]
             +    (fz)* S[nx+1][ny+1][nz+1];
     }
   }
   return Sxyz;
}
```

6.7 Cube File Format

```
# Truelight Cube v2.0
# lutLength 101
# iDims     3
# oDims     3
# width     16 16 16

# InputLUT
 0.000000 0.000000 0.000000
 0.096929 0.490357 0.092795
 ( 97 lines deleted )
 14.969607 14.985015 14.972384
 15.000000 15.000000 15.000000

# Cube
 0 0 0
 0.0184206 0 0
 0.0542529 0 0
 ( 4092 lines deleted )
 0.920355 0.985279 0.997871
 0.994231 0.995776 0.996806
 1 1 1

# end
```

Truelight can read and write cubes as a human-readable ASCII file. Here is an example…

The lines beginning with '#' should not have any spaces or tabs before this character.

The first line identifies the file as a Truelight cube.

The next few lines define some necessary parameters…

- 'lutLength' is the length of the input LUT.
- 'iDIms' is the number of input dimensions (1-4)
- 'oDims' is the number of output dimensions (1-4)
- 'width' gives the size of the cube in each input dimension. The Truelight hardware cannot handle widths larger than 16, but other render options use larger cubes.

These lines can be in any order, provided they come before the two data sections.

'# InputLUT' is followed by the input LUT data. This is iDims * lutLength floating point entries which map input values ranged 0-1 onto cube intervals and fractions.

'# Cube' is followed by the output values ranged 0-1 for the 16x16x16 sets of RGB cube values. The first channel index increments the quickest.

The file format is flexible. The formatting of the data sections is ignored provided the values are in the right order. You can add extra lines anywhere but on the first line or in the middle of a data section – if they cannot be interpreted they are ignored. Truelight may stick extra lines at the end.

The input LUT values should be monotonic increasing, but the values do not have to fit the cube range. Out of range values are clipped when we fit the cube to the precision of the actual input data.

The output LUT values should be valid floats, but are not confined to the range 0-1. Out of range values may be clipped when we fit the cube to the precision of the actual output data. This will happen with all unsigned integer formats.

6.7.1 Input LUTs only

Truelight supports cubes without the multi-dimensional interpolation table. Here is an example…

```
# Truelight Cube v2.0
# lutLength 101
# iDims     3

# InputLUT
 0.000000 0.000000 0.000000
 0.096929 0.490357 0.092795
 ( 97 lines deleted )
 0.969607 0.985015 0.972384
 1.000000 1.000000 1.000000

# end
```

We have left out the number of output dimensions because this must match the number of inputs. You can put the line in, but it must contain the right value.

We have left out the 'width' line because there is no table to have a width.

The '# InputLUT' data section now contains output RGB values ranged 0-1, and not cube index values.

We have left out the '# Cube' data section.

The Truelight hardware has to have a cube. When we must have a cube, the software can generate a unity cube that effectively passes the LUT data unchanged.

6.7.2 No input LUTs

Truelight can also handle cubes without input LUTs. Here is an example…

```
# Truelight Cube v2.0
# iDims     3
# oDims     3
# width     16 16 16

# Cube
 0 0 0
 0.0184206 0 0
 0.0542529 0 0
 ( 4092 lines deleted )
 0.920355 0.985279 0.997871
 0.994231 0.995776 0.996806
 1 1 1

# end
```

We have left out the input LUT length and the data section.

The software may handle this option by making its own input LUT with a length of 2, to fit the cube ranges exactly.

When the software saves a cube, it may recognize these special length 2 LUTs and omit them from the file.

6.7.3 Hardware Cubes

The hardware cube needs a fixed-precision 16x16x16 cube. We can make the hardware cubes from the previous file formats, but the data is different. Only the first part of the file is in ASCII – the rest of the file is binary data. Here is an example of a hardware cube file generated by Truelight…

```
thc{
version{1.1}
created{Fri 04 Feb 2005  11:11:00}
tl_version{2.1.690}
params{
greyValue{445,445,445}
cubeHardware{10-bit}
cubeFile{./temp.cub}
}
recorderChecksum{4604245E}
printChecksum{7B0F00DC}
lampChecksum{7C824180}
displayChecksum{98BBF0CB}
}( start of binary data )
```

The Truelight Hardware Cube header is 'thc{…}' section.

'version{..}' gives the version of the hardware file. This is the only line that has to be present.

'tl_version{..}' gives the version of the Truelight library that created the file.

'params' contains the Truelight parameter settings used to make the cube. The following lines contain the checksums of any calibrations used to make the cube.

We can replace a calibration with another of the same name, or we can change the Truelight root directory. This would change the Truelight transform without changing any of the commands in the profile. The checksums can be used to check whether the current files are the same as the ones used to generate the transform. These checksums are used by the Truelight viewer hardware sync utility to verify that the Truelight calibrations on host computer are the same as the ones that made the cube in the hardware.

The binary data starts directly after the 'thc{..}' close bracket. This data includes a checksum for all the file data, including the ASCII header. If you edit the ASCII header, then the checksum will no longer agree.