Specification

S-2014-006

A Common File Format for Look-Up Tables

The Academy of Motion Picture Arts and Sciences

Science and Technology Council

Academy Color Encoding System (ACES) Project Committee

March 29, 2016

Summary: This document introduces a human-readable text file format for the interchange of color transformations using an XML schema. The XML format supports Look-Up Tables of several types: 1D LUTs, 3D LUTs, and 3by1D LUTs, as well as additional transformation needs such as matrices, range rescaling, and 'shaper LUTs'. The document defines a processing model for color transformations where each transformation is defined by a 'Node' that operates upon a stream of image pixels. A node contains the data for a transformation, and a sequence of nodes can be specified in which the output of one transform feeds into the input of another node. The XML representation allows saving in a text file both a chain of multiple nodes or a single node representing a unique transform. The format is extensible and self-contained so the XML file may be used as an archival element.

NOTICES

Revision History

Version	Date	Description
1.0	05/11/2008	Academy-ASC Common LUT Format, Version 1.0
2.0	12/19/2014	Roll-up of ACES LUT Task Force comments
2.0.1	04/24/2015	Formatting and typo fixes
	03/29/2016	Remove version number - to use modification date as UID

Related Academy Documents

Document Name	Description

Table of Contents

Acknowledgements

The Science and Technology Council wishes to acknowledge the following key contributors to the drafting and validation of this document.

Jim Houston Ana Benitez Doug Walker

The Science and Technology Council wishes to acknowledge the following people for their participation in meetings that led to this specification.

Jim Houston	Ana Benitez	Doug Walker	Ray Feeney
Richard Antley	Scott Barbour	Al Barton	Lars Borg
Curtis Clark ASC	Bob Currier	Steve Crouch	Simon Cuff
Alex Forsythe	Walter Arrigheti	Hugh Heinsohn	Henry Gu
Carlo Hume	Joachim Zell	Lin Sebastian Kayser	Neil Kempt
Richard Kirk	Dennis M. Kornegay	Tom Lianza	Thomas Maier
Andy Maltz	Stu Maschwitz	Milton Adamou	Robert Monaghan
Kevin Mullican	Bruno Munger	Yuri Neyman	Joshua Pines
Peter Polit	Peter Postma	Steve Roach	Steve Shaw
David Stump ASC	Lucas Wilson	Roland Wood	Glenn Woodruff
David McClure	Scott Dyer	Greg Cotten	Will McCown
Hans Lehmann	Haarm-Pieter Duiker	Chris Davies	Jim Whittlesey

This document was drafted in cooperation with the American Society of Cinematographer's Technology committee. The Science and Technology Council thanks the ASC for their ongoing contributions in support of this project.

Introduction

Color Look Up Tables (hereafter just referred to as LUTs) are a common method for transforming color values from one set of codes to another, and also for quickly providing results of a computation using interpolation between precomputed values.

Recent advances in color management systems have led to an increasingly important role for LUTs in production workflows. LUTs are in common usage for device calibration, bit depth conversion, print film simulation, color space transformations, and on-set look modification. With a large number of product developers providing software and hardware solutions for LUTs, there is an explosion of unique vendor-specific LUT file formats, which are often only trivially different from each other.

Recognizing a need to reduce the complexity of interchanging LUTs files, the Science and Technology Council of the Academy of Motion Pictures Arts and Sciences and the Technology Committee of the American Society of Cinematographers sponsored a project to bring together interested parties from production, post-production, and product developers to agree upon a common LUT file format. This document is the result of those discussions.

In their earliest implementation, LUTs were designed into hardware to generate red, green, and blue values for a display from a limited set of bit codes. Recent implementations see LUTs used in many parts of a pipeline in both hardware devices and software applications. LUTs are often pipeline specific and device dependent. The Common LUT File format is a mechanism for exchanging the data contained in these LUTs and expects the designer, user, and application(s) to properly apply the LUTs within an application for the correct part of a pipeline.

All applications that use LUTs have a software or hardware transform engine that digitally processes a stream of pixel code values. The code values represent colors in some color space which may be device-dependent or which is defined in image metadata. For each pixel color, the transform engine calculates the results of a transform and outputs a new pixel color. Defining a method for exchanging these color transforms in a text file is the purpose of this document.

Saving a single LUT into a file to send to another part of production working on the same content is expected to be the most common use of the common LUT file format.

Since color space manipulation is also an essential part of designing color transformations, matrices and other color processing elements are also supported. In addition, the format is extensible so that additional color transformations may be added later while maintaining backwards compatibility.

The Common LUT File format is an XML text file that can contain single or multiple color transforms represented as matrices, LUTs, or other color processing elements. A LUT designer can create an XML list containing multiple transforms which are “daisy chained” together to achieve an end result: the output result of the first transform is used as input to the second transform which then calculates another output that is fed to the next transform, and so on.

Only a small set of common color transforms can be saved in the file format. This document sometimes uses the word LUT as a general stand-in for any of the color transforms that can be represented within the XML schema.

As this document is intended both as a specification and a guide for implementation, it contains a fair amount of complexity for what is a straightforward proposition: saving arrays of numbers in an XML text file. This document assumes the reader has knowledge of LUT creation, color transforms and XML, and therefore the information in the document rarely provides tutorial information.

1 Scope

This document introduces a human-readable text file format for the interchange of color transformations using an XML schema. The XML format supports Look-Up Tables of several types: 1D LUTs, 3D LUTs, and 3by1D LUTs, as well as additional transformation needs such as matrices, range rescaling, and 'shaper LUTs.'

2 References

The following standards, specifications, articles, presentations, and texts are referenced in this text:

IETF RFC 3066: IETF (Internet Engineering Task Force). RFC 3066: Tags for the Identification of Languages, ed. H. Alvestrand. 2001 IEEE DRAFT Standard P123

Academy S-2014-002, Academy Color Encoding System – Versioning System

Academy TB-2014-002, Academy Color Encoding System Version 1.0 User Experience Guidelines

3 Terms and Definitions

The following terms and definitions are used in this document.

3.1 LUT definition

Look-up tables are an array of size n where each entry has an index associated with it starting from 0 and ending at $(n-1)$, (e.g. $lut[0], lut[1], \ldots, lut[n-1]$). Each entry contains the output value of the LUT for a particular input value. In a fully enumerated LUT, there is an entry for each possible input code. With a particular input bit depth b, the size n equals 2^b.

3.2 Sampled LUT

In a sampled LUT, there is a smaller set of entries representing sampled positions of the input code range. By default, the minimum input code value is looked up in $lut[0]$ and the maximum input code value is looked up in $lut[n-1]$. Other output code values are found directly if the input code is one of the sampled positions, otherwise the output values are interpolated between the two nearest sample entries in the LUT.

3.3 Interpolation

When an input value falls between two sampled positions in a LUT, the output value must be calculated as a proportion of the distance along some function that connects the two nearest values in the LUT. Multiple interpolation types are possible, and higher-order interpolations such as piece-wise cubic interpolation use more than just the adjacent entry to determine the shape of the function. The simplest interpolation type, linear, is a straight line between the points. An example of linear interpolation is provided below.

Ex: with a table of the sampled input values in $inValue[i]$ where i ranges from 0 to $(n-1)$, and a table of the corresponding output values in $outValue[j]$ where j is equal to i,

index i	inValue	index j	outValue
0	0	0	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n-1$	1	$n-1$	1000

the $output$ resulting from $input$ can be calculated after finding the nearest $inValue[i] < input$.

When $inValue[i] = input$, the result is evaluated directly.

$$output = \frac{input - inValue[i]}{inValue[i+1] - inValue[i]} \times (outValue[j+1] - outValue[j]) + outValue[j]$$

3.4 Tetrahedral Interpolation

This is a type of interpolation for 3D LUTs in which the color space is subdivided by tetrahedra arranged to cover the entire 3D volume. There are multiple ways to subdivide this space, however for color processing, this specification uses the form where each cube is split along the main (and usually neutral) diagonal into 6 tetrahedra.

3.5 `IndexMap` definition

The mapping of input code values to indexes of the table is modifiable allowing for remapping the range of applicable input values, changing the spacing of the input sampling function, and reshaping of the index function for lookups into the LUT. An `IndexMap` in its simplest form allows the definition of the range of input and output floating-point values that are normalized to 0 to 1.0 for accessing a 1DLUT or a 3DLUT. This extension to the LUT format is fully described in Appendix C. The `IndexMap` is primarily intended for improved implementation of floating-point LUTs and is not necessary for integer LUTs.

3.6 1D LUT definition

A color transform using a 1D LUT has as input a 1-component color value from which it finds the nearest index position whose $inValue$ is less than or equal to the implicit input values for the table. The transform algorithm then calculates the output value by interpolation between $outValue$ entries in the table. A 1DLUT shall be applied equally to all channels in a 3-component color calculation.

3.7 3by1D LUT definition

A 3by1DLUT is a particular case of a 1DLUT in which a 3-component pixel is the input value, and each component is looked up separately in its own 1DLUT which may differ from each other.

3.8 3DLUT definition

In a 3D LUT, the value range of the 3 color components defines the coordinate system of a 3D cube. A single position is found within the volume of the cube from the 3 input values, and a set of the nearest corresponding table entries are identified (between 4 to 8 table entries, depending on the interpolation algorithm). The 3-component output value is calculated by interpolating those nearest table entries from the 3DLUT.

NOTE: For examples of 3D cube interpolation, look at "Efficient color transformation implementation" by Bala and Klassen in Digital Color Imaging Handbook, ed: Sharma, CRC Press, 2003, pg 694-702

3.9 MATRIX definition

A matrix can be used for linear conversion of 3 color component pixels from one color space to another.

The 3-component input value vector is multiplied by the matrix to create the new 3-component output value vector. Matrix color transforms in this format use the column vector convention:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

where $[R, G, B]$ is the result of multiplying input $[r, g, b]$ by the matrix $[a]$.

3.10 ASC CDL

The American Society of Cinematographers' Color Decision List (ASC CDL). This provides a universal method of exchanging basic color correction parameters and specifies a particular color processing equation.

4 Specification

Color transformations (LUTs) are stored in text files that have a defined XML structure.

The top level element in the XML file defines a `ProcessList` which represents a sequential set of color transformations. The result of each single color transformation feeds into the next transform in the list to create a daisy chain of transforms.

An application reads the XML file and initializes a transform engine to perform the transformations in the list. The transform engine reads as input a stream of code values of pixels, performs the calculations and/or interpolations, and writes an output stream representing a new set of code values for the pixels.

In this document, the sequence of transformations is described as a node-graph where each `ProcessNode` performs a transform on a stream of pixel data and only one input line (input pixel values) may enter a node and only one output line (output pixel values) may exit a node. A `ProcessList` may be defined to work on either 1-component or 3-component pixel data, however all transforms in the list must be appropriate especially in the 1-component case (black-and-white) where only 1D LUT operations are allowed.

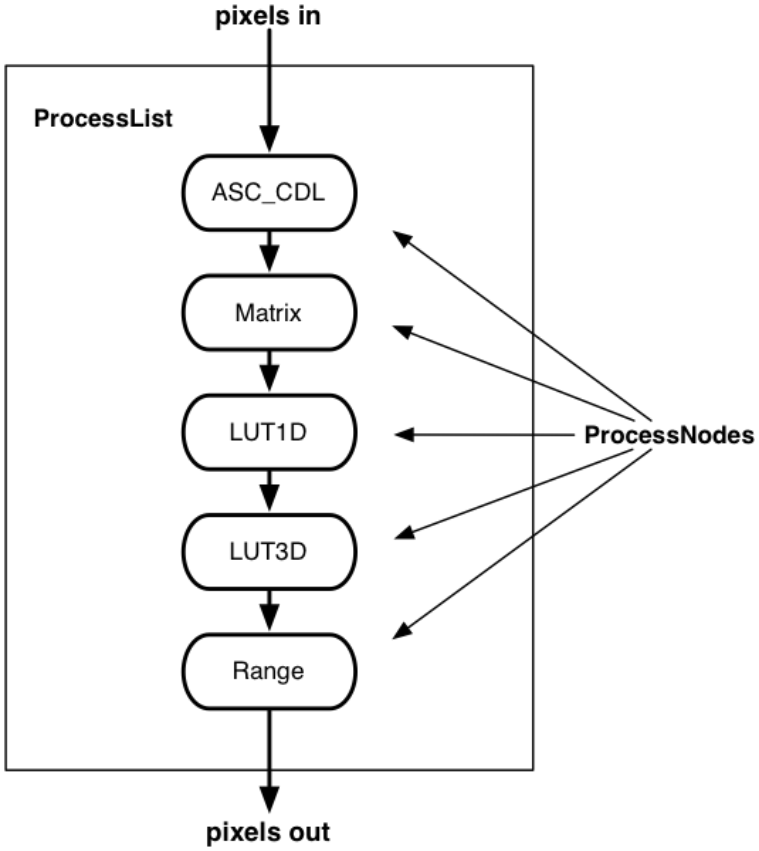

Figure 1 – Process List

The system described in this document assumes that the transform engine performs the set of calculations represented by the color transform nodes. The transform engine has internal objects containing procedures for each type of transform, and the data or data array for each node in the file. The engine determines the appropriate color calculations needed for each of the transform nodes and establishes the sequence of transforms according to the sequence in the list.

Each `ProcessNode` must indicate for its input and output whether the pixel values are floats or integers, and must also specify the bit depth. In software applications, it is assumed that all processing is performed

in floating point and the indication of an integer value represents normalized values (e.g. 10 bit integer represents values from 0.0 to 1.0).

NOTE: See Section 9 for notes on floating point processing.

The input format of each node must be matched by the output format of the previous node in the processing chain. Conversion of values between floating-point and integer ranges must be explicitly performed within a `ProcessNode`. Conversions between integer and floating-point ranges can be made explicit either in the LUT array output, or with a special `Range` node. Floating-point values may require handling large ranges of values, and so the `Range` node is provided for the designer of transforms so that floating-point values may have their ranges altered before feeding into the next node of the list.

It should be noted that input floating-point values may not be assumed to be within any particular range, and due to the need for high dynamic range processing, may cover the entire bit depth of the floating-point type. Selection of the appropriate range for processing in LUTs is achieved for floating-point values with a Range node. Similarly on the output, floating-point values of -65504.0 to 65504 could be expected from certain types of LUTs. A LUT designer can design a processing path that uses all normalized floating-point values, but other ranges of output in the `ProcessList` are also possible.

The last node of the `ProcessList` is expected to be the final output of the LUT. A LUT designer can allow floating-point values to be interpreted by applications and thus delay control of the final encoding through user selections.

A `Range` node as the very last node in the `ProcessList` allows specification of minimum and maximum output values and clamping if needed.

The file format does not provide a mechanism to assign color transforms to either image sequences or image regions. However, the XML structure defining the LUT transform, a `ProcessList`, may be encapsulated in a larger XML structure potentially providing that mechanism. This mechanism is outside the scope of this document.

Each XML file shall be completely self-contained needing no external information or metadata. Each list must be given a unique ID for reference, however, a color transform may not be incorporated by reference to another XML LUT file. The full content of a color transform must be included in each file. This insures that each LUT file can be an independent archival element.

The data for a LUT is specified with an ordered array that is either all floats or all integers. When three RGB color components are present, it is assumed that these are red, green, and blue in that order.

There is only one order for how the data array elements are specified in a LUT, which is in general from black to white (from the minimum input value position to the maximum input value position). Arbitrary ordering of list elements is not provided in the format (see Section 7 for details).

For 3DLUTs, the indexes to the cube are assumed to have regular spacing across the range of input values unless otherwise specified with an `IndexMap`. The transform designer at a minimum may specify the first and last index positions of the LUT relative to the input code value range (see Section 7.3 for details). All other uses of the `IndexMap` are considered extensions.

For simplicity's sake, the standard LUT format does not keep track of color spaces, or require the application to convert to a particular color space before use. 3x3 and 3x4 matrices may be defined in a `ProcessNode` for color conversion needs. Comment fields are provided so that the designer of a transform can indicate the intended usage. The application carries the burden of properly using the transform and/or maintaining pixels in the proper color space.

5 Conventions and Constraints

Some characteristics of the XML format are constrained to improve interoperability and interchange between the different systems that use LUTs. These constraints may be loosened over time as implementations and computer capabilities improve. It is desirable that implementations support the full capabilities of the speci-

fication, but it is recognized that portability and ease of implementation are needed in current deployments.

The constraints listed in this section serve to identify key parameters that an implementation reader must be able to use.

Among these agreed upon conventions are:

This specification uses the `ProcessList` as the root node in an XML file. Storing multiple `ProcessLists` in a file requires definition of a higher-level XML schema for that purpose which is not addressed here.

Matrices are 3x3, 3x4, or 4x4. 3x4 matrices provide an offset value in the 4th column to be added to the results of the previous 3 columns.

3-component processing is the normal mode of operation. 1-component processing is possible with only a `LUT1D` node and if needed a `Range` node, but is not a required feature.

`LUT3Ds` have the same dimension on all axes (i.e. `Array` dimensions are of the form "n n n 3"). A `LUT3D` with axial dimensions greater than 128x128x128 should be avoided.

Software applications are required to support two 3DLUT interpolation types: "trilinear" and "tetrahedral". "Linear" interpolation is required as a default for 1DLUTs.

`LUT1D` lengths should be limited to at most 65536 entries.

As a general principle, the LUT output values should make explicit the clipping behavior for values outside of the region of interest. The `Range` node can be used for clamping of values.

Supported bit depth values are "`8i`", "`10i`", "`12i`", "`16i`", "`16f`" and "`32f`".

"`12i`" is present for processing of digital cinema 12-bit DCDMs. "`16f`" are half floats as described in IEEE 754-1985. All integer values are unsigned ints. "`32f`" LUTs should never be fully enumerated.

6 XML Structure

LUTs are stored in XML files each of which must have the same XML root element, `ProcessList`, regardless of the number of LUTs in the file. The `ProcessList` root element contains a sequence of `ProcessNodes` which are typically either LUTs or matrices. Any `ProcessList` must also contain at least one `ProcessNode`. An example of the overall structure of a LUT file is thus:

```
<ProcessList id="123">
    <Matrix id="1">
        data & metadata
    </Matrix>
    <LUT1D id="2">
        data & metadata
    </LUT1D>
    <Matrix id="3">
        data & metadata
    </Matrix>
</ProcessList>
```

The order and number of transforms is determined by the designer of the transform.

The XML file may contain other information that is useful to XML interpreters. This includes a starting line that identifies the XML version number and Unicode values. This line is mandatory once in a file and looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
```

The file may also contain XML comments that may be used to describe the structure of the file or save information that would not normally be exposed to a database or to a user. XML comments are enclosed in brackets like so,

```
<!--   This is a comment    -->
```

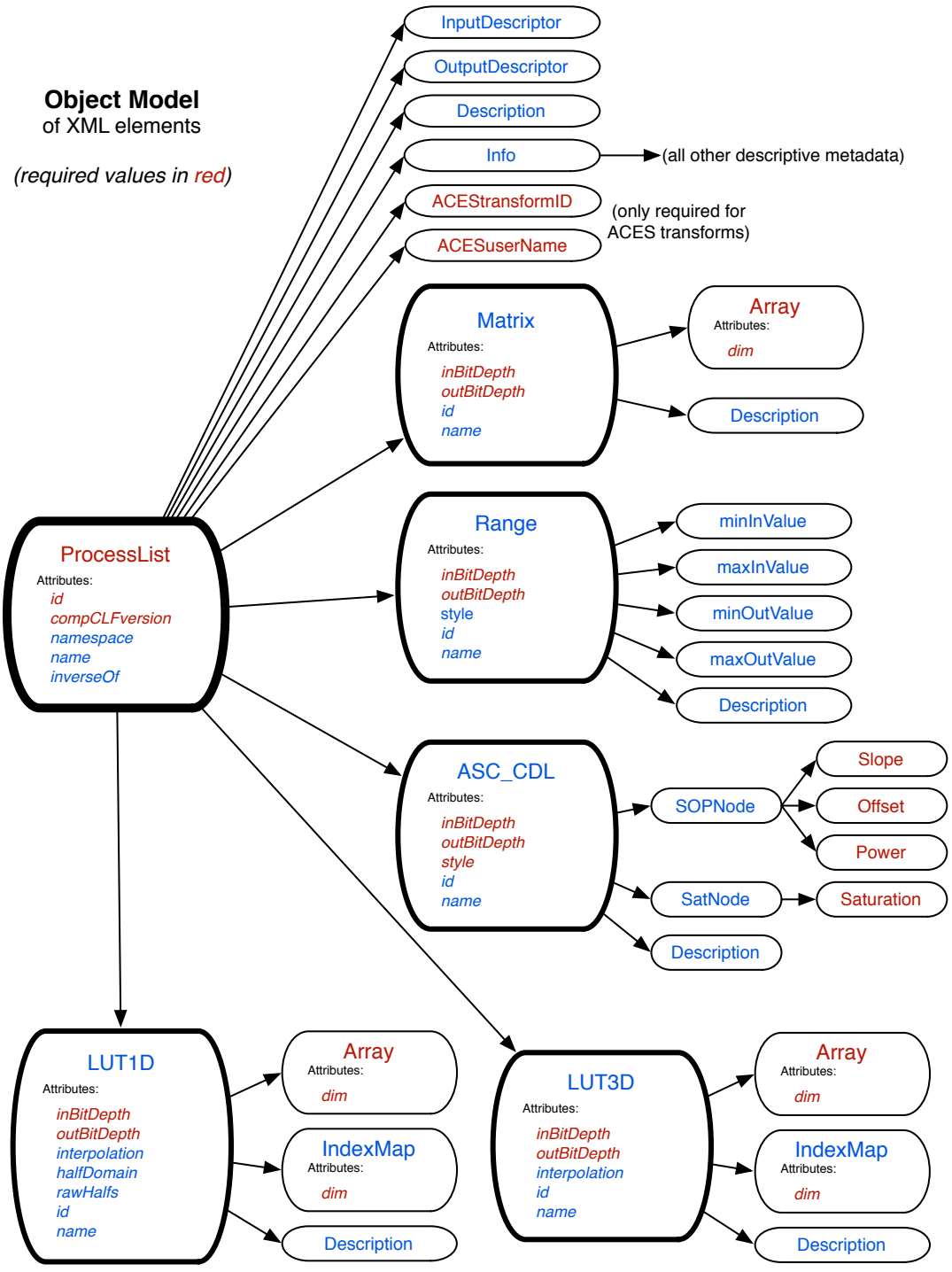

Figure 2 – Object Model of XML Elements

It is often useful to identify the natural or formal language in which text strings of XML documents are written. The special attribute named xml:lang may be inserted in XML documents to specify the language used in the contents and attribute values of any element in an XML document. The values of the attribute are language identifiers as defined by IETF RFC 3066. In addition, the empty string may be specified.

The language specified by xml:lang applies to the element where it is specified (including the values of its attributes), and to all elements in its content unless overridden with another instance of xml:lang. In particular, the empty value of xml:lang can be used to override a specification of xml:lang on an enclosing element, without specifying another language.

7 XML Elements

7.1 `ProcessList`

This is the root element for any LUT XML file and is required even if only one `ProcessNode` will be present. A `ProcessList` is composed of one or more `ProcessNode`s.

Attributes:

`id` (required)
Unique identifier of the `ProcessList`.

`name` (optional)
Text name of the `ProcessList` for display or selection from an application's user interface.

`compCLFversion` (required)
Minimum compatible CLF spec version needed to read this file.

`inverseOf` (optional)
Optional field for linking to another `ProcessList` `id` (unique) which is the inverse of this one.

Elements:

`Description` (required)
Comment field for an arbitrary string describing the function, usage, or notes about the `ProcessList`. A `ProcessList` can have one or more `Description`s.

`Info` (optional)
optional element for addition of custom metadata not needed to interpret the transforms. Includes:

`AppRelease` (optional)
application software release level

`CalibrationInfo` (optional)
calibration metadata used when making the LUT for a device

`<DisplayDeviceSerialNum>`

`<DisplayDeviceHostName>`

`<OperatorName>`

`<CalibrationDateTime>`

`<MeasurementProbe>`

`<CalibrationSoftwareName>`

`<CalibrationSoftwareVersion>`

`InputDescriptor` (optional)
Comment field describing the intended source code values of the `ProcessList`.

`OutputDescriptor` (optional)
Comment field describing the intended output target of the `ProcessList` (e.g., target display).

`ProcessNode` (required)
At least one `ProcessNode` must be in the list. The `ProcessNode` is described in Section 7.2.

7.2 `ProcessNode`

This element represents a process node; it is the primary color transformation element for LUT interchange. Different types of processing are expressed by substitution from this basic element including `LUT1D`, `LUT3D`, `Range` and `Matrix`. In the future, other kinds of `ProcessNode`s can be defined as part of the standard to handle alternate forms of processing. All `ProcessNode` substitutes inherit the attributes and elements below.

Attributes:

`id` (optional)
Unique identifier of the `ProcessNode`.

`name` (optional)
Text name of the `ProcessNode` for display or selection from an application's user interface.

`inBitDepth` (required)
Number of bits and type of the input values to which the `ProcessNode` is applied. The input values can be either integers or floats. Supported values are "`8i`", "`10i`", "`12i`", "`16i`", "`16f`" or "`32f`".

`outBitDepth` (required)
Number of bits and type of the output values which the `ProcessNode` generates. The output values can be either integers or floats, independent of the input values. Supported values are "`8i`", "`10i`", "`12i`", "`16i`", "`16f`" or "`32f`".

Elements:

`Description` (optional)
Comment field for an arbitrary string describing the function, usage, or notes about the `ProcessNode`. A `ProcessNode` can have one or more `Descriptions`.

7.2.1 `LUT1D` (substitute for `ProcessNode`)

This element specifies a 1D LUT or a 3by1D LUT. A 1D LUT transform uses an input pixel value, finds the two nearest index positions in the LUT, and then interpolates the output value using the entries associated with those positions. If the input to a `LUT1D` is an RGB value, the same LUT will be applied to all three color components. If a 3by1D LUT is supplied, each color component is looked up in a separate `LUT1D` of the same length. In both cases, the ordered entries of the LUT are provided in `Array`. In a 3by1D LUT, by convention, the `LUT1D` for the first component goes in the first column of `Array`, etc.). The lookup operation may be altered by redefining the mapping of the input values to index positions of the LUT using an `IndexMap`.

Attributes:

`interpolation` (optional)
Name or description of the preferred calculation used to interpolate values from the `LUT1D`. This attribute is optional with a default of "`linear`" if the attribute is not present. Systems that utilize LUTs may use different types of interpolation; therefore, this attribute is intended as a guide to an application if it wants to attempt recreating the exact outputs of the originating application. Typical values for this attribute would be "`linear`" or "`cubic`".

`rawHalfs` (optional)

If this attribute is present, its value must be "`true`". In this case, the output array values in the form of unsigned 16-bit integers are interpreted as the equivalent bit pattern, half floating-point values. (e.g. to represent the value 1.0, enter the integer 15360 in the `Array` element because it has the same bit-pattern. This allows specification of exact half-float values without relying on conversion from decimal text strings.)

`halfDomain` (optional)

If this attribute is present, its value must be "`true`". In this case, the input domain to the node is all possible 16-bit floating-point values, and there must be exactly 65536 entries in the LUT1D `Array` element (e.g. the unsigned integer 15360 has the same bit-pattern (0011110000000000) as the half-float value 1.0, so the 15360th, zero-indexed, entry in the `Array` element is the output value corresponding to an input value of 1.0). The `IndexMap` may not be used in this case.

Elements:

`IndexMap` (optional)

Table that maps input values to index positions of the `LUT1D`'s `Array`.

In its simplest form, with a dim=2, this element can be used to define the input floating-point range that will be used by the LUT.

An extension to the LUT format described in Appendix C.

`Array` (required)

Table that provides the entries of the `LUT1D` from the minimum to the maximum input values. In a 3by1D LUT, each column in `Array` provides the 1D LUT for a color component; for RGB, the 1st column corresponds to Rs 1D LUT, the 2nd column corresponds to Gs 1D LUT, etc.

7.2.2 `LUT3D` (substitute for `ProcessNode`)

This element specifies a 3D LUT. In a `LUT3D` element, the 3 color components of the input value are used to find the nearest indexed values along each axis of the 3D cube. The 3-component output value is calculated by interpolating within the volume defined by the nearest corresponding positions in the LUT. The lookup operation may be altered by redefining the mapping of the input values to index positions into the LUT (see `IndexMap`). An interpolation comment field is provided to indicate the preferred interpolation calculation to perform within the volume.

Attributes:

`interpolation` (optional)

Name or description of the preferred calculation used to interpolate values in the `3DLUT`. This attribute is optional with a default of "`trilinear`" if the attribute is not present. Systems that utilize LUTs may use different types of interpolation; therefore, this attribute is intended as a guide to an application if it wants to attempt recreating the exact outputs of the originating application. Typical values for this attribute would be "`trilinear`" or "`tetrahedral`".

Elements:

`IndexMap` (optional)

Table that maps input values to index positions of the `LUT3D`'s `Array`. In its simplest form, with a dim=2, this element can be used to define the input floating-point range that will be used by the LUT.

An extension to the LUT format described in Appendix C.

`Array` (required)

Table comprised of the entries for the `LUT3D` from the minimum to the maximum input values, the third component index changing fastest.

Ex: order of entries for a 2x2x2 cube by index $[0 \ldots 1]$

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

7.2.3 `Matrix` (substitute for `ProcessNode`)

This element specifies a matrix transformation to be applied to the input values. The input and output of a `Matrix` are always 3-component values. All matrix calculations should be performed in floating point, and input bit depths of integer type should be treated as scaled floats. If the input bit depth and output bit depth do not match, the coefficients in the matrix must incorporate the results of the 'scale' factor that will convert the input bit depth to the output bit depth (e.g. input of `10i` with an output of `12i` requires the matrix coefficients already have a factor of 4095/1023 applied). Changing the input or output bit depth requires creation of a new set of coefficients for the LUT.

The output values are calculated using the row-order convention:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{aligned} R &= (r \cdot a_{11}) + (g \cdot a_{12}) + (b \cdot a_{13}) \\ G &= (r \cdot a_{21}) + (g \cdot a_{22}) + (b \cdot a_{23}) \\ B &= (r \cdot a_{31}) + (g \cdot a_{32}) + (b \cdot a_{33}) \end{aligned}$$

An offset matrix may be defined as a 3x4 array in which the input value is typically defined as (r, g, b, 1.0). The 4th column of the matrix may then be used to add an offset term to the conversion of the matrix. The output of the 3x4 matrix is (R, G, B). Matrices using an offset calculation will have one more column than rows.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & off1 \\ a_{21} & a_{22} & a_{23} & off2 \\ a_{31} & a_{32} & a_{33} & off3 \end{bmatrix} \begin{bmatrix} r \\ g \\ b \\ 1.0 \end{bmatrix} = \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Elements:

`Array` (required)

Table that provides the coefficients of the transformation matrix. The matrix dimensions are either

3x3 or 3x4. The matrix is serialized row by row from top to bottom and from left to right, i.e., "$a_{11}\ a_{12}\ a_{13}\ a_{21}\ a_{22}\ a_{23}\ \ldots$" for a 3x3 matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

7.2.4 `Range` (substitute for `ProcessNode`)

The `Range` element maps the input domain to the output range by scaling and offsetting values and must be calculated in floating point. The `Range` element is also used to clamp values.

If a `minInValue` is present, then `minOutValue` must also be present and the result is clamped at the low end. Similarly, if `maxInValue` is present, then `maxOutValue` must also be present and the result is clamped at the high end. If none of `minInValue`, `minOutValue`, `maxInValue`, or `maxOutValue` are present, then the `Range` operator performs only scaling.

The scaling of `minInValue` and `maxInValue` depends on the input bit depth, and the scaling of `minOutValue` and `maxOutValue` depends on the output bit depth.

Elements:

`minInValue` (optional)

`maxInValue` (optional)

`minOutValue` (optional)

`maxOutValue` (optional)

Attributes:

`style` (optional)

Description of the preferred handling of the scaling calculation of the `Range` node. If the style attribute is not present, clamping is performed.

``noClamp''

If present, do not apply the clamping in the equations below (i.e. floating point values above `maxOutValue` are allowed).

``Clamp''

If present, clamping is applied according to:

$$\text{MIN}(\texttt{maxOutValue}, \text{MAX}(\texttt{minOutValue}, RGB_{out}))$$

The range between `minInValue` and `maxInValue` is scaled to the `minOutValue` and `maxOutValue` range using the following equation:

$$scale = \frac{(\texttt{maxOutValue} - \texttt{minOutValue})}{(\texttt{maxInValue} - \texttt{minInValue})}$$
$$RGB_{out} = RGB_{in} \times scale + \texttt{minOutValue} - \texttt{minInValue} \times scale$$
$$RGB_{out_{clamped}} = \text{MIN}(\texttt{maxOutValue}, \text{MAX}(\texttt{minOutValue}, RGB_{out}))$$

If the input and output bit depths are not the same, a conversion should take place using the range elements. If the elements defining an `InValue` range or `OutValue` range are not provided, then the default behavior

is to use the full range available with the `inBitDepth` or `outBitDepth` attribute used in place of the missing input range or missing output range, respectively, as calculated with these equations:

In the formulae below, if the bit depth is integral, then rangescalar() is defined as:

$$\text{rangescalar}(bitDepthInteger) = 2^{bitDepth} - 1$$

If the bit depth is specified as floating point, then:

$$\text{rangescalar}(float) = 1.0$$

If only minimum values are specified, the formula is:

$$scale = \frac{\text{rangescalar}(outBitDepth)}{\text{rangescalar}(inBitDepth)}$$
$$RGB_{out} = \text{MAX}(\texttt{minOutValue}, RGB_{in} \times scale + \texttt{minOutValue} - \texttt{minInValue} \times scale)$$

If only maximum values are specified, the formula is:

$$scale = \frac{\text{rangescalar}(outBitDepth)}{\text{rangescalar}(inBitDepth)}$$
$$RGB_{out} = \text{MIN}(\texttt{maxOutValue}, RGB_{in} \times scale + \texttt{maxOutValue} - \texttt{maxInValue} \times scale)$$

7.2.5 `ASC_CDL` (substitute for `ProcessNode`)

This node contains the parameters for processing pixels using the ASC CDL color correction equations. The ASC CDL equations are designed to work on an input domain of floating-point values of [0 to 1.0] although values greater than 1.0 can be present. The output data may or may not be clamped depending on the processing configuration below.

Attributes:

`id` (optional)
: This should match the id attribute of the ColorCorrection element in the ASC CDL XML format.

`style` (optional)
: Determines the formula applied by the operator. The valid options are:

  `"Fwd"`
  : implementation of v1.2 ASC CDL equation

  `"Rev"`
  : inverse equation

  `"FwdNoClamp"`
  : similar to the `Fwd` equation, but without clamping

  `"RevNoClamp"`
  : inverse equation

  The first two implement the math provided in version 1.2 of the ASC CDL specification. The second two omit the clamping step and are intended to provide compatibility with the many applications that take that alternative approach.

Elements:

`SOPNode` (optional)
: The `SOPNode` is optional, but if present, must contain one each of the following sub-elements:

`Slope`
If not provided, the default is 1.0 for all channels.

`Offset`
If not provided, the default is 0.0 for all channels.

`Power`
If not provided, the default is 1.0 for all channels.

`SatNode` (optional)
The `SatNode` is optional, but if present, must contain one of the following sub-element:

`Saturation`
If not provided, the default is 1.0 for all channels.

NOTE: The structure of this `ProcessNode` matches the structure of the XML format described in the v1.2 ASC CDL specification. However, unlike the ASC CDL XML format, there are no alternate spellings allowed for these elements.

The math for `style="Fwd"` is:

$$\begin{aligned} SAT_{in} &= \text{clamp}(RGB_{in} \times slope + offset)^{power} \\ luma &= 0.2126 * SAT_{in_R} + 0.7152 * SAT_{in_G} + 0.0722 * SAT_{in_B} \\ RGB_{out} &= \text{clamp}(luma + saturation * (SAT_{in} - luma)) \end{aligned}$$

where clamp() clamps the argument to [0,1].

The math for `style="FwdNoClamp"` is the same as for `"Fwd"` but the two clamp() functions are omitted. Also, if $(input * slope + offset) < 0$, then no power function is applied.

The math for `style="Rev"` is:

$$\begin{aligned} RGB_{in_{clamp}} &= \text{clamp}(RGB_{in}) \\ luma &= 0.2126 * R_{in_{clamp}} + 0.7152 * G_{in_{clamp}} + 0.0722 * B_{in_{clamp}} \\ RGB_{sat_{out}} &= luma + \frac{1}{saturation} * (RGB_{in_{clamp}} - luma) \\ RGB_{out} &= \text{clamp}\left(\frac{\text{clamp}(RGB_{sat_{out}})^{\frac{1}{power}} - offset}{slope}\right) \end{aligned}$$

The math for `style="RevNoClamp"` is the same as for `"Rev"` but the two clamp() functions are omitted. Also, if $RGB_{sat_{out}} < 0$, then no power function is applied.

NOTE 2: The equations above assume that the input and output bit depths are floating-point. For integer inputs and outputs, the values must be normalized to or from [0.0, 1.0] scaling. In other words, the slope, offset, power, and saturation values stored in the `ProcessNode` do not depend on `inBitDepth` and `outBitDepth` they are always interpreted as if the bit depths were float.

7.3 `IndexMap` element

This element defines a list that is a new mapping of input code values, $inValues$, to index positions, n, in a LUT's `Array`. The format of each item in the list is $newInValue@n$. For example, the first `IndexMap` item might be $64@0$ which assigns the $inValue = 64.0$ to position 0 in the LUT.

In its simplest form, with a `dim=2`, this element can be used to define the input floating point range that will be used by the LUT.

An extension to the LUT format described in Appendix C.

Attributes:

`dim` (required)

Field that specifies the number of items in the list.

7.4 `Array` element

This element contains the table entries of a LUT in order from minimum value to maximum value with a single line for each color component triple. The `dim` attribute specifies the dimensions of the cube, the length of the LUT, or the size of the array.

The `dim` attribute and the type of node should match.

Attributes:

`dim` (required)

Specifies the dimension of the LUT or the matrix and the number of components. The data points for a `ProcessNode` are contained in an XML array element. The `dim` attribute provides the dimensionality of the indexes, where:

4 entries have the dimensions of a 3D cube plus the number of components per entry.

e.g. `dim = 17 17 17 3` indicates a 17-cubed 3D lookup table with 3 component color

3 entries have the matrix dimensions and component value.

e.g. `dim = 3 3 3` is a 3 by 3 matrix acting on 3-component values

e.g. `dim = 3 4 3` is a 3 by 4 matrix acting on 3-component values

2 entries have the length of the LUT and the component value (1 or 3).

e.g. `dim = 256 3` indicates a 256 element 1D LUT with 3 components (a 3by1DLUT)

e.g. `dim = 256 1` indicates a 256 element 1D LUT with 1 component (1DLUT)

8 Examples

This section illustrates some of the typical forms of the LUT format. It should be noted that these are not real examples.

The simplest form is an XML file containing a single node:

```
<?xml version="1.0" encoding="UTF-8"?>
<ProcessList xmlns="urn:NATAS:ASC:LUT:v1.2" id="ex1" name="example 1 transform">
    <Description> Turn 4 grey levels into 4 inverted codes using a 1D </Description>
    <LUT1D id="lut-23" name="4valueLut" inBitDepth="12i" outBitDepth="12i">
        <Description> 1D LUT </Description>
        <Array dim="4 1">
            3
            2
            1
            0
        </Array>
    </LUT1D>
</ProcessList>
```

Example 1 – Example of a "LUT1D"

The `LUT1D` ProcessNode could be replaced with a 3D LUT (Example 2) or a matrix (Example 3).

```
<LUT3D id="lut-24" name="green look" interpolation="trilinear"
    inBitDepth="12i" outBitDepth="16f">
    <Description> 3D LUT </Description>
```

```
    <Array dim="2 2 2 3">
        0.0 0.0 0.0
        0.0 0.0 1.0
        0.0 1.0 0.0
        0.0 1.0 1.0
        1.0 0.0 0.0
        1.0 0.0 1.0
        1.0 1.0 0.0
        1.0 1.0 1.0
    </Array>
</LUT3D>
```

Example 2 – Example of a `LUT3D`

```
<Matrix id="lut-25" name="colorspace conversion" inBitDepth="10i" outBitDepth="10i" >
    <Description> 3x4 Matrix , 4th column is offset </Description>
    <Array dim="3 4 3">
        1.2     0.0     0.0     0.002
        0.0     1.03    0.001   -0.005
        0.004   -0.007  1.004   0.0
    </Array>
</Matrix>
```

Example 3 – Example of a `Matrix`

"Shaper LUTs" require a bit more of an explanation. This is once again an illustration of the technique using an `IndexMap` and not a real world example. These can also be implemented with `1DLUT`s.

```
<LUT1D id="lut-25" name="shaper LUT" inBitDepth="10i" outBitDepth="16f">
    <Description> 1D LUT with shaper </Description>
    <IndexMap dim=4>0@0 10@100 20@250 30@360 40@440 445@445
                    700@600 800@700 900@850 950@1023</IndexMap>
    <Array dim="1024 1">
        0.00
        0.32
        0.50
        <1020 entries omitted>
        1.0
    </Array>
</LUT1D>
```

Example 4 – Example of a partially enumerated "Shaper LUT"

```
<ASC_CDL id="cc01234" inBitDepth="16f" outBitDepth="16f" style="Fwd">
    <Description>scene 1 exterior look</Description>
    <SOPNode>
        <Slope>1.000000  1.000000  0.900000</Slope>
        <Offset>-0.030000  -0.020000  0.000000</Offset>
        <Power>1.2500000  1.000000  1.000000</Power>
    </SOPNode>
    <SatNode>
        <Saturation>1.700000</Saturation>
    </SatNode>
</ASC_CDL>
```

Example 5 – Example of an ASC CDL node

A full example of an XML file (Example 6) shows three nodes in a ProcessList.

```
<?xml version="1.0" encoding="UTF-8"?>
<ProcessList xmlns="urn:NATAS:ASC:LUT:v1.2" id="luts-23+24+25" name="lut chain 34">
    <Description> Turn 4 grey levels into 4 codes for a monitor using a 3by1D LUT
            into 3D LUT into 3x1D LUT </Description>
    <OutputDescriptor> Sony BVM CRT </OutputDescriptor>
    <LUT1D id="lut-23" name="input lut" inBitDepth="12i" outBitDepth="12i">
```

```
        <Description> 3by1D LUT </Description>
        <Array dim="4 3">
            1 1 1
            1 1 1
            2 2 2
            2 2 2
        </Array>
    </LUT1D>
    <LUT3D id="lut-24" name="green look output rendering" interpolation="trilinear"
                inBitDepth="12i" outBitDepth="16f">
        <Description> 3D LUT </Description>
        <Array dim="4 4 4 3">
            0.0 0.0 0.0
            0.0 0.0 1.0
            0.0 1.0 0.0
            0.0 1.0 1.0
            1.0 0.0 0.0
            1.0 0.0 1.0
            1.0 1.0 0.0
            1.0 1.0 1.0
            [ed:  ...abridged:   64 total entries...]
            1.0 1.0 1.0
        </Array>
    </LUT3D>
    <LUT1D id="lut-25" name="output conversion" inBitDepth="16f" outBitDepth="12i">
        <Description> 3x1D LUT </Description>
        <IndexMap dim=2>0.0@0 3.0@65504.0</IndexMap>
        <Array dim="4 3">
            0 0 0
            1 1 1
            2 2 2
            3 3 3
        </Array>
    </LUT1D>
</ProcessList>
```

Example 6 – Full example of an XML LUT file

9 Implementation Notes

9.1 Efficient Processing

The transform engine may merge some or all of the transforms and must maintain appropriate precision in the calculations so that output values are correct. It is recommended that all calculations be done in at least 32-bit floating point. High accuracy for 16-bit half float output is required.

The existence of a common LUT format cannot guarantee that the resulting images will look the same on all implementations as numerical accuracy issues in implementations can have a significant effect on image appearance.

The engine may create a single LUT concatenating the output result of all of the node calculations but this may introduce some inaccuracies to the result due to LUT sampling errors. It is up to the user to determine whether these approximate results are sufficient.

9.2 Indexing Calculations

A `ProcessNode` using LUT tables must perform an index calculation to take the range of input values and ratio them to the input 'index' range of the table (i.e. the minimum and maximum index positions into the table). This allows the LUT location calculation to be easily achieved as the normalized index function can be multiplied by the number of entries in the LUT to get a direct hash function to the appropriate LUT locations. For integer inputs, this is straightforward as the `inBitDepth` attribute may be used to apply the whole range of input across the whole range of index positions.

9.3 Floating-point Output of the `ProcessList`

The output of the LUT chain should be intended for real devices, therefore a transform designer and/or the application should insure that output floating-point values do not contain infinities and NaN codes. The minimum and maximum representable integer values should be used instead. It is the responsibility of the application to handle overflows and underflows correctly.

In most applications, it is either the internal requirements of the application or the end-user who controls the bit depth of pixels being processed through a `ProcessList`. So transform authors should not assume that applications will necessarily clamp and quantize based on the settings of the `outBitDepth` attribute for the last node of the `ProcessList`.

Floating-point processing:

Although it has been traditional practice in converting from integer to floats to normalize the top integer code to a value of 1.0 in floating point, there is an increasing need for calculations in high-dynamic range imaging systems where this assumption might be flawed, so the assumption in the `ProcessList` element is that integer values are normalized floats where for example of 1023 in integer is output from a node as 1.0. A `Range` node is provided to make explicit the choice of scaling that a LUT designer wants to provide in his transform, including the normalization step. Floating-point values may exceed this range, but LUTs designed for output devices should also aim towards a final normalized result compatible with the integer range.

9.4 Half-float 1DLUTs

When the input to a `LUT1D` is 16f, the `halfDomain` attribute may be used to indicate that the value to be used for the lookup into the array treats the 16f value as an integer. As an example, the half-float value 1.0 will be the 15360th (zero-indexed) entry in the array. The simplest implementation of this requires that the LUT1D thus be fully enumerated with 65536 entries. The design of this 1D array must take into account the presence of negative numbers, infinities, and NaNs in the original 16f bit pattern.

(code values 31744-32767 and 64513-65535 are infinity or NaNs)

Similarly, the `rawHalf` attribute may be used to indicate that the integer 16 bit output values in each position of the output array represent the bit-equivalent, half floating-point values.

The floating-point values input to a node may have been manipulated via a `Range` node which may clamp values to a specific smaller range of floating-point values (the values of 0.0 to 1.0 being a common choice).

9.5 Rescaling operations for input and output to a `ProcessNode`

The input and output bit depths of a single node may not match each other. For 1DLUTs and 3DLUTs, the output array values are assumed to be in the output bit depth. For `Matrix` elements, a scale factor must be applied within the coefficients of the matrix.

9.6 Rescaling operations for input and output to a `ProcessList`

There are scenarios where applications must change the input and output bit depth (`inBitDepth` and `outBitDepth`, respectively) of a `ProcessList`. One scenario is when the pixels supplied to or returned, when applying a transform, do not match the bit depth used by the transform creator. Another scenario is when combining two `ProcessList`s. When changing or converting float to integer bit depths, a scale factor of $2^N - 1$ is applied. The inverse scale factor is applied when converting integer to float. Conversions between differing integer bit depths may be derived from these scale factors.

9.7 Type conversion between integer and float

Conversions from float to integer if required must be done by rounding. The output bit depth does not require that any node convert and quantize to integer values. The decision to quantize is made by the application invoking the `ProcessList`.

Conversions from integers with bit depth n to float should be done by:

$$floatValue = \frac{intValue}{(2^n - 1)}$$

9.8 Interpolation Types

When an interpolation type is not listed, it is usually assumed to be linear interpolation (see example in definitions under Sampled LUT) or tri-linear in the case of a 3DLUT. Applications must support tetrahedral interpolation for 3DLUTs.

A comment field is provided to identify the type of interpolation used for a particular cube. In many cases, the details of an interpolation used within a product may not be available. The interpolation type field is currently only a hint. Therefore, there is no assurance that images processed with the same XML LUT file on different hardware or software systems will yield the same resulting image. At the moment, this problem is outside the scope of the committees work. Hopefully, details on interpolations will be published or become more widely available in the future. Further, a common reference implementation of the LUT format may be able to achieve some standardization of common interpolation types.

9.9 XML File White Space

It is desirable that the `Array` elements keep single lines per entry so that a file can quickly be scanned by a human reader. There are some difficulties with this though as XML has some non-specific methods for handling white-space and thus if files are re-written from an XML parser, exact white spacing is not necessarily maintained. XML style sheets may be used for reviewing and checking the LUT's entries to keep the line layout the same.

There is also the issue that is known to exist between Mac, Unix, and PC text files that have differing end-of-line conventions <CR><LF> vs. <CR>. This may cause collapse of the values into one long line. The 'newline' string, i.e. the byte(s) to be interpreted as ending each line of text, is the single code value $10_{10} = 0A_{16}$ (ASCII 'Line Feed' character), also indicated <lf>; this newline convention for text files is native to all *nix operating systems (including Linux and current Apple OSs).

In addition to the above, parsers for this file format can be allowed to interpret either Microsoft's <cr><lf> or old-style MacOS' <cr> newline conventions, but they should not generate CommonLUT files with this encoding.

9.10 Extensions

It is recommended that if readers find an unrecognized element outside of the `Info` block that they either raise an error or at least provide a warning message to the user. This could be an indication that there is a `ProcessNode` that is not recognized by the reader.

The `Info` block in the `ProcessList` should be used for custom metadata that is unlikely to be recognized by other applications.

10 Conclusions

The XML format provides an extensible and straightforward method for exchanging LUT and matrix data between users and facilities. New types of `ProcessNode`s can be defined in the future to handle other desired forms of color processing. Among possible additions are nodes that provide CTL (Color Transformation Language) processing or custom formula transforms such as the ACES RRT.

Feedback from users and from implementers may lead to changes in the specification or schema, so readers should verify usage of the latest version of this document. At this time, there is no reference implementation available for distribution.

The Academy/ASC Common LUT Format is a powerful tool for handling the exchange of LUTs and color

transformations between facilities. It has the potential to become an archival element that would be as useful to the industry as the paper tape timing lights still found today in film cans.

Appendix A

(normative)

XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>

<schema targetNamespace="urn:NATAS:AMPAS:LUT:v2.0"
    xmlns="http://www.oscars.org/aces/ref/CLFschema_v2.0"
    xmlns:lut="urn:NATAS:AMPAS:LUT:v2.0"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

<!--  Process List definition  -->
    <element name="ProcessList" type="lut:ProcessListType"/>

    <complexType name="ProcessListType">
        <sequence>
            <element name="Description" type="string" minOccurs="0"
                maxOccurs="unbounded"/>
            <element name="InputDescriptor" type="string" minOccurs="0" maxOccurs="1"/>
            <element name="OutputDescriptor" type="string" minOccurs="0" maxOccurs="1"/>
            <element ref="lut:InfoType" minOccurs="1" maxOccurs="1"/>
            <element ref="lut:ProcessNode" minOccurs="1" maxOccurs="unbounded"/>
        </sequence>
        <attribute name="id" type="anyURI" use="required"/>
        <attribute name="name" type="string" use="optional"/>
        <attribute name="compatibleCLFversion" type="string" use="required"/>
        <attribute name="inverseOf" type="string" use="optional"/>
    </complexType>

<!--  Info element definition  -->
    <element name="Info" type="lut:InfoType"/>

    <complexType name="InfoType">
        <sequence>
            <element name="AppRelease" type="string" minOccurs="0" maxOccurs="1"/>
            <element ref="lut:CalibrationInfoType" minOccurs="0" maxOccurs="1"/>
        </sequence>
    </complexType>

<!--  Info element definition  -->
    <element name="CalibrationInfo" type="lut:CalibrationInfoType"/>

    <complexType name="CalibrationInfoType">
        <sequence>
            <element name="DisplayDeviceSerialNum" type="string" minOccurs="0"
                maxOccurs="1"/>
            <element name="DisplayDeviceHostName" type="string" minOccurs="0"
                maxOccurs="1"/>
            <element name="OperatorName" type="string" minOccurs="0" maxOccurs="1"/>
            <element name="CalibrationDateTime" type="string" minOccurs="0"
                maxOccurs="1"/>
            <element name="MeasurementProbe" type="string" minOccurs="0"
                maxOccurs="1"/>
            <element name="CalibrationSoftwareName" type="string" minOccurs="0"
                maxOccurs="1"/>
            <element name="CalibrationSoftwareVersion" type="string" minOccurs="0"
                maxOccurs="1"/>
        </sequence>
    </complexType>

    <!--  ProcessNode definition  -->
    <element name="ProcessNode" type="lut:ProcessNodeType"/>
```

```
<complexType name="ProcessNodeType" abstract="true">
    <sequence>
        <element name="Description" type="string" minOccurs="0"
            maxOccurs="unbounded"/>
    </sequence>
    <attribute name="id" type="anyURI" use="optional"/>
    <attribute name="name" type="string" use="optional"/>
    <attribute name="inBitDepth" type="lut:bitDepthType" use="required"/>
    <attribute name="outBitDepth" type="lut:bitDepthType" use="required"/>
</complexType>

<!--  ProcessNode: LUT1D definition  -->
<element name="LUT1D" type="lut:LUT1DType" substitutionGroup="lut:ProcessNode"/>

<complexType name="LUT1DType">
    <complexContent>
        <extension base="lut:ProcessNodeType">
            <sequence>
                <element name="IndexMap" type="lut:IndexMapType" minOccurs="0"
                    maxOccurs="3"/>
                <element name="Array" type="lut:ArrayType" minOccurs="1"
                    maxOccurs="1"/>
            </sequence>
            <attribute name="interpolation" type="string" use="optional"/>
            <attribute name="rawHalfs" type="string" use="optional"/>
            <attribute name="halfDomain" type="string" use="optional"/>
        </extension>
    </complexContent>
</complexType>


<!--  ProcessNode: LUT3D definition  -->
<element name="LUT3D" type="lut:LUT3DType" substitutionGroup="lut:ProcessNode"/>

<complexType name="LUT3DType">
    <complexContent>
        <extension base="lut:ProcessNodeType">
            <sequence>
                <element name="IndexMap" type="lut:IndexMapType" minOccurs="0"
                    maxOccurs="3"/>
                <element name="Array" type="lut:ArrayType" minOccurs="1"
                    maxOccurs="1"/>
            </sequence>
            <attribute name="interpolation" type="string" use="optional"/>
        </extension>
    </complexContent>
</complexType>

<!--  ProcessNode: Range definition  -->
<element name="Range" type="lut:RangeType" substitutionGroup="lut:ProcessNode"/>

<complexType name="RangeType">
    <complexContent>
    <extension base="lut:ProcessNodeType">
        <sequence>
            <element name="minValueIn" type="float" minOccurs="0" maxOccurs="1"/>
            <element name="maxValueIn" type="float" minOccurs="0" maxOccurs="1"/>
            <element name="minValueOut" type="float" minOccurs="0" maxOccurs="1"/>
            <element name="maxValueOut" type="float" minOccurs="0" maxOccurs="1"/>
        </sequence>
       <attribute name="style" type="string" use="optional"/>
    </extension>
    </complexContent>
</complexType>

<!--  ProcessNode: Matrix definition  -->
```

```
    <element name="Matrix" type="lut:MatrixType" substitutionGroup="lut:ProcessNode"/>

    <complexType name="MatrixType">
        <complexContent>
            <extension base="lut:ProcessNodeType">
                <sequence>
                    <element name="Array" type="lut:ArrayType" minOccurs="1"
                        maxOccurs="1"/>
                </sequence>
            </extension>
        </complexContent>
    </complexType>

    <complexType name="ArrayType">
        <simpleContent>
            <extension base="lut:floatListType">
                <attribute name="dim" type="lut:dimType" use="optional"/>
            </extension>
        </simpleContent>
    </complexType>

    <complexType name="IndexMapType">
        <simpleContent>
        <extension base="lut:indexMapItemsType">
            <attribute name="dim" type="lut:dimType" use="optional"/>
        </extension>
        </simpleContent>
    </complexType>

    <simpleType name="indexMapItemsType">
        <restriction base="lut:indexMapItemListType">
            <minLength value="2"/>
        </restriction>
    </simpleType>

    <simpleType name="indexMapItemListType">
        <list>
           <simpleType>
                <restriction base="string">
                    <pattern value="[0-9]+(\.[0-9]+)?@[0-9]+(\.[0-9]+)?"/>
                </restriction>
           </simpleType>
           </list>
    </simpleType>

    <simpleType name="floatListType">
        <list itemType="float"/>
    </simpleType>

    <simpleType name="dimType">
        <restriction base="lut:positiveIntegerListType">
            <minLength value="1"/>
        </restriction>
    </simpleType>

    <simpleType name="positiveIntegerListType">
        <list itemType="positiveInteger"/>
    </simpleType>

    <simpleType name="bitDepthType">
        <restriction base="string">
            <pattern value="[0-9]+[fi]"/>
        </restriction>
    </simpleType>

</schema>
```

Appendix B

(normative)

Use of the Common LUT format for implementation of ACES Color Transforms

The `ProcessList id` attribute may be used to contain the `ACEStransformID` which is mandatory for ACES transforms.

The following elements may be used within the `Info` block of the `ProcessList` element.

`ACEStransformID` (required for ACES transforms)
This element contains an ACES transform identifier as described in Academy S-2014-002. If the transform is the combination of several ACES component transforms, this element may contain several ACES Transform Identifiers, separated by white space or line separators.

`ACESuserName` (required for ACES transforms)
This element contains the user-friendly name recommended for use in product user interfaces as described in Academy TB-2014-002.

Appendix C

(normative)

`IndexMap` Element

Full support for an arbitrary `IndexMap` is now considered an extension to the CLF format as it complicated some implementation decisions. A reader should still recognize the token `IndexMap` structure and if only two elements are present should treat them as described. All other dimensions of an `IndexMap` are considered extensions and a user should be notified if the current reader does not support this extension.

`IndexMap` definition

The mapping of input code values to indexes of the table is modifiable allowing for remapping the range of applicable input values, changing the spacing of the input sampling function, and reshaping of the index function for lookups into the LUT. For example, with a 16-bit floating-point code value, the table may be defined to only operate upon input values in the range from 0.0 to 1.0. Values outside of this range are not able to be looked up within the LUT range and extrapolation is not required. A reshaping function is useful to modify the density of sampled values in different regions of the LUT. Reshaping changes the relationship between input code values and the lookup position into the LUT.

An `IndexMap` is specified with a list containing pairs of values, $inValue@i$, which replaces the original $inValue$ with the new $inValue$ at each entry i in the table.

In this example of reshaping, the precision of the higher input values is increased with more samples in that region of the function with the following set of index values:

IndexMap: $0@0 \quad 512@1 \quad 756@2 \quad 900@3 \quad 1023@4$

This changes a table from the behavior of the left side to the new behavior on the right side. Note that the pixel output results of each of these transforms would be different.

Without `IndexMap`		
index	inValue	outValue
0	0	0
1	255	10
2	511	100
3	767	1000
4	1023	1023

With `IndexMap`		
index	inValue	outValue
0	0	0
1	512	10
2	756	100
3	900	1000
4	1023	1023

XML Example

As an example of an `IndexMap`, in a 5 position LUT1D with 10-bit input values, the IndexMap could be the following:

```
<IndexMap dim=5> 60@0 301@1 542@2 782@3 1023@4 </IndexMap>
```

An input pixel value of 301 would be calculated using the output value in lut position ‘1.’

The first value in the `IndexMap` represents the minimum expected input value for the LUT and must contain the new input value for $lut[0]$. The last value in the `IndexMap` must always define the maximum expected input value, which is associated with the last entry in the LUT, $lut[n-1]$. Values outside of the range of the `IndexMap` are looked up at either the minimum or the maximum input values.

In a default `IndexMap`, the range of available inValues is spread equally (i.e. linear interpolation) across all of the available array indexes. Each index position in the array has a new inValue corresponding to that index position.

In the simplest case, an `IndexMap` listing only two values, $inMin$ and $inMax$ defines the range of inValues that the LUT operates on where the $inValue$ for each index in the LUT ($n = 4$) is linearly calculated as:

i	$inValue$	$outValue$
0	$inMin$	5
1	$\left(\frac{i}{(n-1)} \times (inMax - inMin)\right) + inMin$	100
2	$\left(\frac{i}{(n-1)} \times (inMax - inMin)\right) + inMin$	1000
3	$inMax$	10000

Values outside of the range of the `IndexMap` are looked up at either the minimum or the maximum input values. For example, with 16-bit floating-point codes (`inBitDepth="16"`), a LUT may have been constructed to use only the input values of $[0 \ldots 1.0]$, in which case, the minimum and maximum input values will be 0.0 and 1.0, respectively. If there was an input value of -1.0, the lookup for the minimum input value would be used.

In the absence of an `IndexMap`, the default case, the minimum and maximum input values are determined using the full range of available code values based on the input bit depth and type.

In a complete `IndexMap`, there is a value provided for each index into the table. This is desired particularly in the case of a LUT3D so that the exact input value of each entry is available.

An `IndexMap` will never contain more entries than the lookup table itself.

If only one `IndexMap` is present in a LUT node, it is applied for all the color components. If there is more than one `IndexMap`, there must be one `IndexMap` for each component (in RGB order) for a total of three.

Example

```
<LUT1D id="lut-25" name="shaper LUT" inBitDepth="10i" outBitDepth="16f">
    <Description> 1D LUT with shaper </Description>
    <IndexMap dim=4>0@0 10@100 20@250 30@360 40@440 445@445
        700@600 800@700 900@850 950@1023</IndexMap>
    <Array dim="1024 1">
        0.00
        0.32
        0.50
        <1020 entries omitted>
        1.0
    </Array>
</LUT1D>
```

Example 7 – Example of a partially enumerated "Shaper LUT"

An `IndexMap` (optional extension) is used to reshape the sampling function in the LUT. While it is sometimes possible to combine LUT calculations so that a single LUT would suffice, there are also cases where it is convenient to separate the array data from the sampling function.

In a partial `IndexMap`, there are greater than 2 items in the list but fewer items than there are index positions into the table. This is the case, for example, where you want to use a 10-point function to change the input shape of a 10-bit LUT that has 1023 entries.

```
<IndexMap dim=10>0@0 10@100 20@250 30@360 40@440 445@445 700@600 800@700 900@850
    950@1023</IndexMap>
```

Each index position in the table will have a new $inValue$ calculated using the function in the `IndexMap` list.

In this example, the input sample function is changed to stretch the regions of the LUT that process the shadows and highlights. An input value of 10 is instead placed at position 100 in the LUT allowing greater definition in the LUT for all values between 0 and 10 (positions $0 \ldots 100$ in the LUT). In the highlights, an input value of 800 is instead placed at 700, and 900 is instead at position 850. This gives the highlights 150 positions in the LUT as compared to the original 100 positions. Needless to say the accuracy of the middle region of the LUT is compressed.

Another possible use is to apply an overall delta to the data in the array, perhaps with an offset or gain function. If the input data is 'density' (log with a gamma), and the LUT represents a film print emulation, then applying an offset to the lookup with a shaper LUT will simulate the effect of a printer light change.

Indexing Calculations with `IndexMap`

A `ProcessNode` using LUT tables must perform an index calculation to take the range of input values and ratio them to the input 'index' range of the table (i.e. the minimum and maximum index positions into the table). This allows the LUT location calculation to be easily achieved as the normalized index function can be multiplied by the number of entries in the LUT to get a direct hash function to the appropriate LUT locations. For integer inputs, this is straightforward as the inBitDepth attribute may be used to apply the whole range of input across the whole range of index positions.

For floating-point inputs, the `IndexMap` element can be used to select the beginning and end values of the input floats, and apply them across the range of index positions. The LUT designer may specify range selection with an `IndexMap`, and/or convert floats in a prior `Range` node or 1DLUT. In the absence of an `IndexMap`, the [0 to 1.0] range is applied across the input index range.

Two methods are possible for increasing the accuracy of the LUT calculation by changing the density of samples in a particular region of the LUT. One is to use a 1DLUT node as a 'shaper LUT' ahead of a main LUT. The other method which has a more compact representation is to use the `IndexMap` as a re-indexing function. The array in the main LUT node and the `IndexMap` are specified at the same time to provide better sampling using the selected interpolation method. The `IndexMap` defines values at specific array coordinates. For this second method, the `IndexMap` values are first used to calculate a hash function (the `IndexMap` table) that should have as many entries as the number of entries in the target LUT. This table contains output entries that are fractional coordinates into the array of the main LUT. The first lookup into the `IndexMap` uses the standard LUT access method ($rangefraction \times numEntries$) to find the second LUT entry point. This second LUT output value is used for direct access to the main LUT whether a 1DLUT or a 3DLUT where the fractional amount provides the interpolation ratio to the next coordinate of the LUT. A different `IndexMap` will require revision to the main LUT node to place entries in new positions.

Appendix D

(informative)

Changes between v1.0.1 and v2.0

- `IndexMaps` with dim > 2 are an extension and are not required
- Add `compCLFversion` to `ProcessList`
- Add `Clamp` vs `noClamp` styles to `Range` node
- ASC CDL processing node added
- `Info` element added to `ProcessList`
- `CalibrationInfo` element added to `ProcessList`
- appendix for ACES metadata added
- `rawHalf` and `HalfDomain` processing for `LUT1Ds` is added